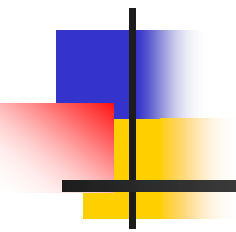
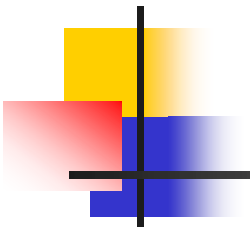


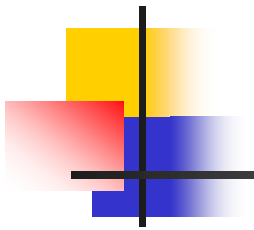
Observer Pattern

Von Claas Reim



Inhalt

1. Motivation
2. Anwendbarkeit
3. Struktur eines Beobachter-Pattern
4. Teilnehmer
5. Konsequenzen
6. Beispielcode
7. Implementierung
8. Verwandte Muster
9. Verwendungsbeispiele

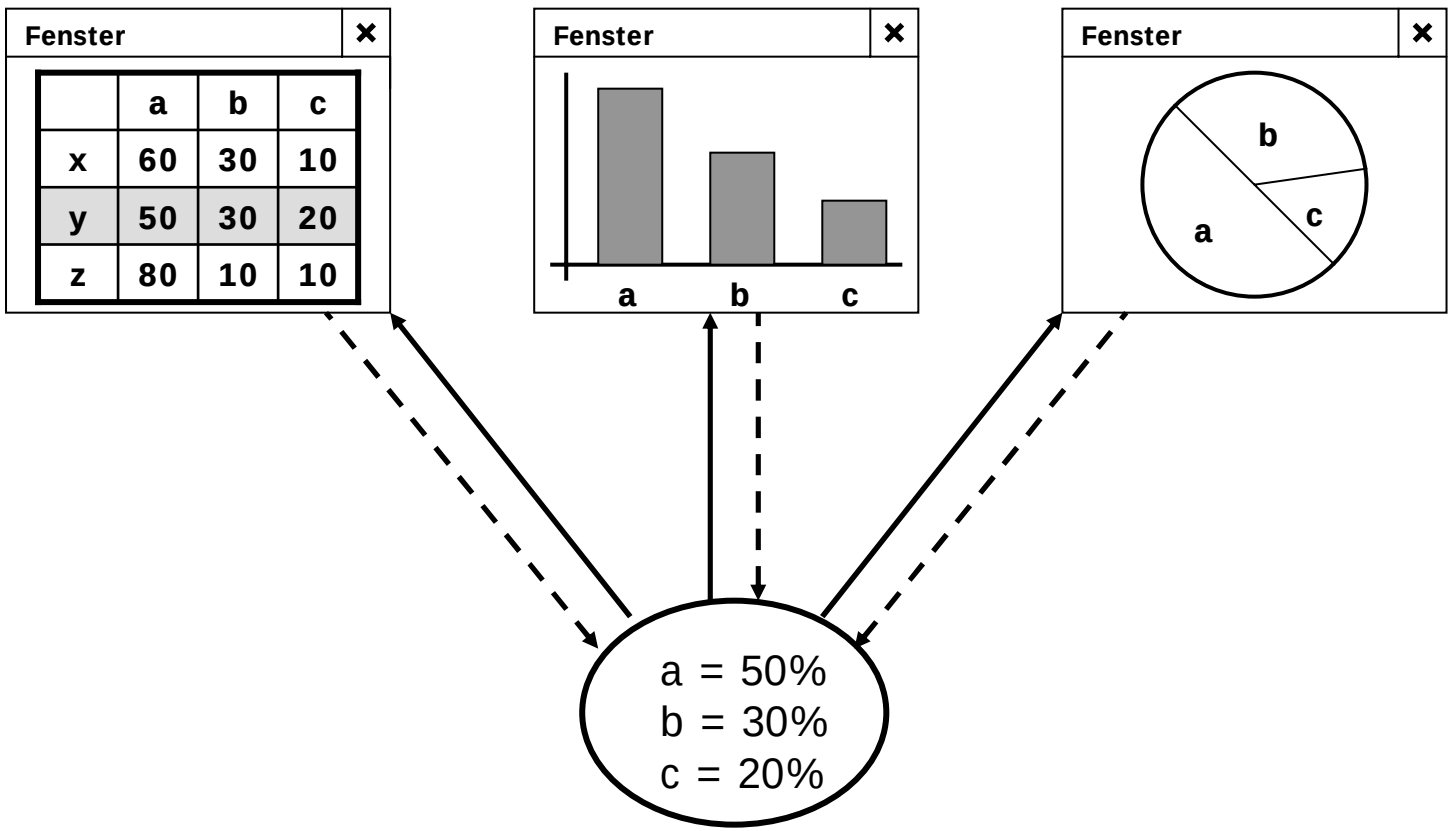


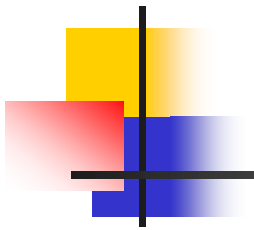
Motivation

Problem und Kontext:

- Konsistenz zwischen den Objekten einer Menge von interagierenden Klassen eines Systems soll aufrechterhalten werden
- bei enger Kopplung der Objekte schränkt man deren Wiederverwendbarkeit ein

Motivation

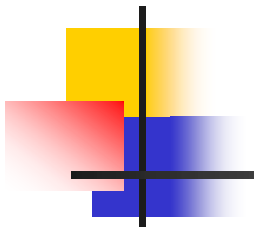

Fenster
a b c
x 60 30 10
y 50 30 20
z 80 10 10
Fenster
a b c
Fenster
b
a
c
a = 50%
b = 30%
c = 20%



Motivation

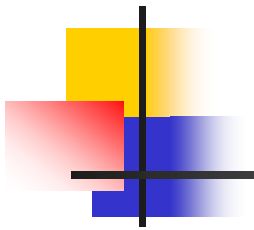
Ziel:

- 1:N Beziehung zwischen Klassen
- Möglichkeit, die abhängigen Objekte über die Zustandsänderung zu informieren
- Konsistenzsicherung zwischen kooperierenden Objekten (ohne zu nahe Bindung)
- Notifikation ohne die Beobachter zu kennen



