Entwurfsmuster

Entwurfsmuster (Design Patterns),
Visitor-, Singleton, Decorator-,
Observer-Pattern. Objektterminierte Listen

OO-Modellierung

- Datenkapselung
  - Klassen
  - Sichtbarkeit
- Vererbung
  - Gemeinsame Dinge in Oberklassen
  - Spezialisierung in Unterklassen
  - in Java: Einfachvererbung
- Polymorphie
  - Methoden in Unterklassen adaptieren
    - überschreiben, erweitern
- Abstrakte Klassen
  - Klassen zusammenfassen
- Interfaces
  - gemeinsame Funktionalität zusichern
  - in Java: Mehrfachvererbung

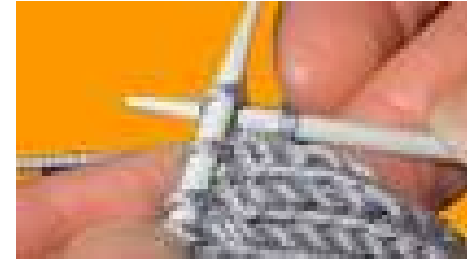
© H. Peter Gumm, Philipps-Universität Marburg

Prinzipien

- Don‘t change a running System
  - füge Funktionalität hinzu
  - auch wenn kein Source-Code vorhanden
  - Bewährte Klassen nicht anfassen
    - nicht rekompilieren
- Klassen lose koppeln
  - Veränderung einer soll sich nicht auf andere auswirken

© H. Peter Gumm, Philipps-Universität Marburg

Entwurfsmuster



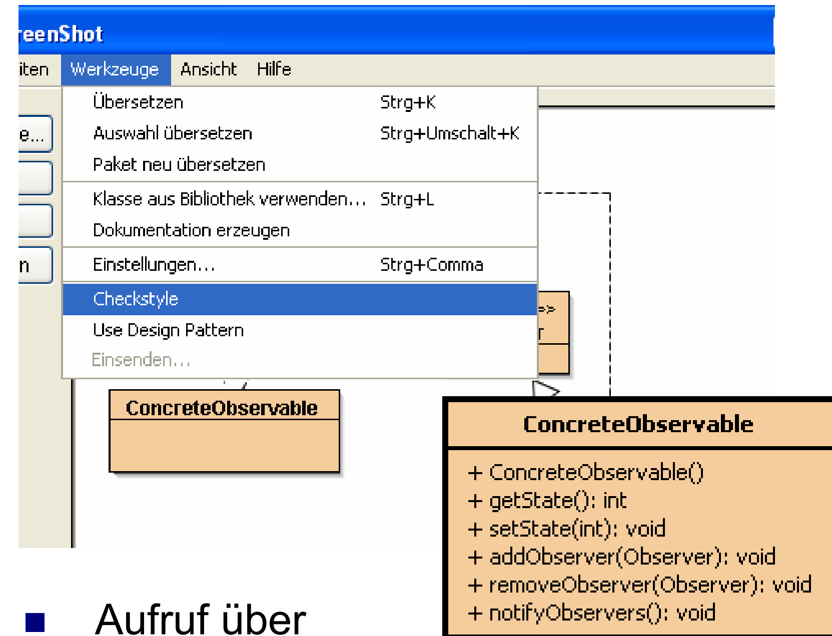
- Schemata für OO-Problemlösungen
  - Häufig vorkommende Probleme
  - einfache bewährte Lösungen

- Beispielmuster
  - Visitor Pattern
    - Durchlaufe eine Objekthierarchie.
    - Reagiere auf jedes Objekt entsprechend seiner Rolle in der Hierarchie
  - Observer Pattern
    - Beobachte und reagiere auf Ereignisse und Veränderungen.
    - Trenne Modell von Sichten
  - Decorator Pattern
    - Modifiziere Objekte
    - Kombiniere Modifikationen

© H. Peter Gumm, Philipps-Universität Marburg

BlueJ-Extensions

- BlueJ hat Extension Mechanismus
  - UML-Extension
  - Checkstyle extension
  - Design Pattern Extension
  - etc. …
- Installation
  - jar-Datei kopieren nach
    - **<BLUEJ_HOME>\lib\extensions**
      - ...for all users of this system in all projects.
    - **<USER_HOME>\bluej\extensions**
      - ...for a single user for all projects.
    - **<BLUEJ_PROJECT>\extensions**
      - ...for this project only.



- Aufruf über
  - Werkzeuge-Menü, oder
  - Rechtsklick auf Klassen-Icons
    - UML-Extension
- Extension API ist offen
  - programmieren Sie eigene Extensions

© H. Peter Gumm, Philipps-Universität Marburg

Design Pattern Extension

- Design Pattern zur Auswahl
- Nenne Klassen um
- Klassen mit relevanten Methoden werden automatisch erzeugt
  - Wizard
- Erweiterbar
  - neue Pattern
  - neue Methoden
  - neue Wizards
  - XML-Beschreibung

Design Patterns Extension [Gebühr: 0,130 Euro Onlinezeit: 00:19:42 Taktende in: 18 s...

Rename Observer (Step 2 of 5)

Use the text field to enter a relevant name for the Observer Interface.

Observer Pattern

Component: ObserverIF

Name: Observer

Component R

The Observe
an an interest
The Observ
Observable o

BlueJ: ScreenShot [Gebühr: 0,150 Euro Onlinezeit: 00:22:02 Taktende in: 58 se...

Projekt Bearbeiten Werkzeuge Ansicht Hilfe

Neue Klasse...

