Analyse und Modellierung von Informationssystemen

Dr. Klaus Höppner

Hochschule Darmstadt – Wintersemester 2014/2015

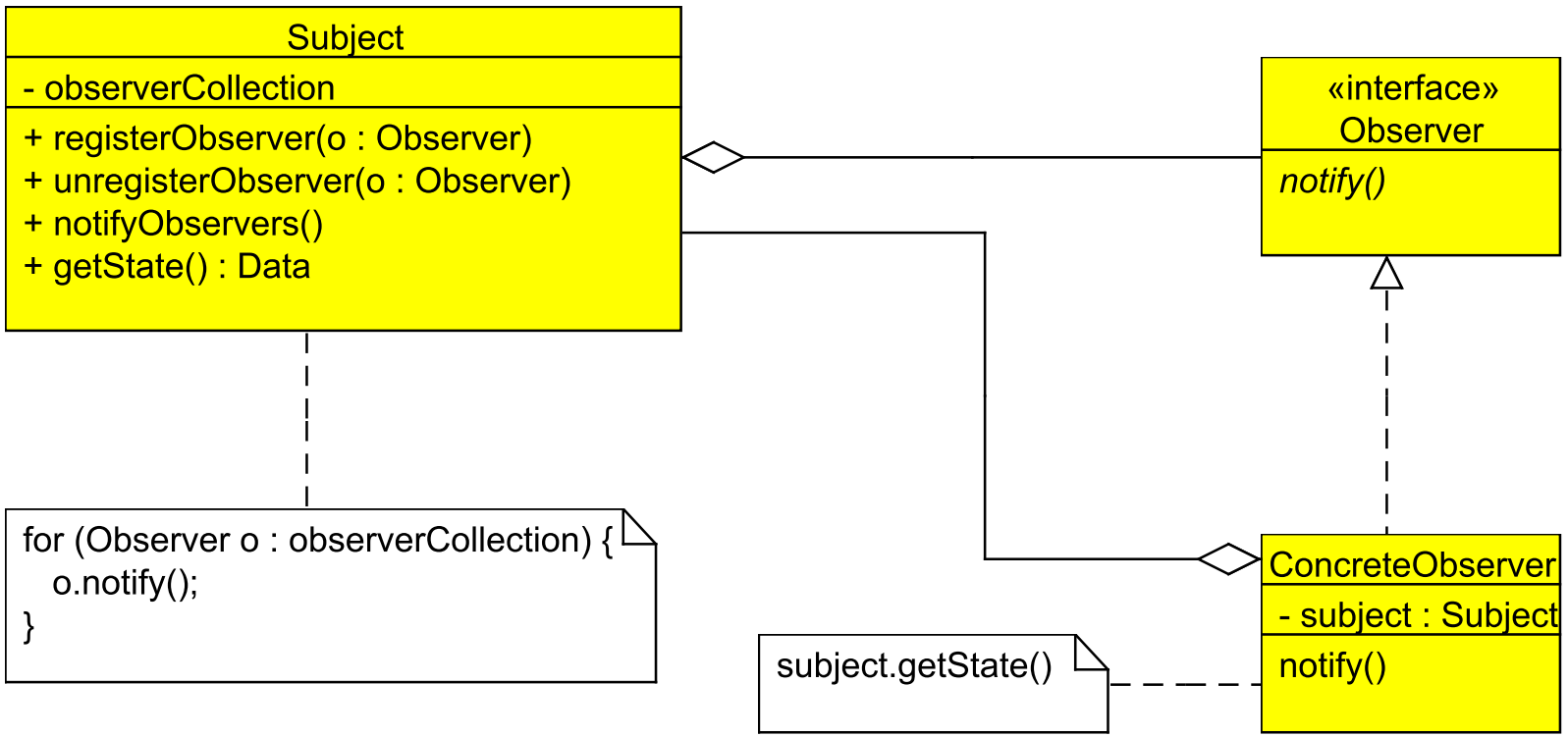
Beobachter-Muster

Besucher-Muster

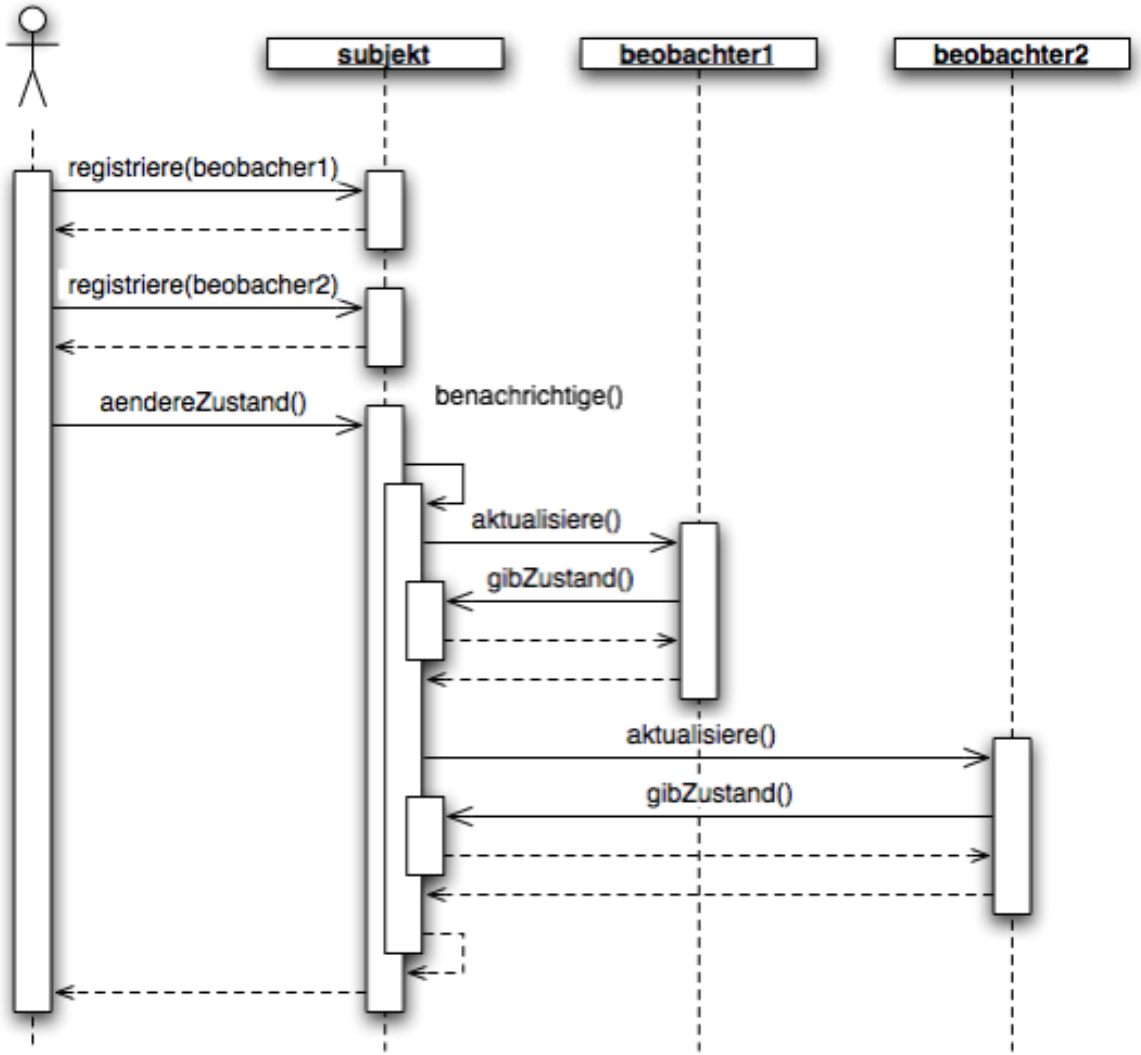
Das Beobachter-Muster

Das Beobachter-Muster (*observer*, manchmal auch *publish-subscribe*) ist ein Verhaltensmuster, das einen so genannten Push-Algorithmus realisiert. Von einem interessierenden Objekt wird ein Zustand nicht regelmäßig abgefragt (Poll), sondern den Beobachtern wird Bescheid gesagt, wenn eine relevante Änderung auftritt.

