1. You are developing a vacation package planner system. Customers can create a custom vacation package by choosing options such as destination, transportation mode, hotel type, number of days, and additional activities (e.g., sightseeing, adventure sports). Create the system must ensure that a vacation package

1. Which is the design pattern suitable for this scenario and why?
2. Write a code example in a programming language of your choice to implement the Builder pattern for the vacation package planner.

Answer:

1. Why Builder is suitable:
The Builder design pattern is suitable because:

- It allows for step-by-step creation of a complex object (vacation package).
- It provides flexibility to add or skip specific options while ensuring the object remains valid.
- It separates the construction process from the final representation, making the code modular and reusable.

```
// VacationPackage class

public class VacationPackage {

   private String destination;

   private String transportation;

   private String hotelType;

   private int numberOfDays;

   private String activities;


   // Private constructor to be called by the Builder

   private VacationPackage(VacationPackageBuilder builder) {

      this.destination = builder.destination;

      this.transportation = builder.transportation;

      this.hotelType = builder.hotelType;

      this.numberOfDays = builder.numberOfDays;
```

```
    this.activities = builder.activities;
}

@Override
public String toString() {
    return "VacationPackage [Destination: " + destination +
            ", Transportation: " + transportation +
            ", Hotel Type: " + hotelType +
            ", Number of Days: " + numberOfDays +
            ", Activities: " + activities + "]";
}

// Static Builder Class
public static class VacationPackageBuilder {
    private String destination;
    private String transportation;
    private String hotelType;
    private int numberOfDays;
    private String activities = "";

    public VacationPackageBuilder setDestination(String destination) {
        this.destination = destination;
        return this;
    }

    public VacationPackageBuilder setTransportation(String transportation) {
        this.transportation = transportation;
        return this;
```

```
    }

    public VacationPackageBuilder setHotelType(String hotelType) {
      this.hotelType = hotelType;
      return this;
    }

    public VacationPackageBuilder setNumberOfDays(int numberOfDays) {
      this.numberOfDays = numberOfDays;
      return this;
    }

    public VacationPackageBuilder addActivity(String activity) {
      if (!activities.isEmpty()) {
        activities += ", ";
      }
      activities += activity;
      return this;
    }

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

// Main Class to Test
public class VacationPackagePlanner {
```

```
    public static void main(String[] args) {
        // Build a custom vacation package
        VacationPackage tropicalVacation = new
VacationPackage.VacationPackageBuilder()
            .setDestination("Hawaii")
            .setTransportation("Flight")
            .setHotelType("5-Star Resort")
            .setNumberOfDays(7)
            .addActivity("Snorkeling")
            .addActivity("Hiking")
            .addActivity("Beach Party")
            .build();

        System.out.println(tropicalVacation);

        // Build another vacation package
        VacationPackage cityBreak = new VacationPackage.VacationPackageBuilder()
            .setDestination("New York")
            .setTransportation("Train")
            .setHotelType("3-Star Hotel")
            .setNumberOfDays(3)
            .addActivity("Museum Visit")
            .addActivity("Broadway Show")
            .build();

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

2.You are tasked with designing a system for assembling custom computer configurations. Each computer can have different components such as a CPU, GPU, RAM, storage, power supply, and additional features like RGB lighting or a liquid cooling system. The system should allow customers to create a computer step-by-step based on their preferences.

1. Why is the Builder design pattern suitable for this scenario?
2. Write a code example in a programming language of your choice to implement the Builder pattern for the computer system.

Answer:

1. Why Builder is suitable:
The Builder design pattern is suitable because:

- It allows step-by-step creation of complex objects (custom computer configurations).
- It provides flexibility in choosing components while ensuring a valid computer is built.
- It separates the construction logic from the representation, making the code more maintainable.