Übersetzen

<<interface>> Observable

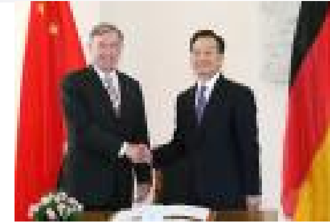
<<interface>> Observer

ConcreteObservable

ConcreteObserver

© H. Peter Gumm, Philipps-Universität Marburg

Das Visitor Pattern



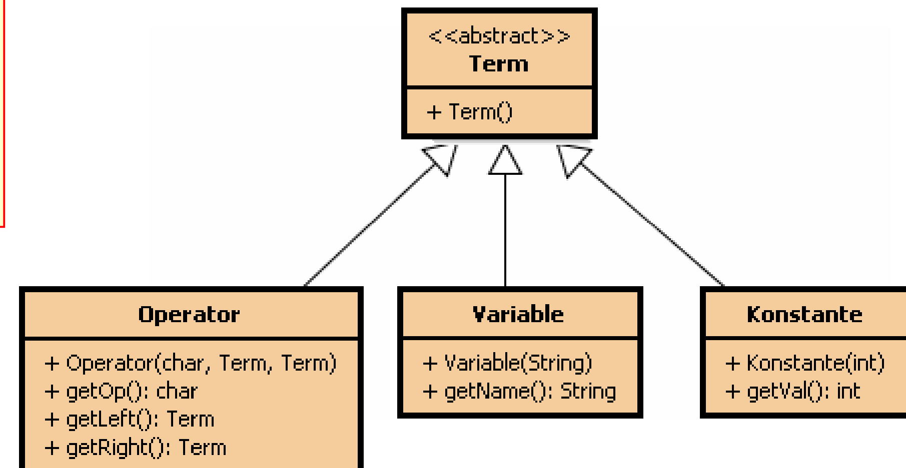
- Bearbeite Objekte verschiedener Unterklassen einer gemeinsamen Klasse
  - Je nach Klasse tue was anderes
- Bearbeitungsmethoden können im Laufe der Zeit neue hinzukommen
  - Klasse soll sich nicht ändern müssen.
- Beispiel: Baumklasse
  - im Laufe der Zeit kommen Wünsche für neue Methoden
  - preorder, postorder
  - leftMostElement
  - depth
- Die Baumklasse soll aber nicht mehr angefasst werden

© H. Peter Gumm, Philipps-Universität Marburg

Die Baumklasse als ADT

- Definition

Ein Term ist
- eine Variable
- eine Konstante
- ein Operator mit Term-Argumenten



- Beispiele
  - x + 3 * 4
  - betrag*(1+zins/100)

- Implementierung

x + 3 * 4

```
Term t = new
  Operator('+', new Variable("x"),
                new Operator( '*',new Konstante(3),
                                  new Konstante(4)));
t
<object reference>   (Operator)
```


© H. Peter Gumm, Philipps-Universität Marburg

Baumdurchlauf

```
<<abstract>>
Term
+ Term()
+ istOperator(): boolean
+ istVariable(): boolean
+ istKonstante(): boolean
```

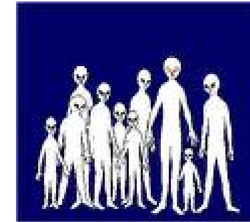
- Die funktionale Methode
  - Term erhält abstrakte Selektoren
    - istOperator()
    - istVariable()
    - istKonstante()
  - Konkret implementiert in Unterklassen
  - Durchläufer erhält Term als Argument

```java
static void preorder(Term t){
    PrintStream out = System.out;
    if(t.istOperator()){
        out.println(((Operator)t).getOp());
        preorder(((Operator)t).getLeft());
        preorder(((Operator)t).getRight());
    }else if (t.istVariable())
        out.println(((Variable)t).getName());
    else out.println(((Konstante)t).getVal());
}
```

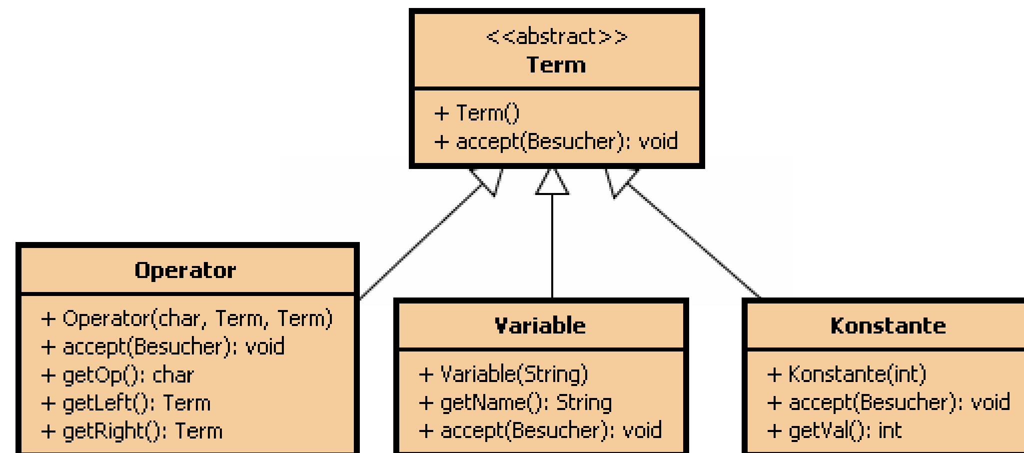
- Nachteile
  - Casts notwendig
    - warum ?
  - Nicht OO
    - static void preorder
  - Baum ist Argument, nicht Akteur
  - Einführung neuer Terme (z.B. UnOp) erfordert Modifikation aller Klassen

© H. Peter Gumm, Philipps-Universität Marburg

Akzeptiere Besucher



- Akzeptiere Besucher
  - **`accept(Besucher v)`**
- vertrau Dich ihm an
  - er macht es richtig
- Spezielle Besucher
  - preorder
  - leftmost
  - depth
  - etc…



kein Selektor mehr nötig

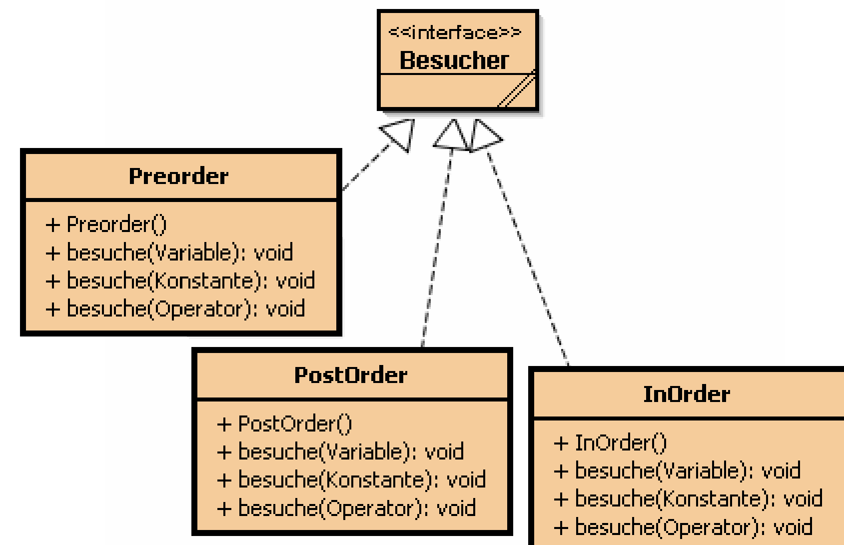
kein Cast

stattdessen **`accept`**-Methode mit Code:

```
// Besucher akzeptieren, sich ihm anvertrauen
public  void accept(Besucher v){
      v.besuche(this);
```


