▲ Hochschule Harz
Hochschule für angewandte Wissenschaften

# Wahlpflichtfach Design Pattern


- Dipl.-Inf., Dipl.-Ing. (FH) Michael Wilhelm
- Hochschule Harz
- FB Automatisierung und Informatik
- miwilhelm@hs-harz.de
- http://www.miwilhelm.de
- Raum 2.202
- Tel. 03943 / 659 338

# Inhalt

# Entwurfsmuster Dekorator

- **Absicht**
  - Der „Dekorator“ stattet ein Objekt dynamisch mit zusätzlichen Eigenschaften aus. Ein Dekorator bietet eine flexible Alternative zu Subklassen.
- **Motivation**
  - Des Öfteren möchte man ein Objekt, und nicht eine ganze Klasse, um einige Eigenschaften erweitern. So sollte es eine grafische Benutzeroberfläche gestatten einem Element individuelle Eigenschaften, wie Rollbalken oder einen Rahmen hinzuzufügen.
  - Dies könnte mittels Vererbung geschehen: Dies platziert etwa einen Rahmen um jede Instanz der entsprechenden Subklasse. Dies ist nicht sehr flexibel, da hier die Wahl des Rahmens statisch vorgenommen wird, d.h. ein Klient kann nicht beeinflussen wie und wann eine Komponente mit einem Rahmen versehen wird.

# Entwurfsmuster Dekorator

- **Motivation (Forts.)**
  - Eine flexiblere Lösung ist die Komponente in ein anderes Objekt einzubetten. Dieses neue Objekt fügt den Rahmen hinzu. Ein derartiges umschließendes Objekt heisst Dekorator. Der Dekorator stellt die Schnittstelle der ursprünglichen Komponente zur Verfügung, daher ist seine Präsenz für die Klienten der Komponente nicht sichtbar.
  - Der Dekorator leitet Anforderungen an die Komponente weiter und führt optional zusätzliche Aufgaben aus (mehr Arbeit).
  - Dekoratoren einer Komponente können beliebig tief verschachtelt werden. So kann einer Textkomponente ein Rollbalken und dem neuen Objekt ein Rahmen hinzugefügt werden.

# Entwurfsmuster Dekorator

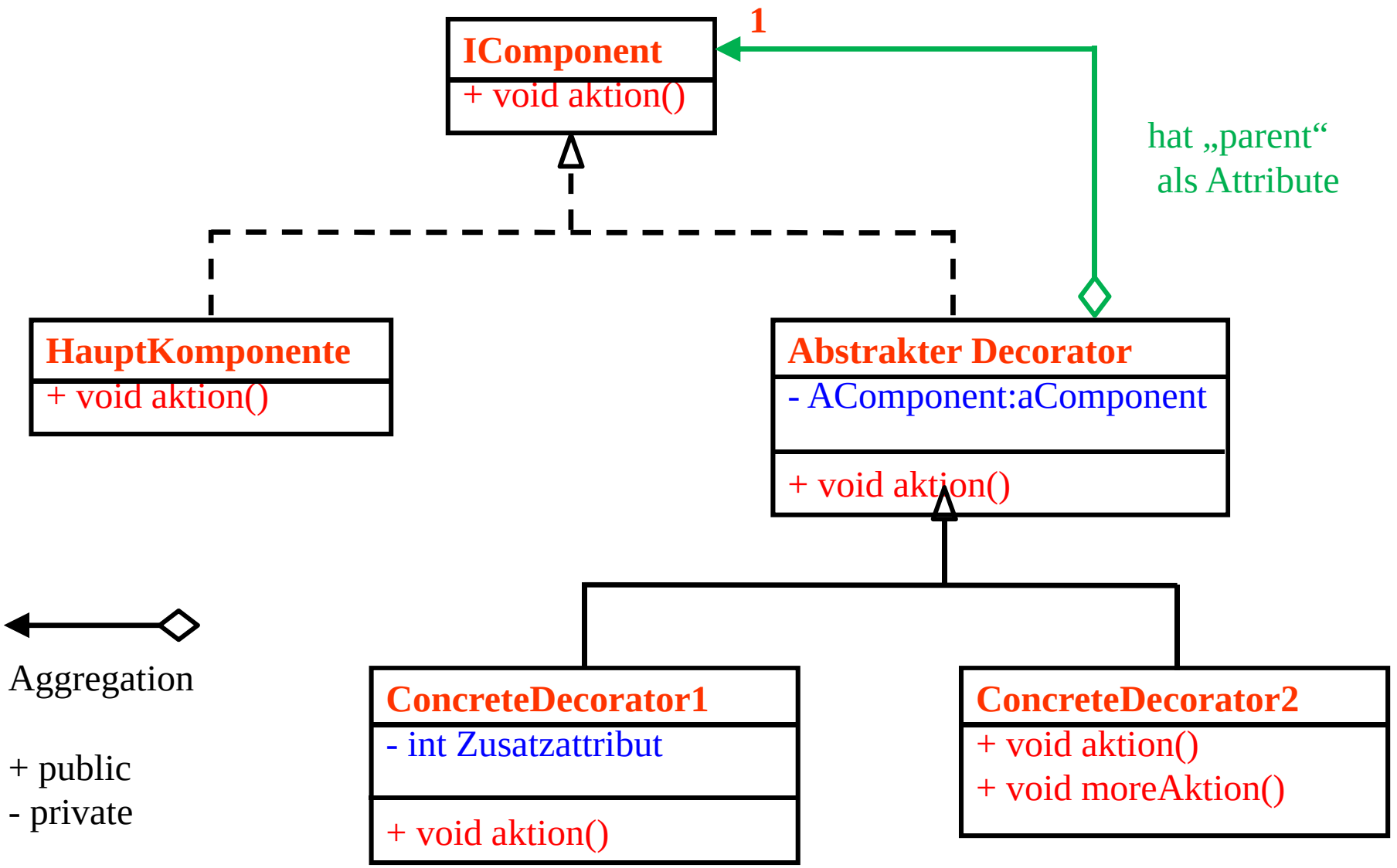
## Anwendbarkeit

- Dynamisches und transparentes Hinzufügen von Verantwortlichkeiten zu einem Objekt, ohne andere Objekte zu beeinflussen.