Klassendiagramm



Sequenzdiagramm



Beispiel: Wetterstation

```
public interface Subjekt {
    public void registriereBeobachter(Beobachter o);
    public void entferneBeobachter(Beobachter o);
    public void benachrichtigeBeobachter();
}

public interface Beobachter {
    public void aktualisieren(float temp, float feucht, float druck);
}

public interface AnzeigeElement {
    public void anzeigen();
}
```

Beispiel: Wetterstation (Forts.)

```
public class WetterDaten implements Subjekt {
    private ArrayList<Beobachter> beobachter = new ArrayList<Beobachter>()
    private float temperatur;
    private float feuchtigkeit;
    private float luftdruck;

    public void registriereBeobachter(Beobachter b) {
        beobachter.add(b);
    }

    public void entferneBeobachter(Beobachter b) {
        beobachter.remove(b);
    }

    public void benachrichtigeBeobachter() {
        for (Beobachter b : beobachter) {
            b.aktualisieren(temperatur, feuchtigkeit, luftdruck);
        }
    }

    public void messwerteGeaendert() {
        benachrichtigeBeobachter();
    }
```

Beispiel: Wetterstation (Forts.)

```
    public void setMesswerte(float temp, float feucht, float druck) {
        this.temperatur = temp;
        this.feuchtigkeit = feucht;
        this.luftdruck = druck;
        messwerteGeaendert();
    }

    // andere WetterDaten-Methoden

    public float getTemperatur() {
        return temperatur;
    }

    public float getFeuchtigkeit() {
        return feuchtigkeit;
    }

    public float getLuftdruck() {
        return luftdruck;
    }
}
```

Beispiel: Wetterstation (Forts.)

```
public class AktuelleBedingungenAnzeige
                    implements Beobachter, AnzeigeElement {
    private float temperatur;
    private float feuchtigkeit;
    private Subjekt wetterDaten;

    public AktuelleBedingungenAnzeige(Subjekt wetterDaten) {
        this.wetterDaten = wetterDaten;
        wetterDaten.registriereBeobachter(this);
    }

    public void aktualisieren(float temp, float feucht, float druck) {
        this.temperatur = temp;
        this.feuchtigkeit = feucht;
        anzeigen();
    }

    public void anzeigen() {
        System.out.println("Aktuelle Bedingungen: " + temperatur
            + " Grad C und " + feuchtigkeit + "% Luftfeuchtigkeit");
    }
}
```

Beispiel: Wetterstation (Forts.)

```
public class WetterStation {

    public static void main(String[] args) {
        WetterDaten wetterDaten = new WetterDaten();

        new AktuelleBedingungenAnzeige(wetterDaten);
        wetterDaten.setMesswerte(30, 65, 30.4f);
        wetterDaten.setMesswerte(32, 70, 29.2f);
        wetterDaten.setMesswerte(28, 90, 29.2f);
    }
}
```

Besonderheiten

- Die Beobachter werden in einer Schleife benachrichtigt. Dies findet synchron statt, d. h. es wird bei jedem registrierten Beobachter gewartet, bis der Aufruf von `notify` zurück kehrt. Dies kann bei vielen Beobachtern Performance-Probleme geben.
- Es gibt i. A. keine Garantie, in welcher Reihenfolge die Beobachter benachrichtigt werden. Dies gilt insbesondere für die von Java mitgelieferten Klasse `Observable`, die mit dem Interface `Observer` zusammen arbeitet.
- In der Praxis wird sowohl verwendet, dass das Subjekt mit der Benachrichtigung schon die geänderten Daten mitschickt, oder der Beobachter selber über vom Subjekt zur Verfügungn gestellte Methoden die Änderungen erfragen muss.

Das Besucher-Muster

Das Besucher-Muster (*visitor*) dient dazu, dass auf einer Menge von Objekten (eines bekannten) Typs Operationen ausgeführt werden können. Hierbei kann es mehrere, voneinander unabhängige Operationen geben, neue Operationen können flexibel hinzugefügt werden.

Beispiel: Eine Bestellung bei einem Versandhaus besteht aus verschiedenen Waren unterschiedlichen Typs (z. B. Bücher und DVDs). Bei der Bearbeitung der Bestellung erfolgen mit der Bestellung mehrere, voneinander unabhängige Operationen: Berechnung der Preisinformationen aus den Einzelpreisen, mit Berücksichtigung unterschiedlicher MWSt-Sätze bei Büchern und DVDs, Auswahl der Versandmethode anhand des Gesamtgewichts und der Warentypen. Es erfolgt somit jeweils eine Iteration über alle bestellte Artikel, wobei die Einzelschritte je nach Artikeltyp unterschiedlich ausfallen.