© H. Peter Gumm, Philipps-Universität Marburg

Die Besucher



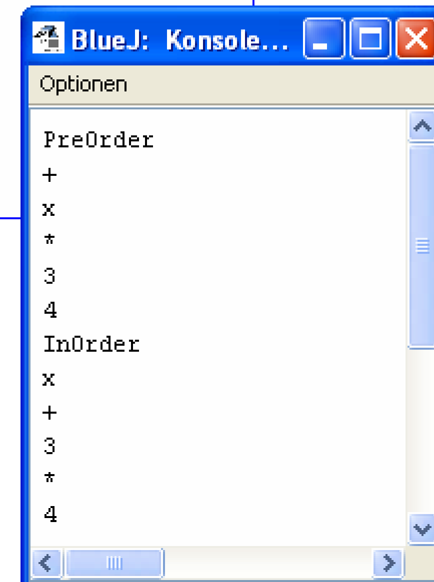
- klopfen an
  - `accept(v)`
- übernehmen das Kommando
  - `v.besuche(this)`
- erledigen ihre Sache im Objekt
  - `println(t.getOp());`
  - `println(t.getName());`
  - `println(t.getVal());`
- klopfen ggf. beim Sohn an
  - lass mich rein:
  - `t.getLeft().accept(this)`

```
public class InOrder implements Besucher{
    public void besuche(Operator t){
                t.getLeft().accept(this);
                System.out.println(t.getOp());
                t.getRight().accept(this);
               }

    public void besuche(Variable t){
            System.out.println(t.getName());
        }
    public void besuche(Konstante t){
            System.out.println(t.getVal());
        }
}// Ende der Klasse InOrder
```


© H. Peter Gumm, Philipps-Universität Marburg

Testbesuch

```
static void test(){
   Term t = new Operator('+',new Variable("x"),
                  new Operator('*',new Konstante(3),
                                   new Konstante(4)));
   System.out.println("PreOrder");
   t.accept(new Preorder());
   System.out.println("InOrder");
   t.accept(new InOrder());
 }
```



- Vorteile des Visitor Patterns
  - einfach, neue Besucher hinzuzufügen
    - besuchte Klasse muss nicht angefasst werden
    - kann compiliert vorliegen
  - wenn neu zu besuchende Klassen hinzukommen
    - nur Besucher updaten
    - notfalls

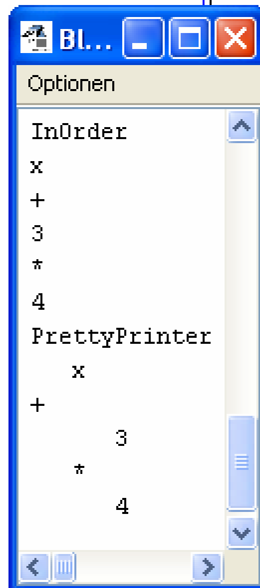
```
void visit(NewClass n) { }
```


© H. Peter Gumm, Philipps-Universität Marburg

PrettyPrinter

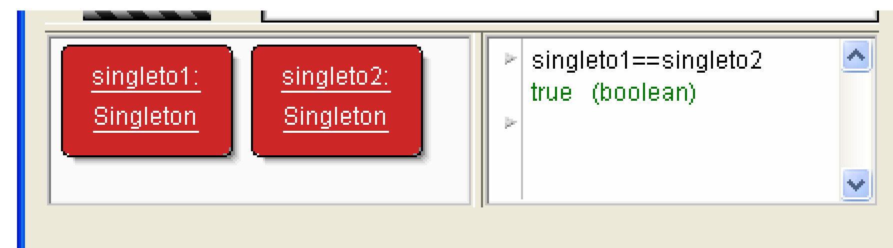
- Weitere mögliche Besucher
  - PrettyPrinter
  - Interpreter
  - Compiler
  - Tiefenmesser
  - …

```
public class PrettyPrinter implements Besucher{

    private int tiefe = 0;

    private void spaces(int n){
        for(int i=0; i<n; i++) System.out.print(" ");
    }
    public void besuche(Operator t){
                tiefe += 3;
                   t.getLeft().accept(this);
                tiefe -= 3;
                spaces(tiefe); System.out.println(t.getOp());
                tiefe += 3;
                  t.getRight().accept(this);
                tiefe -= 3;
               }
    public void besuche(Variable t){
            spaces(tiefe); System.out.println(t.getName());
        }
    public void besuche(Konstante t){
            spaces(tiefe); System.out.println(t.getVal());
        }
}// Ende der Klasse PrettyPrinter
```



Das Singleton Muster

- Aufgabe:
  - Sorge dafür, dass eine Klasse genau eine Instanz erzeugen kann
- Anwendung
  - Eine Hauptklasse einer Anwendung
    - Das Objekt repräsentiert die Anwendung
- Muster: *Singleton*
  - Mathematisch: Singleton = Menge mit genau einem Element



© H. Peter Gumm, Philipps-Universität Marburg

Singleton Lösung

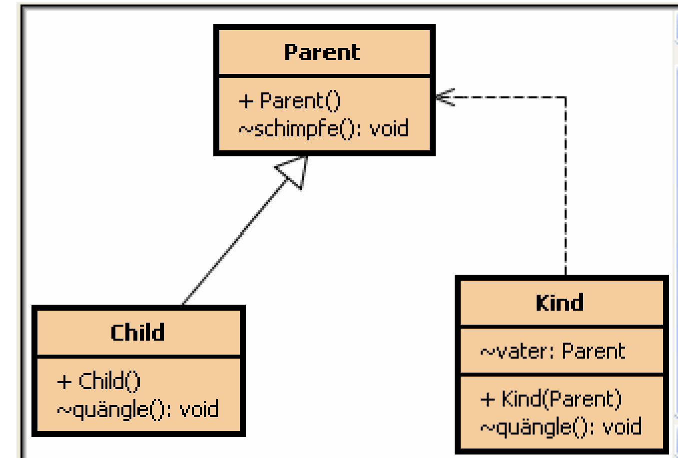
Singleton
+ getInstanz(): Singleton

- Konstruktor *private* erklären
- statische Variable vom Typ *Singleton* erhält die Instanz
  - das passiert beim Laden der Klasse
- statische Methode getInstanz() ersetzt *new*
- liefert immer wieder dieselbe Instanz

```
public class Singleton {

  private static Singleton instanz
                        = new Singleton();
  private Singleton(){
      // whatever
    }
  public static Singleton getInstanz(){
      return instanz;
  }
}// Ende der Klasse Singleton
```


