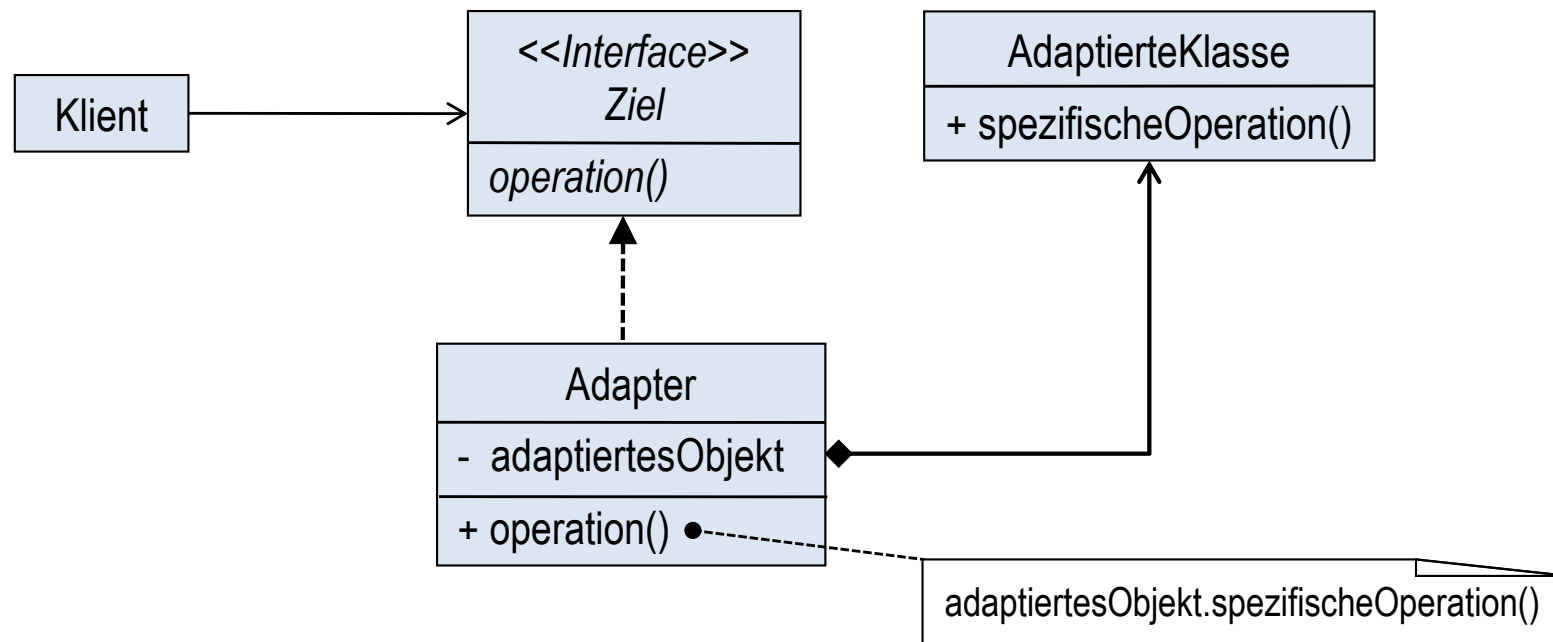


Kapitel 15: Entwurfsmuster

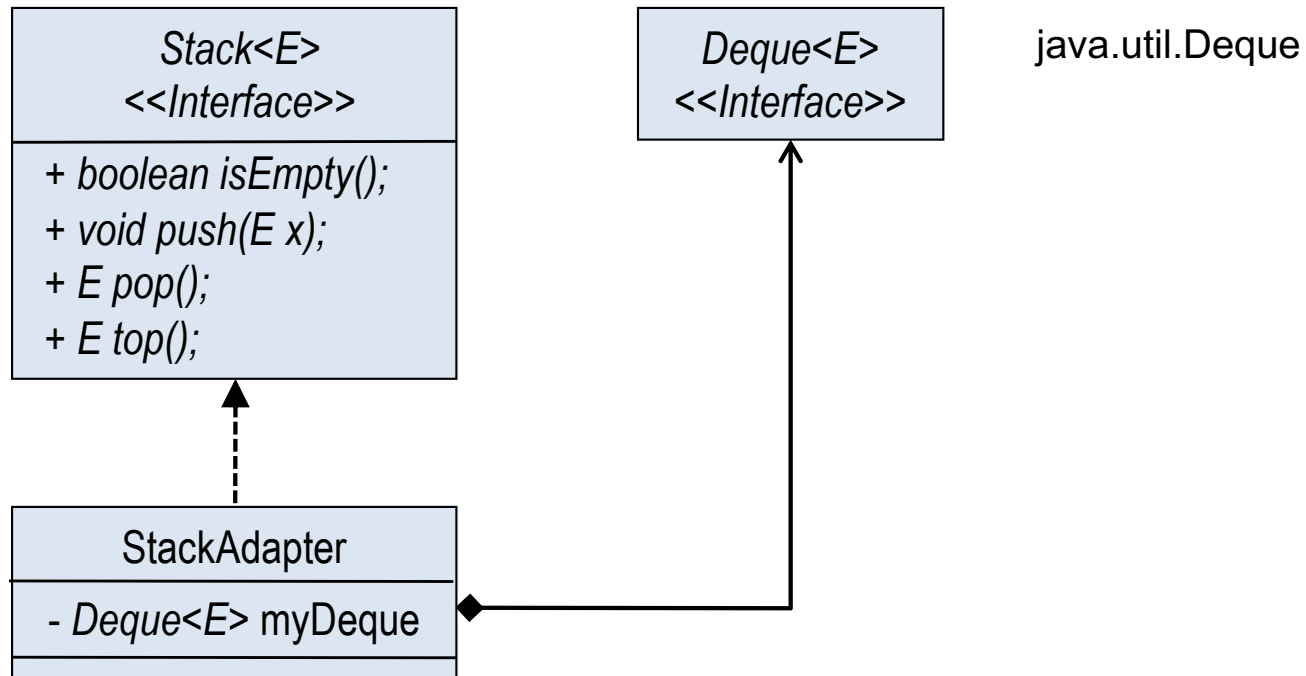
- Adapter
- Assoziationen
- Beobachter (Observer)
- Dekorierer (Decorator)
- Kompositum (Composite)
- Schablonenmethode (Template Method)
- Strategie (Strategy)

Zweck und Struktur

- Die Schnittstelle einer gegebenen Klasse (AdaptierteKlasse) wird an eine andere vom Klienten erwartete Schnittstelle (Ziel) angepasst.



Beispiel (1)



Beispiel (2)

```
public interface Stack<E> {  
    boolean isEmpty();  
    void push(E x);  
    E pop();  
    E top();  
}
```

```
public class StackAdapter<E> implements Stack<E> {  
    private Deque<E> deque;  
    public StackAdapter(Deque<E> d) {deque = d;}  
    public boolean isEmpty() {return deque.isEmpty();}  
    public void push(E x) {deque.offerFirst(x);}  
    public E pop() {return deque.pollFirst();}  
    public E top() {return deque.peekFirst();}  
}
```

// ...

```
public static void main(...) {  
    Stack<Integer> s = new StackAdapter<>(new LinkedList<Integer>());  
    s.push(3);  
    s.push(2);  
    s.push(3);  
    while (! s.isEmpty())  
        System.out.println(s.pop());  
}
```

Auswahl der Deque-Implementierung
zur Laufzeit.

