

# Verhaltensmuster

## Observer, Command, Visitor

Johannes Pfann

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

25.05.2016

# Gliederung

1 Verhaltensmuster

2 Observer

3 Command

4 Visitor

5 Quellen

# Gliederung

## 1 Verhaltensmuster

## 2 Observer

## 3 Command

## 4 Visitor

## 5 Quellen

# Verhaltensmuster – Was ist das?

## Verhaltensmuster ...

- Verhaltensmuster definieren Strukturen um das Verhalten von Software flexibler zu gestalten
- Dabei kapseln sie das Verhalten und lagern dieses aus
- Verhalten lässt sich somit flexibel zuordnen

# Typen von Verhaltensmustern

## Klassenbasiert

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

- Template Method
- Interpreter

# Typen von Verhaltensmustern

## Objektbasiert

Bei den objektbasierten Verhaltensmustern wird die *Objektkomposition* für die Verhaltenszuordnung genutzt

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

# Gliederung

1 Verhaltensmuster

2 Observer

3 Command

4 Visitor

5 Quellen

# Definition

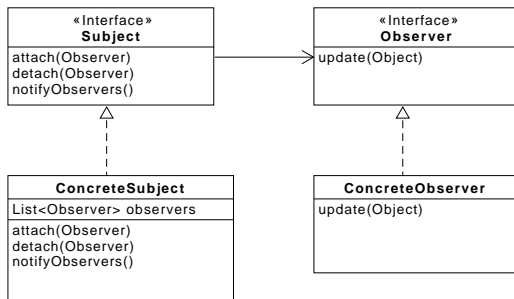
## Zweck

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



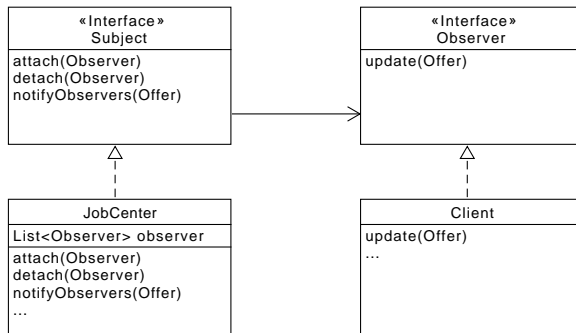
# Klassendiagramm

- 1-zu-n-Abhängigkeit
- Subject  
benachrichtigt  
Observer
- Zustandsänderung  
ist Auslöser
- Clients  
aktualisieren sich  
selbst



# Jobcenter

- 1-zu-n-Kommunikation zwischen Jobcenter und Client
- Jobcenter kennt weder Anzahl noch konkrete Clients
- Clients bearbeiten ihre Benachrichtigung unterschiedlich



# Beispiel - Jobcenter

```
1 public interface Subject {
2     void attach(Observer aObserver);
3     void detach(Observer aObserver);
4     void notifyObservers();
5 }
6
7 public class JobCenter implements Subject {
8
9     List<Observer> mObserver = new LinkedList<Observer>();
10
11     public void attach(Observer aObserver) {
12         mObserver.add(aObserver);
13     }
14     public void detach(Observer aObserver) {
15         mObserver.remove(aObserver);
16     }
17     public void notifyObservers(Offer aOffer) {
18         for(Observer observer : mObserver){
19             observer.update(aOffer);
20         }
21     }
22 }
```

# Beispiel - Client

```
1 public interface Observer{
2     void update(Offer aOffer);
3 }
4
5 public class Client implements Observer {
6
7     public void update(Offer aOffer) {
8         if(aOffer instanceof DeveloperJob){
9             doSomething(aOffer);
10        }
11    }
12 }
```

# Implementierungsmöglichkeiten

## Push Modell

- Subject übergibt detaillierte Informationen
- Observer hat keinen Zugriff auf Subject
- Subjekt muss Interesse der Observer kennen

## Pull Modell

- Subject informiert nur auf Veränderung und übergibt keine Daten
- Observer muss sich Daten vom Subject holen
- Observer müssen Subject kennen

Beides kann auch gemischt werden!

