Analyse und Modellierung von Informationssystemen

Dr. Klaus Höppner

Hochschule Darmstadt – Wintersemester 2014/2015

Strukturmuster

Kompositum

Dekorierer

Adapter

Einführung: Strukturmuster

Zu den von der Gang of Four präsentierten Mustern gehören mehrere *Strukturmuster*, die über die Definition von Relationen zwischen Klassen übergeordnete Strukturen definieren:

- Adapter
- Bridge (Brücke)
- Composite (Kompositum)
- Decorator (Dekorierer)
- Facade (Fassade)
- Flyweight (Fliegengewicht)
- Proxy (Stellvertreter)

Beispiel

Im Folgenden wird eine Anwendung entwickelt, die Text mit Auszeichnungen wie Fett und Kursiv versehen kann, die dann als HTML exportiert werden können.

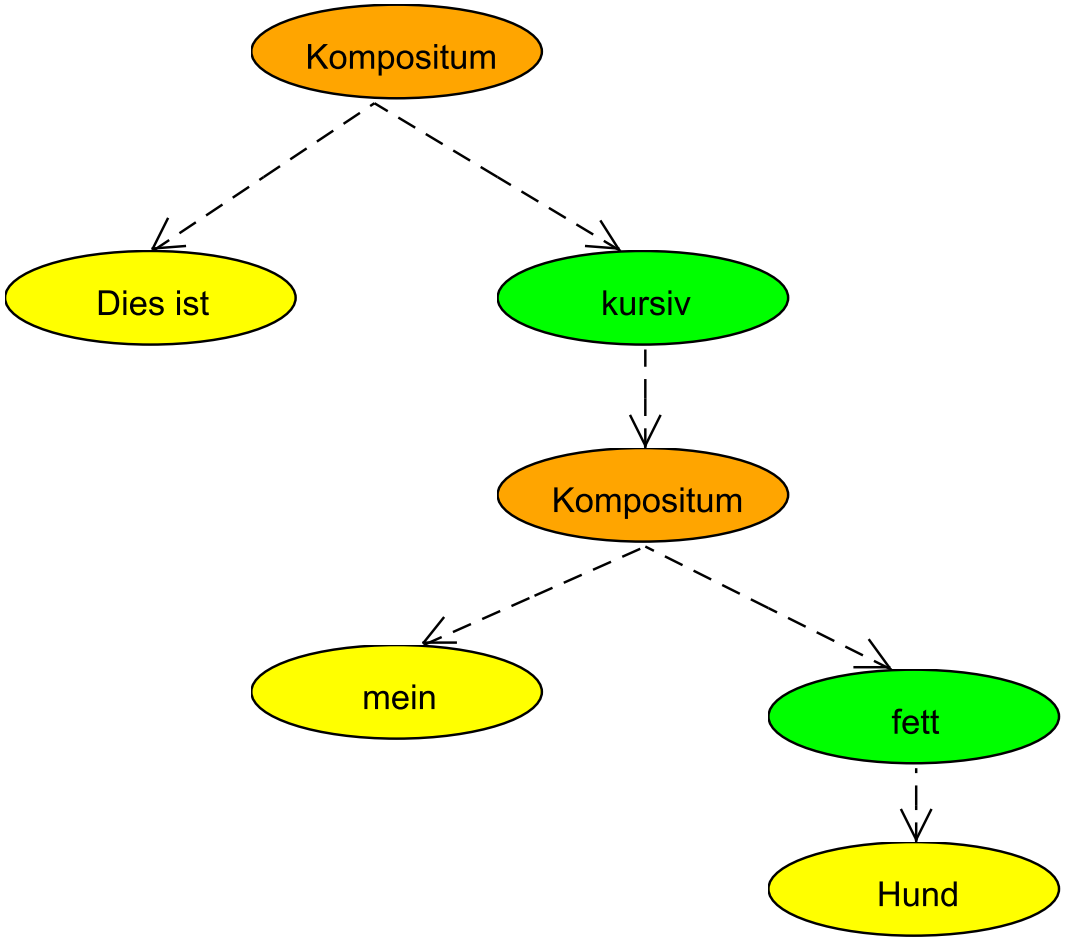
Hierbei kommen folgende Elementtypen vor:

- Normaler Text ohne Auszeichnung.
- Text, der fett oder kursiv formatiert wird.
- Kombinationen aus verschieden formatierten Textstücken.
- Wiederum fett oder kursive formatierte Kombinationen aus formatierten Textstücken.
- Usw. usw.

