# Vortragsthema

Johannes Pfann

Lehrstuhl für Software Engineering
Friedrich-Alexander-Universität Erlangen-Nürnberg

# Gliederung

# Gliederung

# Verhaltensmuster – Was ist das?

## Verhaltensmuster ...

- Strukturieren oder vereinfachen komplexe Abläufe
- Befassen sich mit ..
  - den Zuständigkeiten von Objekten
  - den Muster von Interaktionen

# Typen von Erzeugungsmustern

## Klassenbasiert

Klassenbasierte Verhaltensmuster wenden für die Verhaltenszuordnung zu den Klassen das Vererbungsprinzip an.

- Template Method
- Interpreter

# Typen von Erzeugungsmustern

## Objektbasiert

Bei der Objektbasierten Verhaltensmuster wir die Objektkomposition für die Verhaltenszuordnung genutzt

- Observer
- Command
- Visitor
- Strategy
- Mediator
- Iterator
- Memento
- State
- Chain of Responsibility

# Gliederung

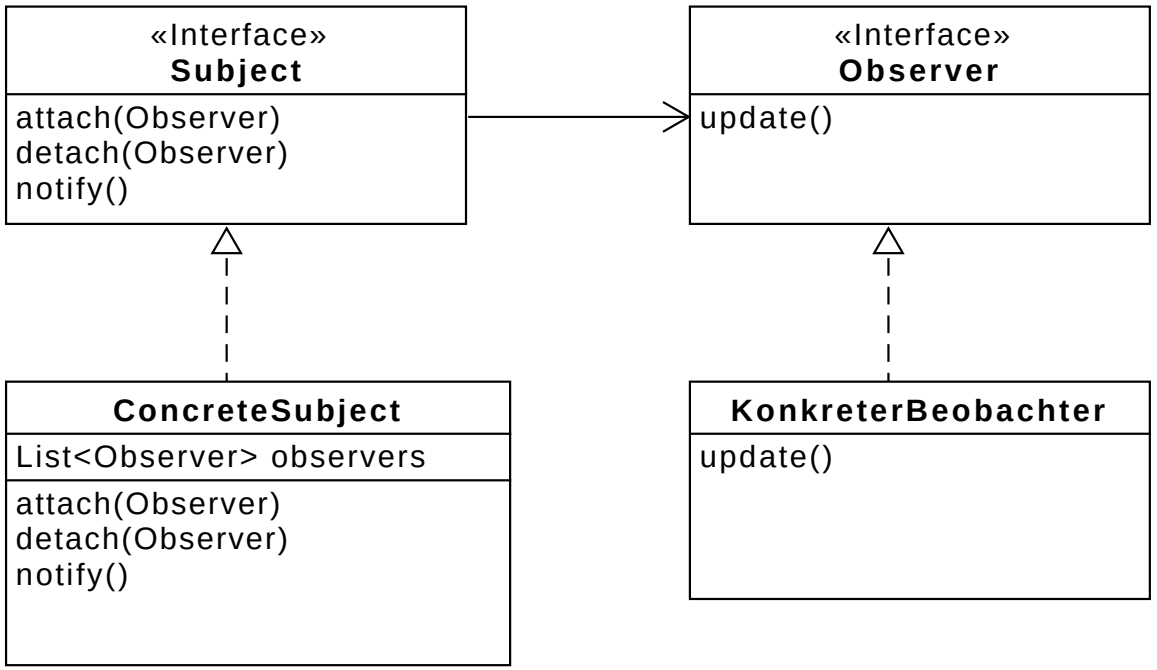
# Definition

## Zweck

Definition einer 1-zu-n-Abhängigkeit zwischen Objekten, damit im Fall einer Zustandsänderung eines Ojbekts alle davon abhängigen Objekte entsprechend benachrichtigt und automatisch aktualisiert werden.