Ansatz mit Fallunterscheidung

```
double price = 0.0;
double tax7 = 0.0;
double tax19 = 0.0;
for (Item item : items) {
    price += item.getPrice();
    if (item instanceof Book) {
        tax7 += .07*item.getPrice()/1.07;
    } else if (item instanceof DVD) {
        tax19 += .19*item.getPrice()/1.19;
    }
}
```

Je nach Zahl der konkreten Artikelklassen wird der Code durch die Fallunterscheidung sehr unübersichtlich.

Auslagerung der Operationen nach Art des Befehlsmusters

Nach Vorbild des Befehlsmusters kapseln wir die Einzeloperationen für die verschiedenen Artikeltypen:

```
public interface Visitor {
    public void operation(Book book);
    public void operation(DVD dvd);
}
```

wobei Befehlsklassen für eine konkrete Operation die typabhängigen Einzeloperationen implementieren.

Preisberechnung

```
public class PriceVisitor implements Visitor {
    private double price = 0.0;
    private double tax7 = 0.0;
    private double tax19 = 0.0;

    @Override
    public void operation(Book book) {
        price += book.getPrice();
        tax7 += .07*book.getPrice()/1.07;
    }

    @Override
    public void operation(DVD dvd) {
        price += dvd.getPrice();
        tax19 += .19*dvd.getPrice()/1.19;
    }

    public void showTotalPrice() {
        System.out.format(
                "Preis: %.2f\nEnthaltene MWSt. 7%%: %.2f\nMWSt. 19%%: %.2f
                price, tax7, tax19);
    }
}
```

Problem

Leider tritt nun das Problem auf, dass bei der Bearbeitung der Bestellung über eine Menge von `Item` iteriert wird, die Methoden `operation(...)` jedoch Referenzen auf die konkreten Klassen, die `Item` implementieren, als Parameter erwarten, so dass weiterhin eine Fallunterscheidung erforderlich ist.

```
PriceVisitor visitor = new PriceVisitor();
for (Item item : items) {
    if (item instanceof Book) {
        visitor.operation( (Book) item);
    } else if (item instanceof DVD) {
        visitor.operation( (DVD) item);
    }
}
```