© H. Peter Gumm, Philipps-Universität Marburg

Komposition *



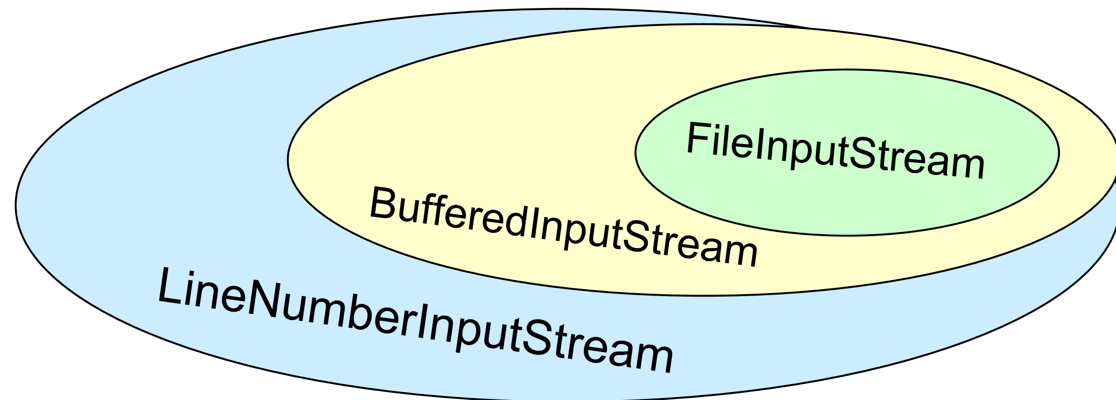
- Erweiterung der Funktionalität einer Klasse
  - Neue Felder, neue Methoden
- Normalerweise:
  - Klasse erweitern
    - `class Child extends Parent{ … }`
    - is-a Beziehung
    - erbt vom Vater: `schimpfe()`
- Stattdessen
  - Neue Klasse mit Referenz auf „Ober"klasse
    - `class Kind{ Parent vater; …. }`
    - has-a Beziehung
    - kann auch: `vater.schimpfe()`
  - Statt `new Kind()[ super() ; … }`
    - jetzt `new Kind(new Parent());`
    - Beispiel:
      `new BufferedReader(new FileReader("text.txt"));`

```
public class Kind {
  Parent vater;
  public Kind(Parent vater){
        this.vater = vater;
   }
  void quängle(){
   System.out.print("Bääääh");
}
```

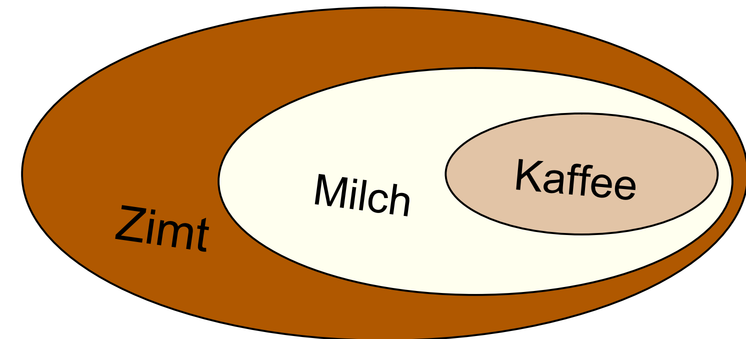
* Diese Folie beschreibt das Konzept der *Komposition*, nicht das *Composite Pattern*

© H. Peter Gumm, Philipps-Universität Marburg

Decorator



- Aufgabe
  - füge einem Objekt dynamisch neue Funktionalität hinzu
  - Kombiniere Funktionalitäten
- Anwendungen
  - FileInputStream in Java-Api
  - Zusätzliche Funktionalität
    - BufferedInputStream
    - LineNumberInputStream
  - Typische Kombination

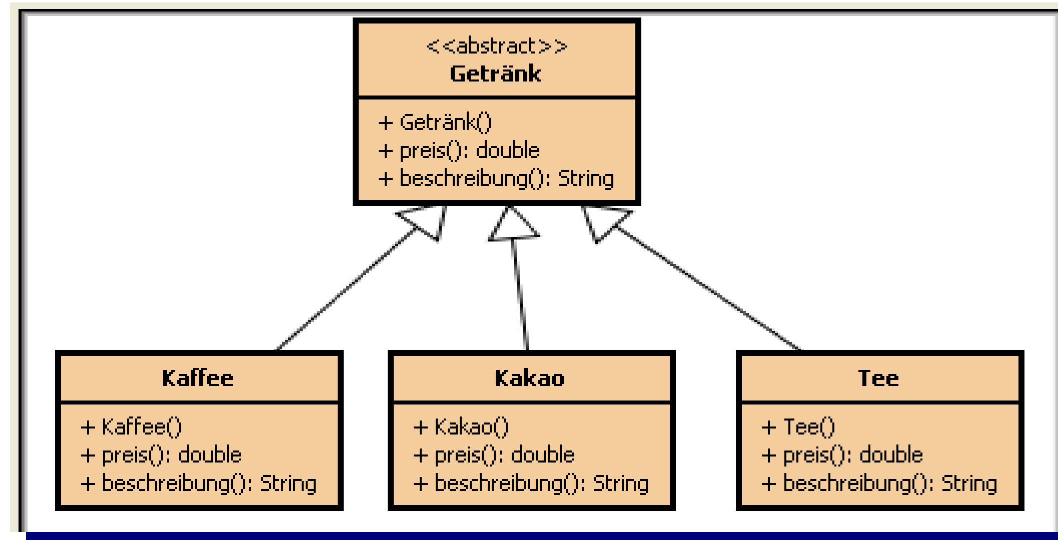



```
InputStream in = new LineNumberInputStream(
                     new BufferedInputStream(
                          new FileInputStream(„Text.txt“)));
```


© H. Peter Gumm, Philipps-Universität Marburg

The Coffee & Tea Company*

- Getränke
  - Kaffee
  - Tee
  - Kakao
- Modifikation
  - Milch
  - Sahne
  - Soja
  - Zimt
- Kombinationen
  - Kaffee mit Milch und Zimt
  - Kaffee mit Zimt und doppelter Sahne
  - Tee mit Milch und Zimt …



Nichts besonderes, aber wie bringt man die Modifikationen hinein ?