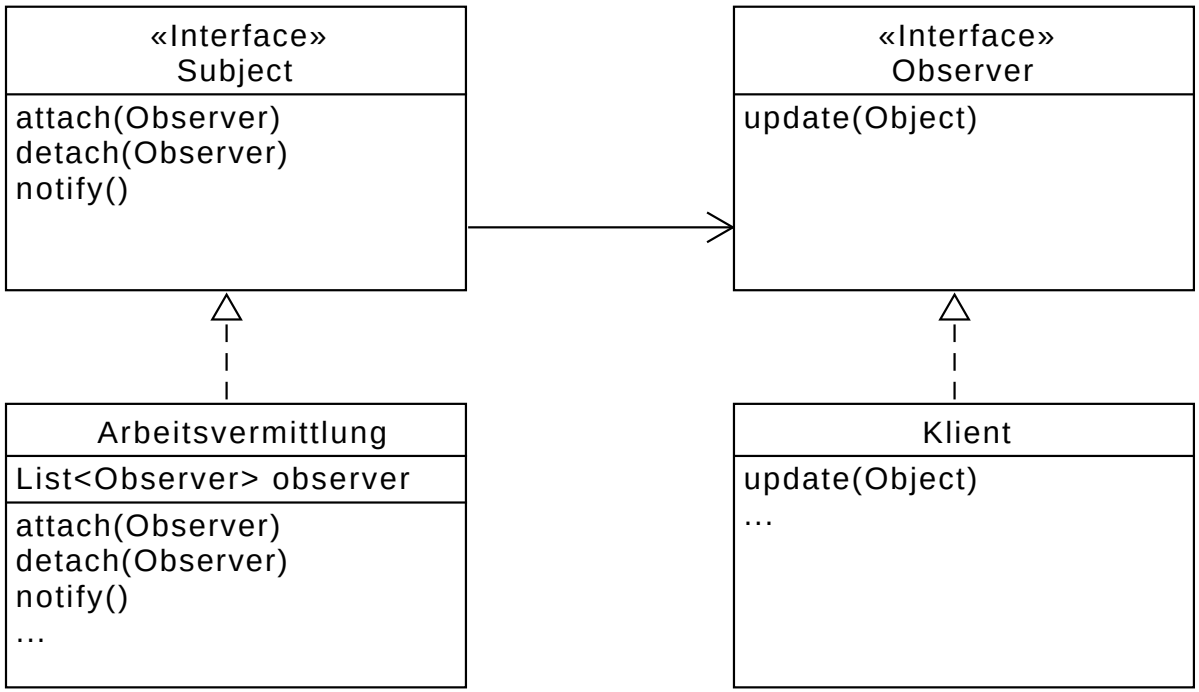
# Observer Pattern

- Basiert auf Rollen
- Stellt Mechanismus zur Broadcast-Kommunikation dar



# Arbeitsvermittlung

- 1-zu-n-Kommunikation zwischen Vermittlung und Klienten (Broadcast)
- Arbeitsvermittlung kennt weder Anzahl noch konkrete Klienten
- Klienten melden sich nur an, wenn sie daran interessiert sind.



# Beispiel - Klient

```java
public interface Subject{
  void attach(Observer aObserver);
  void detach(Observer aObserver);
  void notifyObservers();
}
```

```java
public class Klient implements Observer {

    public void update(Angebot aAngebot) {
        if(aAngebot instanceof InformatikJob){
            doSomething(aAngebot);
        }
    }
}
```

# Beispiel - Arbeitsvermittlung

```java
public interface Observer{
  void update(Object aObject);
}
```

```java
public class Arbeitsvermittlung implements Subject {

    List<Observer> mObserver = new LinkedList<Observer>();

    public void attach(Observer aObserver) {
        mObserver.add(aObserver);
    }
    public void detach(Observer aObserver) {
        mObserver.remove(aObserver);
    }
    public void notifyObservers(Angebot aAngebot) {
        for(Observer observer : mObserver){
            observer.update(aAngebot);
        }
    }
}
```

# Implementierungsmöglichkeiten

## Push Modell

- Daten nur in der update-Methode
- Zugriff auf Subject nicht erlaubt
- Subjekt muss Interesse der Observer kennen.

## Pull Modell

- update-Methode ohne Parameter
- Zugriff auf Subjekt erwünscht
- Observer müssen Subject kennen

Beides kann auch gemischt werden!

