Verhaltensmuster: Observer, Command und Visitor

Johannes Pfann

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

25.05.2016

Gliederung

Gliederung

Verhaltensmuster – Was ist das?

Verhaltensmuster ...

- Verhaltensmuster legen Strukturen um das Verhalten von Software flexibler zu gestalten
- Dabei kapseln sie das Verhalten und lagern dieses aus
- Legen einen Fokus auf ...
  - den Zuständigkeiten von Objekten und Klassen
  - den Interaktionen zwischen Objekte und Klassen

Typen von Verhaltensmuster

Klassenbasiert

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

- Template Method
- Interpreter

Typen von Verhaltensmuster

Objektbasiert

Bei den Objektbasierten Verhaltensmuster wird die `Objektkomposition` für die Verhaltenszuordnung genutzt

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

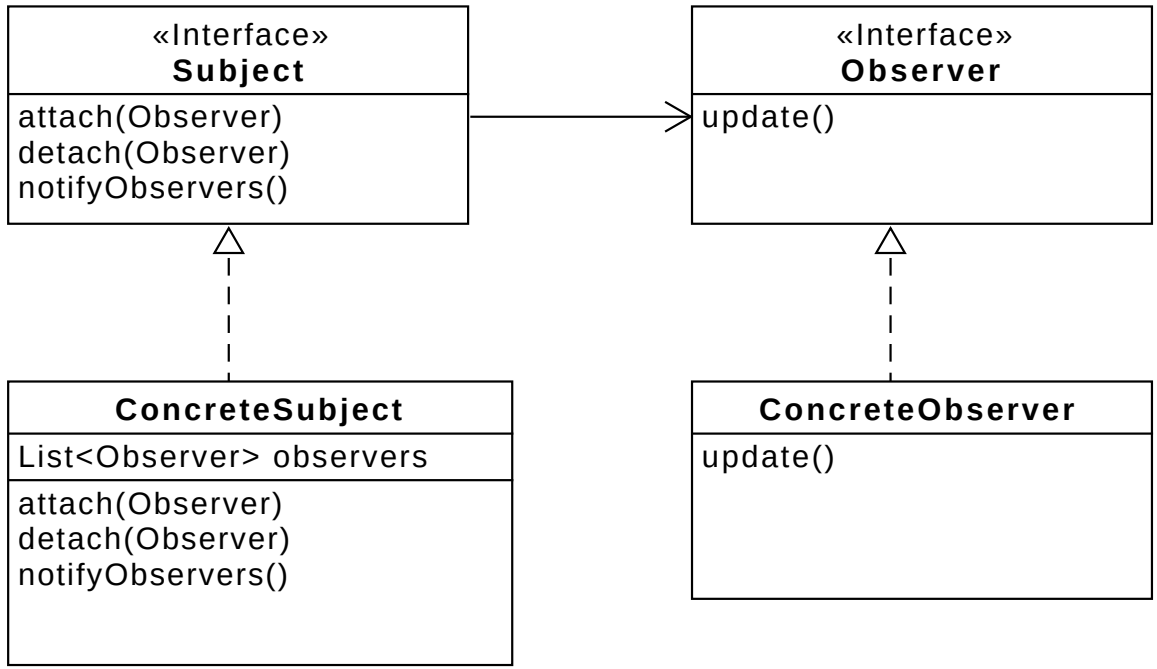
Gliederung

Definition

Zweck

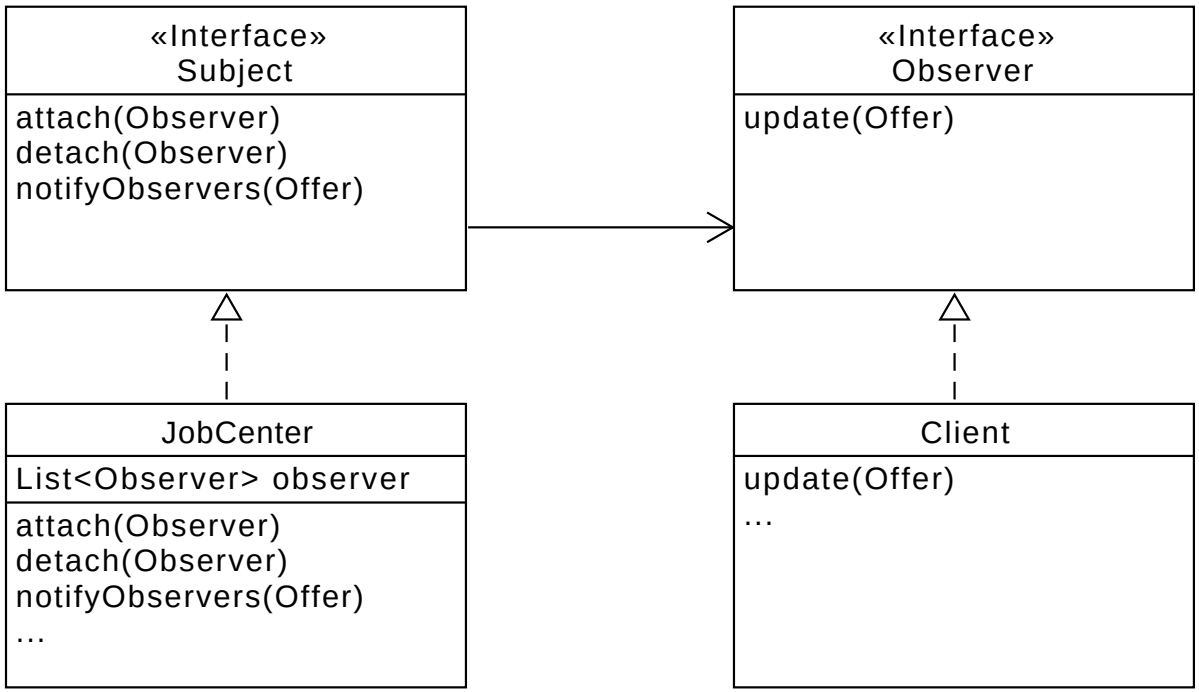
Definition einer 1-zu-n-Abhängigkeit zwischen Objekten, damit im Fall einer Zustandsänderung eines Objekts alle davon abhängigen Objekte entsprechend benachrichtigt und automatisch aktualisiert werden.

1. 1-zu-n-Anbhänigkeit
2. im Fall einer Zustandsände-rung
3. benachrichtigen
4. automatisch



Job center

- 1-zu-n-Kommunikation zwischen JobCenter und Client (Broadcast)
- JobCenter kennt weder Anzahl noch konkrete Clients
- Clients melden sich nur an, wenn sie einen Job suchen



Beispiel - Client

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

```java
public class JobCenter 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(Offer aOffer) {
        for(Observer observer : mObserver){
            observer.update(aOffer);
        }
    }
}
```

Beispiel - JobCenter

```java
public interface Observer{
  void update(Offer aOffer);
}
```

```java
public class Client implements Observer {

    public void update(Offer aOffer) {
        if(aOffer instanceof DeveloperJob){
            doSomething(aOffer);
        }
    }
}
```

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, 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