Insgesamt hat der formatierte Text eine Baumartige Struktur.

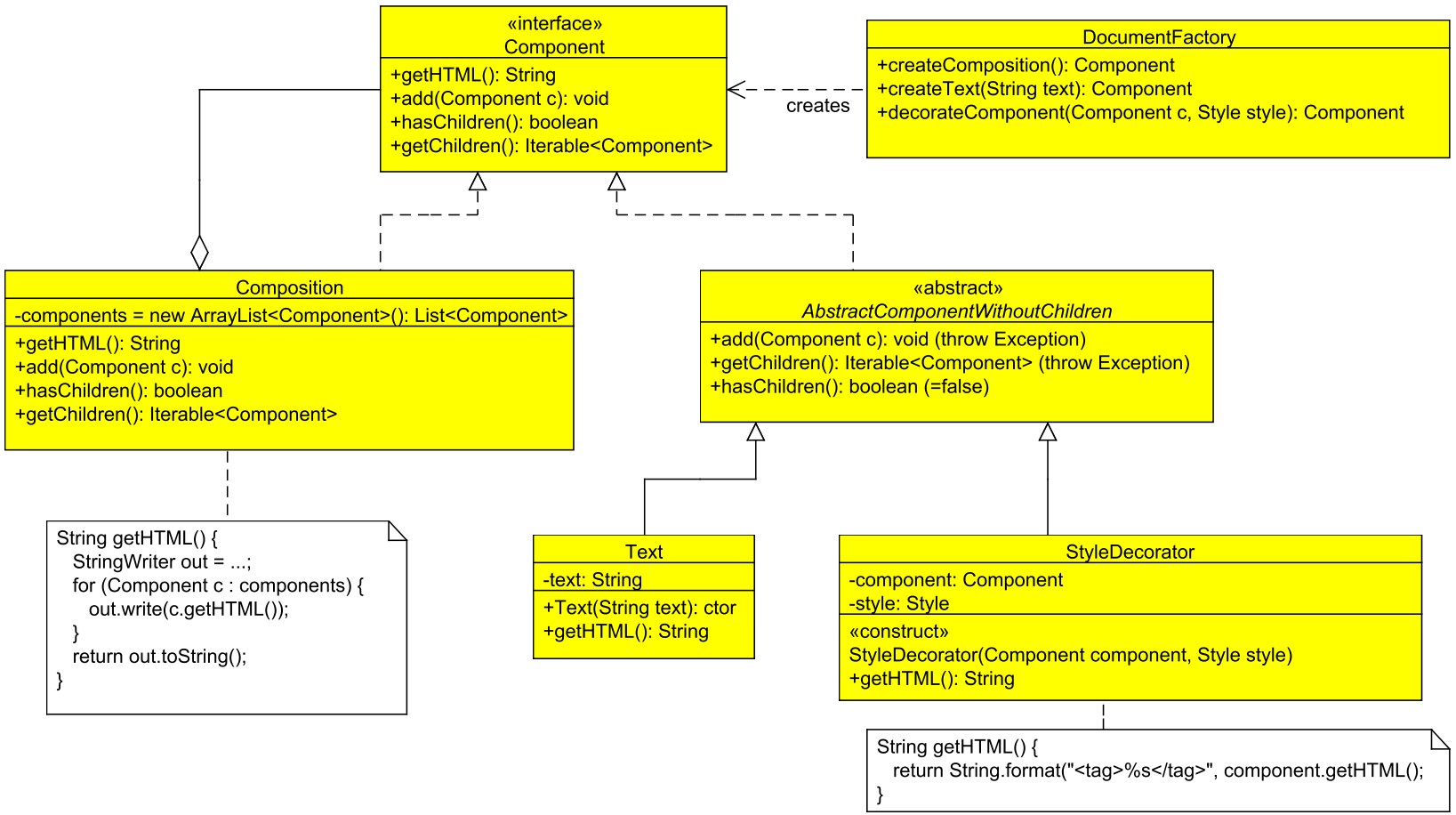
Beispiel

Dies ist *mein* ***Hund***



ergibt: `Dies ist <it>mein <b>Hund</b></it>`

Klassendiagramm

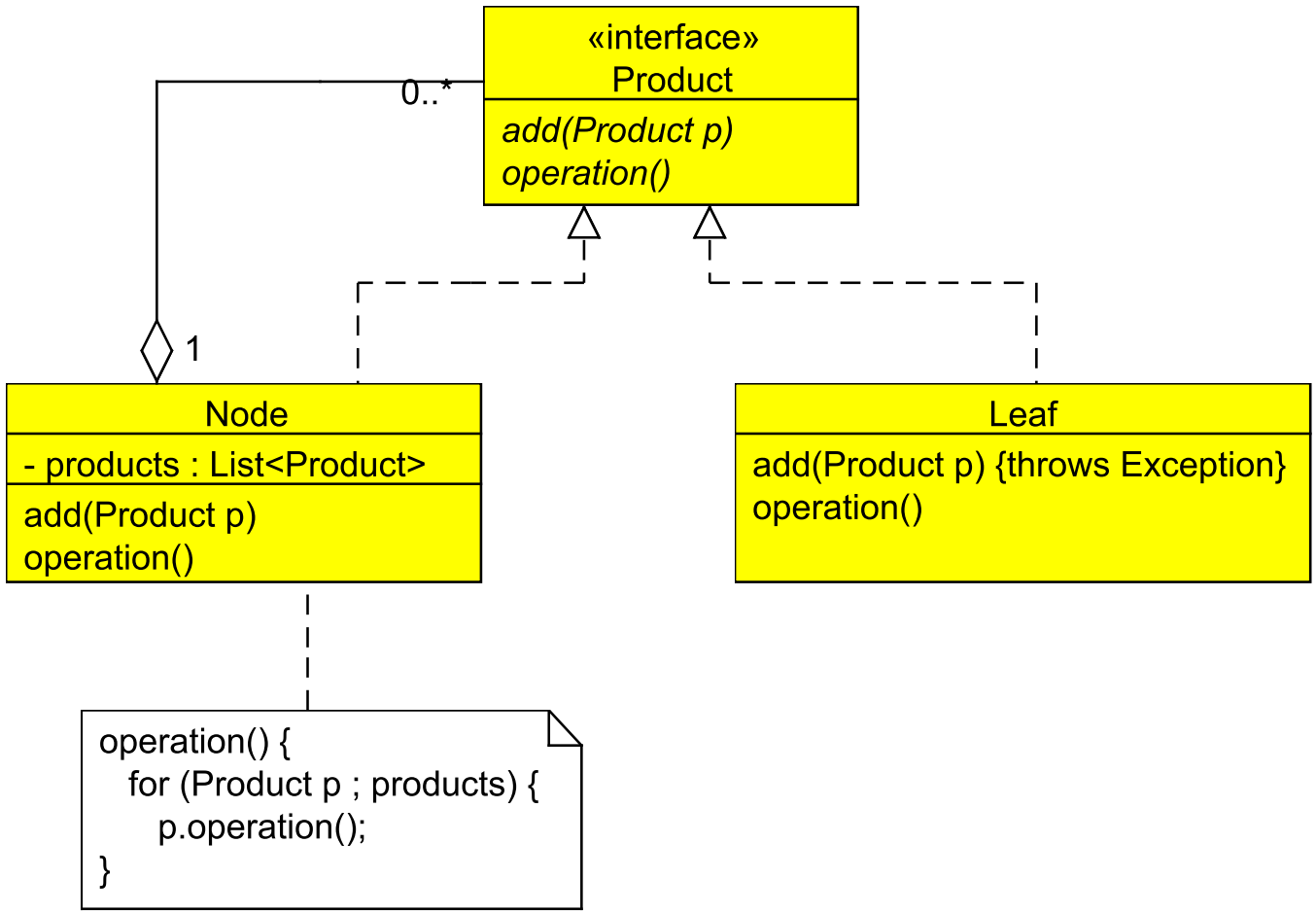


Verwendete Strukturmuster

In dem Beispiel werden zwei Strukturmuster verwendet:

- Das Kompositum (*composite*) beschreibt die baumartige Struktur mit Verästelungen (die Kindelemente haben können) und Blättern (die keine Kinder haben können). Das Typische am Kompositum ist, dass es sich beliebig häufig selbst enthalten kann.
- Der Dekorierer (*decorator*) beschreibt hier die Klasse `StyleDecorator`, die das Interface `Component` implementiert, indem sie auf einer anderen Komponente aufsetzt, derer HTML-Code durch `<b>...</b>` bzw. `<it>...</it>` dekoriert wird.
  Die Methode `getHTML()` wird nicht eigenständig implementiert, sondern leitet den Aufruf an die intern benutzte Komponente weiter.