```
// Computer class

public class Computer {

   private String CPU;

   private String GPU;

   private int RAM;

   private int storage;

   private String powerSupply;

   private String additionalFeatures;


   // Private constructor

   private Computer(ComputerBuilder builder) {

      this.CPU = builder.CPU;

      this.GPU = builder.GPU;

      this.RAM = builder.RAM;
```

```
    this.storage = builder.storage;
    this.powerSupply = builder.powerSupply;
    this.additionalFeatures = builder.additionalFeatures;
  }

  @Override
  public String toString() {
    return "Computer [CPU: " + CPU + ", GPU: " + GPU + ", RAM: " + RAM + "GB, Storage: "
+ storage + "GB, " +
        "Power Supply: " + powerSupply + ", Additional Features: " + additionalFeatures +
"]";
  }

  // Static Builder Class
  public static class ComputerBuilder {
    private String CPU;
    private String GPU;
    private int RAM;
    private int storage;
    private String powerSupply;
    private String additionalFeatures = "";

    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 ComputerBuilder setPowerSupply(String powerSupply) {
    this.powerSupply = powerSupply;
    return this;
}

public ComputerBuilder addAdditionalFeature(String feature) {
    if (!additionalFeatures.isEmpty()) {
        additionalFeatures += ", ";
    }
    additionalFeatures += feature;
    return this;
}

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

```
    }
  }
}

// Main Class to Test
public class ComputerBuilderDemo {
  public static void main(String[] args) {
    // Build a high-performance gaming PC
    Computer gamingPC = new Computer.ComputerBuilder()
      .setCPU("Intel Core i9")
      .setGPU("NVIDIA RTX 4090")
      .setRAM(32)
      .setStorage(2000)
      .setPowerSupply("850W Gold")
      .addAdditionalFeature("RGB Lighting")
      .addAdditionalFeature("Liquid Cooling")
      .build();

    System.out.println(gamingPC);

    // Build an office PC
    Computer officePC = new Computer.ComputerBuilder()
      .setCPU("Intel Core i5")
      .setGPU("Integrated Graphics")
      .setRAM(16)
      .setStorage(512)
      .setPowerSupply("500W Bronze")
      .build();
```

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

Question: Vehicle Manufacturing System

Scenario: You are designing a system for manufacturing vehicles. Each vehicle can have different configurations, such as engine type, number of wheels, color, transmission, and features like air conditioning or a sunroof. The system must allow step-by-step customization of the vehicle while ensuring that all configurations result in a valid vehicle object.

1. Why is the Builder design pattern suitable for this scenario?
2. Write a code example in a programming language of your choice to implement the Builder pattern for the vehicle manufacturing system.

Answer:

1. Why Builder is suitable:
The Builder design pattern is suitable because:

- It simplifies the creation of complex objects (vehicles with multiple configurations).
- It provides a clear and step-by-step process for configuring a vehicle.
- It separates the vehicle construction logic from its representation, allowing flexibility for different vehicle types (e.g., cars, bikes, trucks).

```
// Vehicle class
public class Vehicle {
    private String engine;
    private int wheels;
    private String color;
    private String transmission;
    private String features;

    // Private constructor to be called by the Builder
```

```
private Vehicle(VehicleBuilder builder) {
    this.engine = builder.engine;
    this.wheels = builder.wheels;
    this.color = builder.color;
    this.transmission = builder.transmission;
    this.features = builder.features;
}

@Override
public String toString() {
    return "Vehicle [Engine: " + engine + ", Wheels: " + wheels +
          ", Color: " + color + ", Transmission: " + transmission +
          ", Features: " + features + "]";
}

// Static Builder Class
public static class VehicleBuilder {
    private String engine;
    private int wheels;
    private String color;
    private String transmission;
    private String features = "";

    public VehicleBuilder setEngine(String engine) {
        this.engine = engine;
        return this;
    }
```

```
        public VehicleBuilder setWheels(int wheels) {
            this.wheels = wheels;
            return this;
        }

        public VehicleBuilder setColor(String color) {
            this.color = color;
            return this;
        }

        public VehicleBuilder setTransmission(String transmission) {
            this.transmission = transmission;
            return this;
        }

        public VehicleBuilder addFeature(String feature) {
            if (!features.isEmpty()) {
                features += ", ";
            }
            features += feature;
            return this;
        }

        public Vehicle build() {
            return new Vehicle(this);
        }
    }
}
```

```
// Main Class to Test
public class VehicleBuilderDemo {
    public static void main(String[] args) {
        // Build a custom vehicle
        Vehicle car = new Vehicle.VehicleBuilder()
            .setEngine("V6")
            .setWheels(4)
            .setColor("Red")
            .setTransmission("Automatic")
            .addFeature("Air Conditioning")
            .addFeature("Sunroof")
            .build();

        System.out.println(car);

        // Build another vehicle
        Vehicle bike = new Vehicle.VehicleBuilder()
            .setEngine("Single Cylinder")
            .setWheels(2)
            .setColor("Black")
            .setTransmission("Manual")
            .addFeature("Disc Brakes")
            .build();

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

Question: Book Publishing System

Scenario: You are designing a book publishing system for an online bookstore. Each book can have various attributes such as title, author, ISBN, genre, price, and additional features like hardcover or audiobook format. The system should allow the creation of a book with optional features while ensuring a valid book is created step-by-step.

1. Why is the Builder design pattern suitable for this scenario?
2. Write a code example in a programming language of your choice to implement the Builder pattern for the book publishing system.

Answer:

1. Why Builder is suitable:
The Builder design pattern is suitable because:

- It helps construct a complex object (the book) with multiple attributes.
- It provides flexibility to customize the book's attributes while ensuring the object remains valid.
- It allows optional features to be added without complicating the creation process, such as choosing between paperback and audiobook formats.