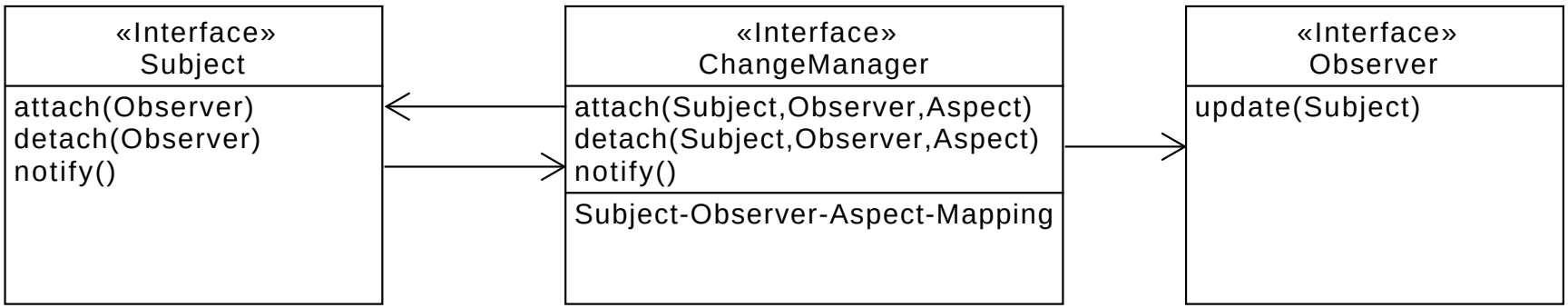
- Observer registrieren sich für ein bestimmtes Event
- Subject kümmert sich um die Zuordnung
- Benachrichtigung wird effizienter
- Subject wird komplexer

```
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 Strukturn

- Zyklische Abhängigkeiten
- Fehlersuche sehr komplex

Fazit

Vorteile

- Lose Kopplung
