Vortragsthema

Johannes Pfann

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

Gliederung

Gliederung

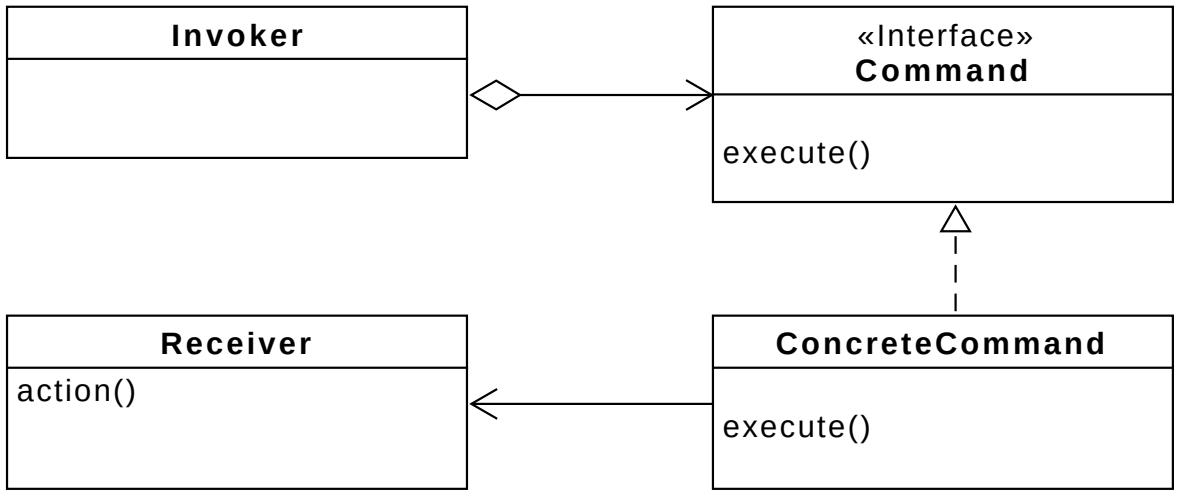
Definition

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