```
// Book class
public class Book {
    private String title;
    private String author;
    private String ISBN;
    private String genre;
    private double price;
    private String additionalFeatures;

    // Private constructor to be called by the Builder
    private Book(BookBuilder builder) {
        this.title = builder.title;
        this.author = builder.author;
        this.ISBN = builder.ISBN;
```

```
        this.genre = builder.genre;
        this.price = builder.price;
        this.additionalFeatures = builder.additionalFeatures;
    }

    @Override
    public String toString() {
        return "Book [Title: " + title + ", Author: " + author + ", ISBN: " + ISBN +
                ", Genre: " + genre + ", Price: $" + price + ", Additional Features: " +
additionalFeatures + "]";
    }

    // Static Builder Class
    public static class BookBuilder {
        private String title;
        private String author;
        private String ISBN;
        private String genre;
        private double price;
        private String additionalFeatures = "";

        public BookBuilder setTitle(String title) {
            this.title = title;
            return this;
        }

        public BookBuilder setAuthor(String author) {
            this.author = author;
```

```
    return this;
}

public BookBuilder setISBN(String ISBN) {
    this.ISBN = ISBN;
    return this;
}

public BookBuilder setGenre(String genre) {
    this.genre = genre;
    return this;
}

public BookBuilder setPrice(double price) {
    this.price = price;
    return this;
}

public BookBuilder addAdditionalFeature(String feature) {
    if (!additionalFeatures.isEmpty()) {
        additionalFeatures += ", ";
    }
    additionalFeatures += feature;
    return this;
}

public Book build() {
    return new Book(this);
```

```
    }
  }
}

// Main Class to Test
public class BookPublishingSystem {
  public static void main(String[] args) {
    // Build a custom book
    Book novel = new Book.BookBuilder()
      .setTitle("The Great Adventure")
      .setAuthor("John Doe")
      .setISBN("978-1-234567-89-7")
      .setGenre("Fiction")
      .setPrice(19.99)
      .addAdditionalFeature("Hardcover")
      .addAdditionalFeature("Signed by Author")
      .build();

    System.out.println(novel);

    // Build another book
    Book textbook = new Book.BookBuilder()
      .setTitle("Java Programming for Beginners")
      .setAuthor("Jane Smith")
      .setISBN("978-1-234567-90-3")
      .setGenre("Education")
      .setPrice(39.99)
      .addAdditionalFeature("Audiobook")
```

```
        .build();


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

Question: Car Configuration System

Scenario: You are designing a car configuration system for a car dealership. Customers can configure their car by selecting various options like car model, engine type, color, wheels, interior design, and additional features (e.g., sunroof, leather seats). The system must allow users to build their car step-by-step while ensuring all required components are selected.

1. Why is the Builder design pattern suitable for this scenario?
2. Write a code example in a programming language of your choice to implement the Builder pattern for the car configuration system.

Answer:

