1. **Behavioural design pattern**

A. Chain of Responsibility

Purpose: This pattern allows a request to pass through a chain of handlers, where each handler decides either to process the request or to pass it along to the next handler in the chain.
Decouples the sender of a request from its receiver by giving multiple objects a chance to handle the request.

```
interface SupportHandler {
    void setNextHandler(SupportHandler next);
    void handleRequest(String request);
}

class BasicSupport implements SupportHandler {
    private SupportHandler nextHandler;

    public void setNextHandler(SupportHandler next) {
        this.nextHandler = next;
    }

    public void handleRequest(String request) {
        if (request.equals("basic")) {
            System.out.println("Basic support handled the request.");
        } else if (nextHandler != null) {
            nextHandler.handleRequest(request);
        } else {
            System.out.println("Request not handled.");
        }
    }
}

class ManagerSupport implements SupportHandler {
    private SupportHandler nextHandler;

    public void setNextHandler(SupportHandler next) {
        this.nextHandler = next;
    }

    public void handleRequest(String request) {
        if (request.equals("complex")) {
            System.out.println("Manager handled the request.");
        } else if (nextHandler != null) {
            nextHandler.handleRequest(request);
        } else {
            System.out.println("Request not handled.");
        }
    }
}
```

```
public class Main {
   public static void main(String[] args) {
      SupportHandler basicSupport = new BasicSupport();
      SupportHandler managerSupport = new ManagerSupport();

      basicSupport.setNextHandler(managerSupport);

      basicSupport.handleRequest("basic");
      basicSupport.handleRequest("complex");
   }
}
```

B. Interpreter

Purpose: The Interpreter pattern defines a grammar for a language and provides an interpreter to process sentences in that language.
A simple calculator that interprets math expressions like "5 + 3 – 2".

```
interface Expression {

   int interpret();

}


class Number implements Expression {

   private int number;


   public Number(int number) {

      this.number = number;

   }


   public int interpret() {

      return number;

   }

}


class Add implements Expression {

   private Expression leftExpression;

   private Expression rightExpression;
```

```
    public Add(Expression left, Expression right) {
        this.leftExpression = left;
        this.rightExpression = right;
    }

    public int interpret() {
        return leftExpression.interpret() + rightExpression.interpret();
    }
}

class Subtract implements Expression {
    private Expression leftExpression;
    private Expression rightExpression;

    public Subtract(Expression left, Expression right) {
        this.leftExpression = left;
        this.rightExpression = right;
    }

    public int interpret() {
        return leftExpression.interpret() - rightExpression.interpret();
    }
}

public class Main {
    public static void main(String[] args) {
        Expression five = new Number(5);
        Expression three = new Number(3);
        Expression two = new Number(2);

        Expression addExpr = new Add(five, three);
        Expression subExpr = new Subtract(addExpr, two);

        System.out.println("Result: " + subExpr.interpret());
    }
}
```

2. Creational design pattern

A. **Singleton pattern** ensures that a class has only one instance and provides a global point of access to that instance.

Use Case:

Consider a **logging system** where we want to ensure that only one instance of the logger exists throughout the application to avoid multiple loggers writing to the same file or console in a concurrent environment.

```
class Logger {

    private static Logger instance;
    private Logger() {
        System.out.println("Logger initialized");
    }
    public static Logger getInstance() {
        if (instance == null) {
            instance = new Logger();
        }
        return instance;
    }
    public void log(String message) {
        System.out.println("Log message: " + message);
    }
}

public class Main {
    public static void main(String[] args) {
        Logger logger1 = Logger.getInstance();
        logger1.log("First log message");


        Logger logger2 = Logger.getInstance();
        logger2.log("Second log message");
        System.out.println(logger1 == logger2);  // Output: true
    }
}
```

B. Factory Method Pattern

The **Factory Method pattern** defines an interface for creating objects but lets subclasses alter the type of objects that will be created. It is useful when the creation logic is complex or when the system needs to decide which subclass or object to instantiate at runtime.