# Implementierungsmöglichkeiten

## Ausführung der Benachrichtigungsmethode durch Subject

- z.B. in Add-Methoden
- Weniger fehleranfällig
- Jedoch zu häufige Updates

## Ausführung der Benachrichtigungsmethode von Extern

- Fehleranfälliger
- Regulierung der Updates

# Implementierungsmöglichkeiten

## Observer beobachten mehrere Subjects

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

```
1 public void update(Subject aSubject, Offer aOffer) {  
2     if(aSubject instanceof JobCenterA) {  
3         doSomething(aOffer);  
4     }  
5     if(aSubject instanceof JobCenterB) {  
6         doSomething(aOffer);  
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

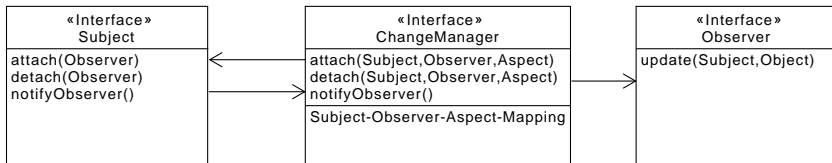
```
1 public void attach(Observer aObserver, Aspect aInterest){  
2     sortToObserverlist(aInterest, aObserver);  
3 }
```



# Implementierungsmöglichkeiten

## Einführung eines ChangeManager

- Subject delegiert Aufgaben zu ChangeManager
- 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

## Komplexe Struktur

- Zyklische Abhängigkeiten
- Komplexe Fehlersuche

# Fazit

## Vorteile

- Lose Kopplung
- Flexibilität
- Automatische Benachrichtigung

## Nachteile

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

# Gliederung

1 Verhaltensmuster

2 Observer

**3 Command**

4 Visitor

5 Quellen

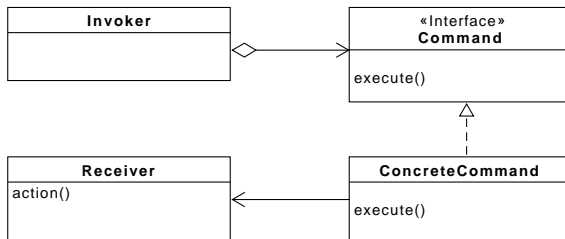
# Definition

## Zweck

Kapselung eines Requests als Objekt, um so die Parametrisierung von Clients mit verschiedenen Requests zu ermöglichen.

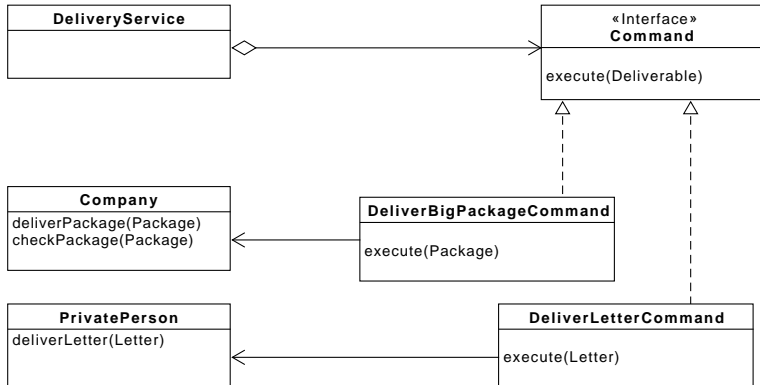
# Command Pattern

- Kapselt Request in ein eigenes Command-Object
- Implementierung von Command kennt den Empfänger



# Beispiel Klassendiagramm

- Verschiedene Objekte die unterschiedlich ausgeliefert werden müssen
- Unterschiedliche Kunden



# Command