Kaffee mit Zimt und doppelter Sahne

Vererbung würde zu unglaublich vielen Klassen führen

Idee aus: *Head First Design Patterns*. O'Reilly 2006.
http://www.oreilly.de/catalog/hfdesignpat/chapter/ch03.pdf

© H. Peter Gumm, Philipps-Universität Marburg

Implementierung

- Decorator ist ein Getränk
  - Vererbung
- Decorator hat ein Getränk
  - Komposition
  - das Getränk, das dekoriert wird
  - kann selber ein dekoriertes Getränk sein
- Alternativen für Vererbung von
  - preis
  - beschreibung

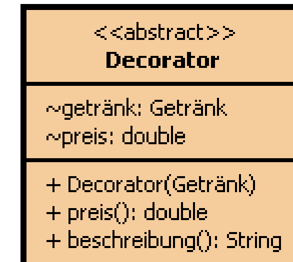
```
public abstract class Decorator extends Getränk{
    Getränk getränk;
    double preis;

    public Decorator(Getränk g){
        getränk = g;
    }
    public String beschreibung(){
        return getränk.beschreibung();
    }
    public double preis(){
      return getränk.preis() + preis;
    }
}// Ende der K
```

<<abstract>> Decorator
~getränk: Getränk ~preis: double
+ Decorator(Getränk) + preis(): double + beschreibung(): String

```
public class Milch extends Decorator{

   public String beschreibung(){
        return super.beschreibung()
              + " mit Milch ";
   }
   public Milch(Getränk g){
        super(g);
        preis = .25;
    }
}// Ende der Klasse Milch
```

Milch
+ Milch(Getränk) + beschreibung(): String

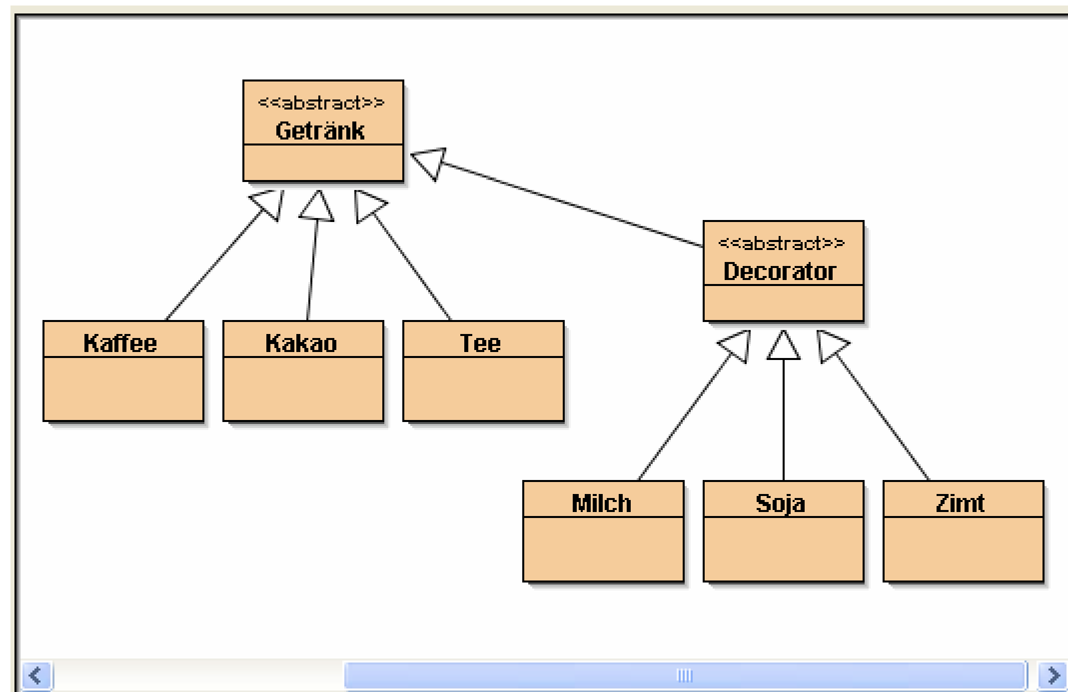
Decorator Pattern



- Decorator ist ein Getränk
  - notwendig für Schachtelung

```
new Zimt(
 new Milch(
  new Milch(
   new Kaffee())));
```