- Eine zu große Erweiterung durch Subklassen ist inpraktikabel, da diese zeitintensiv sind.

- Eine Klassendefinition kann auch nicht zur Verfügung stehen oder nicht möglich sein (Klasse ist als final deklariert).

# Struktur des Entwurfsmuster Dekorator



# Entwurfsmuster Dekorator

- **Teilnehmer**
  - IComponent
    - definiert die Schnittstelle für Objekte die dynamisch erweitert werden können.
  - ConcreteComponent
    - definiert ein Objekt das für zusätzliche Eigenschaften verantwortlich ist.
  - Decorator
    - unterhält eine **Referenz** auf eine IComponent und definiert eine IComponent entsprechende Schnittstelle.
  - ConcreteDecorator 1/2
    - fügt IComponent neue Eigenschaften hinzu

# Dekorator

- **Beispiel**

```
interface IComponent {
  public void doStuff();
}
class ConcreteComponent implements IComponent {
  public void doStuff() { }
  public void doMoreStuff() { }
}
abstract class Decorator implements IComponent {
  private Component comp;
  public Decorator (Component c) { comp = c ; }
  public void doStuff() { comp.doStuff(); }
}

public class ConcreteDecorator extends Decorator {
  public void doEvenMoreStuff() { }
}
```

# Dekorator

- **Zusammenarbeit**
  - Ein Dekorator leitet Anforderungen an seine IComponent weiter.
  - Zusätzlich kann er andere Operationen vor oder nach einer solchen Weiterleitung ausführen.

# Entwurfsmuster Dekorator

- **Konsequenzen: Vorteile**
  - Größere Flexibilität als Vererbung. Das Dekorator-Muster stellt eine flexiblere Methode einem Objekt zusätzliche Eigenschaften zu geben als (Mehrfach-)Vererbung.
  - Mittels Dekoratoren können Eigenschaften einfach **zur Laufzeit** hinzugefügt und auch wieder zurückgenommen werden.
  - Im Gegensatz erfordert Vererbung die Erzeugung einer neuen Klasse für jede neue unabhängige Eigenschaft. Dies führt zu vielen Klassen und entsprechender Komplexität.
  - Durch verschiedene Dekoratoren für eine Komponente können mehrere Eigenschaften nach Bedarf hinzugefügt werden. Zudem erlauben es Dekoratoren, im Gegensatz zur Vererbung, einfach eine Eigenschaft mehrmals hinzuzufügen.

# Entwurfsmuster Dekorator

- **Konsequenzen: Vorteile (Forts.)**
  - Vermeidet mit Eigenschaften überladene Klassen (fat classes) nahe der Hierarchiewurzel. **Statt alle möglichen Eigenschaften in einer komplexen, anpassbaren Klasse vorherzusehen**, wird eine einfache Klasse definiert und diese inkrementell mit Dekoratoren erweitert.
  - Dies erlaubt Funktionalität mit einfachen Einzelbausteinen zu erzeugen. Daher entstehen einer Anwendung keine Kosten für ungenutzte Eigenschaften.
  - Dekoratoren erlauben eine einfache und unabhängige Erweiterung einer Klasse, während neue Subklassen dazu neigen für die neue Eigenschaft unwichtige Einzelheiten offen zu legen.

# Entwurfsmuster Dekorator

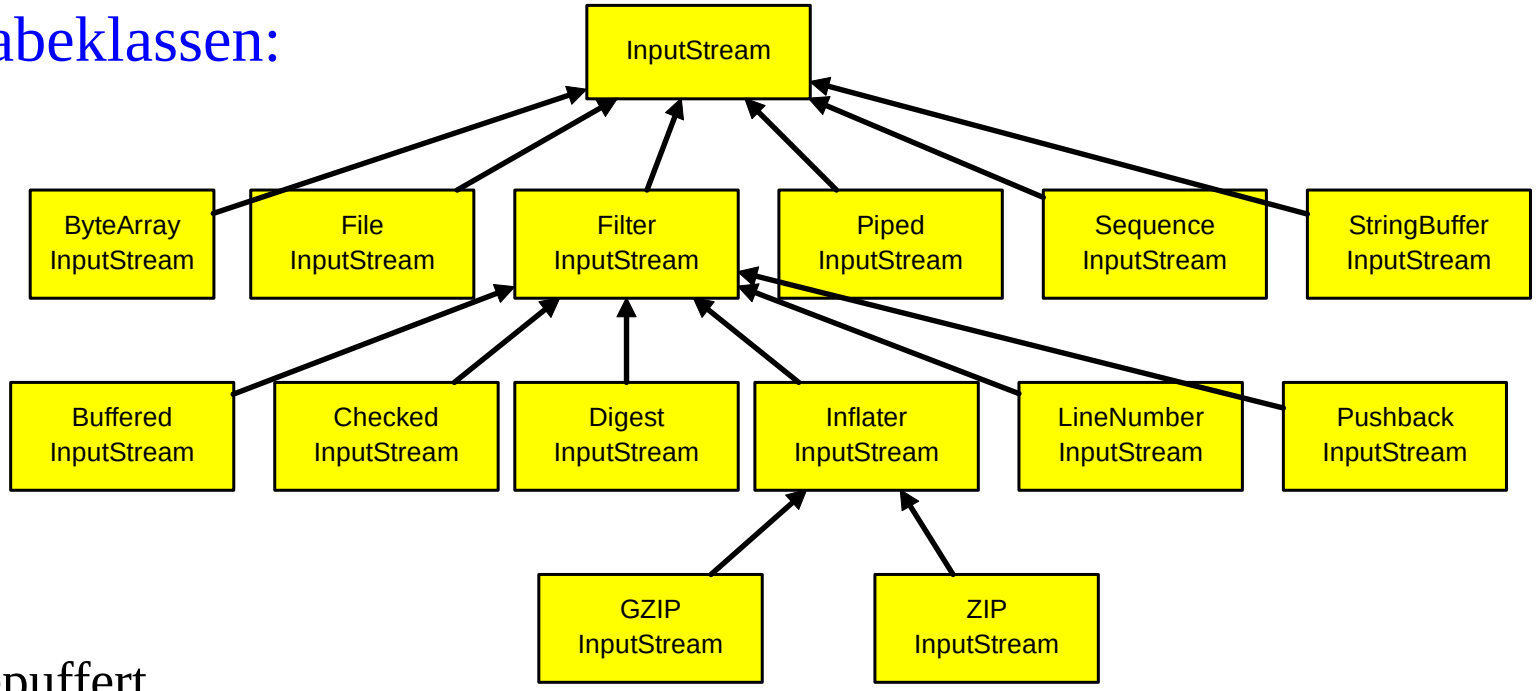
- **Konsequenzen: Nachteile**
  - Ein Dekorator und seine „Component“ sind nicht identisch. Ein Dekorator agiert als transparente Hülle. Vom Objektstandpunkt ist die dekorierte Komponente nicht identisch mit der Komponente. Daher sollten Abfragen nach Objekt-Identität bei der Benutzung von Dekorator vermieden werden.