- Flexibilität
- Automatische Benachrichtigung

Nachteile

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

Gliederung

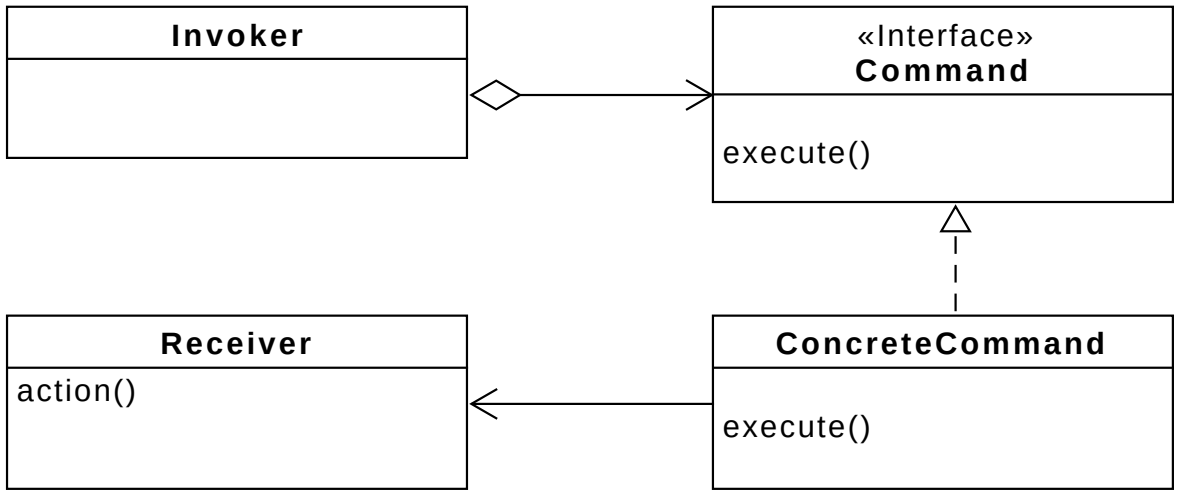
Definition

Zweck

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

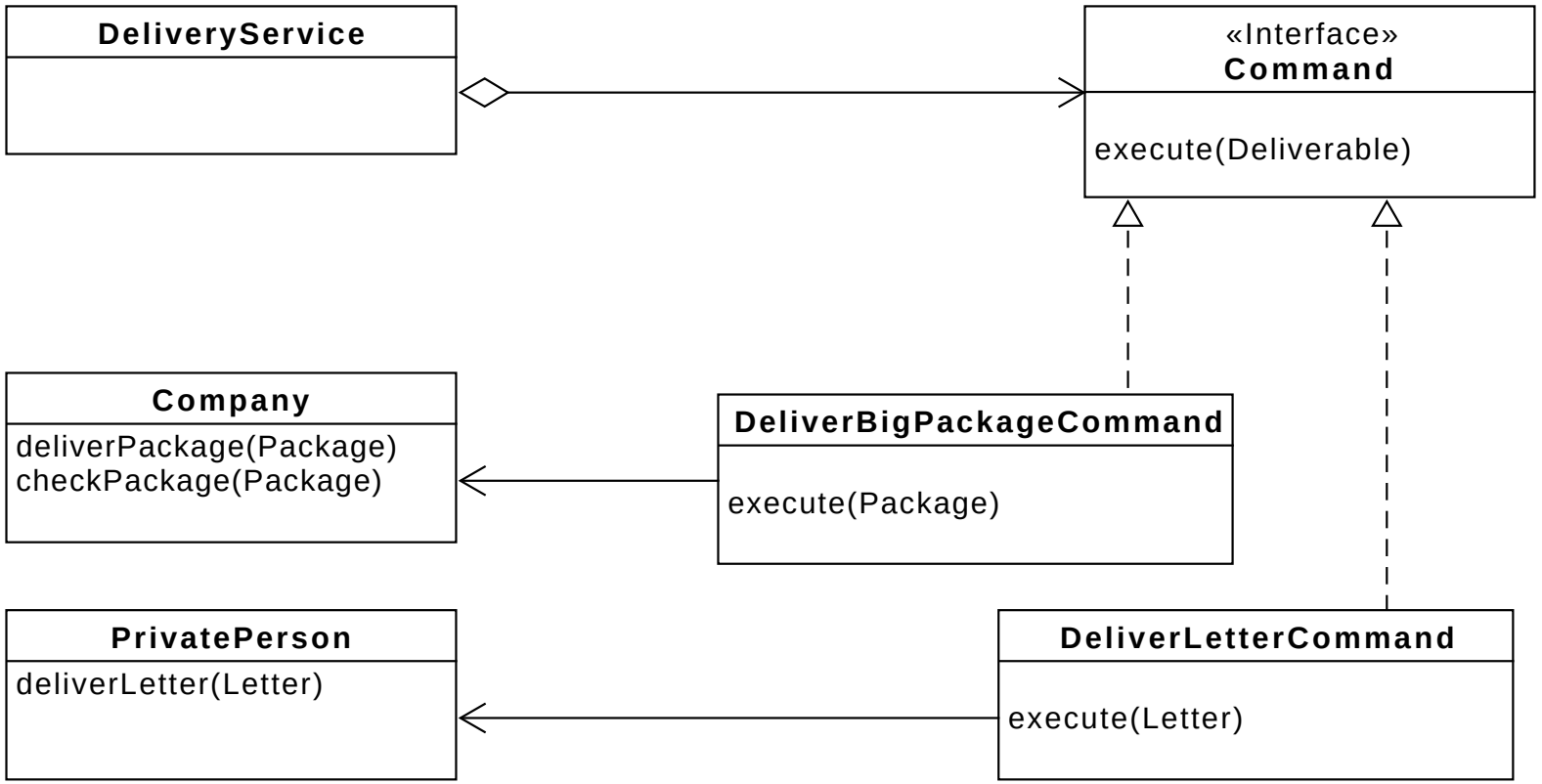
Observer Pattern

- 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