Lösung

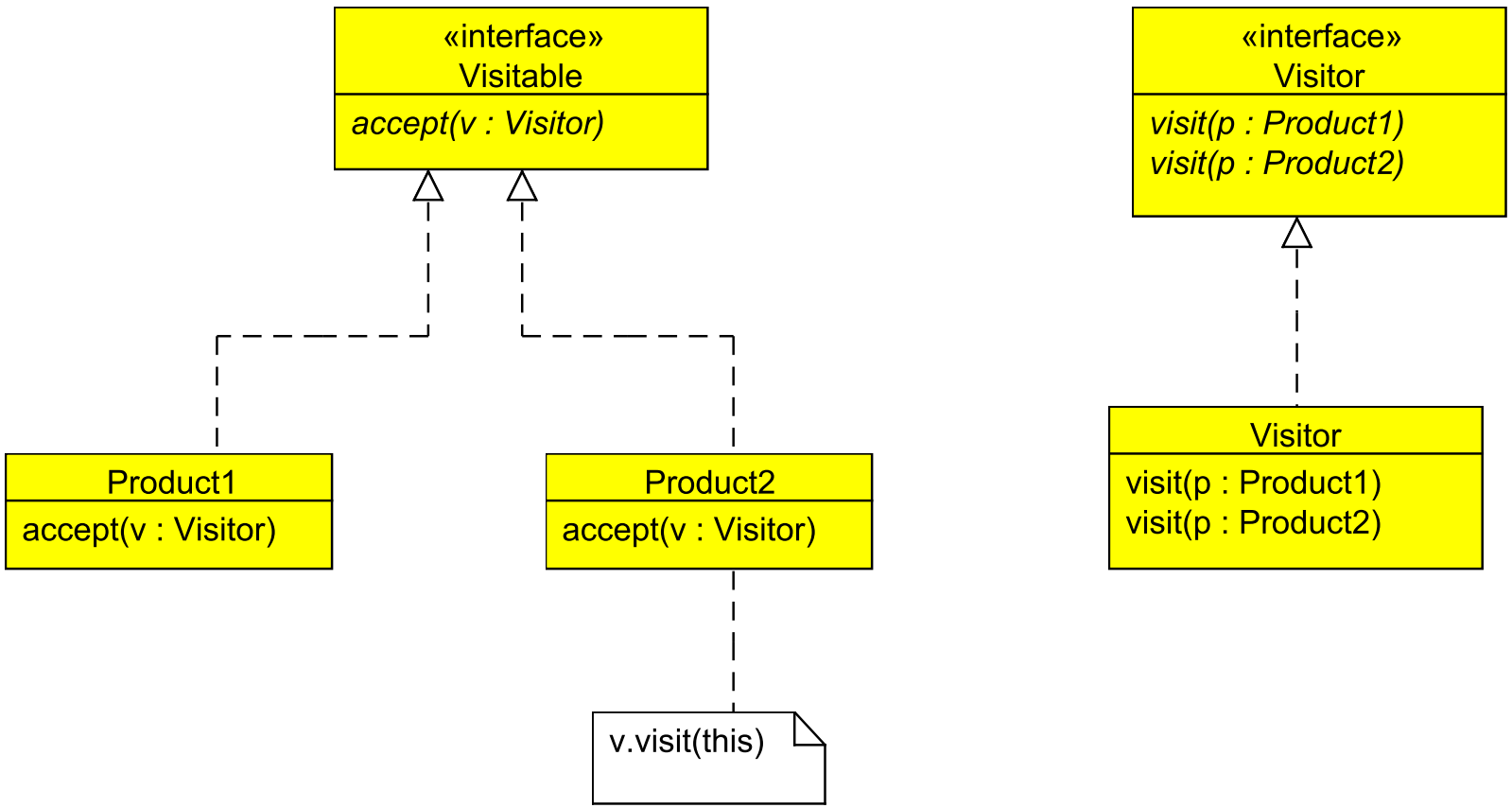
Das Problem wird nun gelöst, indem das Interface `Item` eine Methode zur Verfügung stellt, die einen Besucher akzeptiert, wobei in den konkreten Implementierungen jeweils die Operation des Besuchers mit einer Referenz auf sich selbst aufgerufen wird, für den Besucher nun als Referenz auf die konkrete Klasse erkennbar.

Muster der besuchbaren Objekte

```
public interface Item {
    ...
    void accept(Visitor v);
}

public class Book implements Item {
    void accept(visitor v) {
        v.operation(this);
    }
}
```

Klassendiagramm



Vor- und Nachteile

- Der Besucher und die besuchbaren Klassen sind voneinander entkoppelt.
- Effektiv realisiert das Muster doppelte Polymorphie, denn beim Aufruf von `accept` findet die dynamische Bindung an die Implementation innerhalb der Produktklasse statt, hier wird wiederum `visit` aufgerufen, das dynamisch an die Implementierung in der konkreten Besucher-Klasse bindet: Multimethode (*double dispatch*).
- Neue Besucher lassen sich einfach implementieren, ohne Änderung an den zu besuchenden Klassen.
- Das Hinzufügen neuer zu besuchender Klassen erzeugt hingegen den Aufwand, bei den Besuchern die Operation für diese neue Klasse hinzuzufügen.
- Aus diesem Grund eignet sich das Besucher-Muster für Szenarien, in denen keine oder nur wenig Änderungen an der Struktur der Produktklassen zu erwarten sind.

Alternative in Java: Reflections

```
public class PriceVisitorReflections implements Visitor {
    private double price = 0.0;
    private double tax7 = 0.0;
    private double tax19 = 0.0;

    public void operation(Item item) {
        System.out.println("Foo");
        Class<?> c = item.getClass();
        try {
            Method m = this.getClass().getDeclaredMethod("operation", c);
            m.invoke(this, item);
        } catch (Exception e) {}
    }

    public void operation(Book book) {
        price += book.getPrice();
        tax7 += .07*book.getPrice()/1.07;
    }

    public void operation(DVD dvd) {
        price += dvd.getPrice();
        tax19 += .19*dvd.getPrice()/1.19;
    }
```