  - Objektinflation. Ein Dekorator-basiertes Design resultiert oft in Systemen mit vielen, nahezu gleich aussehenden Objekten. Solche Systeme sind für Eingeweihte leicht an unterschiedliche Erfordernisse anzupassen, für andere aber schwer zu lernen und von Fehlern zu befreien.: fout, dout etc.

# Datenverarbeitung in Java

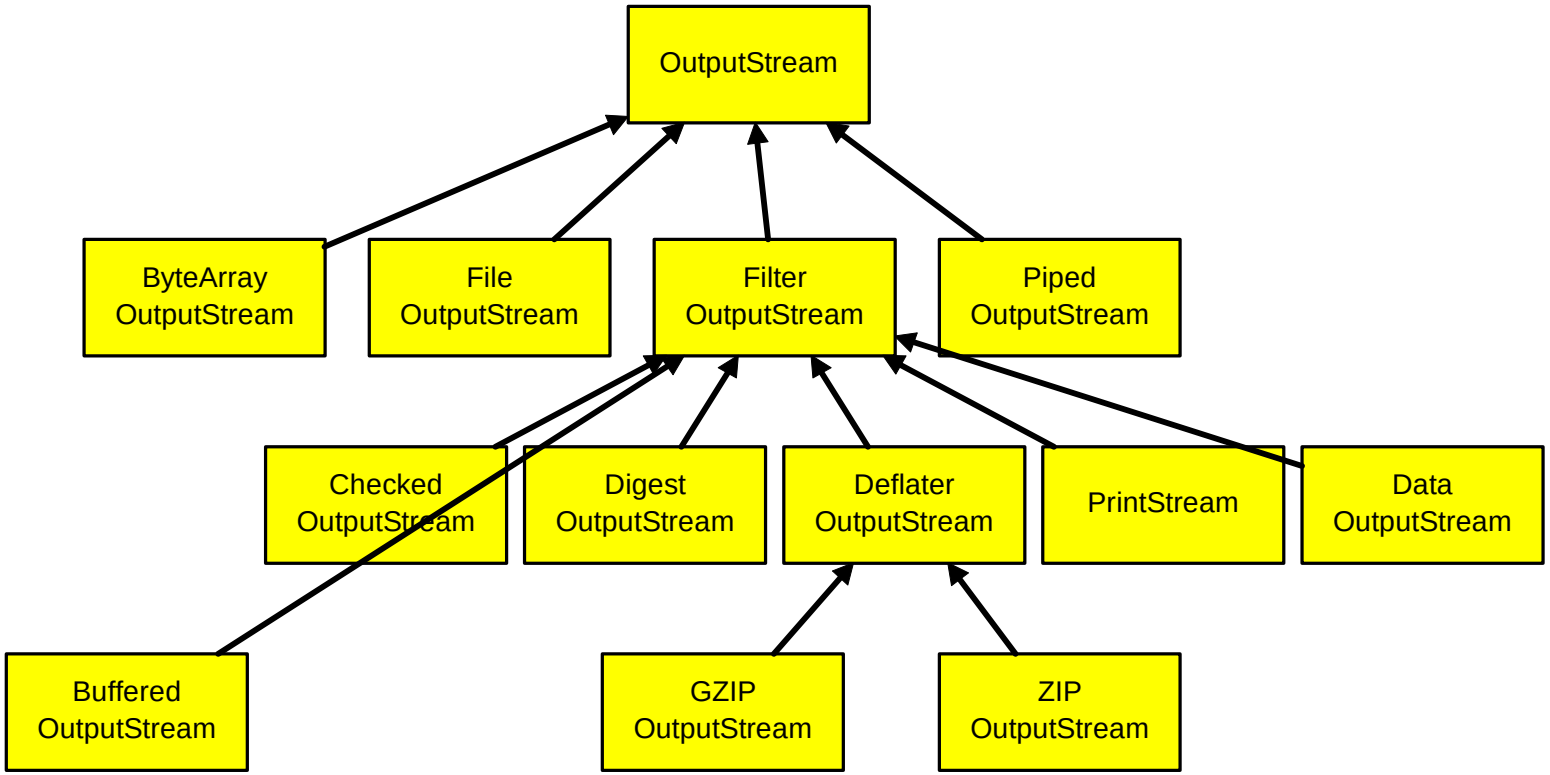
Es steht eine Vielzahl von Klassen/Modulen zur Verfügung (58):