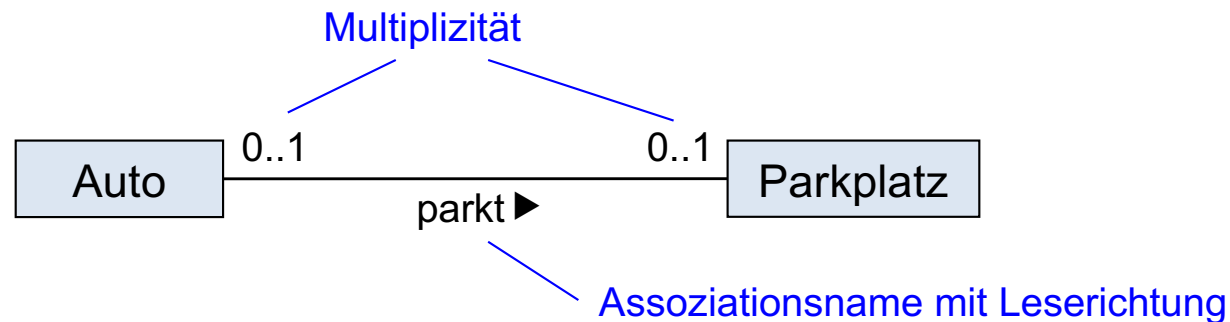
Kapitel 15: Entwurfsmuster

- Adapter
- Assoziationen
- Beobachter (Observer)
- Dekorierer (Decorator)
- Kompositum (Composite)
- Schablonenmethode (Template Method)
- Strategie (Strategy)

Assoziation

- Unter einer **binären Assoziation** versteht man eine beliebige Beziehung zwischen Objekten zweier Klassen.
- Mit der **Multiplizität** wird festgelegt, wieviele Objekte an einer Assoziation beteiligt sein können.

1-1-Assoziation



n-m-Assoziation



Ein Student besucht beliebig viele, aber wenigstens eine Lehrveranstaltung.
Eine Lehrveranstaltung wird von wenigstens 5 Studenten besucht.

Gerichtete Assoziation

- Eine Assoziation hat eine Richtung.
- Sie kann entweder **unidirektional** oder **bidirektional** sein.

Unidirektionale Assoziation

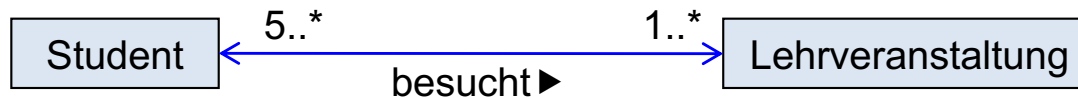


Navigation nur von Student in Richtung Lehrveranstaltung möglich:

Ein Student kennt seine Lehrveranstaltungen.

Eine Lehrveranstaltung kennt nicht die sie besuchenden Studenten.

Bidirektionale Assoziation

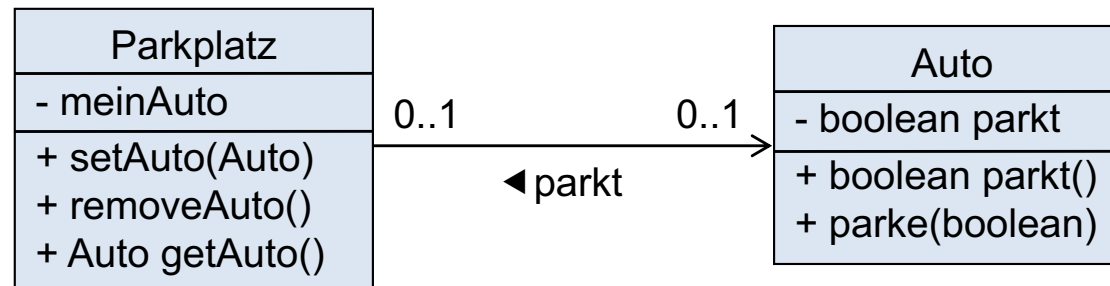


Navigation ist in beiden Richtungen möglich:

Ein Student kennt seine Lehrveranstaltungen.

Eine Lehrveranstaltung kennt die sie besuchenden Studenten.

Unidirektionale 1-1-Assoziation



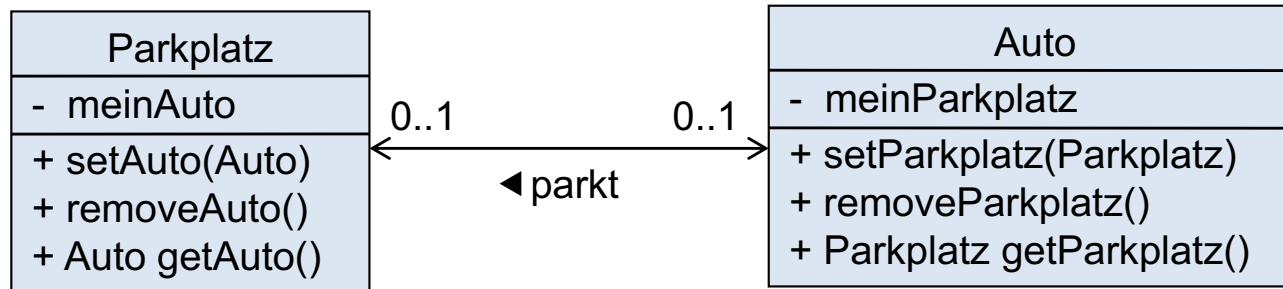
- Ein Parkplatz hat eine Referenz auf sein assoziiertes Objekt (d.h. geparktes Auto)
- Die Assoziationsbeziehung kann nur von Parkplatz aus über die Methode `setAuto(Auto)` bzw. `removeAuto()` geändert werden.
- Um überprüfen zu können, ob die Multiplizitäten eingehalten werden, muss ein Auto wissen, ob es assoziiert (d.h. geparkt) ist.
- Wird eine Assoziationsbeziehung eingerichtet oder gelöscht, dann muss das beim assoziierten Auto mit `parke(boolean)` gespeichert werden.

Unidirektionale 1-1-Assoziation in Java

```
public class Parkplatz {  
    private String name;  
    public Parkplatz(String name) {  
        this.name = name;}  
    public String getName() {  
        return name;}  
  
    private Auto meinAuto;  
    public void setAuto(Auto a) {  
        if (a.parkt()) // Auto parkt bereits  
            return;  
        if (meinAuto != null) // Parkplatz ist bereits besetzt  
            return;  
        meinAuto = a;  
        meinAuto.parke(true);  
    }  
    public void removeAuto() {  
        if (meinAuto == null) // kein Auto parkt  
            return;  
        meinAuto.parke(false);  
        meinAuto = null;  
    }  
    public Auto getAuto() {  
        return meinAuto;}  
}
```

```
public class Auto {  
    private String name;  
    public Auto(String name) {  
        this.name = name;}  
    public String getName() {  
        return name;}  
  
    private boolean parkt = false;  
  
    public boolean parkt() {  
        return this.parkt;  
    }  
  
    public void parke(boolean b) {  
        parkt = b;  
    }  
}
```

Bidirektionale 1-1-Assoziation



- Die Assoziationsbeziehungen können nun sowohl von Parkplatz als auch von Auto aus verändert werden.
- Die Assoziationsmethoden werden vollständig symmetrisch implementiert.
- Die Assoziationsmethoden rufen sich gegenseitig auf:
beim Aufruf von `setAuto(a)` wird `a.setParkplatz(this)` aufgerufen und umgekehrt.
(`removeAuto` und `removeParkplatz` analog).
Besondere Vorsicht ist geboten, um Endlosrekursion zu verhindern!

Bidirektionale 1-1-Assoziation in Java (1)

```
public class Parkplatz {
    // ...

    private Auto meinAuto;

    public void setAuto(Auto a) {
        if (meinAuto != null)
            return; // Parkplatz ist bereits besetzt.
        if (a.getParkplatz() != null && a.getParkplatz() != this)
            return; // Auto steht bereits woanders.
        meinAuto = a;
        a.setParkplatz(this);
    }

    public void removeAuto() {
        if (meinAuto == null) return; // Parkplatz ist bereits frei.
        Auto a = meinAuto;
        meinAuto = null;
        a.removeParkplatz();
    }

    public Auto getAuto() {
        return meinAuto;
    }
}
```

Bidirektionale 1-1-Assoziation in Java (2)

```
public class Auto {  
    // ...  
  
    private Parkplatz meinParkplatz;  
  
    public void setParkplatz(Parkplatz p) {  
        if (meinParkplatz != null)  
            return; // Auto parkt bereits  
        if (p.getAuto() != null && p.getAuto() != this)  
            return; // Parkplatz von anderem Auto besetzt  
        meinParkplatz = p;  
        p.setAuto(this);  
    }  
  
    public void removeParkplatz() {  
        if (meinParkplatz == null) return; // Auto parkt nicht  
        Parkplatz p = meinParkplatz;  
        meinParkplatz = null;  
        p.removeAuto();  
    }  
  
    public Parkplatz getParkplatz() {  
        return meinParkplatz;  
    }  
}
```

