Eduardo Pons

WeatherStation Observer Pattern Implementation

Overview

This document outlines the implementation and advantages of the Observer pattern applied to the `WeatherStation` project. The Observer pattern is a behavioral design pattern that defines a one-to-many dependency between objects, allowing one object (the subject) to notify multiple dependent objects (observers) of changes to its state.

Problem Statement

In meteorological systems, multiple displays and systems need to be updated with the latest weather data simultaneously. Managing these updates can become complex, especially when the number of displays and systems increases. Without a proper design pattern, updates might lead to tight coupling between the weather data source and the displays, making the system less flexible and harder to maintain.

Solution

The Observer pattern provides an effective solution by decoupling the weather data source from the displays. This pattern allows the `WeatherStation` (the subject) to notify all registered displays (the observers) about changes in weather data. This approach improves flexibility, scalability, and maintainability of the system.

Code Implementation

Observer Interface

The `Observer` interface defines the method for receiving updates from the subject. It ensures that all observers implement the `update` method to handle changes in the weather data.

java
Copiar código
```
public interface Observer {
    void update(float temperature, float humidity, float pressure);
}
```

WeatherStation Class

The `WeatherStation` class represents the subject that maintains a list of observers. It notifies observers about changes in weather data.

java
Copiar código

```
import java.util.ArrayList;
import java.util.List;

public class WeatherStation {
    private List<Observer> observers = new ArrayList<>();
    private float temperature;
    private float humidity;
    private float pressure;

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

    public void removeObserver(Observer observer){
        observers.remove(observer);
    }

      public void setWeatherData(float temperature, float humidity,
float pressure){
        this.temperature = temperature;
        this.humidity = humidity;
        this.pressure = pressure;
        notifyObservers();
    }

    private void notifyObservers(){
        for(Observer observer : observers){
            observer.update(temperature, humidity, pressure);
        }
    }
}
```

CurrentConditionDisplay Class

The `CurrentConditionDisplay` class is an observer that displays the current weather conditions.

java
Copiar código

```
public class CurrentConditionDisplay implements Observer {

    private float temperature;
    private float humidity;

    @Override
      public void update(float temperature, float humidity, float
pressure){
        this.temperature = temperature;
```

```
        this.humidity = humidity;
        display();
    }

    public void display(){
          System.out.println("Current conditions: " + temperature +
"°C and " + humidity + "% humidity");
    }
}
```

ForecastDisplay Class

The `ForecastDisplay` class is an observer that provides weather forecasts based on pressure changes.

java
Copiar código

```
public class ForecastDisplay implements Observer {
    private float currentPressure = 29.92f;
    private float lastPressure;

    @Override
      public void update(float temperature, float humidity, float
pressure){
        lastPressure = currentPressure;
        currentPressure = pressure;
        display();
    }

    public void display(){
        System.out.println("Forecast: ");
        if (currentPressure > lastPressure) {
            System.out.println("Weather is getting warmer");
        }
        else if(currentPressure < lastPressure){
            System.out.println("Weather is getting cooler");
        }
        else if(currentPressure == lastPressure){
            System.out.println("Weather remains the same");
        }
    }
}
```

Main Class

The `Main` class demonstrates the use of the Observer pattern with a `WeatherStation`, `CurrentConditionDisplay`, and `ForecastDisplay`.

java
Copiar código

```
public class Main {

    public static void main(String[] args) {
        WeatherStation weatherStation = new WeatherStation();

                    CurrentConditionDisplay currentDisplay = new
CurrentConditionDisplay();
        ForecastDisplay forecastDisplay = new ForecastDisplay();

        weatherStation.addObserver(currentDisplay);
        weatherStation.addObserver(forecastDisplay);

        weatherStation.setWeatherData(25.5f, 65f, 30.4f);
        weatherStation.setWeatherData(27.8f, 70f, 29.2f);
        weatherStation.setWeatherData(22.3f, 90f, 28.5f);
    }
}
```

Benefits of Using the Observer Pattern

1. **Decoupling of Components**: The Observer pattern separates the `WeatherStation` from its observers, reducing the dependencies and making it easier to manage and modify the system.
2. **Scalability**: New types of displays or systems can be added without modifying the `WeatherStation` class, enabling easy extension and scaling of the system.
3. **Dynamic Updates**: Observers are updated dynamically whenever there is a change in the weather data, ensuring that all displays and systems reflect the latest information.
4. **Improved Maintainability**: The pattern improves maintainability by allowing the addition or removal of observers independently, without affecting the core weather data logic.
5. **Flexibility**: Different types of observers can be implemented and attached to the `WeatherStation`, offering a flexible solution to handle various display and reporting requirements.

Conclusion

The Observer pattern is an effective design choice for managing the distribution of weather data in the `WeatherStation` project. It enhances flexibility, scalability, and maintainability by decoupling the data source from the various displays and systems that depend on it. This approach provides a robust and adaptable solution to the challenges of managing dynamic updates in a weather monitoring system.