Implementierung von Raumdaten-Abonnements mit dem Observer-Pattern

Verfasser:
Ian Risch & Julius Lange

Kurs/Projekt:
BSIT22a

Institution:
CBS Koblenz

Inhaltsverzeichnis

Observer Pattern erklärt

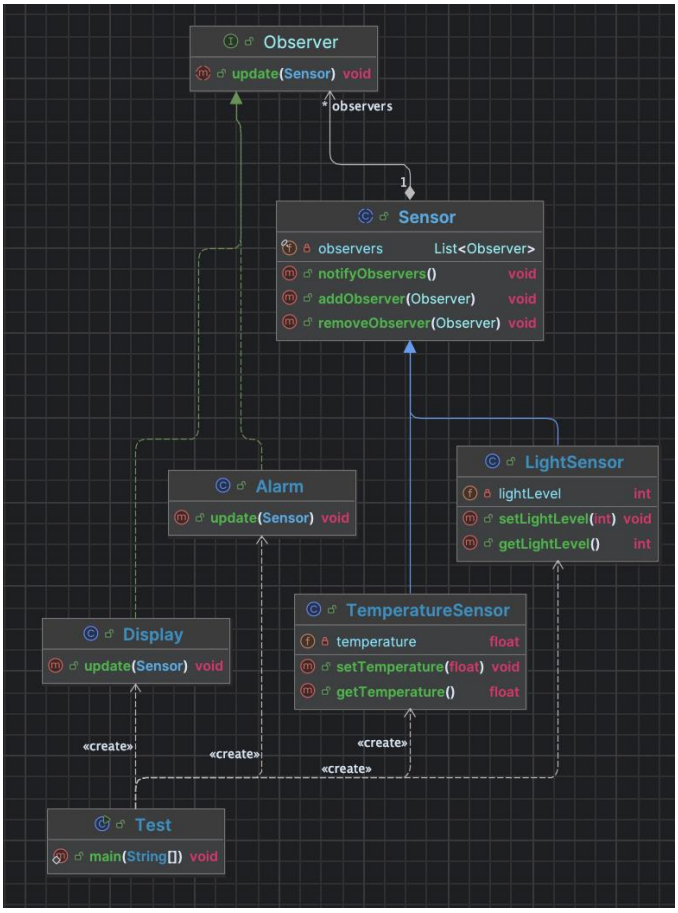
Das Observer-Pattern ist ein Entwurfsmuster aus der Softwareentwicklung, das verwendet wird, um eine 1:n-Beziehungzwischen Objekten zu realisieren. Dabei registrieren sich sogenannte Observer (Beobachter) bei einem Subject (Subjekt), um über Änderungen des Zustands benachrichtigt zu werden. Wenn sich der Zustand des Subjekts ändert, werden alle registrierten Beobachter automatisch aktualisiert.

Warum haben wir uns für das Observer Pattern entschieden?

Das Observer-Pattern und das MQTT-Protokoll beruhen auf ähnlichen Prinzipien, da beide ein Publish-Subscribe-Konzept implementieren. Im Observer-Pattern registrieren sich Beobachter (Observers) bei einem Subjekt (Subject), um über Änderungen benachrichtigt zu werden. Ebenso ermöglicht MQTT, dass Abonnenten (Subscribers) sich auf Themen (Topics) registrieren, um automatisch Benachrichtigungen zu erhalten, wenn neue Daten veröffentlicht (published) werden.

Da MQTT bereits das Publish-Subscribe-Modell nutzt, bietet es sich an, das Observer-Pattern zusätzlich im Code anzuwenden. Dies ermöglicht eine saubere Trennung zwischen der Logik zur Verarbeitung der Raumdaten und der Datenquelle. So können neue Beobachter flexibel hinzugefügt werden, ohne bestehende Komponenten zu ändern, was die Erweiterbarkeit und Wartbarkeit des Systems fördert.

Uml Diagramm der Anwendung



Bezug zum Uml Diagramm

Das UML-Diagramm illustriert die Struktur der Implementierung. Im Mittelpunkt steht die Klasse Sensor, die als Subjekt fungiert. Sie verwaltet eine Liste von Beobachtern (observers) und implementiert Methoden wie addObserver(), removeObserver() und notifyObservers(). Dadurch können Beobachter wie Alarm und Display registriert werden, um bei Zustandsänderungen des Sensors (z. B. Temperatur- oder Lichtänderungen) automatisch aktualisiert zu werden.

Die Klassen LightSensor und TemperatureSensor erweitern die Basisklasse Sensor und repräsentieren spezifische Sensortypen, die ihre Daten durch entsprechende Getter- und Setter-Methoden (setTemperature(), getLightLevel()etc.) bereitstellen. Die Beobachter wie Display oder Alarm implementieren die Observer-Schnittstelle, was sicherstellt, dass jede Änderung des Sensors durch die Methode update(Sensor sensor) verarbeitet wird. Diese Struktur ist flexibel und skalierbar, da weitere Sensortypen oder Beobachter leicht hinzugefügt werden können.

Quellcode

```
    public static void main(String[] args) {
        // Sensoren erstellen
        TemperatureSensor tempSensor = new TemperatureSensor();
        LightSensor lightSensor = new LightSensor();

        // Beobachter erstellen
        Display display = new Display();
        Alarm alarm = new Alarm();

        // Beobachter an die Sensoren binden
        tempSensor.addObserver(display);
        tempSensor.addObserver(alarm);

        lightSensor.addObserver(display);
        lightSensor.addObserver(alarm);

        // Sensoren auslösen
        System.out.println("Ändere Temperatur auf 25°C:");
        tempSensor.setTemperature(25);

        System.out.println("Ändere Temperatur auf 35°C:");
        tempSensor.setTemperature(35);

        System.out.println("Ändere Lichtlevel auf 80 Lumen:");
        lightSensor.setLightLevel(80);

        System.out.println("Ändere Lichtlevel auf 30 Lumen:");
        lightSensor.setLightLevel(30);
    }
}
```

```
public abstract class Sensor {  8 usages  2 inheritors
    private final List<Observer> observers = new ArrayList<>();  3 usages

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

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

    public void notifyObservers() {  2 usages
        for (Observer observer : observers) {
            observer.update( sensor: this);
        }
    }
}
```


public interface Observer {
void update(Sensor sensor);
}