## Eingabeklassen:




- gepuffert
- Filter für Dateinamen
- für Zeichen, Zeichenketten, Objekte, Token
- mit Pipe-Verfahren

# Ausgabe in eine Datei




- gepuffert
- Filter für Dateinamen
- für Zeichen, Zeichenketten, Objekte, Token
- mit Pipe-Verfahren

# Klassen des io-Packages

| | |
|---|---|
| **BufferedInputStream** | Zwischenbuffer |
| **BufferedOutputStream** | Zwischenbuffer |
| **BufferedReader** | Zeilenweise |
| **BufferedWriter** | Zwischenbuffer |
| ByteArrayInputStream | ZwischenArray |
| ByteArrayOutputStream | ZwischenArray |
| CharArrayReader | CharacterStream |
| CharArrayWriter | CharacterStream |
| Console | à la DOS |
| **DataInputStream** | int byte double |
| **DataOutputStream** | int byte double |
| File | abtrakte Klasse |
| FileDescriptor | arbeitet mit handles |
| **FileInputStream** | byte-weise |
| **FileOutputStream** | byte-weise |
| FilePermission | Rechte |
| FileReader | Charakterbasierend |
| FileWriter | Charakterbasierend |

# Klassen des io-Packages

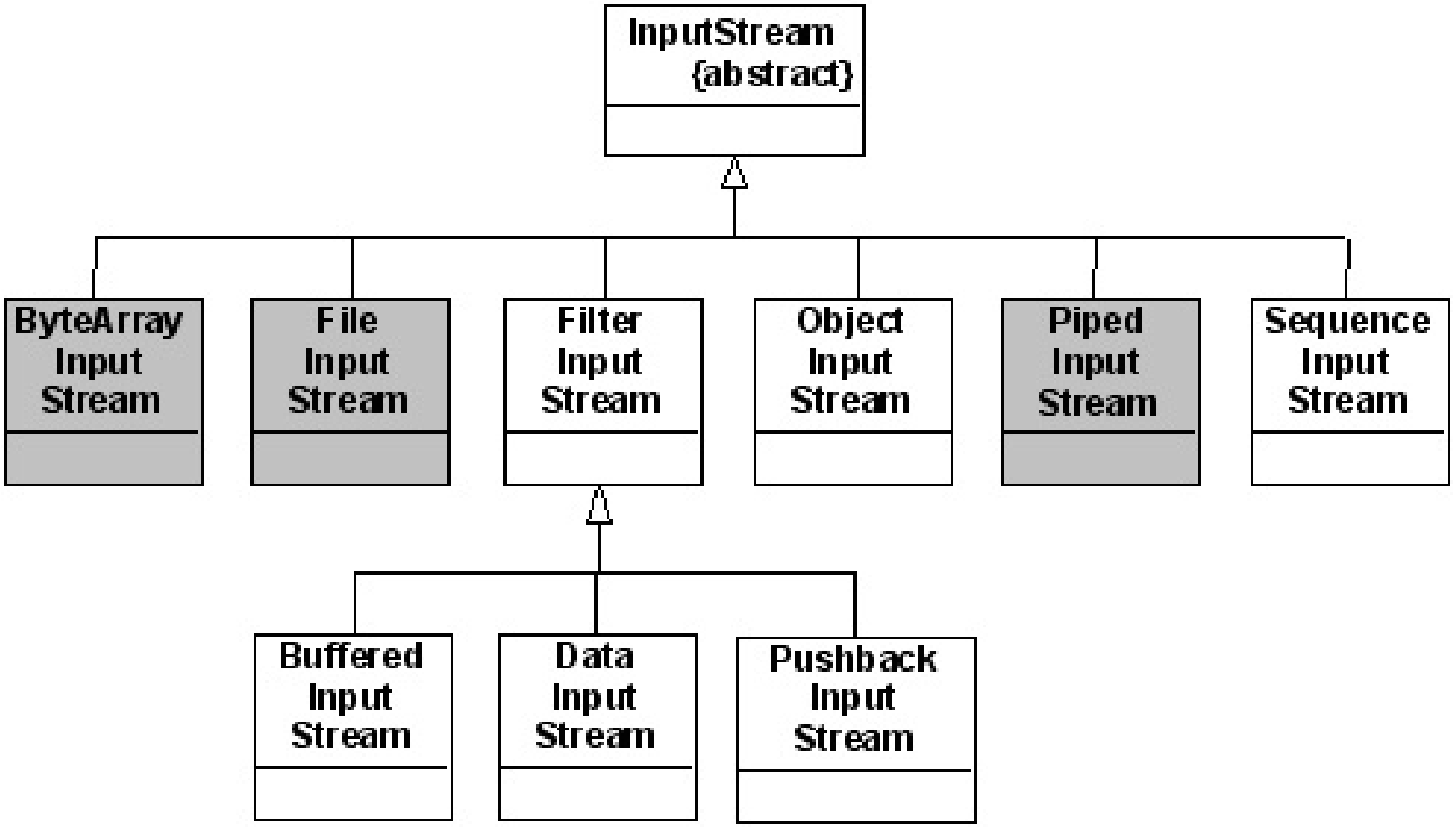
| | |
|---|---|
| FilterInputStream | Transformierung |
| FilterOutputStream | Transformierung |
| FilterReader | Transformierung und Charakterbasierend |
| FilterWriter | Transformierung und Charakterbasierend |
| InputStream | Superclass |
| InputStreamReader | Charakterbasierend |
| **LineNumberInputStream** | Zeilenweise |
| **LineNumberReader** | Zeilenweise |
| **ObjectInputStream** | Serialize |
| **ObjectOutputStream** | Serialize |
| ObjectStreamClass | Serialize |
| ObjectStreamField | Serialize |
| OutputStream | Superclass |
| OutputStreamWriter | Charakterbasierend |
| **PipedInputStream** | \| zwei Kanäle |
| **PipedOutputStream** | \| zwei Kanäle |
| PipedReader | \| zwei Kanäle |
| PipedWriter | \| zwei Kanäle |