Bidirektionale 1-1-Assoziation in Java (3)

```
public static void main(String[] args) {
```

```
    Auto a = new Auto("KN-AA 111");  
    Parkplatz p = new Parkplatz("pp");
```

```
    p.setAuto(a);  
    p.removeAuto();
```

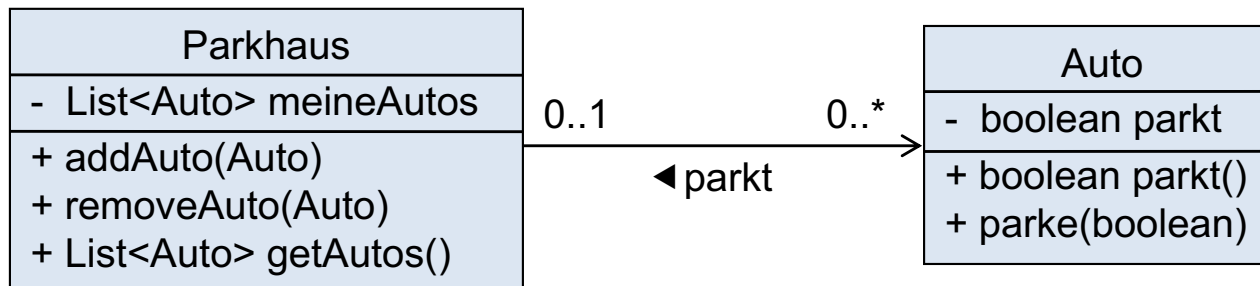
Parken und Parkplatz verlassen
über **Parkplatz p** veranlasst.

```
    a.setParkplatz(p);  
    a.removePrakplatz();
```

Parken und Parkplatz verlassen
über **Auto a** veranlasst.

```
}
```

Unidirektionale 1-n-Assoziation



- Alle assoziierten (geparkten) Autos werden in einem Listen-Container verwaltet.
- Alternativ geht auch ein Set-Container. Hierbei ist zu beachten, dass die Verwendung von `TreeSet` im Gegensatz zu `HashSet` eine `compareTo`-Methode erfordert.
- Beachte, dass die `removeAuto`-Methode nun einen Parameter haben muss.
- Wie bei der unidirektionalen 1-1-Assoziation muss ein `Auto` über seine Assoziationsbeziehung Auskunft geben bzw. seine Assoziationsbeziehung verändern können.

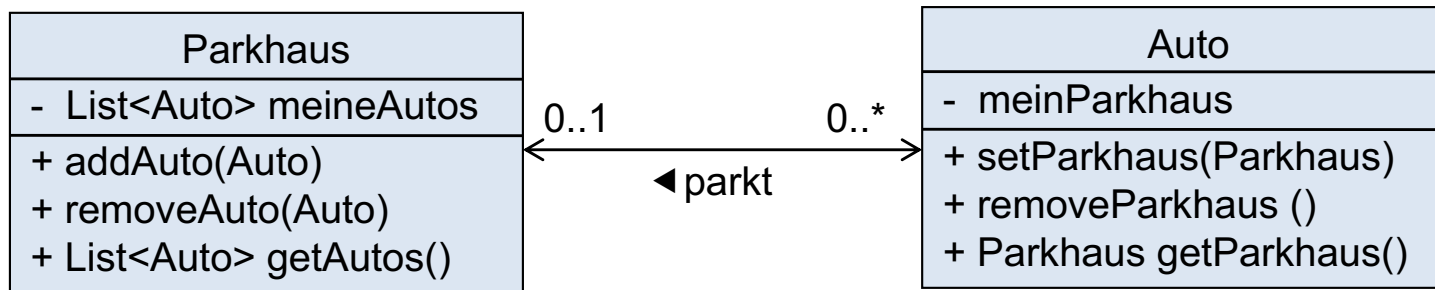
Unidirektionale 1-n-Assoziation in Java

```
public class Parkhaus {  
    // ...  
  
    private List<Auto> meineAutos  
        = new LinkedList<Auto>();  
  
    public void addAuto(Auto a) {  
        if (a.parkt()) return; // Auto parkt bereits  
        meineAutos.add(a);  
        a.parke(true);  
    }  
  
    public void removeAuto(Auto a) {  
        if (meineAutos.remove(a)) // Auto hat hier geparkt  
            a.parke(false);  
    }  
  
    public List<Auto> getAuto() {  
        return meineAutos;  
    }  
}
```

```
public class Auto {  
    // ...  
  
    private boolean parkt = false;  
  
    public boolean parkt() {return this.parkt;}  
  
    public void parke(boolean b) {parkt = b;}  
}
```

```
// ...  
public static void main(...) {  
    Auto a1 = new Auto("KN-AA 111");  
    Auto a2 = new Auto("KN-BB 222");  
    Auto a3 = new Auto("KN-CC 333");  
    Parkhaus ph = new Parkhaus("ph");  
  
    ph.addAuto(a1);  
    ph.addAuto(a1);  
    ph.addAuto(a2);  
    ph.addAuto(a3);  
    ph.removeAuto(a2);  
    System.out.println(ph);  
}
```

Bidirektionale 1-n-Assoziation



- Die Assoziationsbeziehungen können nun sowohl von Parkhaus als auch von Auto aus verändert werden.
- Die Assoziationsmethoden werden fast symmetrisch implementiert.
- Die Assoziationsmethoden rufen sich nun gegenseitig auf:
beim Aufruf von `addAuto(a)` wird auch `a.setParkhaus(this)` aufgerufen und umgekehrt.
(`removeAuto(a)` und `removeParkhaus()` analog).
Besondere Vorsicht ist notwendig, um Endlosrekursion zu verhindern!

Bidirektionale 1-n-Assoziation in Java

```
public class Parkhaus {
    // ...

    private List<Auto> meineAutos = new LinkedList<Auto>();

    public void addAuto(Auto a) {
        if (meineAutos.contains(a))
            return; // Auto parkt bereits hier
        if (a.getParkhaus() != null && a.getParkhaus() != this)
            return; // Auto steht bereits woanders.
        meineAutos.add(a);
        a.setParkhaus(this);
    }

    public void removeAuto(Auto a) {
        if (!meineAutos.contains(a))
            return; // Auto parkt hier nicht
        meineAutos.remove(a);
        a.removeParkhaus();
    }

    public List<Auto> getAuto() {
        return meineAutos;
    }
}
```

```
public class Auto {
    // ...

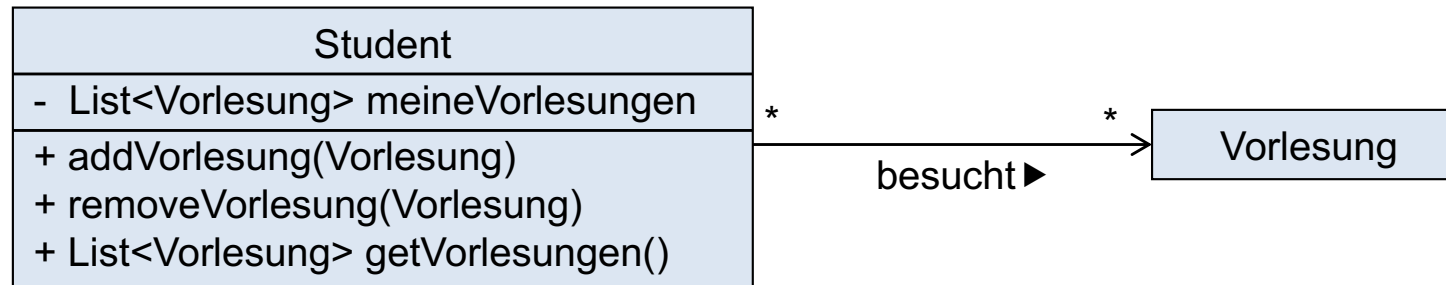
    private Parkhaus meinParkhaus;

    public void setParkhaus(Parkhaus p) {
        if (meinParkhaus != null)
            return; // Auto parkt bereits
        meinParkhaus = p;
        p.addAuto(this);
    }

    public void removeParkhaus() {
        if (meinParkhaus == null)
            return; // Auto parkt nicht
        Parkhaus p = meinParkhaus;
        meinParkhaus = null;
        p.removeAuto(this);
    }

    public Parkhaus getParkhaus() {
        return meinParkhaus;
    }
}
```