1. Why Builder is suitable:
The Builder design pattern is suitable because:

- It allows step-by-step configuration of a car with multiple customizable options.
- It provides flexibility to include optional features like a sunroof, leather seats, or a specific wheel type.
- It ensures that all the necessary components are included, and that the car configuration is valid before it is finalized.

```
// Car class
public class Car {
  private String model;
  private String engineType;
  private String color;
  private String wheels;
  private String interiorDesign;
  private String additionalFeatures;
```

```
// Private constructor
private Car(CarBuilder builder) {
    this.model = builder.model;
    this.engineType = builder.engineType;
    this.color = builder.color;
    this.wheels = builder.wheels;
    this.interiorDesign = builder.interiorDesign;
    this.additionalFeatures = builder.additionalFeatures;
}

@Override
public String toString() {
    return "Car [Model: " + model + ", Engine: " + engineType + ", Color: " + color +
        ", Wheels: " + wheels + ", Interior: " + interiorDesign +
        ", Additional Features: " + additionalFeatures + "]";
}

// Static Builder Class
public static class CarBuilder {
    private String model;
    private String engineType;
    private String color;
    private String wheels;
    private String interiorDesign;
    private String additionalFeatures = "";

    public CarBuilder setModel(String model) {
```

```
    this.model = model;
    return this;
}

public CarBuilder setEngineType(String engineType) {
    this.engineType = engineType;
    return this;
}

public CarBuilder setColor(String color) {
    this.color = color;
    return this;
}

public CarBuilder setWheels(String wheels) {
    this.wheels = wheels;
    return this;
}

public CarBuilder setInteriorDesign(String interiorDesign) {
    this.interiorDesign = interiorDesign;
    return this;
}

public CarBuilder addAdditionalFeature(String feature) {
    if (!additionalFeatures.isEmpty()) {
        additionalFeatures += ", ";
    }
```

```
        additionalFeatures += feature;
        return this;
    }

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

// Main Class to Test
public class CarConfigurationSystem {
  public static void main(String[] args) {
    // Build a custom car configuration
    Car luxuryCar = new Car.CarBuilder()
        .setModel("Mercedes-Benz S-Class")
        .setEngineType("V8")
        .setColor("Midnight Blue")
        .setWheels("19-inch Alloy")
        .setInteriorDesign("Leather")
        .addAdditionalFeature("Panoramic Sunroof")
        .addAdditionalFeature("Heated Seats")
        .build();

    System.out.println(luxuryCar);

    // Build another custom car configuration
    Car sportsCar = new Car.CarBuilder()
```

```
        .setModel("Porsche 911")
        .setEngineType("Turbocharged 6-cylinder")
        .setColor("Carmine Red")
        .setWheels("20-inch Sports")
        .setInteriorDesign("Alcantara")
        .addAdditionalFeature("Bose Sound System")
        .build();

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

Factory Method

Question: Notification System

Scenario: You are designing a notification system for an application that can send different types of notifications to users. The system should support multiple notification types, such as **Email**, **SMS**, and **Push Notifications**. Each notification type has its own sending mechanism and may include different formatting requirements. The system should provide a way to create the appropriate notification object without exposing the instantiation logic to the client code.

1. Why is the Factory design pattern suitable for this scenario?
2. Write a code example in a programming language of your choice to implement the Factory pattern for the notification system.

Answer:

1. Why Factory is suitable:
The Factory design pattern is suitable for this scenario because:

- It provides a centralized mechanism for creating instances of different notification types (Email, SMS, Push).

- The pattern hides the complexity of object creation and allows the system to be easily extended to support new types of notifications without modifying existing code.
- It promotes loose coupling, allowing the client code to interact with a common interface, not worrying about how individual notification types are instantiated.

```
// Notification Interface
public interface Notification {
    void send(String message);
}

// Concrete Email Notification Class
public class EmailNotification implements Notification {
    @Override
    public void send(String message) {
        System.out.println("Sending Email: " + message);
    }
}

// Concrete SMS Notification Class
public class SMSNotification implements Notification {
    @Override
    public void send(String message) {
        System.out.println("Sending SMS: " + message);
    }
}

// Concrete Push Notification Class
public class PushNotification implements Notification {
    @Override
```

```
    public void send(String message) {
        System.out.println("Sending Push Notification: " + message);
    }
}

// Notification Factory Class
public class NotificationFactory {
    public static Notification createNotification(String type) {
        switch (type.toUpperCase()) {
            case "EMAIL":
                return new EmailNotification();
            case "SMS":
                return new SMSNotification();
            case "PUSH":
                return new PushNotification();
            default:
                throw new IllegalArgumentException("Unknown notification type: " + type);
        }
    }
}

// Main Class to Test
public class NotificationSystem {
    public static void main(String[] args) {
        // Create and send Email notification
        Notification email = NotificationFactory.createNotification("EMAIL");
        email.send("Welcome to our service!");
```

```
        // Create and send SMS notification
        Notification sms = NotificationFactory.createNotification("SMS");
        sms.send("Your verification code is 12345.");

        // Create and send Push notification
        Notification push = NotificationFactory.createNotification("PUSH");
        push.send("You have a new message in your inbox.");
    }
}
```

Sample output