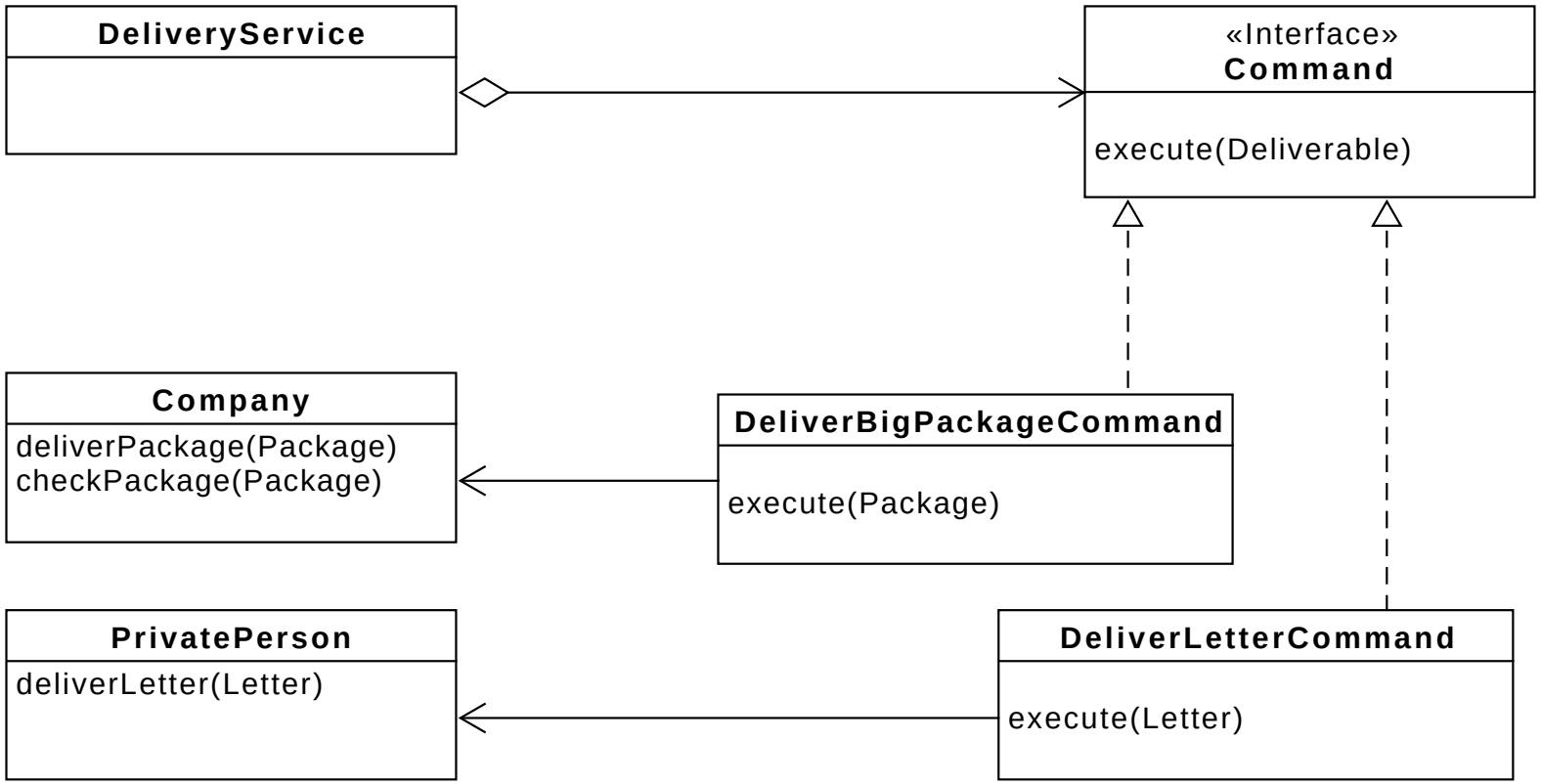
Observer Pattern

- 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

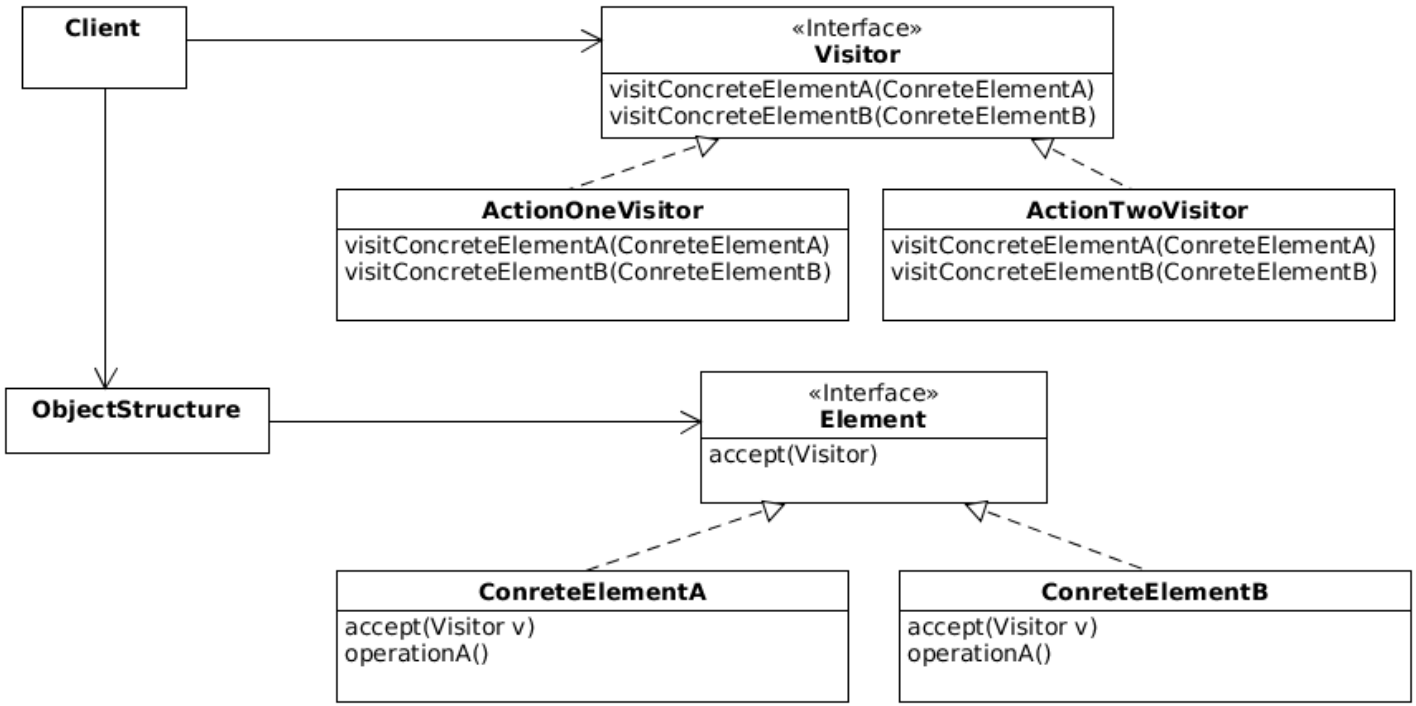
Definition

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.