# Klassen des io-Packages

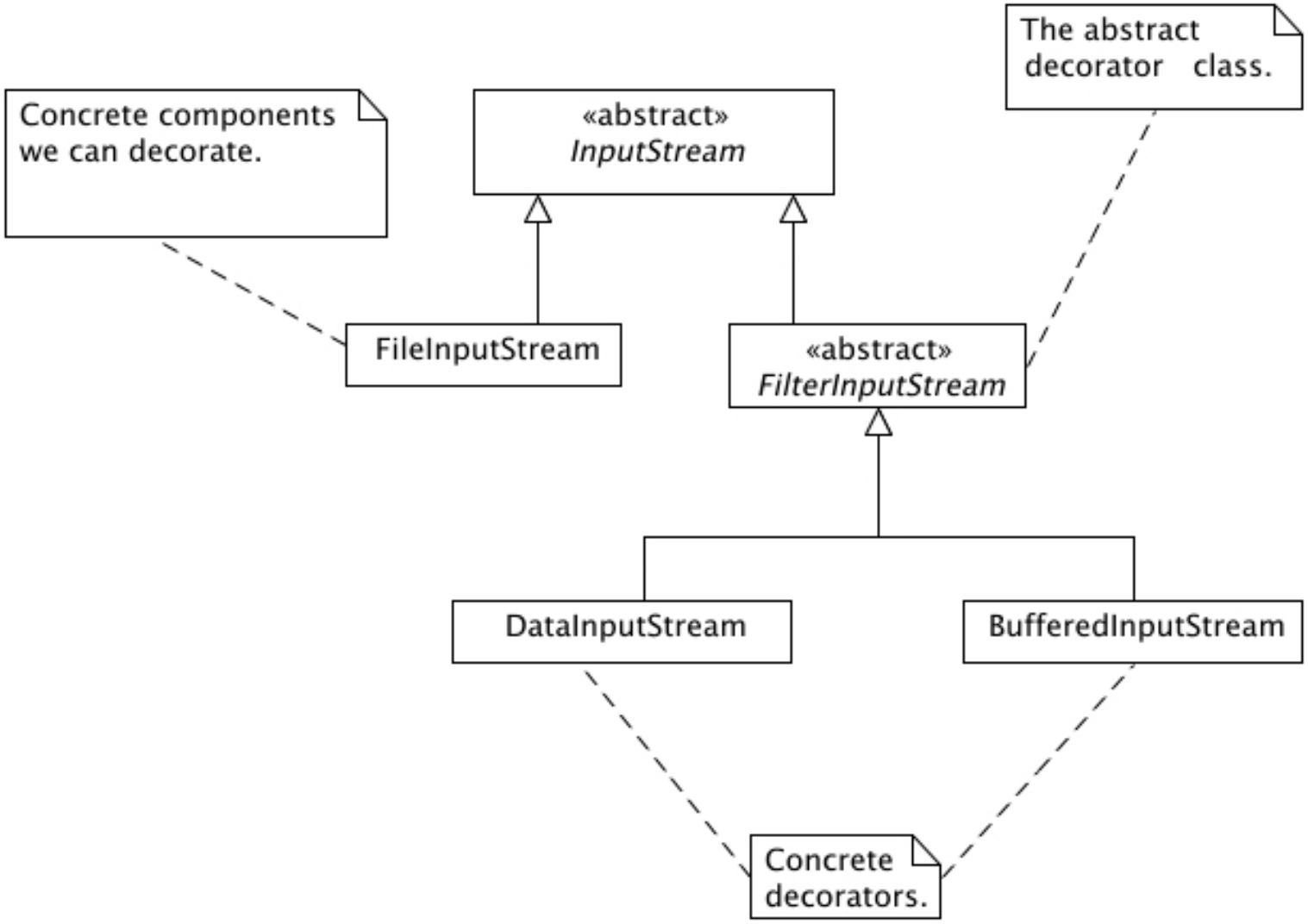
| | |
|---|---|
| **PrintStream** | println, syso |
| PrintWriter | println, syso |
| PushbackInputStream | preview |
| PushbackReader | preview RandomAccessFile seek, HexEditor |
| Reader | abstrakte Klasse |
| SequenceInputStream | liest mehrere Stream hintereinander |
| SerializablePermission | Serialize |
| StreamTokenizer | lesen mit definierten Tokens (Parser) |
| StringBufferInputStream | deprecated |
| StringReader | Liest aus einem String, zeichernweise |
| StringWriter | Schreibt in einem String |
| Writer | abstrakte Klasse |