Diagramm: Kompositum



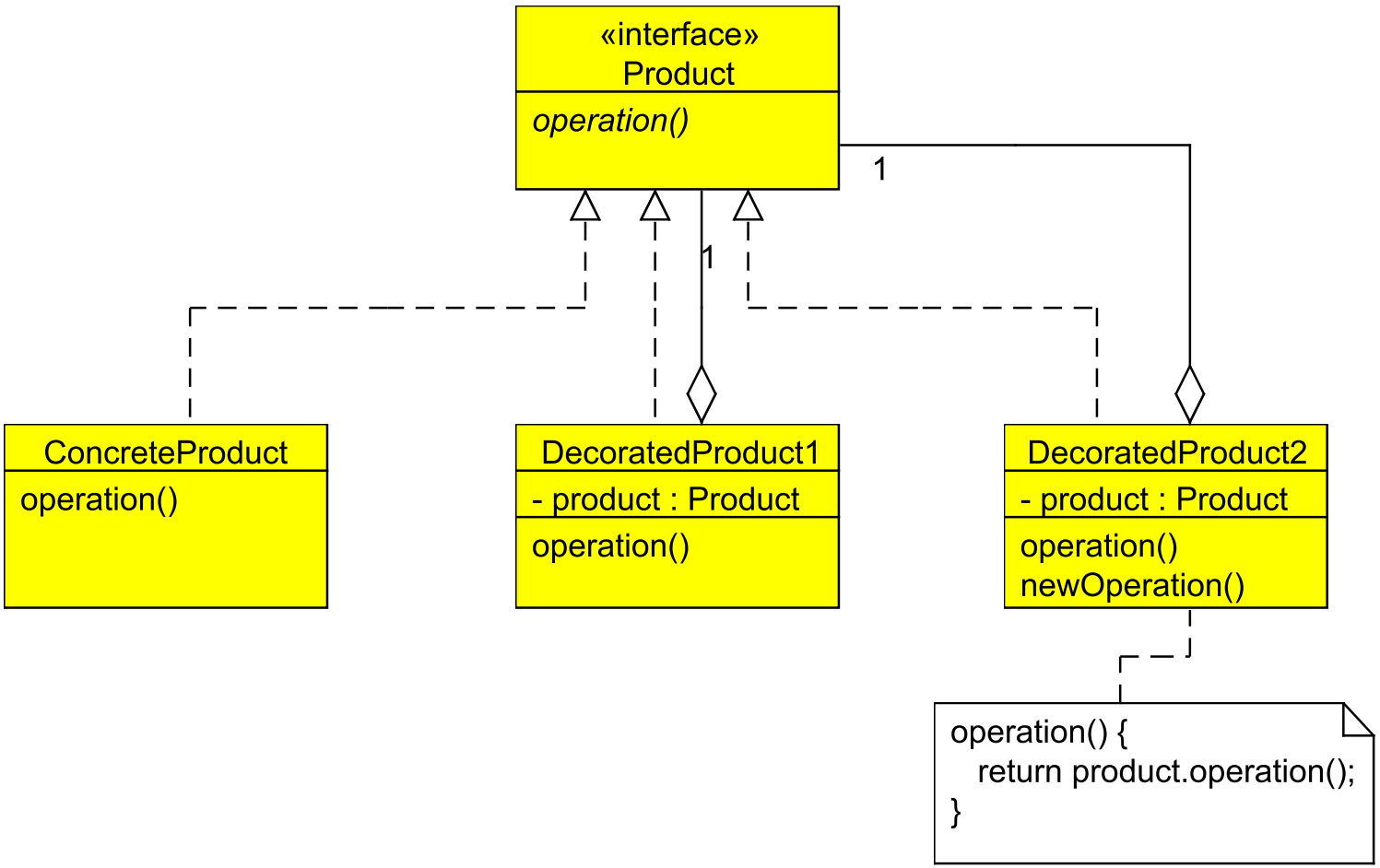
Beispiel

In Swing erben die Komponenten von `JComponent`. Hierfür ist die Methode `add` definiert, über die einer Komponente Kindkomponenten hinzugefügt werden können.