```
1 public interface Command {
2     void execute(Deliverable aObject);
3 }

1 public class DeliverBigPackageCommand implements Command {
2     ...
3     public DeliverBigPackageCommand(Company aCompany) {
4         mCompany = aCompany;
5     }
6     @Override
7     public void execute(Deliverable aObject) {
8         mCompany.checkPackage((Package) aObject);
9         mCompany.deliverPackage((Package) aObject);
10    }
11 }
```



# Command Interface

```
1 public class DeliveryService {
2     public void sendObject(Command aCommand, Deliverable aObject){
3         aCommand.execute(aObject);
4     }
5 }

1 public class main {
2
3     public final static void main(String[] args){
4         DeliverService deliverService = new DeliverService();
5         DeliverBigPackageCommand commandDeliverToDATEV =
6             new DeliverBigPackageCommand(new Company());
7         DeliverLetterCommand commandDeliverToJohannes =
8             new DeliverLetterCommand(new PrivatePerson());
9
10        deliverService.sendObject(
11            commandDeliverToDATEV, new Package());
12        deliverService.sendObject(
13            commandDeliverToJohannes, new Letter());
14    }
15 }
```

# 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 Originalwerte 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         mPrivatePerson = aPrivatePerson;  
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
- Command 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     receiver.incrementValue(mValue);  
3 }
```

# 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
- Flexibilität indem Commands leicht ausgetauscht werden können

## Nachteile

- Hohe Anzahl von Klassen

# Gliederung

1 Verhaltensmuster

2 Observer

3 Command

**4 Visitor**

5 Quellen

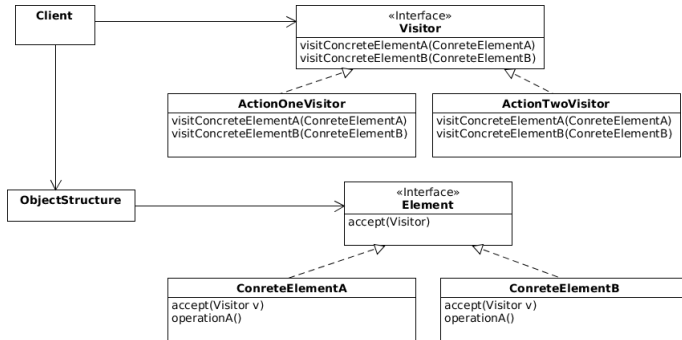
# Definition

## Zweck

Das Design Pattern Visitor ermöglicht die Definition einer neuen Operation, ohne die Klasse der von ihr bearbeiteten Elemente zu verändern.

# Visitor Pattern

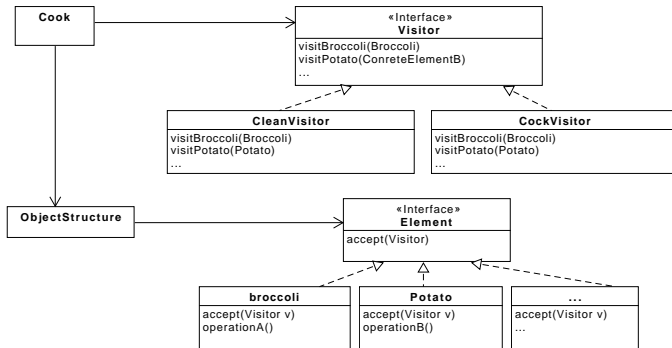
- Elemente der Objektstruktur fest
- Operationen auf Objektstruktur sollen austauschbar und erweiterbar sein





# Klassendiagramm

- Die verschiedenen Sorten die wir bearbeiten möchten, bleiben konstant
- Wie wir allerdings diese bearbeiten ist noch unklar.