- Decorator hat ein Getränk
  - notwendig für Preispropagation

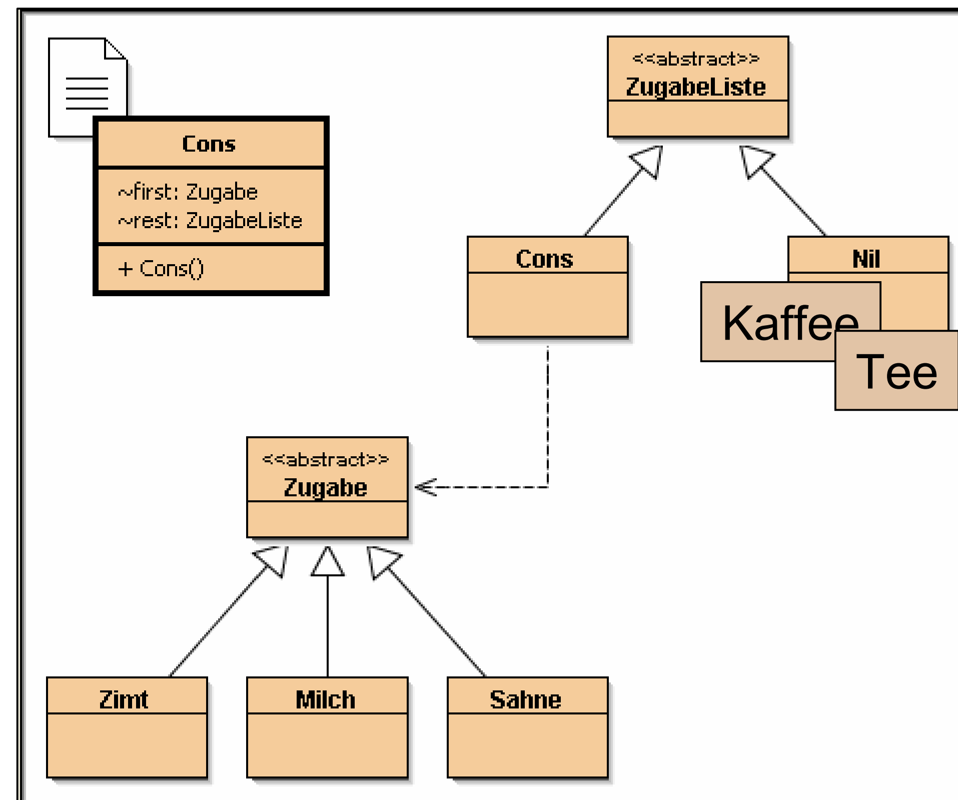



```
public double preis(){
  return getränk.preis() + preis;
}
```


© H. Peter Gumm, Philipps-Universität Marburg

Erinnerung: Listen

- Listen von Zugaben
- Nil ist Listenterminator
  - Trägt keine Information
  - Warum eigentlich nicht ?
- Idee (nächste Folie):
  - Ersetze Nil durch anderes Objekt
  - Statt
    [Sahne, Zimt, Sahne : Nil]
  - Getränk-terminiert:
    - [Sahne, Zimt, Sahne : Kaffee]
    - [Sahne, Zimt, Sahne : Tee]
    - [Sahne, Zimt, Sahne : Kakao]

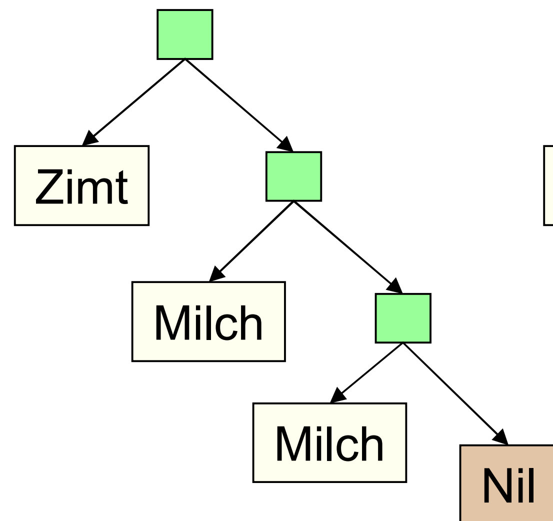


© H. Peter Gumm, Philipps-Universität Marburg

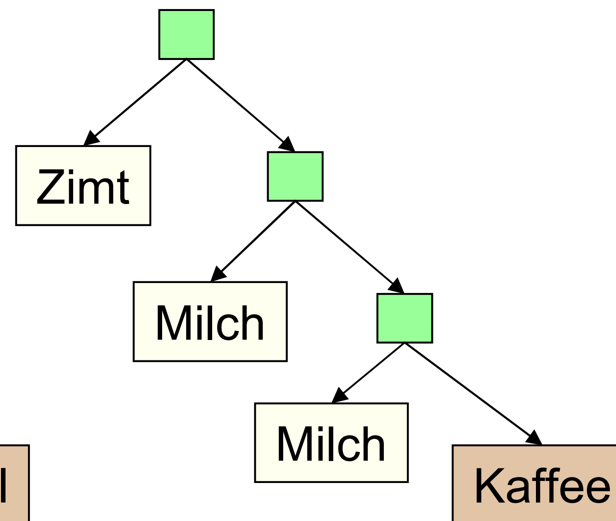
Objektterminierte Listen

- Decorierte Objekte sind eigentlich Objektterminierte Listen von Decorators
  - Normalerweise terminiert man Listen mit Nil
  - Man kann sie aber mit jedem anderen Objekt terminieren
  - oder man kann das terminierende Objekt variieren

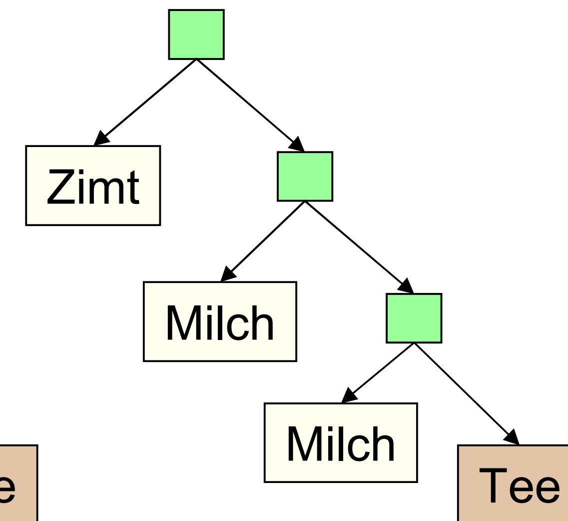
[Zimt, Milch, Milch]



[Zimt, Milch, Milch : Kaffee]