Hierbei gilt

- Einige Komponenten unterstützen das Hinzufügen von Kindkomponenten, z. B. `JPanel`.
- Anderen Komponenten können keine Kindkomponenten hinzugefügt werden, so kann z. B. `JTextField` natürlich keine Kindkomponenten besitzen.
- Bestimmte Kompontenten dürfen nicht anderen Komponenten als Kind hinzugefügt werden, so kann ein Toplevel-Element wie ein `JFrame` nicht Kindkomponente eines `JPanel` sein.

Diagramm: Dekorierer



Das Dekorierer-Muster

Beim Dekorierer (*Decorator*) verwenden einige Klassen zum Implementieren eines Interfaces intern ein Objekt, das ebenfalls das Interface implementiert.

Hierbei greift die Dekorierer zur Durchführung der im Interface deklarierten Operationen auf das interne Objekt zurück, lässt also das verwaltete Objekt die Operation erledigen.

Hierbei kann der Dekorierer

- das Ergebnis der Operation, wie sie vom verwalteten Objekt erledigt wurde, verändern, und/oder
- neue Operationen hinzufügen.

Da der Dekorierer selber das Interface implementiert, kann das Dekorieren über beliebig viele Ebenen stattfinden.

Beispiel

Datei `Mahlzeit.java`:

```
public interface Mahlzeit {
    public double getPreis();
    public String getBeschreibung();
}
```

Datei `AbstractMahlzeit.java`:

```
public abstract class AbstractMahlzeit implements Mahlzeit {
    private String beschreibung;
    private double preis;

    public AbstractMahlzeit(String beschreibung, double preis) {
        this.beschreibung = beschreibung;
        this.preis = preis;
    }
```

Beispiel (Forts.)

Datei `AbstractMahlzeit.java` (Forts.):

```
    @Override
    public double getPreis() {
        return preis;
    }

    @Override
    public String getBeschreibung() {
        return beschreibung;
    }

    @Override
    public String toString() {
        return String.format("%s zu %.2f",
                getBeschreibung(), getPreis());
    }
}
```

Beispiel (Forts.)

Datei `Hauptgericht.java`:

```
public class Hauptgericht extends AbstractMahlzeit {
    public Hauptgericht(String beschreibung, double preis) {
        super(beschreibung, preis);
    }
}
```

Datei `Beilage.java`:

```
public class Beilage extends AbstractMahlzeit {
    private Mahlzeit basisEssen;

    public Beilage(Mahlzeit basisEssen,
                   String beschreibung, double preis) {
        super(beschreibung, preis);
        this.basisEssen = basisEssen;
    }
```

Beispiel (Forts.)

Datei `Beilage.java` (Forts.):

```
    @Override
    public double getPreis() {
        return basisEssen.getPreis()+super.getPreis();
    }

    @Override
    public String getBeschreibung() {
        return String.format("%s, %s",
                basisEssen.getBeschreibung(),
                super.getBeschreibung());
    }
}
```

Verwenden des Dekorierers

Im Beispiel kann nun ein Hauptgericht beliebig mit Beilagen dekoriert werden, Beilagen ohne Hautgericht gibt es jedoch nicht:

```
Mahlzeit m = new Beilage(
    new Beilage(
        new Hauptgericht("Schnitzel", 6.90),
        "Pommes", 2.50),
    "Salat", 1.50);
System.out.println(m);
```

ergibt `Schnitzel, Pommes, Salat zu 10,90`.