Unidirektionale n-m-Assoziation



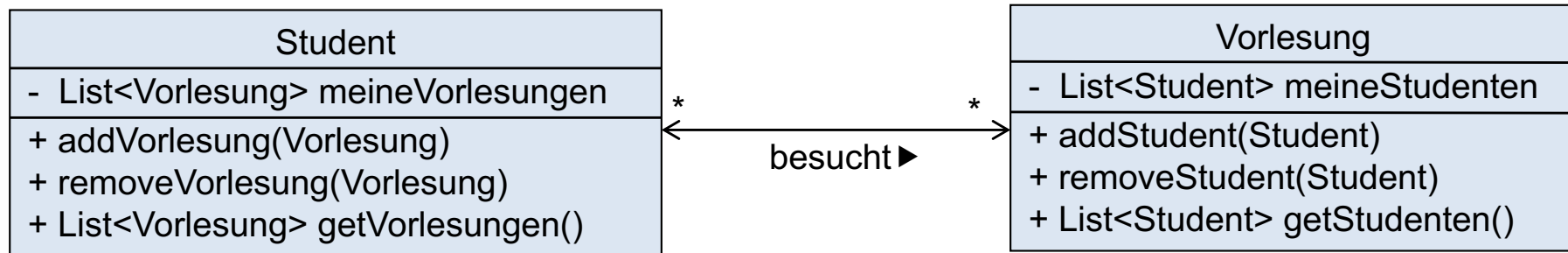
- Da eine Vorlesung bei mehreren Studenten assoziiert werden kann, benötigt sie keine Informationen über ihren Assoziationszustand.

Unidirektionale n-m-Assoziation in Java

```
public class Student {  
    // ..  
  
    private List<Vorlesung> meineVorlesungen  
        = new LinkedList<Vorlesung>();  
  
    public void addVorlesung(Vorlesung v) {  
        if (!meineVorlesungen.contains(v))  
            meineVorlesungen.add(v);  
    }  
  
    public void removeVorlesung(Vorlesung v) {  
        meineVorlesungen.remove(v);  
    }  
  
    public List<Vorlesung> getVorlesungen() {  
        return meineVorlesungen;  
    }  
}
```

```
public class Vorlesung {  
    // ..  
}
```

Bidirektionale n-m-Assoziation



- Student und Vorlesung lassen sich vollständig symmetrisch implementieren.

Bidirektionale n-m-Assoziation in Java

```
public class Student {  
    // ..  
  
    private List<Vorlesung> meineVorlesungen  
        = new LinkedList<Vorlesung>();  
  
    public void addVorlesung(Vorlesung v) {  
        if (!meineVorlesungen.contains(v)) {  
            meineVorlesungen.add(v);  
            v.addStudent(this);  
        }  
    }  
  
    public void removeVorlesung(Vorlesung v) {  
        if (meineVorlesungen.remove(v))  
            v.removeStudent(this);  
    }  
  
    public List<Vorlesung> getVorlesungen() {  
        return meineVorlesungen;  
    }  
}
```

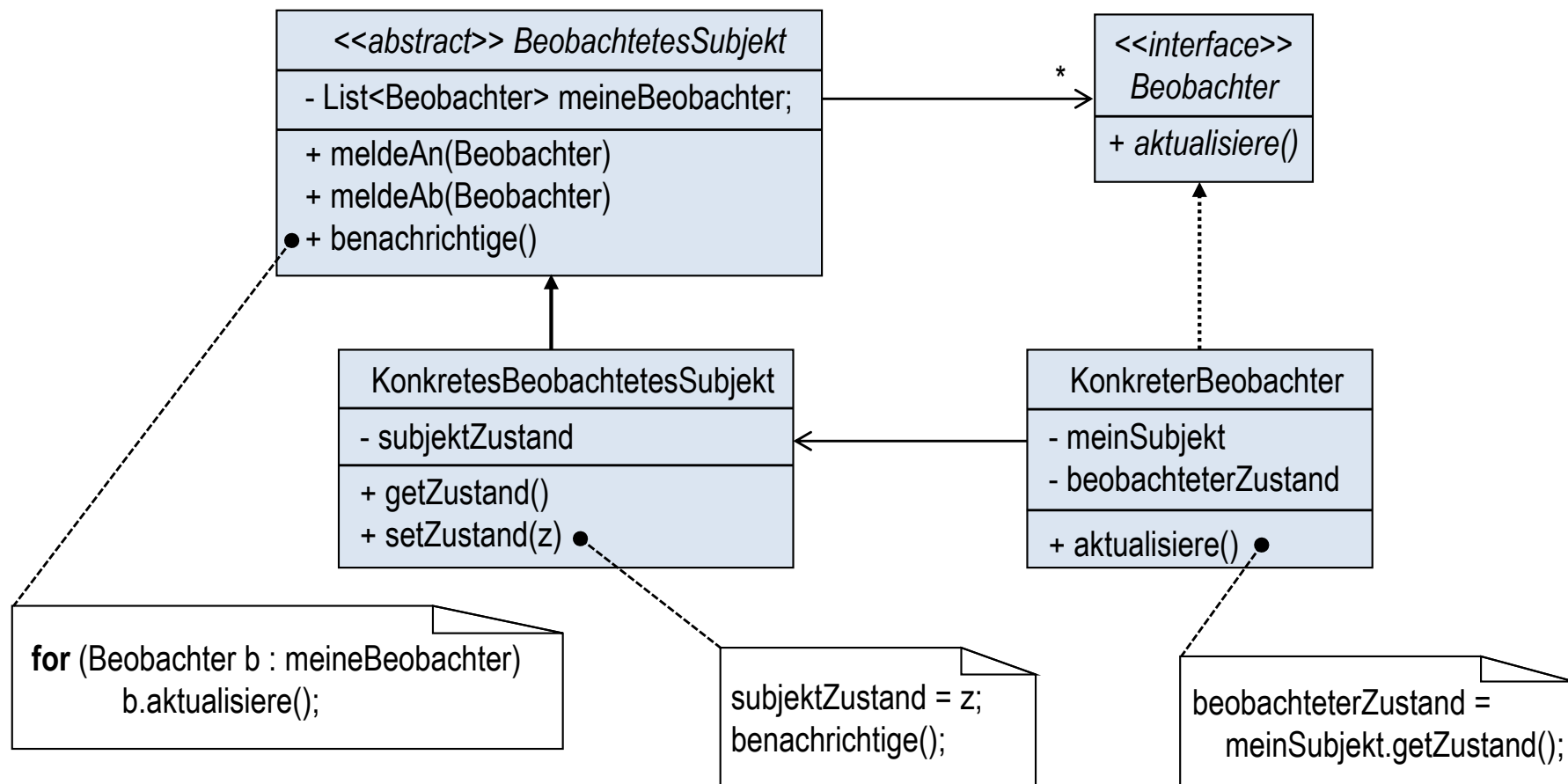
```
public class Vorlesung {  
    // ..  
  
    private List<Student> meineStudenten  
        = new LinkedList<Student>()  
  
    public void addStudent(Student s) {  
        if (!meineStudenten.contains(s)) {  
            meineStudenten.add(s);  
            s.addVorlesung(this);  
        }  
    }  
  
    public void removeStudent(Student s) {  
        if (meineStudenten.remove(s))  
            s.removeVorlesung(this);  
    }  
  
    public List<Student> getStudenten() {  
        return meineStudenten;  
    }  
}
```