```
Sending Email: Welcome to our service!
Sending SMS: Your verification code is 12345.
Sending Push Notification: You have a new message in your inbox.
```

Question: Shape Drawing Application

Scenario: You are designing a drawing application that allows users to draw different shapes, such as Circle, Rectangle, and Triangle. Each shape has a specific way of rendering, but the user should not need to know the details of how each shape is drawn. The system should provide a way to create the appropriate shape object based on user input (e.g., "circle", "rectangle", "triangle").

Why is the Factory design pattern suitable for this scenario?

Write a code example in a programming language of your choice to implement the Factory pattern for the shape drawing application.

```
// Shape Interface
public interface Shape {
    void draw();
```

```
}

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

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

// Concrete Triangle Class
public class Triangle implements Shape {
    @Override
    public void draw() {
        System.out.println("Drawing a Triangle");
    }
}

// Shape Factory Class
public class ShapeFactory {
```

```
    public static Shape createShape(String shapeType) {
        switch (shapeType.toUpperCase()) {
            case "CIRCLE":
                return new Circle();
            case "RECTANGLE":
                return new Rectangle();
            case "TRIANGLE":
                return new Triangle();
            default:
                throw new IllegalArgumentException("Unknown shape type: " + shapeType);
        }
    }
}

// Main Class to Test
public class ShapeDrawingApp {
    public static void main(String[] args) {
        // Create and draw a Circle
        Shape circle = ShapeFactory.createShape("CIRCLE");
        circle.draw();

        // Create and draw a Rectangle
        Shape rectangle = ShapeFactory.createShape("RECTANGLE");
        rectangle.draw();

        // Create and draw a Triangle
        Shape triangle = ShapeFactory.createShape("TRIANGLE");
        triangle.draw();
```

```
    }
}
```

Sample output

Drawing a Circle

Drawing a Rectangle

Drawing a Triangle

Question: Document Management System

Scenario: You are designing a document management system that supports creating different types of documents such as Word Documents, PDF Documents, and Spreadsheet Documents. Each document type has its own specific way of being saved, edited, and displayed. The system should allow users to create the desired document type based on their input, without exposing the details of how each document type is created.

Why is the Factory design pattern suitable for this scenario?

Write a code example in a programming language of your choice to implement the Factory pattern for the document management system.

Answer:

1. Why Factory is suitable:
The Factory design pattern is suitable for this scenario because:

- It provides a centralized mechanism for creating instances of different document types based on user input.
- It encapsulates the logic for creating specific document objects, making the system easy to extend when adding new document types.
- It promotes loose coupling by allowing the client to interact with a unified interface without worrying about how each document is created or initialized.

```
// Document Interface
public interface Document {
    void open();
    void save();
}
```

```
// Concrete WordDocument Class
public class WordDocument implements Document {
    @Override
    public void open() {
        System.out.println("Opening a Word Document...");
    }

    @Override
    public void save() {
        System.out.println("Saving a Word Document...");
    }
}

// Concrete PDFDocument Class
public class PDFDocument implements Document {
    @Override
    public void open() {
        System.out.println("Opening a PDF Document...");
    }

    @Override
    public void save() {
        System.out.println("Saving a PDF Document...");
    }
}

// Concrete SpreadsheetDocument Class
```

```
public class SpreadsheetDocument implements Document {
    @Override
    public void open() {
        System.out.println("Opening a Spreadsheet Document...");
    }

    @Override
    public void save() {
        System.out.println("Saving a Spreadsheet Document...");
    }
}

// DocumentFactory Class
public class DocumentFactory {
    public static Document createDocument(String documentType) {
        switch (documentType.toUpperCase()) {
            case "WORD":
                return new WordDocument();
            case "PDF":
                return new PDFDocument();
            case "SPREADSHEET":
                return new SpreadsheetDocument();
            default:
                throw new IllegalArgumentException("Unknown document type: " +
documentType);
        }
    }
}
```

```
// Main Class to Test
public class DocumentManagementApp {
    public static void main(String[] args) {
        // Create and use a Word document
        Document wordDoc = DocumentFactory.createDocument("WORD");
        wordDoc.open();
        wordDoc.save();

        // Create and use a PDF document
        Document pdfDoc = DocumentFactory.createDocument("PDF");
        pdfDoc.open();
        pdfDoc.save();