# Weitere io-Packages: nio und nio2

# Java I/O als Beispiel für einen Dekorator



# Beispiel eines Dekorator-Pattern mit I/O



# Dekoratorbeispiel Pizza (1)

## Abbilden einer Pizza-Bestellung

**Attribute:**
int radius;

**Zutaten:**
Käse
Salami
Huhn
Ei

**Aktion:**
Bestimmen des Gesamtpreises

# Dekoratorbeispiel Pizza (2)

```
class Pizza {
  public int radius;
  public boolean kaese=false;
  public boolean salami=false;
  public boolean chicken=false;

  public Pizza(int r) {
    radius=r;
  }
  public String toString()  {
    return "Radius: "+radius
         +"\n  Kase: "+kaese
         +"\n  Salami: "+salami
         +"\n  Hühnchen: "+chicken+"\n\n";
  }

}  // Pizza
```

**Aufruf:**

```
Pizza p1 = new Pizza(20);
p1.kaese=true;
p1.ei=true;
System.out.println("p1: "+p1);

 Pizza p2 = new Pizza(32);
 p2.kaese=true;
 p2.chicken=true;
 p2.salami=true;
 System.out.println("p2: "+p2);
```

- **Eigenschaft (Kase, Salami…), aber nicht immer eine Aktionen !!**
- **viele Abfragen, bei der Bestimmung des Gesamtpreises**

# Dekoratorbeispiel Pizza (3): Ableiten von Pizza

```
class Pizza {
  public int radius;
  public Pizza(int r) {
    radius=r;
  }
  public String toString()  {
    return "Pizza: Radius: "+radius+"\n";
  }
}  // Pizza
```

```
class PizzaKaese extends Pizza {
  public PizzaKaese(int r) {
    super(r);
  }
  public String toString()  {
    return "PizzaKaese: "+super.toString()+"\n";
  }
}  // PizzaKaese
```

```
class PizzaSalami extends Pizza {
  public PizzaSalami(int r) {
    super(r);
  }
  public String toString()  {
    return "PizzaSalami: "+super.toString()+"\n";
  }
}  // PizzaSalami
```

**jetzt Aktionen !!**

**Aber man darf nur eine Zutat auswählen**

# Dekoratorbeispiel Pizza (4): Verschachtelte Ableitung

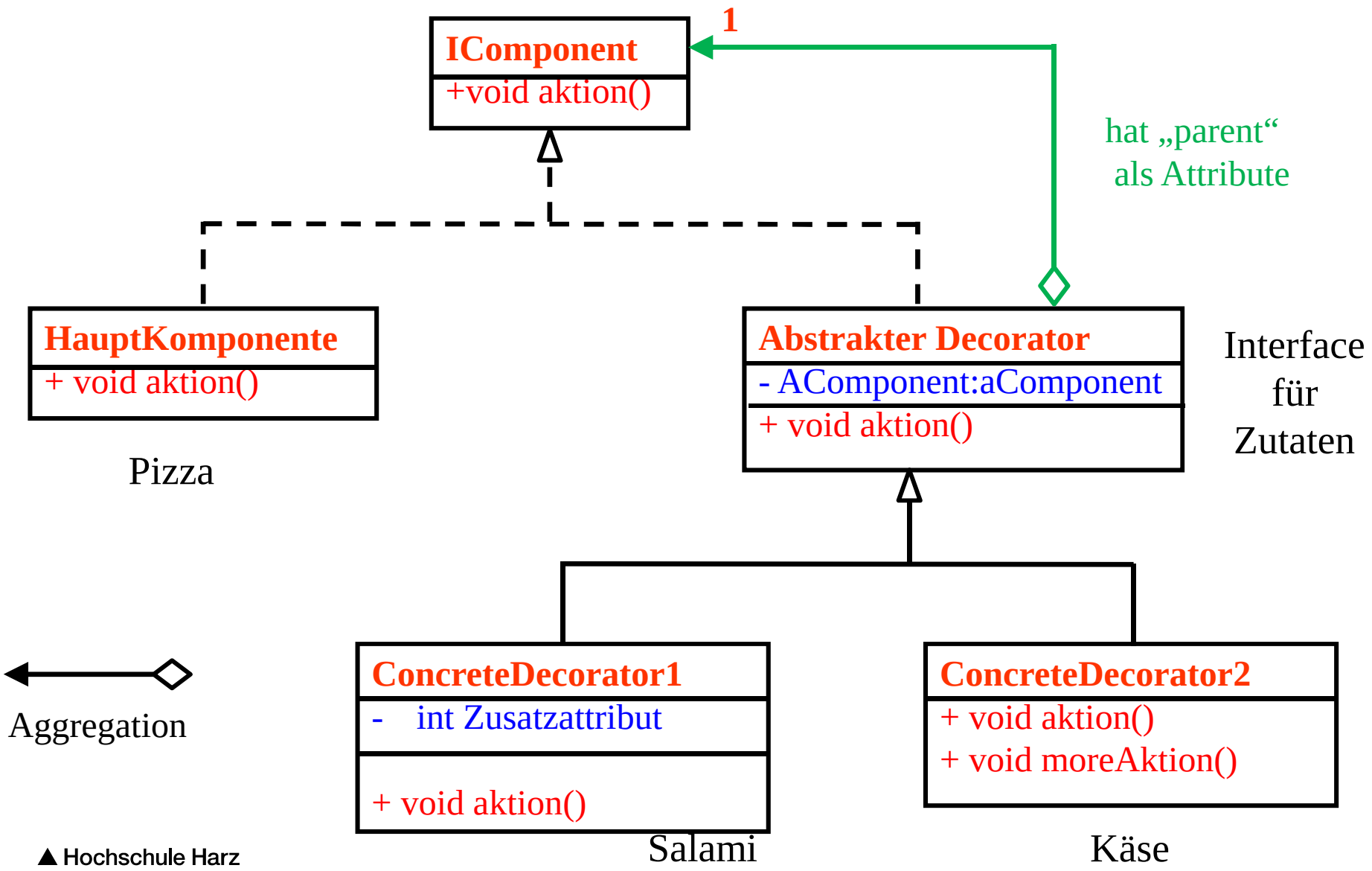
```
class Pizza {
  public int radius;
  public Pizza(int r) {
    radius=r;
  }
  public String toString()  {
    return "Pizza: Radius: "+radius+"\n";
  }
}  // Pizza
```

```
class PizzaKaese extends Pizza {
  public PizzaKaese(int r) {
    super(r);
  }
  public String toString()  {
    return "PizzaKaese: "+super.toString()+"\n";
  }
}  // PizzaKaese
```

```
class PizzaSalami extends PizzaKaese {
  public PizzaSalami(int r) {
    super(r);
  }
  public String toString()  {
    return "PizzaSalami: "+super.toString()+"\n";
  }
}  // PizzaSalami
```