# Implementierungsmöglichkeiten

## Ausführung der Updates durch Subject

- Z.B. in Setter-Methoden
- Weniger fehleranfällig
- Jedoch zu häufige Updates

## Ausführung der Updates durch Client

- Fehleranfälliger
- Regulierung der Updates

# Implementierungsmöglichkeiten

## Observer beobachten mehrere Subects

- Observer registriert sich bei mehreren Subjects
- Muss allerdings unterschiedlich darauf reagieren
- Lösung: erweiterung der update-Methode mit Subject

```
1    public void update(Subject aSubject, Angebot aAngebot) {
2        if(aSubject instanceof ArbeitsvermittlungA){
3            doSomething(aAngebot);
4        }
5        if(aSubject instanceof ArbeitsvermittlungB){
6            doSomething(aAngebot);
7        }
8        ...
9    }
```

# Implementierungsmöglichkeiten

## Observer gibt sein Interesse an

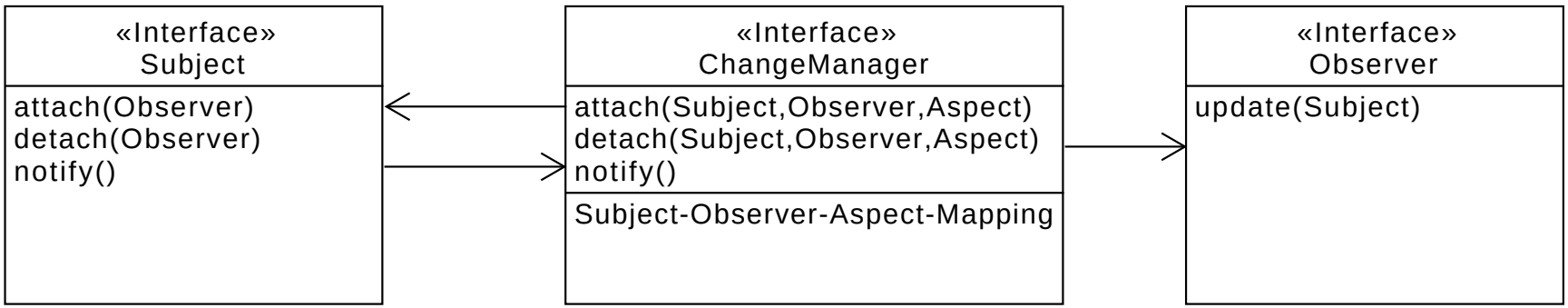
- Observer registrieren sich für ein bestimmtes Event
- Subject kümmert sich um die Zuordnung
- Benachrichtigung wird effizienter
- Subject wird komplexer

```
1 public void attach(Observer aObserver, Aspect aInterest){
2         sortToObserverlist(aInterest, aObserver);
3 }
```

# Implementierungsmöglichkeiten

## Einführung eines `ChangeManager`

- Subject delegiert Aufgaben zu ChangeManger
- ChangeManger hat drei Aufgaben:
  - Legt Zuordnung von Subject, Observer und Aspect fest
  - Legt Aktualisierungsstrategie fest
  - Führt die Aktualisierung aus



# Observer-Pattern als Fehlerquelle

## Konsistenz vor dem Update

- Vorsicht bei Vererbung

## Verwaiste Referenzen auf gelöschte Subjects

- Subject wird gelöscht und Observer wissen nichts darüber
- Java hat Garbage collection
- C++ hat Pointer