Reale Verwendung bei I/O-Klassen in Java

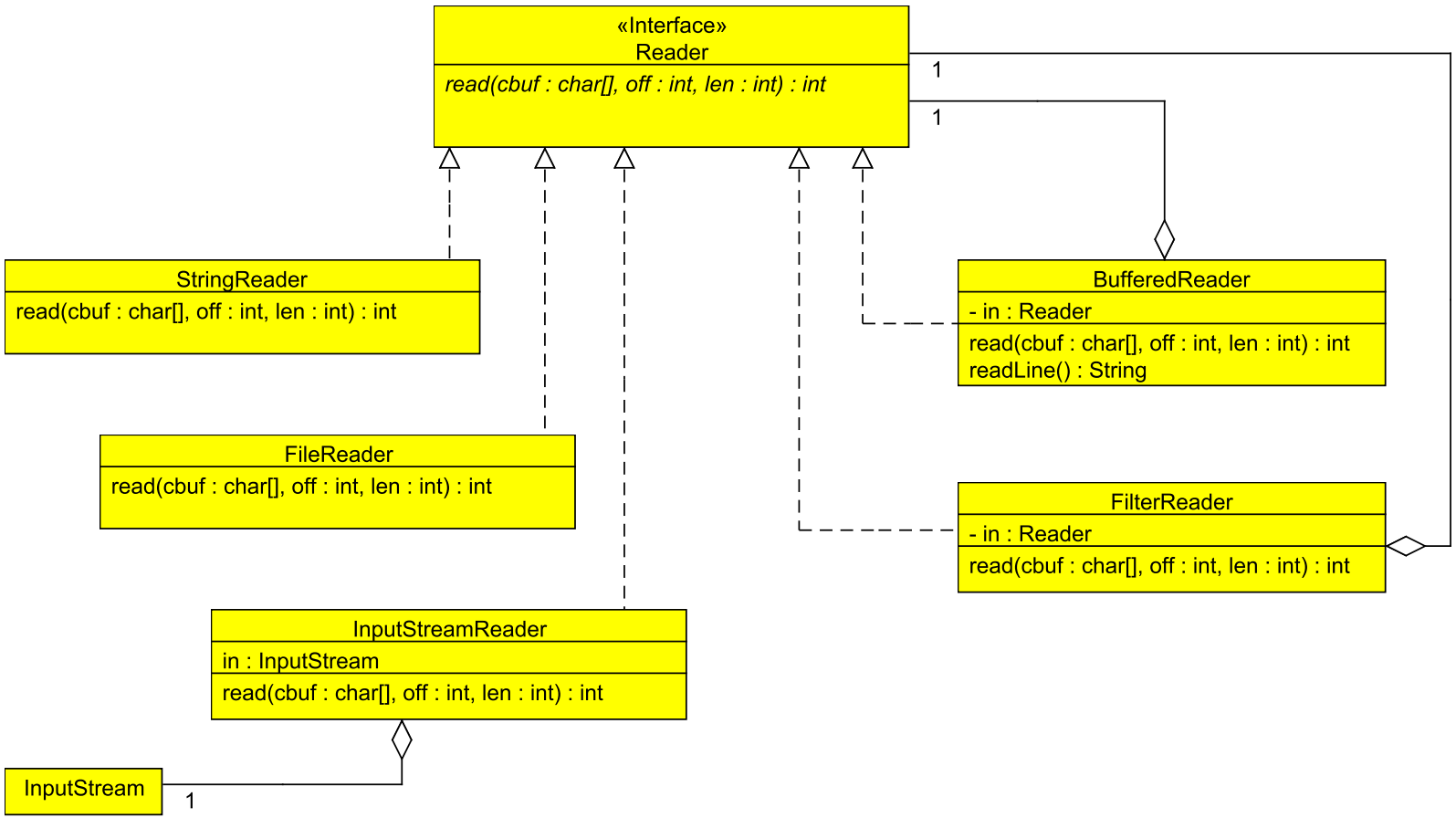
Java benutzt Dekorierer bei den I/O-Klassen. Hier existieren die Interfaces:

- `InputStream`
- `OutputStream`
- `Reader`
- `Writer`

Hierbei existieren als konkrete Klassen so genannte Low-Level-Klassen, die lediglich die Grundfunktionalität zum Lesen/Schreiben einzelner Bytes oder Zeichen erlauben, z. B. dient der `FileWriter` zum Schreiben von Zeichen in eine Datei.

Weiterhin existierende dekorierende Klasse als High-Level-Klassen, die zum Beispiel eine zeilenweise gepufferte Eingabe erlauben.