# Schnittstellen

```
1 public interface Element {  
2     void accept(Visitor aVisitor);  
3 }  
  
1 public interface Visitor {  
2     void visitBroccoli(Broccoli aBroccoli);  
3     void visitPotato(Potato aPotato);  
4     ...  
5 }
```

# Implementierungen

```
1 public class Potato implements Element {
2     @Override
3     public void accept(Visitor aVisitor) {
4         aVisitor.visitPotato(this);
5     }
6 }

1 public class CleanVisitor implements Visitor {
2
3     @Override
4     public void visitBroccoli(Broccoli aBroccoli) {
5         System.out.println("Clean broccoli");
6         modifyBroccoli(aBroccoli);
7     }
8
9     @Override
10    public void visitPotato(Potato aPotato) {
11        System.out.println("Clean potatoes");
12        modifyPotato(aPotato);
13    }
14 }
```

# Client

```
1 public class main {
2
3     public static final void main(String[] args){
4         Visitor cleanVisitor = new CleanVisitor();
5         Visitor cookVisitor = new CookVisitor();
6         Element[] elements = new Element[2];
7         elements[0] = new Broccoli();
8         elements[1] = new Potato();
9
10        for(int i = 0; i < elements.length;i++){
11            elements[i].accept(cleanVisitor);
12        }
13
14        for(int i = 0; i < elements.length;i++){
15            elements[i].accept(cookVisitor);
16        }
17    }
18
19 }
```

# Implementierungen

## Wer ist für die Traversierung der Objektstruktur verantwortlich

- In der Objektstuktur
- Im Visitor-Objekt
- In einem eigenen Iterator-Objekt

# Implementierung

## Objektstruktur

- Objektstruktur kümmert sich um die Traversierung
- Visitor muss der Objektstruktur übergeben werden
- Realisierbar durch z.B Composite-Pattern

```
1 public void accept(Visitor aVisitor) {  
2     visitElement(aVisitor)  
3     for(Element element : mElements) {  
4         element.accept(aVisitor)  
5     }  
6 }
```

# Implementierung

## Iterator

- Zugriff auf Elementen einer Objektstruktur, ohne die Objektstruktur zu kennen.
- Die Visitor-Objekten den Iterator übergeben
- Beim Zugriff eines Elements die accept-Methode aufrufen

## Im Visitor-Objekt

- Objektstruktur wird den konkreten Visitors übergeben
- Visitor übernimmt jetzt die Traversierung
- Nachteil: Traversierung in jedem Visitor
- Vorteil: Flexibler in der Steuerung für bestimmte Aufgaben

# Implementierung

```
1 public class CleanVisitor implements Visitor {
2     ...
3     public CleanVisitor(List<Element> aElement) {
4         mElement = aElement;
5     }
6
7     @Override
8     public void traverse() {
9         traverseBroccoli();
10        traversePotato();
11    }
12
13    @Override
14    public void visitBroccoli(Broccoli aBroccoli) {
15        ...
16    }
17
18    @Override
19    public void visitPotato(Potato aPotato) {
20        ...
21    }
22 }
```



# Fazit

## Vorteile

- Leicht neue Operationen auf eine Objektstruktur zu implementieren
- Selektives bearbeiten einzelner Elemente einer Objektstruktur
- Flexibilität indem Commands leicht ausgetauscht werden können

## Nachteile

- Enge Kopplung des Visitor mit den Elementen einer Objektstruktur
- Elemente müssen über öffentliche Methoden und Attribute dem Visitor den Zugriff bereitstellen

# Gliederung

1 Verhaltensmuster

2 Observer

3 Command

4 Visitor

5 Quellen



Gamma, Helm, Johnson, Vlissides: „Design Patterns - Entwurfsmuster als Elemente wiederverwendbarer Objektorientierter Software“. 1. Auflage, mitp Verlags GmbH, 2015.



Eilebrecht, Starke: „Patterns Kompakt - Entwurfsmuster für effektive Software-Entwicklung“. 3. Auflage, Spektrum Verlag, Heidelberg 2010.



Michael Inden: Der Weg zum Java-Profi. 3 Auflage dpunkt.verlag



Eric Freeman, Elisabeth Freeman, Kathy Sierra, Bert Bates: Entwurfsmuster von Kopf bis Fuß. 1 Auflage O'Reilly Media, Inc