```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 }
```

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

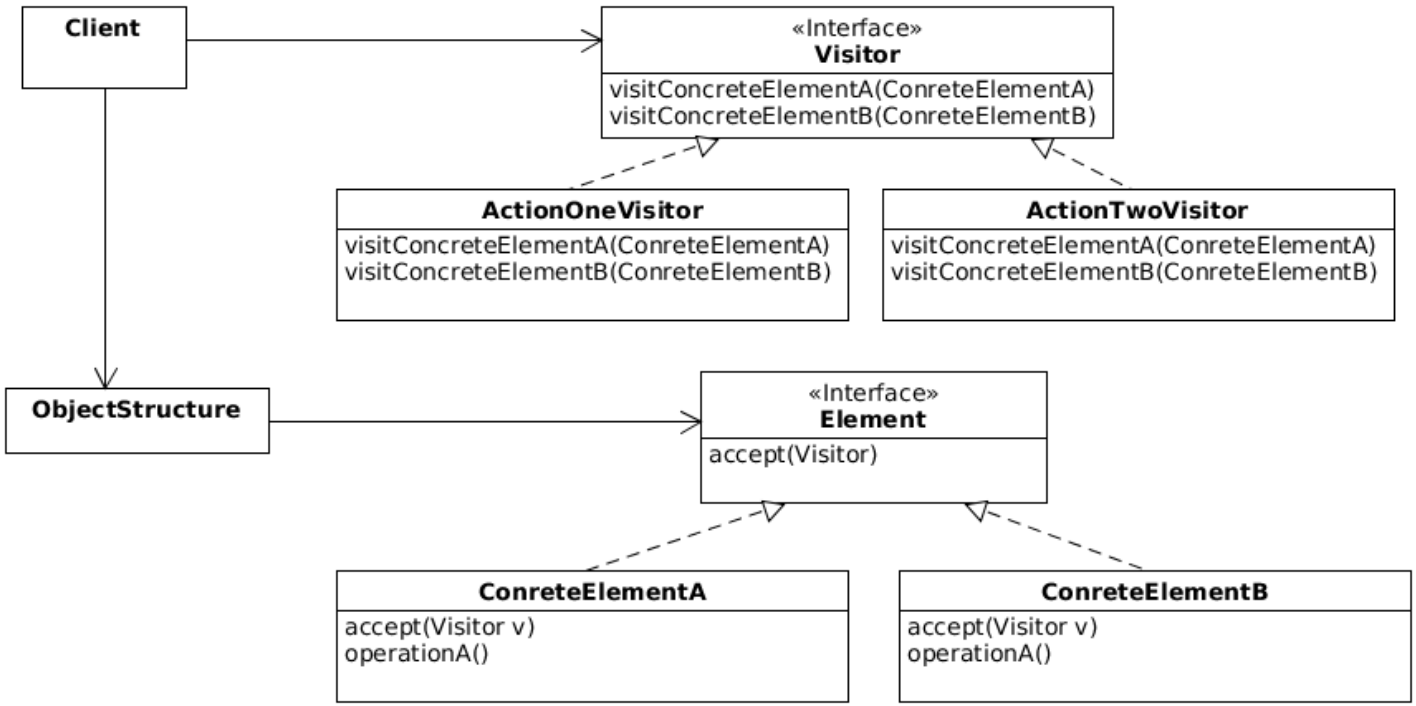
Definition

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.

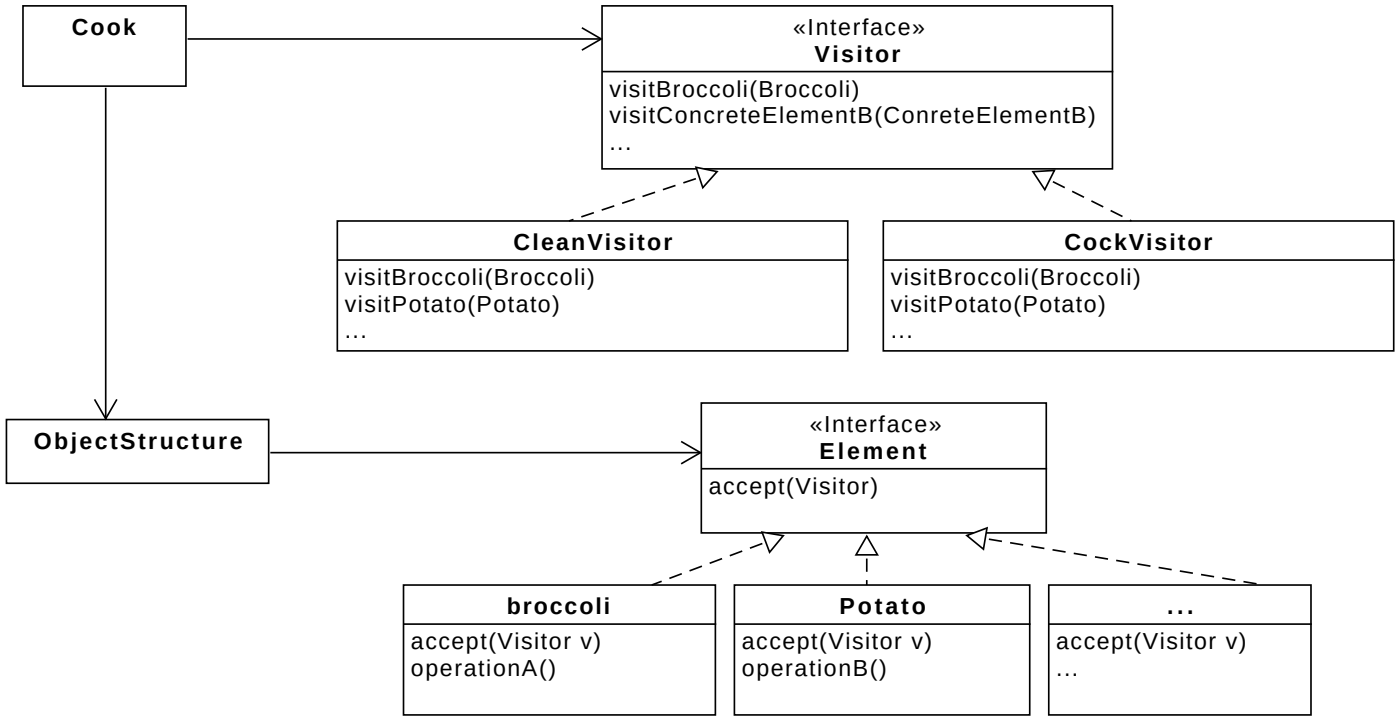
Observer Pattern

- 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 }
```

Implementierung

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 Composide-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
- Flexiblität 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


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