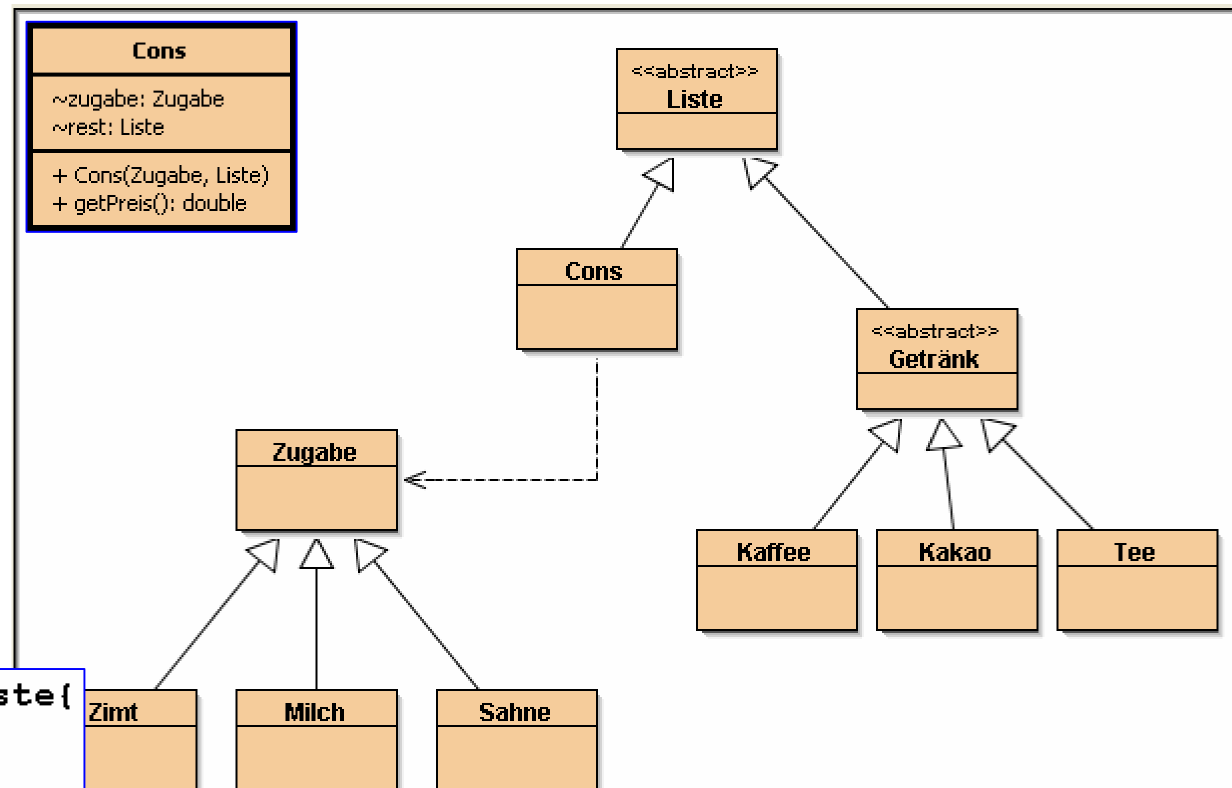
[Zimt, Milch, Milch : Tee]



© H. Peter Gumm, Philipps-Universität Marburg

Objektterminierte Listen statt Decorator

- Implementierung standard
  - jede Klasse hat getPreis()
  - In Cons werden die Preise addiert
- Einziger Nachteil
  - Konstruktorenaufruf innerhalb Cons



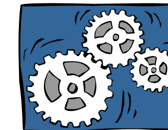

```java
public class Cons extends Liste{
  Zugabe zugabe;
  Liste rest;

  public double getPreis(){
   return zugabe.getPreis()
     + rest.getPreis();
  }
```

```java
Liste l = new Cons(new Zimt(),
           new Cons(new Milch(),
            new Cons(new Milch(),
                 new Kaffee())));
```

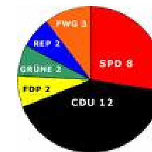

© H. Peter Gumm, Philipps-Universität Marburg

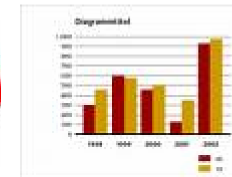
Model-View-Controller

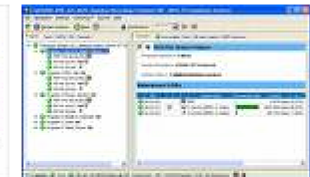
- MVC-Pattern *
  - Modell
    - Verarbeitung der Daten und Prozesse



  - View
    - Sicht der Benutzer
    - Ausgabe, Präsentation
    - ein Modell kann viele Views haben







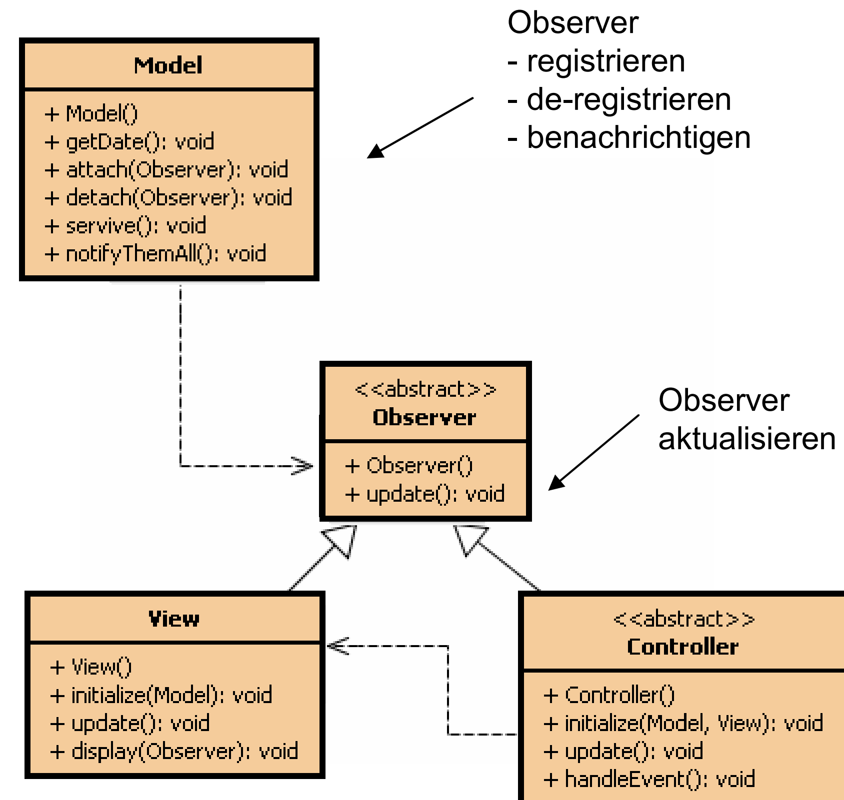
  - Controller
    - Eingabe,
    - Kontrolle, Einflussnahme

* Das MVC-Pattern* stammt wie viele gute Ideen im Zusammenhang mit Java von SmallTalk 80 (das war vor 26 Jahren) Fazit: Auch in der Informatik kann es lange dauern, bis sich gute Ideen durchsetzen.