## Komplexe Strukturn

- Zyklische Abhänigkeiten
- Fehlersuche sehr komplex

# Fazit

## Vorteile

- Lose Kopplung
- Flexibilität
- Automatische Benachrichtigung

## Nachteile

- Komplexität
- Nachvollziehbarkeit (bei Fehlern)
- Gefahr von Zyklen

# Gliederung

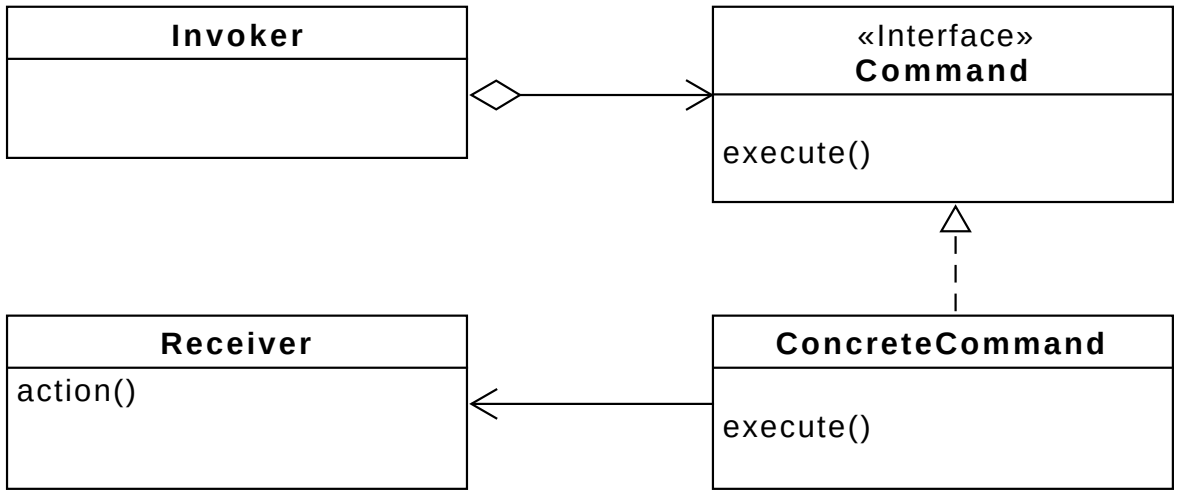
# Definition

## Zweck

Kapselung eines Requests als Objekt, um so die Parametrisierung von Clients mit verschiede- nen Requests, Warteschlangen- oder Logging-Operationen sowie das Rückgängigmachen von Operationen zu ermöglichen