        // Create and use a Spreadsheet document
        Document spreadsheetDoc =
DocumentFactory.createDocument("SPREADSHEET");
        spreadsheetDoc.open();
        spreadsheetDoc.save();
    }
}
```

Output

```
Opening a Word Document...
Saving a Word Document...
Opening a PDF Document...
Saving a PDF Document...
Opening a Spreadsheet Document...
Saving a Spreadsheet Document...
```

Abstract Factory Design Pattern Use Case

Question: GUI Widget Toolkit

Scenario: You are designing a GUI framework that can support multiple operating systems like **Windows**, **MacOS**, and **Linux**. Each operating system has its own style for UI components like **Buttons** and **Checkboxes**. The framework should allow developers to create UI components that match the look and feel of the operating system without knowing the specifics of each OS.

1. Why is the Abstract Factory design pattern suitable for this scenario?
2. Write a code example in a programming language of your choice to implement the Abstract Factory pattern for this GUI framework.

Answer:

1. Why Abstract Factory is suitable:
The Abstract Factory design pattern is suitable for this scenario because:

- It provides a way to create families of related objects (e.g., Buttons and Checkboxes) without specifying their concrete classes.
- It ensures consistency between related objects in a family (e.g., Windows-style components will always match).
- It makes it easy to extend the system to support new operating systems by adding new factories without modifying the existing code.

```
// Abstract Button Interface

public interface Button {

  void render();

}


// Abstract Checkbox Interface

public interface Checkbox {

  void render();

}


// Concrete WindowsButton Class

public class WindowsButton implements Button {
```

```
    @Override
    public void render() {
        System.out.println("Rendering Windows Button");
    }
}

// Concrete WindowsCheckbox Class
public class WindowsCheckbox implements Checkbox {
    @Override
    public void render() {
        System.out.println("Rendering Windows Checkbox");
    }
}

// Concrete MacOSButton Class
public class MacOSButton implements Button {
    @Override
    public void render() {
        System.out.println("Rendering MacOS Button");
    }
}

// Concrete MacOSCheckbox Class
public class MacOSCheckbox implements Checkbox {
    @Override
    public void render() {
        System.out.println("Rendering MacOS Checkbox");
    }
```

```
}

// Abstract GUIFactory Interface
public interface GUIFactory {
    Button createButton();
    Checkbox createCheckbox();
}

// Concrete WindowsFactory Class
public class WindowsFactory implements GUIFactory {
    @Override
    public Button createButton() {
        return new WindowsButton();
    }
    @Override
    public Checkbox createCheckbox() {
        return new WindowsCheckbox();
    }
}

// Concrete MacOSFactory Class
public class MacOSFactory implements GUIFactory {
    @Override
    public Button createButton() {
        return new MacOSButton();
    }

    @Override
```

```
    public Checkbox createCheckbox() {
        return new MacOSCheckbox();
    }
}

// Application Class
public class Application {
    private final Button button;
    private final Checkbox checkbox;

    public Application(GUIFactory factory) {
        button = factory.createButton();
        checkbox = factory.createCheckbox();
    }

    public void render() {
        button.render();
        checkbox.render();
    }
}

// Main Class to Test
public class Main {
    public static void main(String[] args) {
        // Use WindowsFactory
        GUIFactory windowsFactory = new WindowsFactory();
        Application windowsApp = new Application(windowsFactory);
        windowsApp.render();
```

```
        // Use MacOSFactory
        GUIFactory macFactory = new MacOSFactory();
        Application macApp = new Application(macFactory);
        macApp.render();
    }
}
```

Output

```
Rendering Windows Button
Rendering Windows Checkbox
Rendering MacOS Button
Rendering MacOS Checkbox
```

Question: Document Cloning System

Scenario:

You are building a document management system that handles different types of documents, such as reports, invoices, and memos. Each document type can have complex initializations, including headers, footers, watermarks, and other metadata.

To improve performance and simplify the creation of documents with similar properties, the system should support cloning existing documents instead of creating new ones from scratch.

Implement the Prototype pattern in a programming language of your choice to support cloning of documents.

Why Prototype Design Pattern is Suitable:
The Prototype design pattern is suitable for this scenario because:

- It allows you to create new objects by copying an existing object rather than instantiating a new one from scratch.
- It avoids the performance cost of initializing complex objects.
- It provides a way to add new document types without modifying existing code, as new prototypes can be registered dynamically.

