Assignment 2: Observer Pattern and Factory Pattern

Course: Object-Oriented Analysis and Design

Executive Summary

In this assignment, you will implement a **Smart Home Monitoring Dashboard** that demonstrates the practical integration of the **Observer pattern** and the **Factory pattern**.

The system consists of:

- A **monitoring hub** (Subject) that receives and broadcasts sensor data
- Multiple **display panels** (Observers) that react to sensor updates
- **Configuration factories** that create appropriate panel sets for different room types

---

System Scenario

The Smart Home Monitoring System

Your system monitors environmental conditions in different types of rooms (dormitory vs. laboratory). Each room type needs different monitoring panels with different alert thresholds.

Components:

1. **MonitoringHub** (Subject): Receives sensor data and notifies all registered panels
2. **Display Panels** (Observers): Show information and trigger alerts
   - `RealtimePanel`: Shows current sensor readings
   - `AlarmPanel`: Triggers alerts when smoke exceeds a threshold
3. **Configuration Factories**: Create appropriate panel sets for different room types
   - `DormRoomFactory`: Creates panels with relaxed thresholds
   - `LabRoomFactory`: Creates panels with strict thresholds

Task 1: Implement Observer Pattern (40 points)

Objective

Create a loosely-coupled notification system where the hub doesn't need to know about specific panel types.

1.1 Define Observer Pattern Interfaces

Create a generic `Subject` interface:

```
public interface Subject<T> {
    void registerObserver(Observer<T> observer);
    void removeObserver(Observer<T> observer);
    void notifyObservers(T event);
}
```

Create an `Observer` interface:

```
public interface Observer<T> {
    void update(T event);
}
```

1.2 Create SensorEvent Class

This class carries sensor data between the hub and panels.

Required fields:

- `sensorType` (String): "temperature" or "smoke"
- `roomName` (String): which room the data comes from
- `value` (double): the sensor reading

Required methods:

- Constructor with all three parameters
- Getter methods for all fields
- `toString()` method for debugging

1.3 Implement MonitoringHub (Concrete Subject)

Requirements:

- Implements `Subject<SensorEvent>` interface
- Maintains a list of registered observers (use `ArrayList<Observer<SensorEvent>>`)
- Provides a `reportData(String sensorType, String roomName, double value)` method

Key behavior:

- When `reportData()` is called, create a `SensorEvent` object
- Call `notifyObservers()` to inform all registered panels

Design constraints:

- Should NOT import or reference concrete panel classes (RealtimePanel, AlarmPanel)
- Should only work with the `Observer<SensorEvent>` interface

1.4 Implement Display Panels (Concrete Observers)

RealtimePanel

- Implements `Observer<SensorEvent>`
- Constructor: `RealtimePanel(String panelName)`
- In `update()`: Display the sensor reading

Example output: `[Dorm Display] temperature in Dormitory: 23.5`

AlarmPanel

- Implements `Observer<SensorEvent>`
- Constructor: `AlarmPanel(String panelName, double smokeThreshold)`
- In `update()`: Check if event is smoke type and value exceeds threshold
- If yes, print an alarm message

Example output: 🚨 `ALARM [Dorm Alarm]! High smoke detected: 75.0`

Validation Criteria

- MonitoringHub only uses the `Observer` interface, not concrete panel classes
- Panels can be registered and removed dynamically
- All registered panels receive notifications when data is reported
- Adding a new panel type does NOT require modifying MonitoringHub

---

Task 2: Integrate Factory Pattern (60 points)

Objective

Use factories to encapsulate the creation AND registration of monitoring panels, demonstrating how Factory and Observer patterns interact.

Why Use Factories Here?

Different room types need different monitoring configurations:

- **Dormitory**: Relaxed smoke threshold (50.0) - avoid false alarms
- **Laboratory**: Strict smoke threshold (20.0) - safety-critical environment

Without factory, your main program would need to:

1. Know all concrete panel classes
2. Manually create each panel
3. Manually configure thresholds for each room type
4. Manually register panels with the hub

The factory encapsulates all this complexity!

2.1 Define Configuration Factory Interface

```
public interface RoomConfigFactory {
    /**
     * Create and register display panels for this room type
     * IMPORTANT: Panels are automatically registered with hub
     * @param hub The monitoring hub to observe
     * @param roomName The name of the room
     */
    void setupPanels(MonitoringHub hub, String roomName);
}
```

Key Point: The factory method `setupPanels` handles BOTH creation and registration. This is where the two patterns interact!