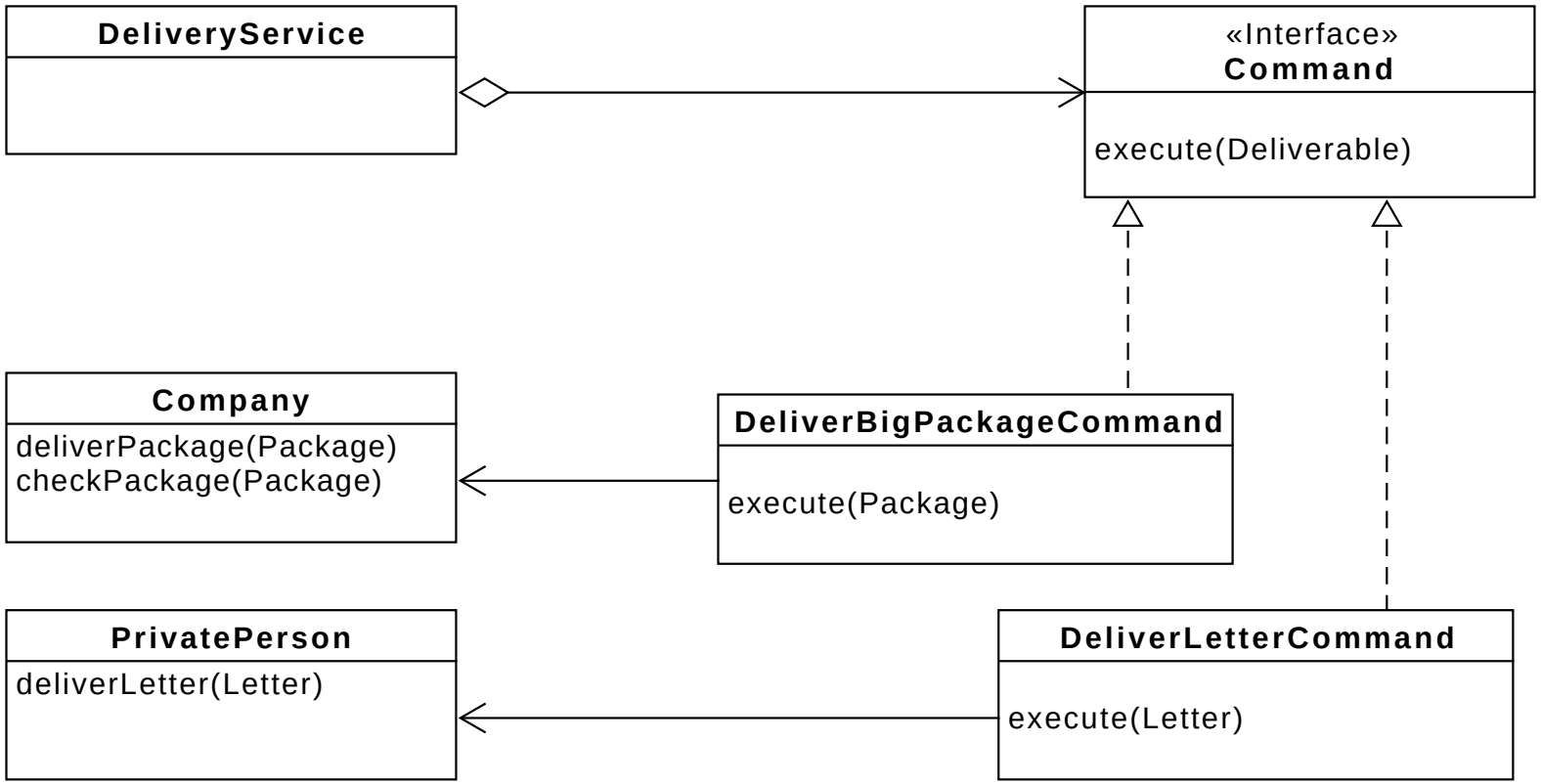
# Observer Pattern

- Kapselt Request in ein eigenes Objekt Command
- Implementierung von Command kennt dann auch den Empfänger



# Klassendiagramm

- Verschiedene Objekte die unterschiedlich ausgeliefert werden müssen
- Unterschiedliche Kunden



# Command

```java
public interface Command {
    void execute(Object aObject);
}
```

```java
public class DeliverBigPackageCommand implements Command {
    ...
    public DeliverBigPackageCommand(Company aCompany){
        mCompany = aCompany;
    }
    @Override
    public void execute(Object aObject) {
        mCompany.cheackPackage((Package)aObject);
        mCompany.deliverPackage((Package)aObject);
    }
}
```

# Command Interface

```java
public class DeliveryService {
    public void sendObject(Command aCommand, Object aObject){
        aCommand.execute(aObject);
    }
}
```

```java
public class main {

    public final static void main(String[] args){
        DeliverBigPackageCommand commandDeliverToDATEV =
        new DeliverBigPackageCommand(new Company());
        DeliverLetterCommand commandDeliverToJohanes =
        new DeliverLetterCommand(new PrivatePerson());

        mDeliverService.sendObject(
        commandDeliverToDATEV, new Package());
        mDeliverService.sendObject(
        commandDeliverToJohanes, new Letter());
    }
}
```

# Erweiterung

## Undo-Funktion

- Erweiterung des Interfaces Command mit undo-Methode
- Der Invoker kann sich Commands merken und auf diese eine undo-Methode aufrufen
- das ConcreteCommand muss dann ggf. Daten speichern:
  - Receiver-Objekt
  - Die Argumente, die für die Ausführung angewendet wurden
  - Alle relevanten Orginalwerte im Receiver-Objekt

# Erweiterung

## Makro-Befehle

- Mehrere Receiver könnten gleichzeitig durch ein Command bearbeitet werden

```
1  public class MacroCommand implements Command {
2
3      public MacroCommand(Company aCompany,
4          PrivatePerson aPrivatePerson){
5          mCompany = aCompany;
6      }
7
8      @Override
9      public void execute(Deliverable aObject) {
10         mCompany.cheackPackage(aObject);
11         mCompany.deliverPackage(aObject);
12
13         mPrivatePerson.deliverLetter(aObject);
14     }
15 }
```

# Implementierungsmöglichkeit

## Intelligenz der Commandobjekte

- Command übernimmt vollständig die Logik
- Comnand delegiert die komplette Logik an den Receiver

```
1 public void execute(){
2     int sum = mValue + 1;
3     System.out.println(sum);
4 }
```


```
1 public void execute(){
2     int sum = mValue + 1;
3     System.out.println(sum);
4 }
```

# Fazit

## Vorteile

- Entkopplung von Befehl und Ausführung
- Aufrufer können mit dem Interface Command arbeiten ohne wissen zu müssen, welche Operationen hinter den konkreten Commands stecken
- Flexiblität indem Commands leicht ausgetauscht werden können

## Nachteile

- Hohe Anzahl von Klassen

# Gliederung

# Gliederung