- **jetzt Aktionen !!**
- **Man hat mehrere Zutaten**
- **Alle sind an oder aus (Bool‘sche Variable)**

# Struktur des Entwurfsmuster Dekorator



# Dekoratorbeispiel Pizza (5): Interface IPizza

```
private void   bnTest1_click() {
    Pizza p1 = new Pizza(20);
    System.out.println("p1: "+p1.backen());
    System.out.println("  Preis: "+p1.getPreis() );

    PizzaKaese p2 = new PizzaKaese(p1);
    System.out.println("p2: "+p2.backen());
    System.out.println("  Preis: "+p2.getPreis() );
```

```
interface IPizzabacken {
  public String backen();
  public double getPreis();
}
```

**Ergebnis:**

```
p1: Pizza: Radius: 20
  Preis: 6.78

p2: Pizza: Radius: 20
  PizzaKaese:
  Preis: 8.120000000000001
```

# Dekoratorbeispiel Pizza (5): Interface IPizza

**Test1:**

**p1: Pizza: Radius: 20**
Preis: 6,78

**p2: Pizza: Radius: 20**
PizzaKaese:
Preis: 8,12

**p3: Pizza: Radius: 20**
PizzaKaese:
PizzaSalami:
Preis: 10,78

**p4: Pizza: Radius: 20**
PizzaKaese:
PizzaSalami:
PizzaChicken:
Preis: 13,23

**Preise:**

| | |
|---|---|
| Pizza: | 6,78; |
| Nudeln: | 7.80; |
| Käse: | + 1.50; |
| Salami: | + 2.50; |
| Schinken | + 2.75; |
| Tomaten: | + 0.56; |

```
import java.text.*;
DecimalFormat df;
df  =   new DecimalFormat  ( ",##0.00" );
Syso("  Preis: "+df.format(b3.getPreis()) + " Euro");
```

# Dekoratorbeispiel Pizza (6): Interface IPizza

```
class Pizza implements IPizzabacken {
  public int radius;
  public Pizza(int r) {
    radius=r;
  }
  public String toString()  {
    return "Pizza: Radius: "+radius+"\n";
  }
  public String backen() {
   return toString();
  }
  public double getPreis() {
    return 6.78;
  }
}  // Pizza
```

```
class PizzaKaese implements IPizzabacken {
  private IPizzabacken p;
  public PizzaKaese(IPizzabacken p) {
    this.p = p;
  }
  public String toString()  {
    return "PizzaKaese: "+"\n";
  }
  public String backen() {
   return p.backen()+"   "+toString();
  }
  public double getPreis() {
    return 1.34+p.getPreis();
  }
}  // PizzaKaese
```

# Dekoratorbeispiel Pizza (7): IPizza, Beilage

```
interface IPizzahaus {
  public String backen();
  public double getPreis();
}



// vereinfacht die Erstellung der Beilagen
abstract class Beilage implements IPizzahaus {
    protected IPizzahaus p;

    public Beilage(IPizzahaus p) {
        this.p=p;
    }
}
```

# Dekoratorbeispiel Pizza (8): IPizza, Beilage

```
// Hauptgerichte
class Pizza implements IPizzahaus {
  public int radius;
  public Pizza(int r) {
    radius=r;
  }
  public String toString()  {
    return "Pizza: Radius: "+radius+"\n";
  }
  public String backen() {
   return toString();
  }
  public double getPreis() {
    return 4.80;
  }
}  // Pizza
```

```
// Hauptgerichte
class Nudeln implements IPizzahaus {
  public double menge;
  public Nudeln(double menge) {
    this.menge = menge;
  }
  public String toString()  {
    return "Nudeln: Menge: "+menge+"\n";
  }
  public String backen() {
   return toString();
  }
  public double getPreis() {
    return 7.80;
  }
}  // Nudeln
```

# Dekoratorbeispiel Pizza (9): IPizza, Beilage

```
abstract class Beilage implements IPizzahaus {
   protected IPizzahaus p;

   public Beilage(IPizzahaus p) {
      this.p=p;
   }
}
```

```
class Kaese extends Beilage  {
  public Kaese(IPizzahaus p) {
     super(p);
  }
  public String toString()  {
    return "Zusatz Kaese: "+"\n";
  }
  public String backen() {
   return p.backen()+"  "+toString();
  }
  public double getPreis() {
    return p.getPreis() + 1.50;
  }
} // Kaese
```