2.2 Implement Concrete Factories

Implement two concrete factories with **clear differences** in configuration:

DormRoomFactory

Requirements:

- Implements `RoomConfigFactory`
- In `setupPanels()`:
    1. Create a `RealtimePanel` with name like "Dorm Display"
    2. Create an `AlarmPanel` with higher smoke threshold (e.g., 50.0)
    3. Register BOTH panels with the hub

Example implementation structure:

```
public class DormRoomFactory implements RoomConfigFactory {
    public void setupPanels(MonitoringHub hub, String roomName) {
        // 1. Create RealtimePanel
        RealtimePanel realtimePanel = new RealtimePanel("Dorm Display");
        hub.registerObserver(realtimePanel);  // ← Factory registers observer!

        // 2. Create AlarmPanel with relaxed threshold
        AlarmPanel alarmPanel = new AlarmPanel("Dorm Alarm", 50.0);
        hub.registerObserver(alarmPanel);  // ← This is the interaction point!
    }
}
```

LabRoomFactory

Requirements:

- Implements `RoomConfigFactory`
- In `setupPanels()`:

1. Create a `RealtimePanel` with name like "Lab Display"
2. Create an `AlarmPanel` with lower threshold (e.g., 20.0) - more sensitive!
3. Register BOTH panels with the hub

Critical Design Point:
The factory encapsulates both creation and registration. This is where Factory and Observer patterns interact!

2.3 Configuration-Based Factory Selection

Option A: Simple Approach (Recommended for simplification)

Create a `FactoryConfig` class with a simple method:

```
public class FactoryConfig {
    public static RoomConfigFactory getFactory(String roomType) {
        if (roomType.equalsIgnoreCase("dormitory")) {
            return new DormRoomFactory();
        } else if (roomType.equalsIgnoreCase("laboratory")) {
            return new LabRoomFactory();
        } else {
            return new DormRoomFactory(); // Default
        }
    }
}
```

Option B: Reflection Approach (OPTIONAL)

Create a `config.properties` file:

```
factory.classname=DormRoomFactory
```

Implement a `FactoryLoader` class that uses reflection to load the factory class from the properties file. Refer to the Abstract Factory tutorial material for reflection examples.

2.4 Main Application

Your main program should:

1. Create a `MonitoringHub` instance
2. Choose a factory (either by type string or from configuration)
3. Use factory to setup panels
4. Simulate sensor data by calling `hub.reportData()`
5. Demonstrate different scenarios

Example Main Structure:

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

```
        // 1. Create hub
        MonitoringHub hub = new MonitoringHub();

        // 2. Setup for dormitory (you can change this to "laboratory")
        String roomType = "dormitory";
        String roomName = "Dormitory A";

        RoomConfigFactory factory = FactoryConfig.getFactory(roomType);
        factory.setupPanels(hub, roomName);

        // 3. Simulate normal conditions
        System.out.println("--- Normal Conditions ---");
        hub.reportData("temperature", roomName, 23.5);
        hub.reportData("smoke", roomName, 15.0);

        // 4. Simulate smoke alert
        System.out.println("\n--- Smoke Alert ---");
        hub.reportData("smoke", roomName, 75.0);
    }
}
```

Validation Criteria

- Two distinct factory implementations with different configurations
- Panels created by factory are automatically registered with hub
- Can switch room type by changing one line of code (or config file)
- Demonstrate different behavior (alarm thresholds) between room types
- Main program doesn't directly instantiate concrete panel classes

---

Example Output

With Dormitory Configuration:

```
--- Normal Conditions ---
[Dorm Display] temperature in Dormitory A: 23.5
[Dorm Display] smoke in Dormitory A: 15.0

--- Smoke Alert ---
[Dorm Display] smoke in Dormitory A: 75.0
🚒 ALARM [Dorm Alarm]! High smoke detected: 75.0
```

With Laboratory Configuration:

```
--- Normal Conditions ---
[Lab Display] temperature in Laboratory: 23.5
[Lab Display] smoke in Laboratory: 15.0

--- Smoke Alert ---
[Lab Display] smoke in Laboratory: 25.0
🚨 ALARM [Lab Alarm]! High smoke detected: 25.0
```

Note: Lab alarm triggers at 20.0, so smoke value 25.0 triggers it, while 15.0 doesn't.

Submission Checklist

Required Files (Without package declarations)

Core Classes (9 files):

1. `Subject.java` - Subject interface
2. `Observer.java` - Observer interface
3. `SensorEvent.java` - Event data class
4. `MonitoringHub.java` - Concrete subject
5. `RealtimePanel.java` - Concrete observer
6. `AlarmPanel.java` - Concrete observer
7. `RoomConfigFactory.java` - Factory interface
8. `DormRoomFactory.java` - Concrete factory
9. `LabRoomFactory.java` - Concrete factory

Configuration Class (choose one):

- `FactoryConfig.java` - Simple factory selector (Option A)
- OR `FactoryLoader.java` + `config.properties` - Reflection-based loader (Option B)

Application:

- `SmartHomeApp.java` - Main application

README.txt

Include:

- How to compile and run your program
- How to switch between dormitory and laboratory configurations
- Brief explanation (3-4 sentences) of how Observer and Factory patterns interact

Class Diagram (optional)

Provide a UML class diagram showing:

- Subject and Observer interfaces with implementing classes
- Factory interface with concrete factory classes
- Key methods and relationships