Kapitel 15: Entwurfsmuster

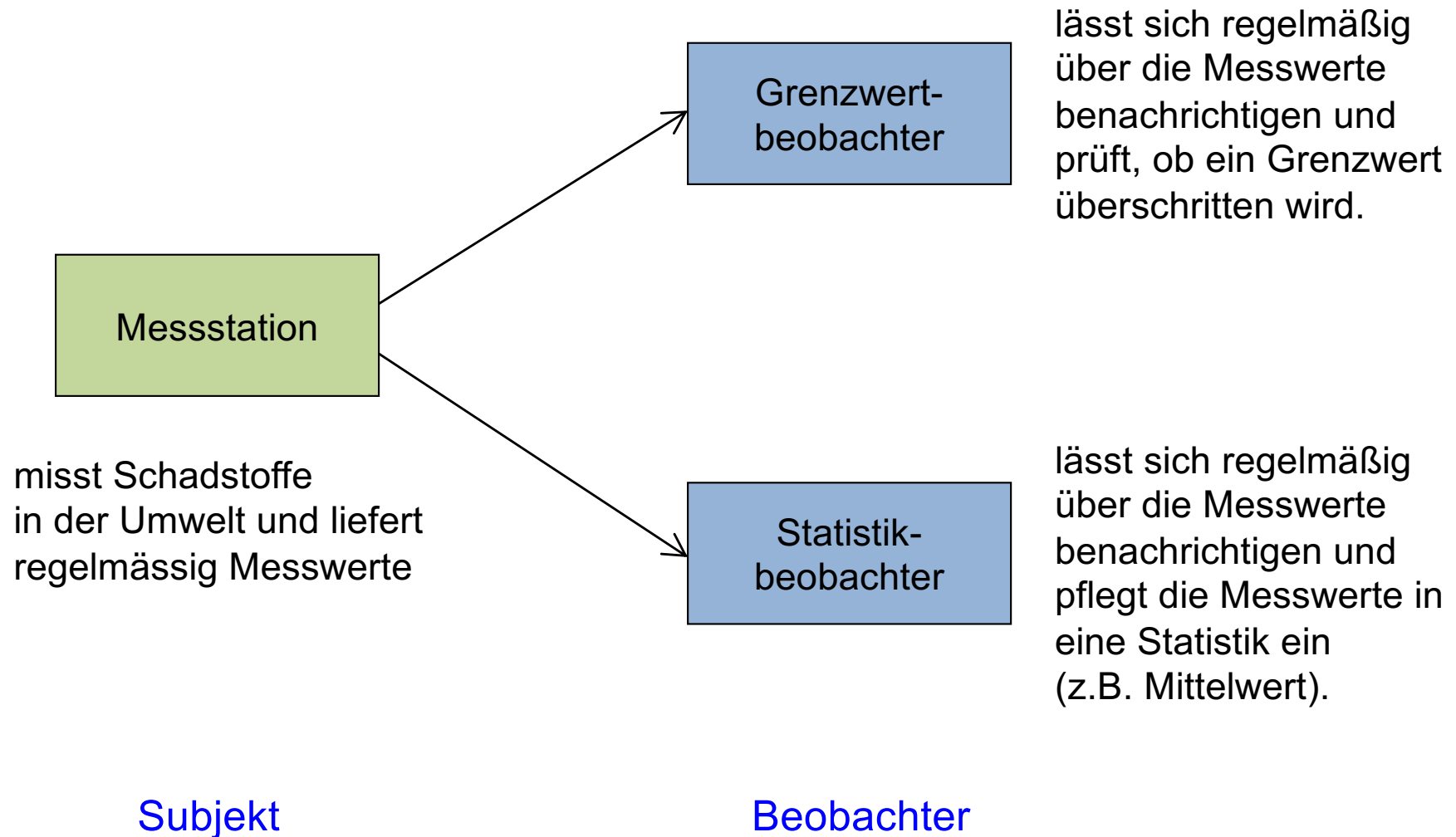
- Adapter
- Assoziationen
- Beobachter (Observer)
- Dekorierer (Decorator)
- Kompositum (Composite)
- Schablonenmethode (Template Method)
- Strategie (Strategy)

Zweck und Struktur

- Definiere eine 1-zu-n-Abhängigkeit zwischen einem Subjekt und n Beobachter-Objekten, die das Subjekt beobachten wollen.
- Ein Beobachter-Objekt lässt sich beim Subjekt als Interessent registrieren und wird dann bei Änderung des Subjektzustands automatisch benachrichtigt.
- Muster wird auch **Publisher-/Subscriber** oder **Listener** genannt.



Beispiel Messstation (1)



Beispiel Messstation: Subjekt

```
public class Subjekt {  
    private List<Beobachter> meineBeobachter =  
        new LinkedList<>();  
  
    public final void meldeAn(Beobachter b) {  
        meineBeobachter.add(b);  
    }  
  
    public final void meldeAb(Beobachter b) {  
        meineBeobachter.remove(b);  
    }  
  
    public final void benachrichtige() {  
        for (Beobachter b : meineBeobachter)  
            b.aktualisiere();  
    }  
}
```

```
public class MessStation extends Subjekt {  
    private double messwert;  
    private Random rand = new Random();  
  
    public void step() throws InterruptedException {  
        Thread.sleep(100);  
        messwert = (double) (Math.abs(rand.nextInt()) % 100);  
        benachrichtige();  
    }  
  
    public double getMesswert() {  
        return messwert;  
    }  
}
```

nach einer Wartezeit von 100 msec wird ein zufälliger Messwert aus dem Intervall [0,100) generiert.

Beispiel Messstation: Beobachter

```
public interface Beobachter {  
    void aktualisiere();  
}
```

```
public class GrenzwertBeobachter implements Beobachter {  
    private MessStation messStation;  
    private final double grenzwert = 30;  
  
    public GrenzwertBeobachter(MessStation m) {  
        messStation = m;  
    }  
  
    public void aktualisiere() {  
        double mw = messStation.getMesswert();  
        if (mw > grenzwert)  
            System.out.printf("Grenzwert überschritten: %5.2f\n", mw);  
        else  
            System.out.printf("Messwert OK: %5.2f\n", mw);  
    }  
}
```

```
public class StatistikBeobachter implements Beobachter {  
    private MessStation messStation;  
    private double mittelwert = 0;  
    private int anzWerte = 0;  
  
    public StatistikBeobachter(MessStation m) {  
        messStation = m;  
    }  
  
    public void aktualisiere() {  
        anzWerte++;  
        double mw = messStation.getMesswert();  
        mittelwert = (mittelwert*(anzWerte-1) + mw)/anzWerte;  
        System.out.printf("Mittelwert: %5.2f\n", mittelwert);  
    }  
}
```

Beispiel Messstation: Anwendung

```
public class Anwendung
{
    public static void main(String[] args) throws InterruptedException {
        MessStation ms = new MessStation();
        Beobachter bg = new GrenzwertBeobachter(ms);
        Beobachter bm = new StatistikBeobachter(ms);
        ms.meldeAn(bg);
        ms.meldeAn(bm);

        for (int i = 0; i < 100; i++)
            ms.step();
    }
}
```

Die beiden Beobachter werden bei der Messstation registriert.

Bemerkungen

- Es gibt grundsätzlich zwei Ansätze, wie den Beobachtern der neue Zustand des beobachteten Subjekts übergeben wird:
- **Pull-Mechanismus:**
der Beobachter holt sich mit einer get-Methode den neuen Wert des beobachteten Subjekts ab.
Wie im Beispiel: die beiden Beobachter holen sich in ihrer aktualisiere-Methode den neuen Messwert von der Messstation mit der Methode getMesswert() ab.
- **Push-Mechanismus:**
die Beobachter erhalten den neuen Zustand direkt über die aktualisiere-Methode über einen zusätzlichen Parameter.
- Ein Beobachter kann selbstverständlich auch mehrere Subjekte beobachten.
Die Nachrichten der einzelnen Subjekte können dann in der aktualisiere-Methode über einen zusätzlichen Subjekt-Parameter auseinander gehalten werden.