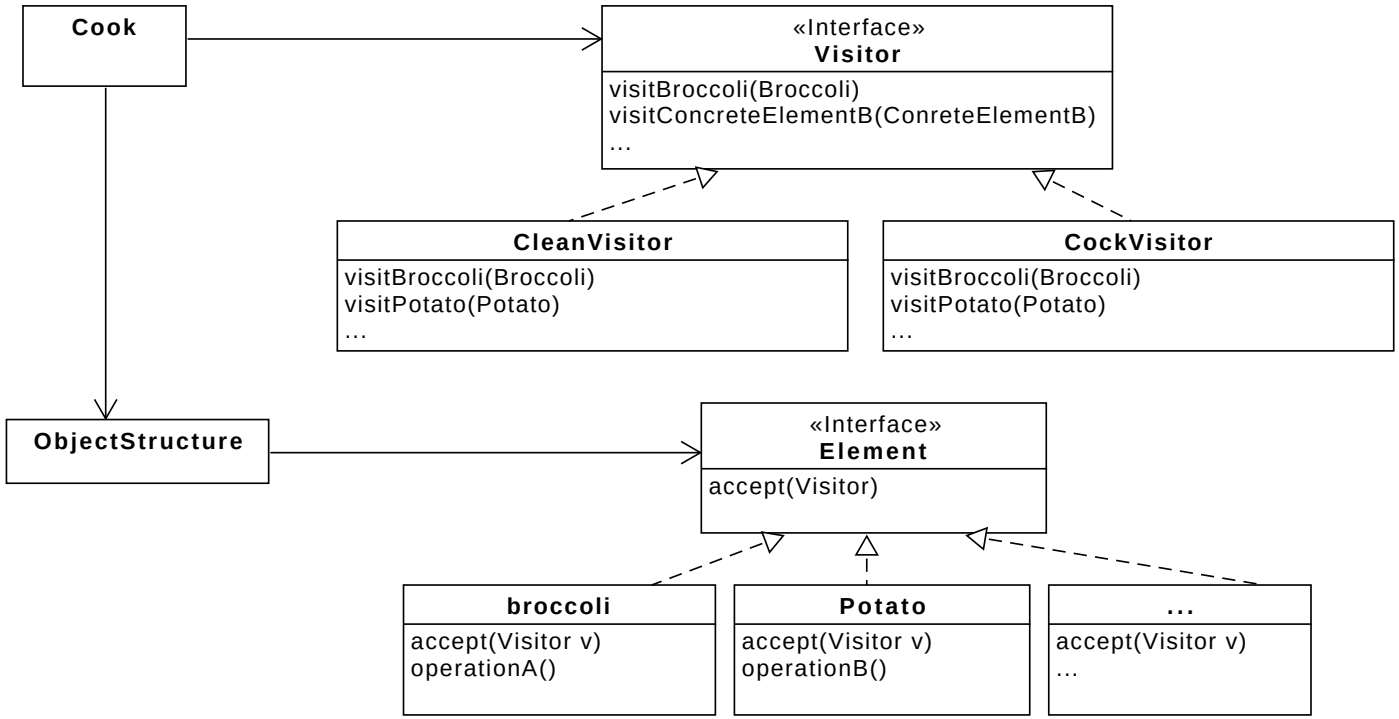
Observer Pattern

- 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

```java
public interface Element {
    void accept(Visitor aVisitor);
}
```

```java
public interface Visitor {
    void visitBroccoli(Broccoli aBroccoli);
    void visitPotato(Potato aPotato);
    ...
}
```

Implementierung

```java
public class Potato implements Element {
    @Override
    public void accept(Visitor aVisitor) {
        aVisitor.visitPotato(this);
    }
}
```

```java
public class CleanVisitor implements Visitor {

    @Override
    public void visitBroccoli(Broccoli aBroccoli) {
        System.out.println("Clean broccoli");
        modifyBroccoli(aBroccoli);
    }

    @Override
    public void visitPotato(Potato aPotato) {
        System.out.println("Clean potatoes");
        modifyPotato(aPotato);
    }
}
```

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 }
```

Gliederung