Anwendbarkeit

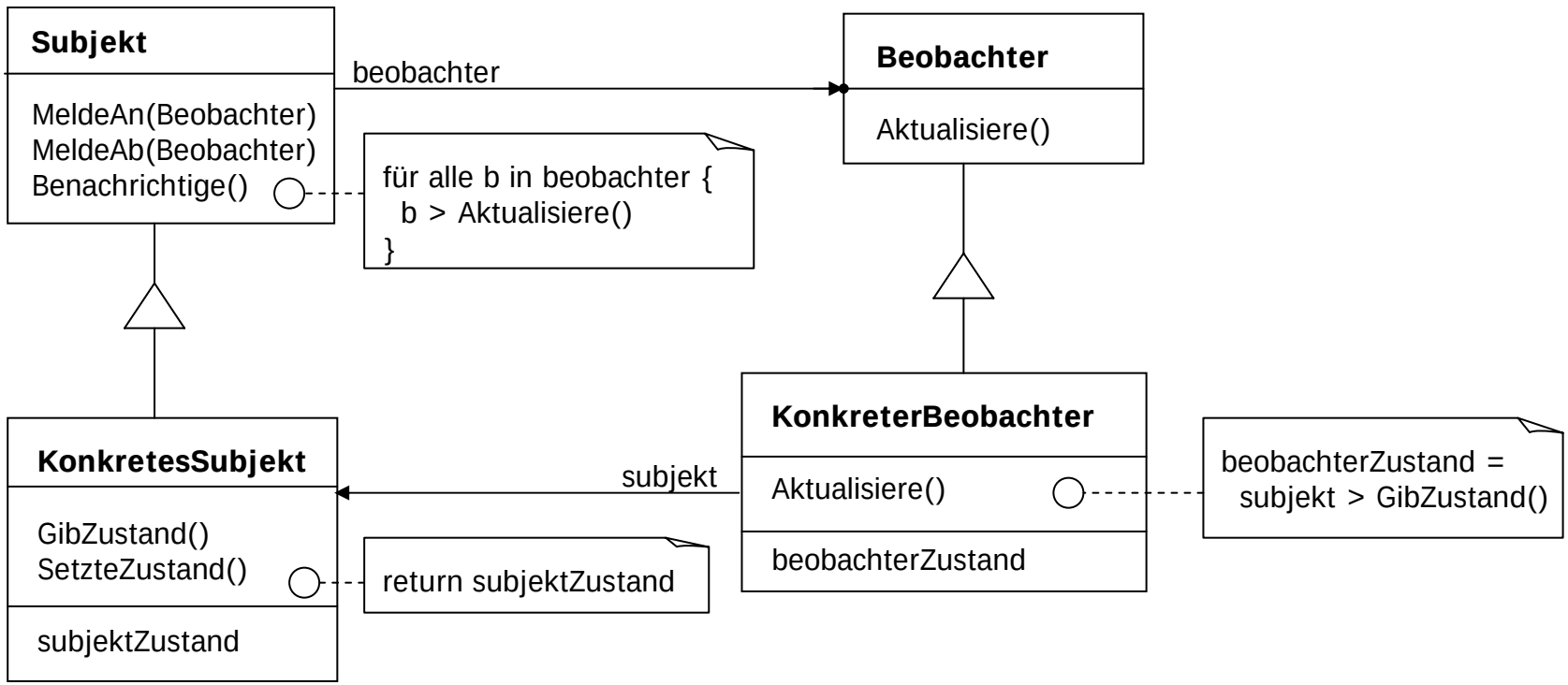
- Abstraktion besitzt zwei Aspekte, von denen der eine vom anderen abhängt
- Änderung eines Objektes verlangt die Änderung anderer Objekte, jedoch ist deren Anzahl nicht bekannt
- ein Objekt soll andere Objekte benachrichtigen, ohne zu wissen, wer diese sind

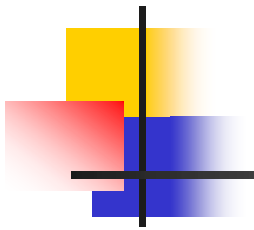


Struktur

- zentrale Objekte sind das Subjekt und der Beobachter
- Subjekt kann beliebige Anzahl von abhängigen Beobachtern haben
- Beobachter kennen sich untereinander nicht
- alle Beobachter werden benachrichtigt, falls das Subjekt seinen Zustand ändert
- darauf synchronisieren sich die Beobachter mit dem aktuellen Zustand durch Anfragen an das Subjekt

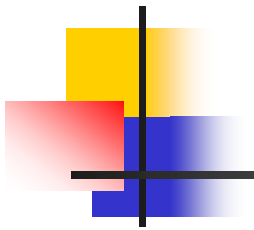
Struktur


Subjekt
MeldeAn(Beobachter)
MeldeAb(Beobachter)
Benachrichtige()
beobachter
Beobachter
Aktualisiere()
für alle b in beobachter {
b > Aktualisiere()
}
KonkreterBeobachter
Aktualisiere()
beobachterZustand
beobachterZustand =
subjekt > GibZustand()
subjekt
KonkretesSubjekt
GibZustand()
SetzteZustand()
subjektZustand
return subjektZustand



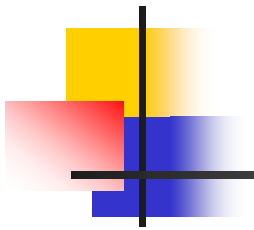
Teilnehmer

- Subjekt
  - kennt Beobachter
  - bietet eine Schnittstelle zum An- und Abmelden von Beobachtern
  - mehrere Beobachter können ein Subjekt beobachten
- Beobachter
  - definiert eine Aktualisierungsschnittstelle für Objekte
  - Schnittstelle soll über Änderungen eines Subjekts benachrichtigt werden