Use Case:

Consider an application that **generates different types of notifications** (e.g., email, SMS, or push notifications). The type of notification is determined at runtime, and we want to encapsulate the creation logic for different notification types.

```
interface Notification {
    void notifyUser();
}

// Concrete Email Notification
class EmailNotification implements Notification {
    public void notifyUser() {
        System.out.println("Sending an Email Notification.");
    }
}

// Concrete SMS Notification
class SMSNotification implements Notification {
    public void notifyUser() {
        System.out.println("Sending an SMS Notification.");
    }
```

```
}

// Concrete Push Notification
class PushNotification implements Notification {
    public void notifyUser() {
        System.out.println("Sending a Push Notification.");
    }
}

// Factory Class
class NotificationFactory {
    public Notification createNotification(String type) {
        if (type == null || type.isEmpty()) {
            return null;
        }
        switch (type.toLowerCase()) {
            case "email":
                return new EmailNotification();
            case "sms":
                return new SMSNotification();
            case "push":
                return new PushNotification();
            default:
                throw new IllegalArgumentException("Unknown notification type " + type);
        }
    }
}

public class Main {
    public static void main(String[] args) {
        NotificationFactory factory = new NotificationFactory();
        // Create and use Email notification
        Notification notification1 = factory.createNotification("email");
        notification1.notifyUser();  // Output: Sending an Email Notification.

        Notification notification2 = factory.createNotification("sms");
        notification2.notifyUser();  // Output: Sending an SMS Notification.

        // Create and use Push notification
        Notification notification3 = factory.createNotification("push");
        notification3.notifyUser();  // Output: Sending a Push Notification.
    }
}
```

3. Structural design pattern

A. Adapter Pattern

The **Adapter pattern** allows incompatible interfaces to work together. It acts as a bridge between two incompatible interfaces, enabling communication.

Use Case:

Consider a scenario where you have an existing system that uses **legacy code** with a specific interface, and you want to integrate a new third-party library that has a different interface.

```
interface OldSystem {
    void oldMethod();
}

// Legacy class implementing the old interface
class LegacySystem implements OldSystem {
    public void oldMethod() {
        System.out.println("Executing old method.");
    }
}
```

```
interface NewSystem {
    void newMethod();
}
class SystemAdapter implements NewSystem {
    private OldSystem oldSystem;

    public SystemAdapter(OldSystem oldSystem) {
        this.oldSystem = oldSystem;
    }

    public void newMethod() {
        oldSystem.oldMethod(); // Adapting to the new method
    }
}

public class Main {
    public static void main(String[] args) {
        OldSystem legacySystem = new LegacySystem();
        NewSystem adaptedSystem = new SystemAdapter(legacySystem);
        adaptedSystem.newMethod(); // Output: Executing old method.
    }
}
```

B. Bridge Pattern

The **Bridge pattern** separates an abstraction from its implementation, allowing them to vary independently. This pattern is particularly useful when both the class hierarchy and the implementation hierarchy can vary.

Use Case:

Consider a scenario where you have different types of **shapes** (e.g., circles, squares) that can be rendered in different **colors** (e.g., red, blue). You want to create a flexible system that can easily accommodate new shapes or colors without modifying existing code.

```
interface Color {
    String fillColor();
```

```
}

class Red implements Color {
   public String fillColor() {
      return "Red";
   }
}

// Concrete implementation for Blue
class Blue implements Color {
   public String fillColor() {
      return "Blue";
   }
}
abstract class Shape {
   protected Color color;

   protected Shape(Color color) {
      this.color = color;
   }

   abstract void draw();
}

class Circle extends Shape {
   public Circle(Color color) {
      super(color);
   }

   void draw() {
      System.out.println("Drawing Circle in " + color.fillColor());
   }
```

```
}

class Square extends Shape {
    public Square(Color color) {
        super(color);
    }

    void draw() {
        System.out.println("Drawing Square in " + color.fillColor());
    }
}

public class Main {
    public static void main(String[] args) {
        Shape redCircle = new Circle(new Red());
        Shape blueSquare = new Square(new Blue());

        redCircle.draw();
        blueSquare.draw();
    }
}
```