Kapitel 15: Entwurfsmuster

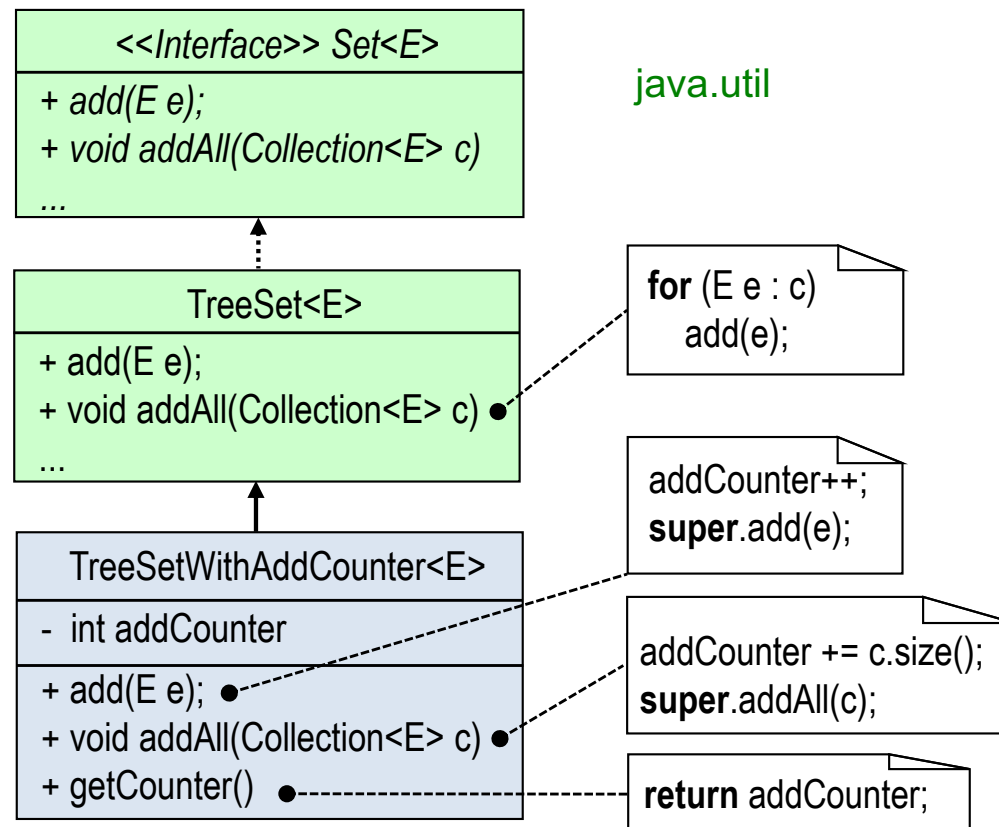
- Adapter
- Assoziationen
- Beobachter (Observer)
- Dekorierer (Decorator)
- Kompositum (Composite)
- Schablonenmethode (Template Method)
- Strategie (Strategy)

Aufgabenstellung: Dekoration einer Klasse

- Bei einem TreeSet-Objekt soll gezählt werden, wieviel Elemente eingefügt werden (add bzw. addAll).
- Definiere dazu eine Klasse TreeSetWithAddCounter, die außer der TreeSet-Funktionalität noch zusätzlich die Zählung übernimmt.

```
List<Integer> l = new LinkedList<>();  
l.add(1);  
l.add(2);  
  
TreeSetWithAddCounter<Integer> s = new TreeSetWithAddCounter<>();  
s.add(3);  
s.addAll(l);  
println(s.getCounter()); // 3
```

Lösung mit Vererbung verstößt gegen Kapselung und ist falsch!



```
List<Integer> l = new ...;
l.add(1);
l.add(2);

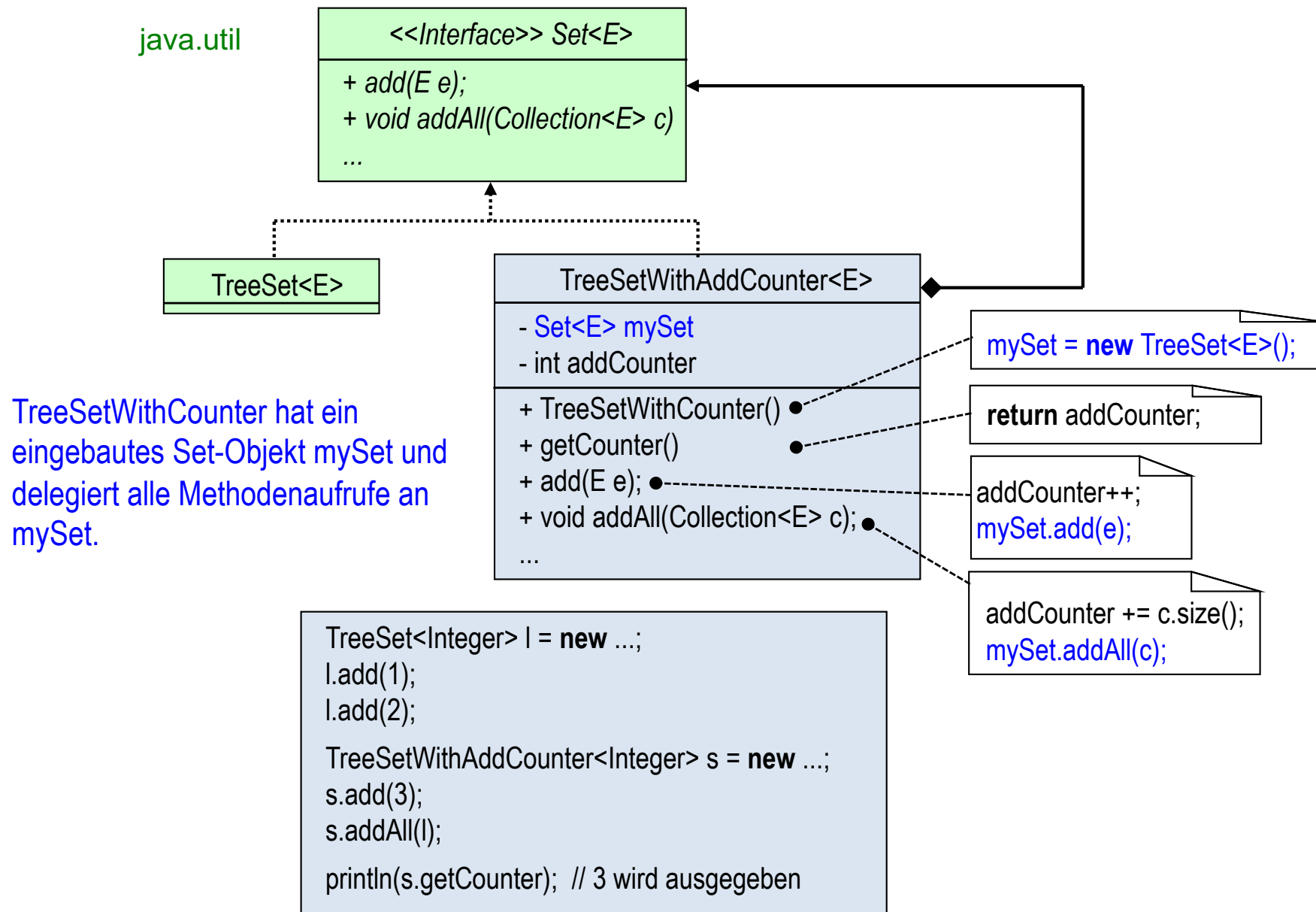
TreeSetWithAddCounter<Integer> s = new ...;
s.add(3);
s.addAll(l);
println(s.getCounter());
```

Es wird 5 ausgegeben statt 3 wie erwartet.

Grund: In TreeSet wird addAll durch mehrmaliges Aufrufen von add realisiert. Somit wird addCounter sowohl bei add als auch bei addAll erhöht.

