EXERCISE 1

1. Behavioral Design Patterns

Use Case 1: Strategy Pattern - Payment Gateway

The Strategy pattern is demonstrated through a payment gateway system that allows users to pay using different payment methods (e.g., credit card, PayPal, bank transfer).

```java
// PaymentGateway.java
public interface PaymentGateway {
    void pay(double amount);
}

// CreditCardStrategy.java
public class CreditCardStrategy implements PaymentGateway {
    @Override
    public void pay(double amount) {
        System.out.println("Paid " + amount + " using credit card");
    }
}

// PayPalStrategy.java
public class PayPalStrategy implements PaymentGateway {
    @Override
    public void pay(double amount) {
        System.out.println("Paid " + amount + " using PayPal");
    }
}

// BankTransferStrategy.java
public class BankTransferStrategy implements PaymentGateway {
    @Override
    public void pay(double amount) {
        System.out.println("Paid " + amount + " using bank transfer");
    }
}

// PaymentGateway.java
public class PaymentGateway {
    private PaymentGateway strategy;

    public PaymentGateway(PaymentGateway strategy) {
        this.strategy = strategy;
    }

    public void pay(double amount) {
        strategy.pay(amount);
    }
}
```

Use Case 2: Observer Pattern - Weather Station

The Observer pattern is demonstrated through a weather station system that notifies users of weather updates.

```java
// WeatherStation.java
public interface WeatherStation {
    void registerObserver(WeatherObserver observer);
    void removeObserver(WeatherObserver observer);
    void notifyObservers();
}

// WeatherObserver.java
public interface WeatherObserver {
    void update(double temperature, double humidity);
}

// ForecastDisplay.java
public class ForecastDisplay implements WeatherObserver {
    @Override
    public void update(double temperature, double humidity) {
        System.out.println("Forecast: Temperature=" + temperature + ", Humidity=" + humidity);
    }
}

// WeatherStation.java
public class WeatherStation implements WeatherStation {
    private List<WeatherObserver> observers;
    private double temperature;
    private double humidity;

    public WeatherStation() {
        observers = new ArrayList<>();
    }

    @Override
    public void registerObserver(WeatherObserver observer) {
        observers.add(observer);
    }

    @Override
    public void removeObserver(WeatherObserver observer) {
        observers.remove(observer);
    }

    }

    public void setMeasurements(double temperature, double humidity) {
        this.temperature = temperature;
        this.humidity = humidity;
        notifyObservers();
    }
```

2. Creational Design Patterns

Use Case 1: Singleton Pattern - Logger

The Singleton pattern is demonstrated through a logger system that ensures only one instance of the logger is created.

```java
// Logger.java
public class Logger {
    private static Logger instance;

    private Logger() {}

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

    public void log(String message) {
        System.out.println("Logged: " + message);
    }
}
```

Use Case 2: Factory Pattern - Vehicle Factory

The Factory pattern is demonstrated through a vehicle factory system that creates different types of vehicles (e.g., car, truck, motorcycle).

```java
// VehicleFactory.java
public interface VehicleFactory {
    Vehicle createVehicle();
}

// Car.java
public class Car implements Vehicle {
    @Override
    public void drive() {
        System.out.println("Driving a car");
    }
}

// Truck.java
public class Truck implements Vehicle {
    @Override
    public void drive() {
        System.out.println("Driving a truck");
```

3. Structural Design Patterns

Use Case 1: Adapter Pattern - File Converter

The Adapter pattern is demonstrated through a file converter system that converts files from one format to another.

```java
// FileConverter.java
public interface FileConverter {
    void convert(String file);
}

// PDFAdapter.java
public class PDFAdapter implements FileConverter {
    @Override
    public void convert(String file) {
        System.out.println("Converting " + file + " to PDF");
    }
}

// WordAdapter.java
public class WordAdapter implements FileConverter {
    @Override
    public void convert(String file) {
        System.out.println("Converting " + file + " to Word");
    }
}

// FileConverter.java
public class FileConverter {
    private FileConverter adapter;

    public FileConverter(FileConverter adapter) {
        this.adapter = adapter;
    }

    public void convert(String file) {
        adapter.convert(file);
    }
}
```

Use Case 2: Composite Pattern - File System

The Composite pattern is demonstrated through a file system system that represents files and directories as a composite structure.

```java
// FileSystem.java
public interface FileSystem {
    void print();
}

// File.java
public class File implements FileSystem {
    private String name;

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

    @Override
    public void print() {
        System.out.println("File: " + name);
    }
}

// Directory.java
public class Directory implements FileSystem {
    private List<FileSystem> files;
    private String name;

    public Directory(String name) {
        this.name = name;
        files = new ArrayList<>();
    }

    public void add(FileSystem file) {
        files.add(file);
    }

    @Override
    public void print() {
        System.out.println("Directory: " + name);
        for (FileSystem file : files) {
            file.print();
        }
    }
}
```