Teilnehmer

- KonkretesSubjekt
  - speichert den für KonkreterBeobachter relevanten Zustand
  - benachrichtigt seine Beobachter bei Zustandsänderung
- KonkreterBeobachter
  - verwaltet Referenz auf ein KonkretesSubjekt
  - speichert den Zustand, der mit dem des Subjekts in Einklang stehen soll
  - implementiert die Aktualisierungsschnittstelle der Beobachterklasse, um die Konsistenz mit dem Zustand des Subjektes zu halten



Konsequenzen

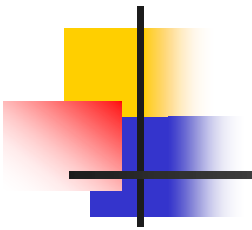
- Beobachter-Muster ermöglichen es, Subjekte und Beobachter unabhängig voneinander zu modifizieren
- Beobachter und Subjekte können einzeln wiederverwendet werden
- neue Beobachter können ohne Änderung des Subjektes hinzugefügt werden

Beispielcode

```
import java.util.*;

interface Observer {                                    // Beobachter
   void update();
}

class Observable {                                      // Subjekt
   private Vector observers = new Vector();
   public void addObserver(Observer o) {
      observers.addElement(o);
   }
   public void removeObserver(Observer o) {
      observers.removeElement(o);
   }
   public void notifyObservers() {
      for(int i=0; i<observers.size(); i++) {
         Observer o = (Observer)observers.elementAt(i);
         o.update();
      }
   }
}
```

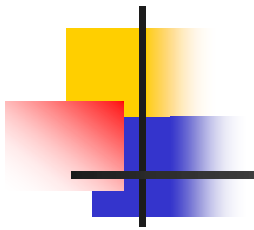
Beispielcode

```
class Value extends Observable {                    //KonkreterSubjekt
   private int m;
   public int getVal() {
      return m;
   }
   public void setVal(int val) {
      m = val;
      notifyObservers();
   }
}

class ValueObserver implements Observer {           //KonkretesBeobachter
   Value val;
   ValueObserver(Value v) {          // Konstruktor
      val = v;
      v.addObserver(this);
   }
   public void update() {
      System.out.println("update called");
   }
}
```



Implementierung

- Auslösen der Aktualisierung
  - es ist notwendig zu wissen welches Objekt ruft die Benachrichtige-Operation auf
  - zwei Möglichkeiten:
  - A. Die zustandsändernde Operation der Subjekts ruft Benachrichtige() nach Ausführen der Änderung auf.
  - B. Die Klienten bestimmen den Zeitpunkt für den Aufruf von Benachrichtige().

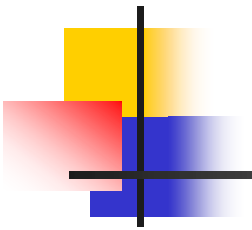


Implementierung

- Kapselung komplexer Aktualisierungssemantik
  - neues Objekt **Änderungsmanager**, dass die Beziehung zwischen Subjekten und Beobachtern verwaltet
  - Hauptaufgaben des Änderungsmanager sind:

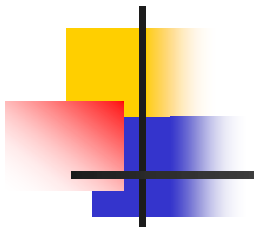
1. Er bildet ein Subjekt auf seine Beobachter ab und bietet eine Schnittstelle zu Verwaltung dieser Abbildung.
2. Die Aktualisierungsstrategie wird durch Ihn definiert.
3. Alle abhängigen Beobachter werden auf Anforderung eines Subjektes aktualisiert.

  - im allgemeinen gibt es nur einen Änderungsmanager und er ist global bekannt



Verwandte Muster

- Vermittler
  - mittels Kapselung komplexer Aktualisierungssemantik wirkt der ÄnderungsManager als Vermittler zwischen Subjekten und Beobachter
- Singleton
  - um einmalig und global zugreifbar zu sein kann der ÄnderungsManager das Singletonmuster verwenden



Verwendung

Beispiele:

- ST: MVC
- AWT Listeners
- java.util.Observer
- JavaBeans: event notification

Vielen Dank für Ihre Aufmerksamkeit.

??? FRAGEN ???