© H. Peter Gumm, Philipps-Universität Marburg

Model-View-Controller Klassen

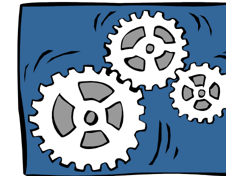
- Model
  - arbeitet autonom
  - unabhängig von Views
- Observer verwaltet
  - Sichten
  - Kontrolle



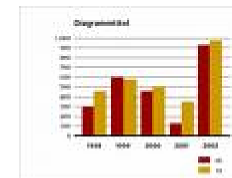
© H. Peter Gumm, Philipps-Universität Marburg

Das Observer Pattern

- Trennung von Modell und Sichten
  - Lose Kopplung
  - Dynamisches de-/registrieren neuer Views



- Beispiel
  - Modell = Datenbank
  - Sichten: Balkendiagramm, Mittelwert, ..

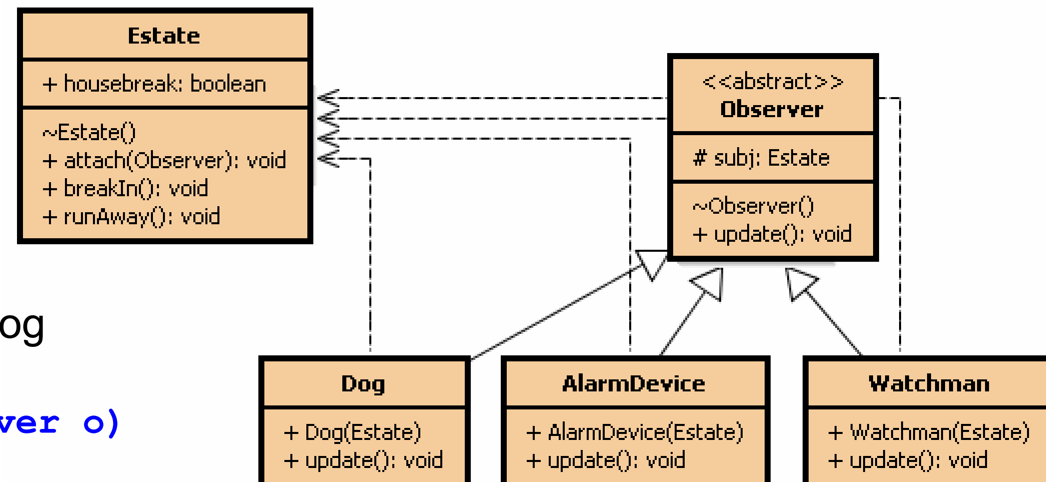


- Methode
  - Sichten registrieren sich beim Modell
    - `register()`
  - Wenn sich am Zustand des Modells etwas ändert, werden alle Sichten informiert
    - `notify()`

© H. Peter Gumm, Philipps-Universität Marburg

Beispiel: Bewachung einer Villa

- Villa kann Zustand ändern
  - jemand bricht ein
    - `breakIn()`
  - Gefahr vorüber
    - `runAway()`
- Observer
  - Watchman, AlarmDevice, Dog
  - registrieren sich in derVilla
    - `attachObserver(Observer o)`
- Bei jeder Zustandsänderung
  - alle Observer benachrichtigen
    - `notifyObservers()`
  - diese reagieren entsprechend
    - `update()`



© H. Peter Gumm, Philipps-Universität Marburg

Die Observer

```
  Every observer has an Estate to look over and an
  update function, which is called in case the state
  of the estate changes.
*/
abstract class Observer {
   protected Estate subj;
   public abstract void update();
}
```

- Konstruktoren
  - registrieren Observer mit zu bewachendem Objekt
- **update()**
  - beliebig
  - je nach Rolle des Observers

```
// The dog is one observer looking after the estate.
class Dog extends Observer {
    // At construction time it stores the estate and
    // notifies the estate of its presence.
    public Dog(Estate e) {
      System.out.println("Cave Canem");
      subj = e;
      subj.attach(this);
    }
    // in case of a statechange it looks whether there
    // was a housebreak or not - and reacts to it.
    public void update() {
      if (subj.housebreak)
        System.out.println("bow-wow! grrrr! wow-wow wuff!");
      else
        System.out.println("dog wags its tail");
    }
}
```

Die Villa

- Verwaltet eine Liste von Observern
  - attach(Observer o)
- Das Haus ändert Zustand
  - breakIn()
  - runAway()
- und benachrichtigt jeweils die Observer
  - notifyObservers()

```
class Estate {
    private Observer[] observers = new Observer[9];
    private int observerCount = 0;
    public boolean housebreak;
    // add a new observer
    public void attach(Observer o) {
      if (observerCount == 8) {
        System.out.println("The estate is already well pr
      } else {
        observers[observerCount] = o;
        observerCount++;
      }
    }
    // Call update-method of all attached observers.
    private void notifyObservers() {
      for (int i = 0; i < observerCount; i++) {
        observers[i].update();
      }
    }
    // changes the state - observers are notified
    public void breakIn() {
      System.out.println("Somenone tries to break in!");
      housebreak = true;
      notifyObservers();
```


© H. Peter Gumm, Philipps-Universität Marburg

TestProgramm

```
public static void main( String[] args ) {
    Estate myVilla = new Estate();
    System.out.println("I'm a little paranoid,"
                + " so I'll protect my Villa!");
    new Watchman(myVilla);
    System.out.println();
    myVilla.breakIn();
    System.out.println();
    myVilla.runAway();
    System.out.println();
    new Dog(myVilla);
    new AlarmDevice(myVilla);
    myVilla.breakIn();
    System.out.println();
    myVilla.runAway();
}
```

BlueJ: Konsole - Observer

Optionen

```
I'm a little paranoid, so I'll protect my Villa!
Watchman Walter watching you're estate!

Somenone tries to break in!
Stop or I'll shoot!

The burglar runs away...
Let's get some donuts.

Cave Canem
The DATE System protects your estate!
Somenone tries to break in!
Stop or I'll shoot!
bow-wow! grrrr! wow-wow wuff!
DRRRRRRRRRRRRRRRRRRRRRRIIIIIIIIIIIIIIIIIIIIIING!

The burglar runs away...
Let's get some donuts.
dog wags its tail
```


© H. Peter Gumm, Philipps-Universität Marburg