

# Vortragsthema

Johannes Pfann

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

# Gliederung

- 1 Command
- 2 Visitor
- 3 Zusammenfassung

# Gliederung

- 1 Command
- 2 Visitor
- 3 Zusammenfassung

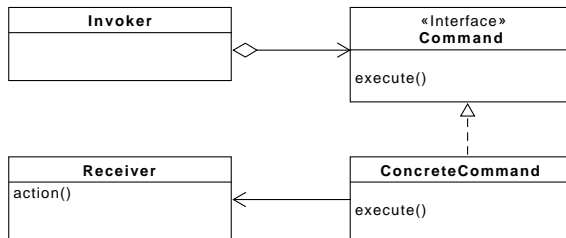
# Definition

## Zweck

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

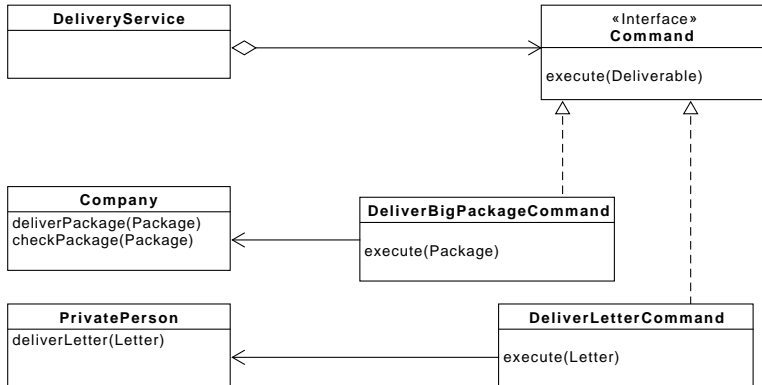
# Observer Pattern

- 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

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

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

# Command Interface

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

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



# 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     }  
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     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
- Flexibilität indem Commands leicht ausgetauscht werden können

## Nachteile

- Hohe Anzahl von Klassen

# Gliederung

- 1 Command
- 2 Visitor
- 3 Zusammenfassung

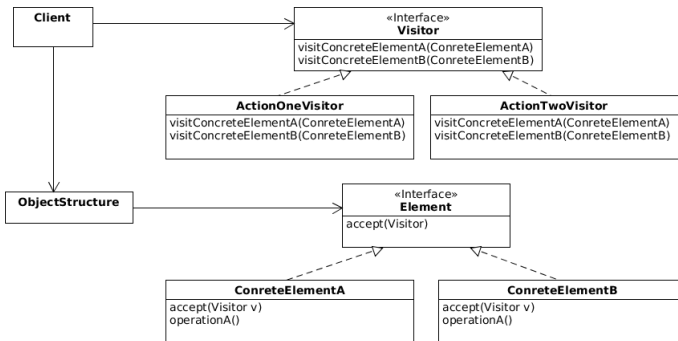
# Definition

## Zweck

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

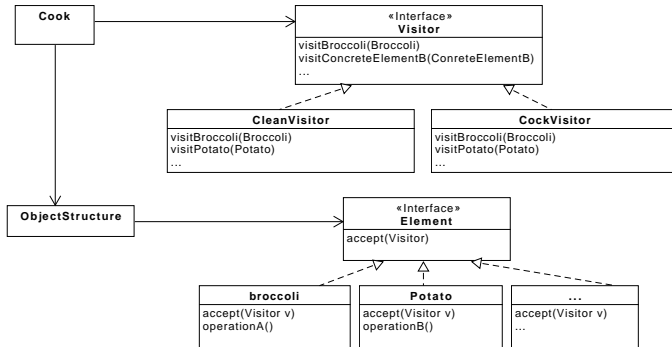
# Observer Pattern

- 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.





# Interfaces

```
1 public interface Element {
2     void accept(Visitor aVisitor);
3 }
4
5 public interface Visitor {
6     void visitBroccoli(Broccoli aBroccoli);
7     void visitPotato(Potato aPotato);
8     ...
9 }
```

# Implementierung

```
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 }
```

# Implementierung

## Wer ist für die Traversierung der Objektstruktur verantwortlich

- In der Objektstruktur
- 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 }
```

# Gliederung

- 1 Command
- 2 Visitor
- 3 Zusammenfassung**