```
import java.util.HashMap;
import java.util.Map;

// Abstract Prototype
abstract class Document implements Cloneable {
    private String title;
    private String content;

    public String getTitle() {
        return title;
    }

    public void setTitle(String title) {
        this.title = title;
    }

    public String getContent() {
        return content;
    }

    public void setContent(String content) {
        this.content = content;
    }

    @Override
    public Document clone() {
        try {
            return (Document) super.clone();
```

```
        } catch (CloneNotSupportedException e) {
            throw new RuntimeException("Cloning not supported", e);
        }
    }

    public abstract void printDocumentDetails();
}

// Concrete Prototype: Report
class Report extends Document {
    private String reportType;

    public String getReportType() {
        return reportType;
    }

    public void setReportType(String reportType) {
        this.reportType = reportType;
    }

    @Override
    public void printDocumentDetails() {
        System.out.println("Report: " + getTitle() + ", Type: " + reportType + ", Content: " +
getContent());
    }
}

// Concrete Prototype: Invoice
```

```
class Invoice extends Document {
    private String invoiceNumber;

    public String getInvoiceNumber() {
        return invoiceNumber;
    }

    public void setInvoiceNumber(String invoiceNumber) {
        this.invoiceNumber = invoiceNumber;
    }

    @Override
    public void printDocumentDetails() {
        System.out.println("Invoice: " + getTitle() + ", Number: " + invoiceNumber + ",
Content: " + getContent());
    }
}

// Prototype Registry
class DocumentRegistry {
    private final Map<String, Document> prototypes = new HashMap<>();

    public void registerPrototype(String key, Document prototype) {
        prototypes.put(key, prototype);
    }

    public Document getPrototype(String key) {
        return prototypes.get(key).clone();
```

```
    }
}

// Client Code
public class Main {
    public static void main(String[] args) {
        // Create prototypes
        Report reportPrototype = new Report();
        reportPrototype.setTitle("Annual Report");
        reportPrototype.setReportType("Financial");
        reportPrototype.setContent("This is the financial report for the year.");

        Invoice invoicePrototype = new Invoice();
        invoicePrototype.setTitle("Invoice Template");
        invoicePrototype.setInvoiceNumber("INV-000");
        invoicePrototype.setContent("This is a standard invoice.");

        // Register prototypes
        DocumentRegistry registry = new DocumentRegistry();
        registry.registerPrototype("report", reportPrototype);
        registry.registerPrototype("invoice", invoicePrototype);

        // Clone and use prototypes
        Document clonedReport = registry.getPrototype("report");
        clonedReport.setContent("This is the financial report for 2023.");
        clonedReport.printDocumentDetails();

        Document clonedInvoice = registry.getPrototype("invoice");
```

```
        clonedInvoice.setContent("This invoice is for client XYZ.");
        clonedInvoice.printDocumentDetails();
    }
}
```

Output

```
Report: Annual Report, Type: Financial, Content: This is the financial report for 2023.
Invoice: Invoice Template, Number: INV-000, Content: This invoice is for client XYZ.
```

Structural design pattern

Adapter Design Pattern

```
package LowLevelDesign.DesignPatterns.AdapterDesignPattern.Adaptee;
public interface WeightMachine {
        //return the weight in pound
        public double getWeightPound();
}
package LowLevelDesign.DesignPatterns.AdapterDesignPattern.Adapter;
        public interface WeigthMachineAdapter {
                public double getWeightInKg();
}
package LowLevelDesign.DesignPatterns.AdapterDesignPattern.Adapter;
import
LowLevelDesignn.DesignPatterns.AdapterDesignPattern.Adaptee.WeightMachine;
public class WeightMachineAdapterImpl implements WeightMachineAdapter {
        WeightMachine weightMachine;
        public WeightMachineAdapterImp1(WeightMachine weightMachine) {
                this.weightMachine=weightMachine;
        }
        public double getWeightInKg() {
```

```
                double weightInPound = weightmachine.getWeightInPound();
                //convert it to kg's
                Double weightInKg = weightInPound * .45;
                return weightInKg;
        }
}
package LowLevelDesign.DesignPatterns.AdapterDesignPattern.Client;
import ...
public class Main {
        public static main(String args[]) {
                WeightMachineAdapter weightMachineAdapter=new
WeightMachineAdapterImpl(new WeightMachineofBabies());
System.out.println(weightMachineAdapter.getWeightInKg());
}
}
```