Diagramm: Reader-Klassen in java.io



Beispiel

Eine Klasse zum Verändern eingelesener Zeichen, hier Umwandeln in Großbuchstaben:

```
public class UppercaseReader extends FilterReader {
    public UppercaseReader(Reader in) {
        super(in);
    }

    @Override
    public int read(char[] cbuf, int off, int len)
                                        throws IOException {
        int count = super.read(cbuf, off, len);
        for (int i=off; i<off+count; i++) {
            cbuf[i] = Character.toUpperCase(cbuf[i]);
        }
        return count;
    }
}
```

Anwenden des Filter-Readers

```
BufferedReader in = new BufferedReader(
    new UppercaseReader(
        new InputStreamReader(System.in)));
System.out.println(in.readLine());
```

Hier werden Zeichen von der Tastatur eingelesen. Das Objekt `System.in` ist jedoch vom Type `InputStream`, liefert also Bytes. Die Klasse `InputStreamReader` übersetzt gelesene Bytes in Zeichen, übersetzt also die Operation aus einem Interface in eine Operation aus einem anderen.

Dies führt zum nächsten Entwurfsmuster, dem *Adapter*.

Der Adapter

Der Adapter übersetzt die Funktionalität eines Objektes so, dass es wie ein anderes erscheint.

Hierbei gibt es zwei Möglichkeiten:

- Der *Klassenadapter* benutzt eine Adapterklasse, die das Zielinterface implementiert, aber von der zu adaptierenden Klasse (!) erbt.
- Beim *Objektadapter* verwendet die Adapterklasse keine Vererbung, sondern delegiert die Funktionalität aus dem zu adaptierenden Interface bzw. der zu adaptierenden Klasse an ein anderes Objekt.

Während der Klassenadapter im Prinzip mit weniger Code zu implementieren ist, ist er unflexibler als der Objektadapter, denn er ist an eine konkrete Klasse gebunden!

Beispiel

aus „Head First Design Patterns“:

```
public interface Duck {
    public void quack();
    public void fly();
}
```

```
public class MallardDuck implements Duck {
    public void quack() {
        System.out.println("Quack");
    }

    public void fly() {
        System.out.println("I'm flying");
    }

}
```

Beispiel (Forts.)

```
public interface Turkey {
    public void gobble();
    public void fly();
}
```

```
public class WildTurkey implements Turkey {
    public void gobble() {
        System.out.println("Gobble gobble");
    }
    public void fly() {
        System.out.println("I'm flying a short distance");
    }
}
```

Adapter Truthahn auf Ente

```
public class TurkeyAdapter implements Duck {
    Turkey turkey;

    public TurkeyAdapter(Turkey turkey) {
        this.turkey = turkey;
    }

    public void quack() {
        turkey.gobble();
    }

    public void fly() {
        for(int i=0; i < 5; i++) {
            turkey.fly();
        }
    }
}
```

Reales Beispiel

In alten Java-Versionen gab es als Container-Klassen z. B. `Vector`, wobei man sich für einen solchen über die Methode `elements()` eine `Enumeration` über die Elemente holen konnte.

Mittlerweile sind in Java die `Collection`s Standard, über die man mit den Interfaces `Iterator` und `Iterable` leicht iterieren kann.

Im folgenden werden zwei Adapterklassen definiert, mit deren Hilfe man über einen `Vector` wie eine `Collection` iterieren kann.

Die Adapterklassen

```
public class Enum2Iterator<E> implements Iterator<E> {
    Enumeration<E> e;

    public Enum2Iterator(Enumeration<E> e) {
        this.e = e;
    }

    @Override
    public boolean hasNext() {
        return e.hasMoreElements();
    }

    @Override
    public E next() {
        return e.nextElement();
    }

    @Override
    public void remove() {
        throw new RuntimeException("not implemented");
    }
}
```

Die Adapterklassen (Forts.)

```
public class Enum2Iterable<E> implements Iterable<E> {
    private Enumeration<E> e;

    public Enum2Iterable(Enumeration<E> e) {
        this.e = e;
    }

    @Override
    public Iterator<E> iterator() {
        return new Enum2Iterator<E>(e);
    }
}
```

Anwendung

```
Vector<String> v = new Vector<String>();
v.add("Hallo");
v.add("Echo");

for (String s : new Enum2Iterable<String>(v.elements())) {
    System.out.println(s);
}
```