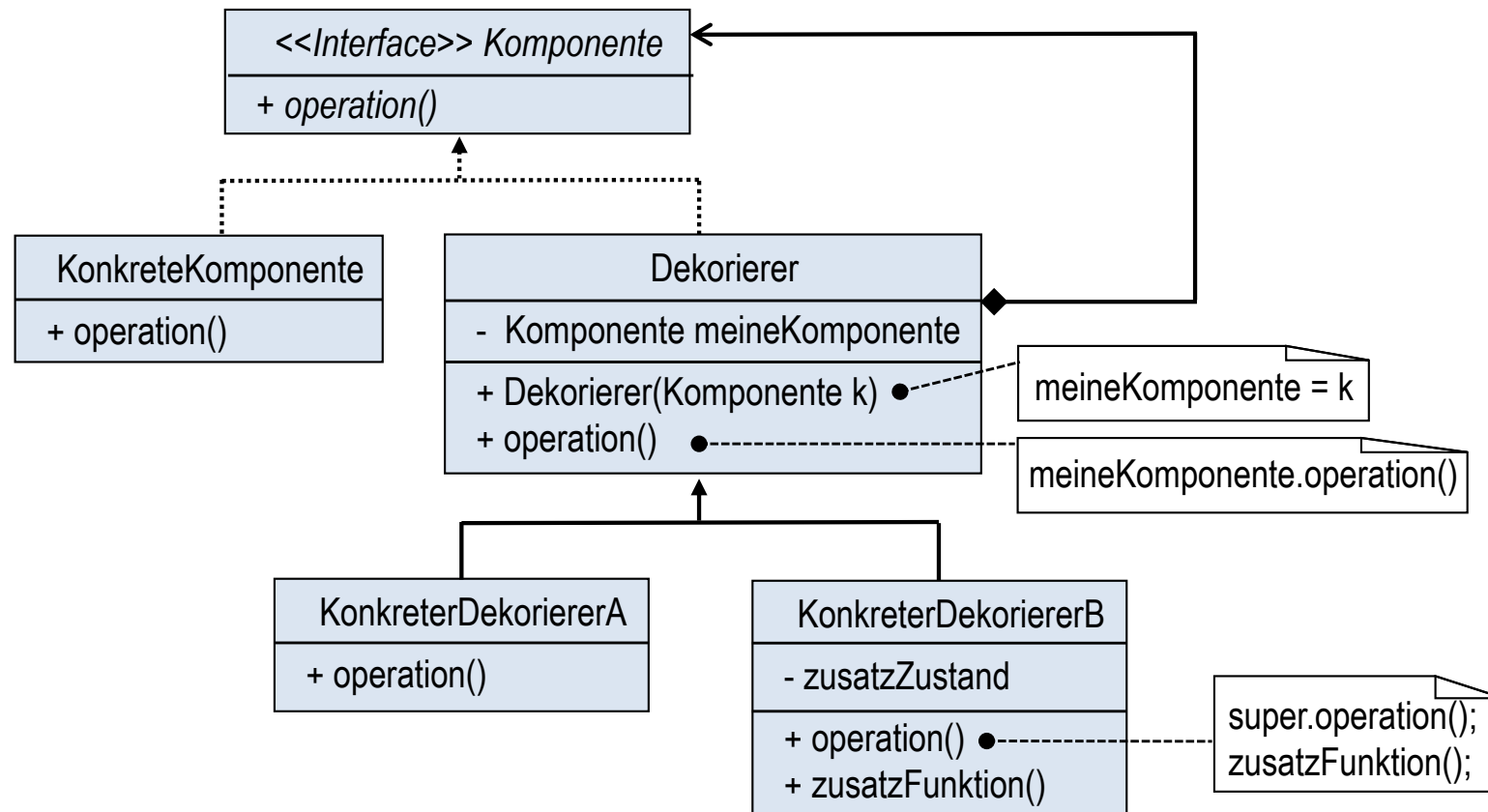
Das Funktionieren von TreeSetWithAddCounter hängt offensichtlich von der Implementierung von TreeSet ab.

Lösung mit Delegation funktioniert

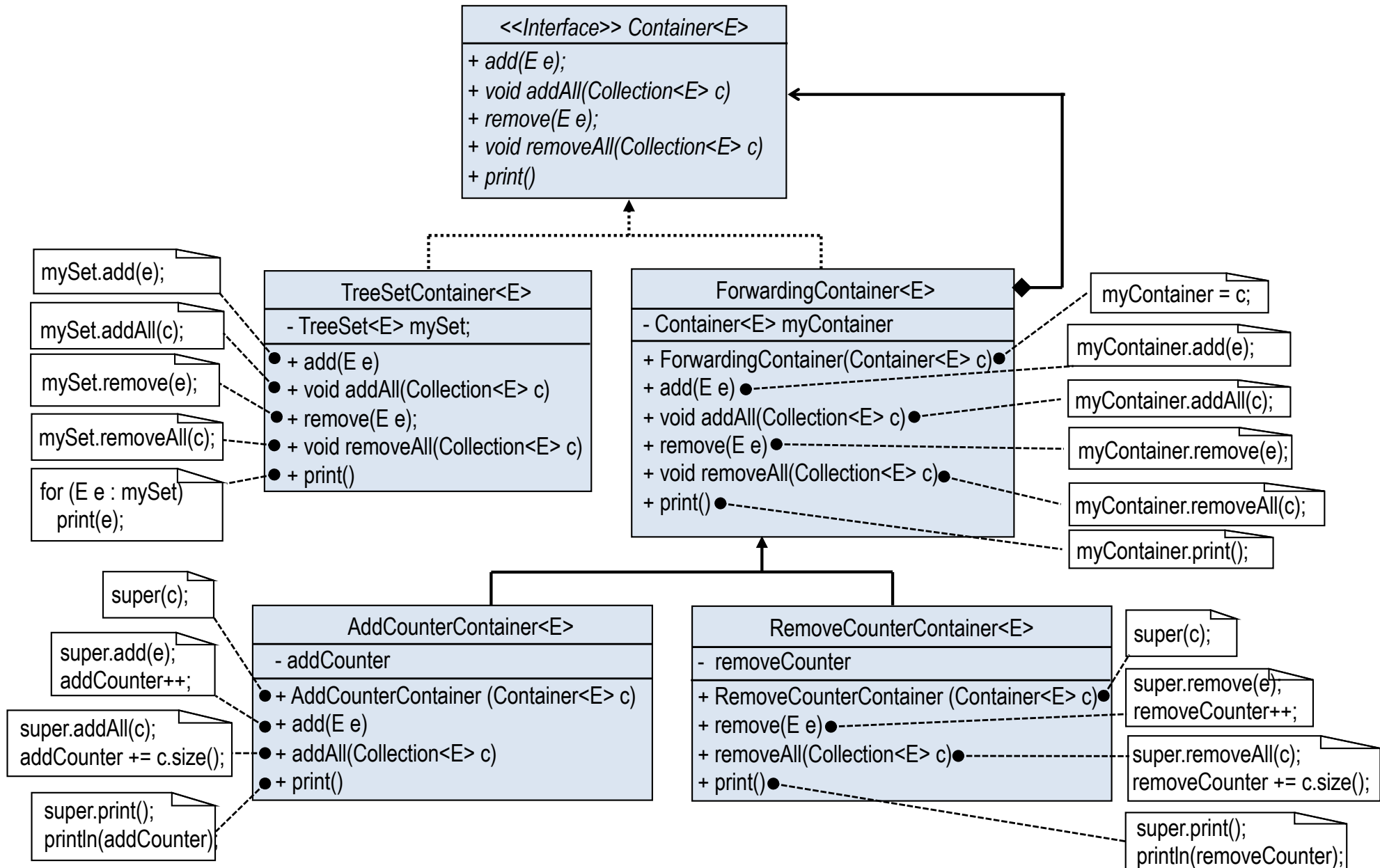


Zweck und Struktur des Dekorierer-Musters

- Eine Komponente soll **dynamisch** um **verschiedene Fähigkeiten** erweitert werden.
- Ein Dekorierer-Objekt umhüllt eine Komponente und delegiert einen operation()-Aufruf an die eingehüllte Komponente.
- Ein Dekorierer wird dann um eine zusätzliche Fähigkeit erweitert zu einem konkreten Dekorierer.
- Dekorierer sind eine flexible Alternative zur Vererbung.

