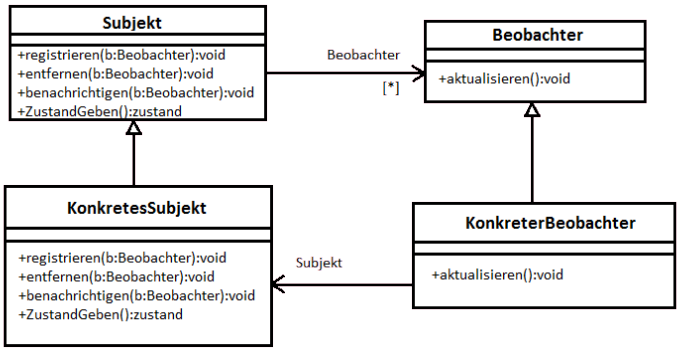
Merkblatt Design Pattern – Observer

Zweck

Durch das Observer Pattern können Daten ohne grossen Aufwand in mehreren unabhängigen Komponenten einer App verwendet und bei Änderungen automatisch aktualisiert werden.

Dadurch bleiben die Daten in jeder Komponente immer auf dem neusten Stand und das System ist bei der Verwendung des Observer Pattern sehr flexibel.

UML



Beschreibung

Behobene Probleme

- Eine One-to-many Abhängigkeit zwischen Objekten wird benötigt, ohne dass die einzelnen Objekte eng verbunden sind.
- Beim Ändern eines Objektes sollten zusätzlich eine unbestimmte Anzahl von abhängigen Objekten automatisch aktualisiert wird.

Lösung

- Subjekt und Observer Objekte werden definiert
- Wenn ein Subjekt geändert wird, werden alle registrierten Beobachter/Observer automatisch benachrichtigt und aktualisiert

Die einzige Verantwortung eines Subjekts besteht darin, eine Liste von Beobachtern zu führen und sie über Änderungen zu informieren, indem sie ihre update()-Methode aufrufen. Die Beobachter müssen sich auf ein Subjekt registrieren und ihren Inhalt aktualisieren, wenn sie vom Subjekt benachrichtigt werden.

Vorteile und Anwendungsfälle

Unabhängig von der Darstellung können die Daten übergeben werden. Die Subjekt Klasse kennt die Implementation der Observer (Beobachter) nicht.

- GUI (User verändert Daten, neue Daten müssen in allen GUI-Komponenten aktualisiert werden)
- Ein Datensatz (Key-Value-Paare) mit mehreren Visualisierungen (Tabelle, Balkendiagramm etc.)
- Allgemein im MVC-Pattern (Model-View-Controller) bei der View-Model-Kommunikation.

Beispiel in Java

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

/**
 * Abstraktes Subject – implementiert das meiste, was das Subjekt können muss
 */
abstract class Subject {
    // Beobachter werden in dieser Liste gespeichert
    private List<Observer> observers = new ArrayList<>();


    /**
     * Methode um die Observer/Beobachter zu informieren, dass es neue Daten
gibt
     *
     * @param event
     */
    void notifyObservers(String event) {
        observers.forEach(observer -> observer.update(event));
    }

    /**
     * Einen neuen Beobachter hinzufügen
     *
     * @param observer
     */
    void addObserver(Observer observer) {
        observers.add(observer);
    }

    /**
     * Einen Beobachter entfernen
     *
     * @param observer
     */
    void removeObserver(Observer observer) {
        observers.remove(observer);
    }

    void addValue(String input) {
        this.notifyObservers(input);
    }
}
```

```java
/**
 * Konkretes Subject welches das Subject erweitert
 */
class TodoItemSubject extends Subject {

    void addValue(String todoitem) {
        if (!todoitem.trim().equals("")) {
            notifyObservers(todoitem.trim());
        }
    }
}
```

```java
/**
 * Observer Interface – muss update Methode implementieren
 */
interface Observer {
    void update(String input);
}
```

```java
/**
 * Klasse, welche as Subject beobachtet
 */
public class TodoItemObserver implements Observer {
    TodoItemSubject todoItemSubject = new TodoItemSubject();

    TodoItemObserver() {
        this.todoItemSubject.addObserver(this);
    }

// Methode des Interfaces wird implementiert
    @Override
    public void update(String item) {
        System.out.println(item);
    }
}
```