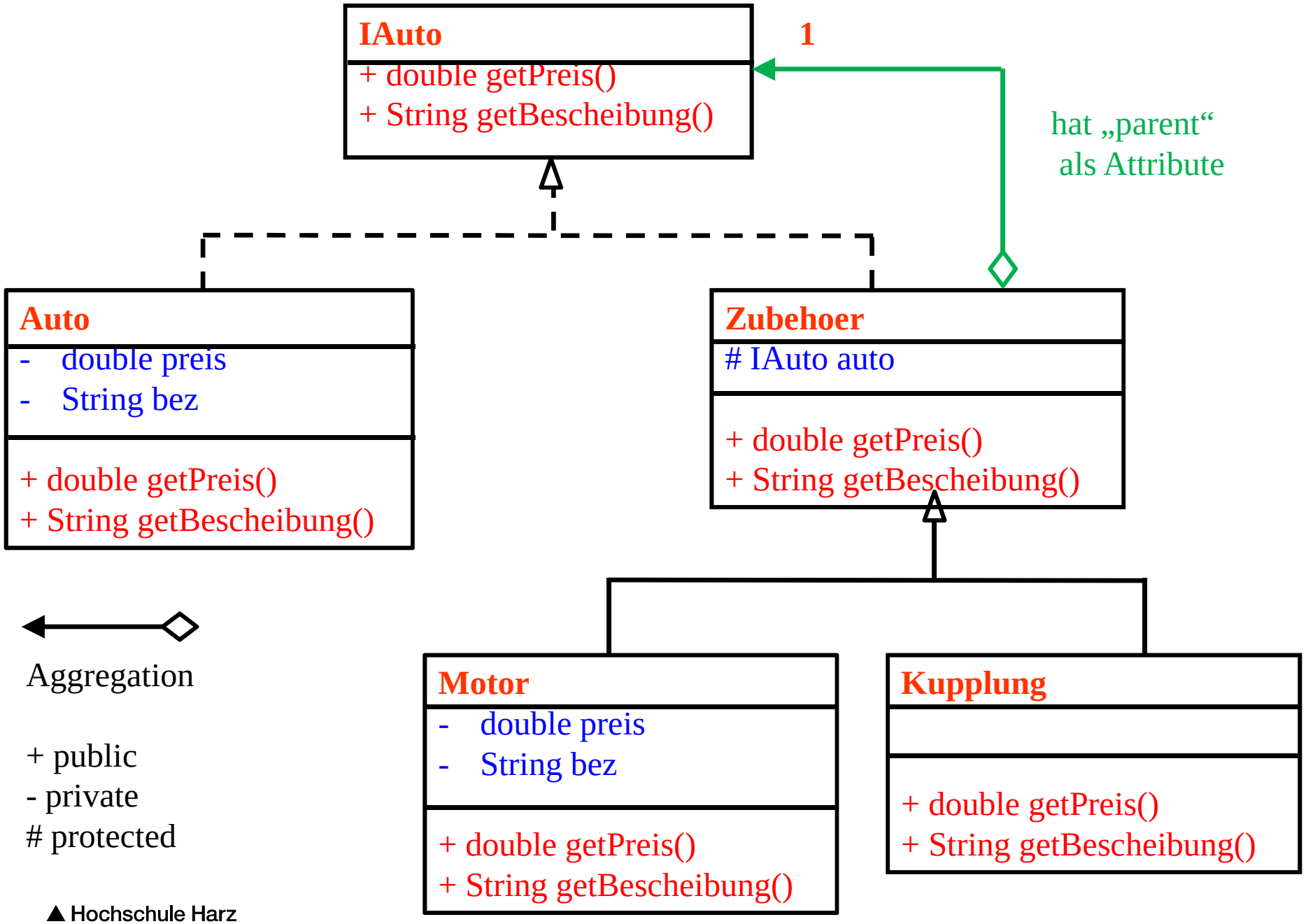
# Dekoratorbeispiel Pizza (5): Interface IPizza

**Test1:**
mit 1. Beilage: Pizza: Radius: 15
--- Zusatz Kaese:
fuer 6,30 Euro
Test2:
**mit 1. Beilage: Pizza: Radius: 18**
--- Zusatz Kaese:
**mit 2. Beilage: Pizza: Radius: 18**
--- Zusatz Kaese:
Zusatz Salami:
fuer 9,05 Euro
**mit 3. Beilage: Pizza: Radius: 18**
--- Zusatz Kaese:
Zusatz Salami:
Zusatz Salami:
fuer 9,61 Euro

**Preise:**

| | |
|---|---|
| Pizza: | 4.80; |
| Nudeln: | 7.80; |
| Käse: | + 1.50; |
| Salami: | + 2.50; |
| Schinken | + 2.75; |
| Tomaten: | + 0.56; |

```
import java.text.*;
DecimalFormat df;
df  =   new DecimalFormat  ( ",##0.00" );
Syso("  Preis: "+df.format(b3.getPreis()) + " Euro");
```


IAuto
+ double getPreis()
+ String getBescheibung()
1
hat „parent“
als Attribute
Auto
- double preis
- String bez
+ double getPreis()
+ String getBescheibung()
Zubehoer
# IAuto auto
+ double getPreis()
+ String getBescheibung()
Aggregation
+ public
- private
# protected
Motor
- double preis
- String bez
+ double getPreis()
+ String getBescheibung()
Kupplung
+ double getPreis()
+ String getBescheibung()

# Dekoratorbeispiel (Stream Decorator Implementierung)

```
import java.io.*;

// eigene Klasse
public class CharUpperInputStream extends FilterInputStream {

        public CharUpperInputStream(InputStream is) {
          super(is);
        }

        public int read() throws IOException {
          int c = super.read();
          return Character.toUpperCase(c);
        }
}
```

# Dekorator Beispiel

```
import java.io.*;

public class CharUpperInputStream {
        private void read(String sFilename) throws IOException {
          InputStream is = new FileInputStream (sFilename);
          InputStream bis = new BufferedInputStream (is);
          InputStream cuis = new CharUpperInputStream( bis );
          int c ;
          while (( c = cuis.read() ) >= 0) {
                System.out.print((char) c);
          }
        }
}
```