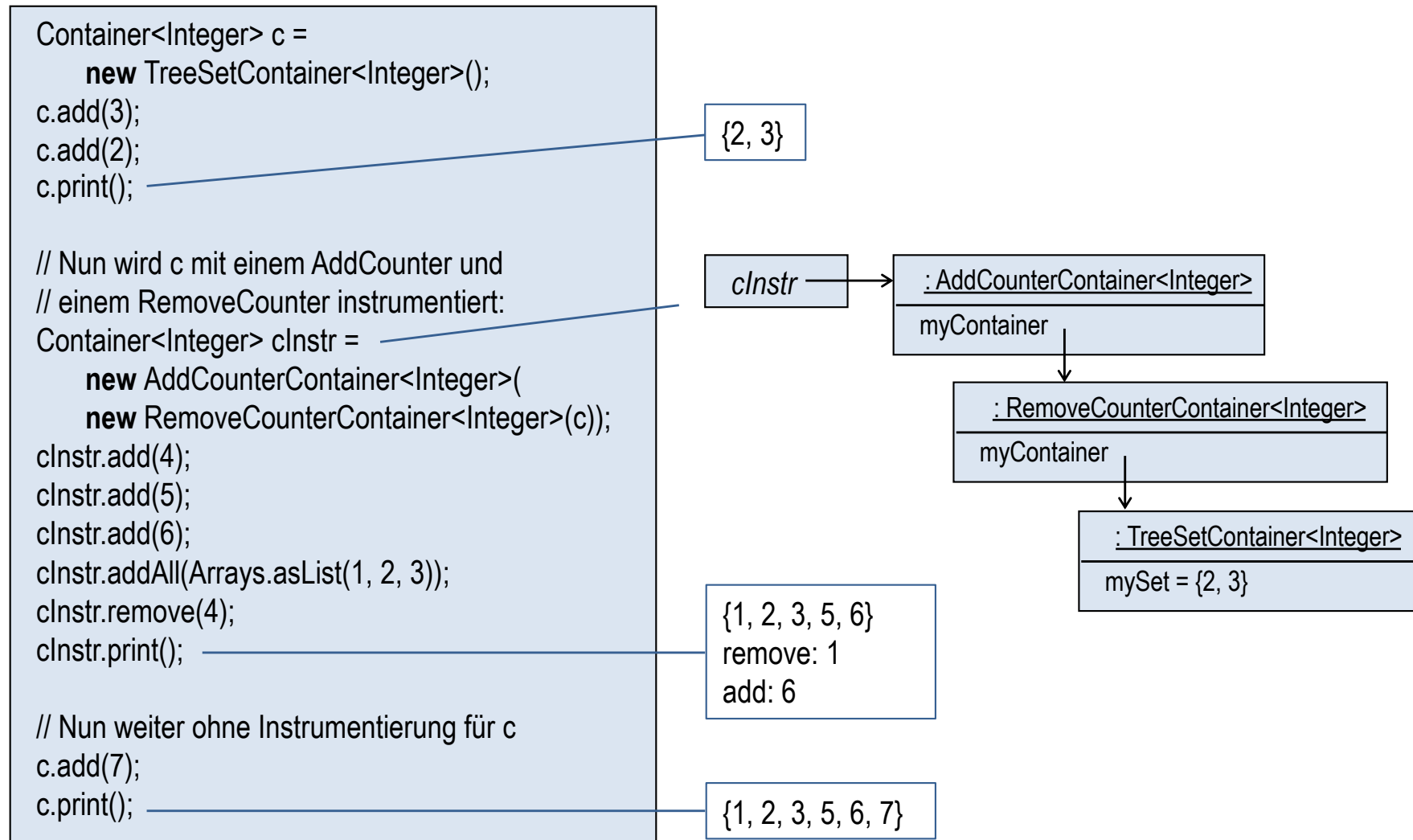


Beispiel: Container mit Add bzw. Remove-Counter (1)



Beispiel: Container mit Add bzw. Remove-Counter (2)

- Damit kann ein Container dynamisch (zur Laufzeit) mit einem AddCounter und/oder einem RemoveCounter instrumentiert werden.

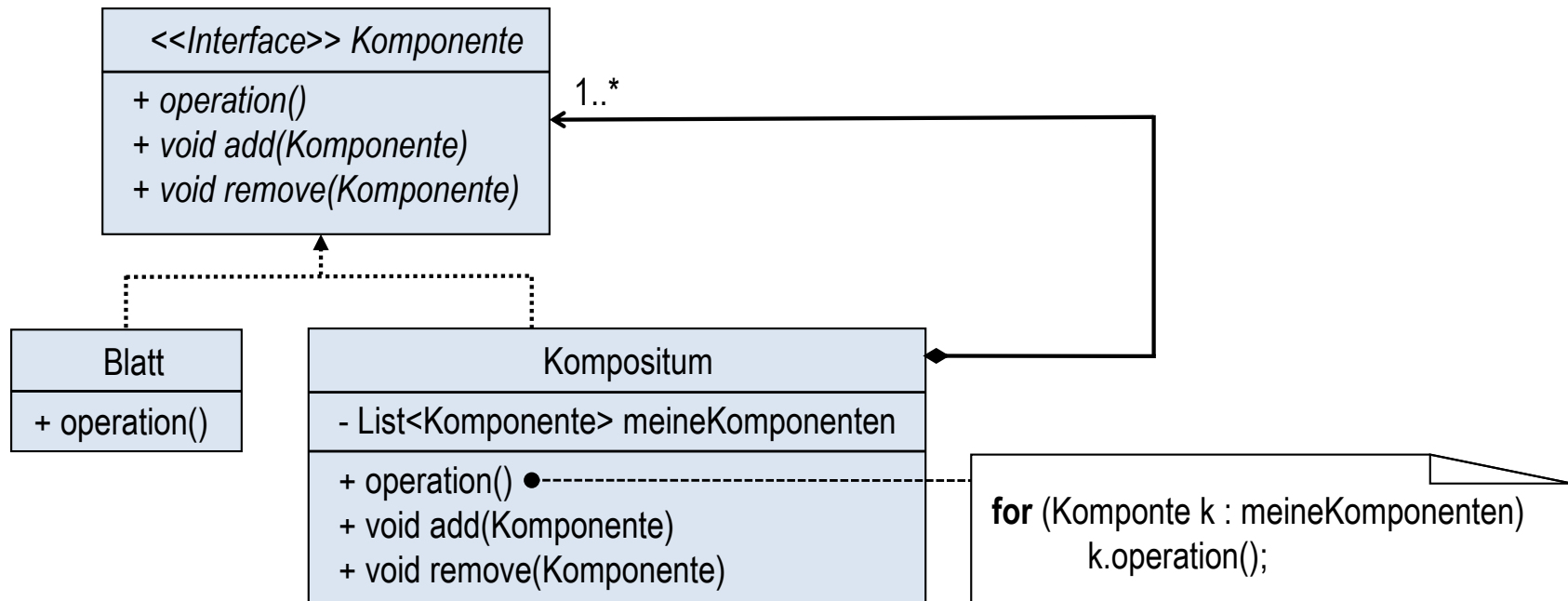


Kapitel 15: Entwurfsmuster

- Adapter
- Assoziationen
- Beobachter (Observer)
- Dekorierer (Decorator)
- Kompositum (Composite)
- Schablonenmethode (Template Method)
- Strategie (Strategy)

Zweck und Struktur

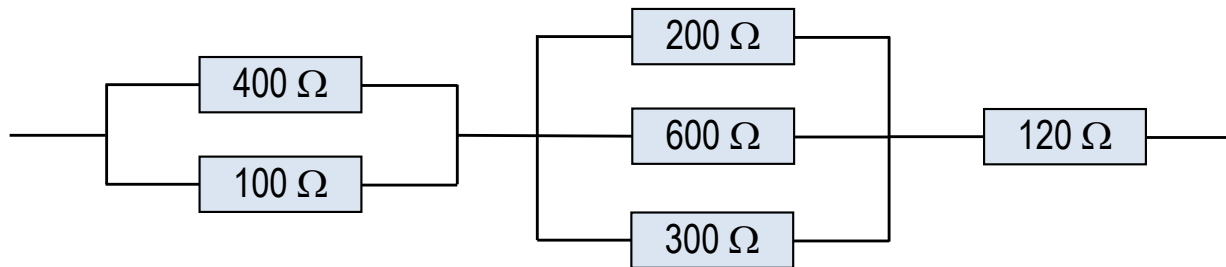
- Füge Objekte (Bausteine, Komponenten) zu Baumstrukturen (Teil-Ganze-Hierarchien) zusammen.



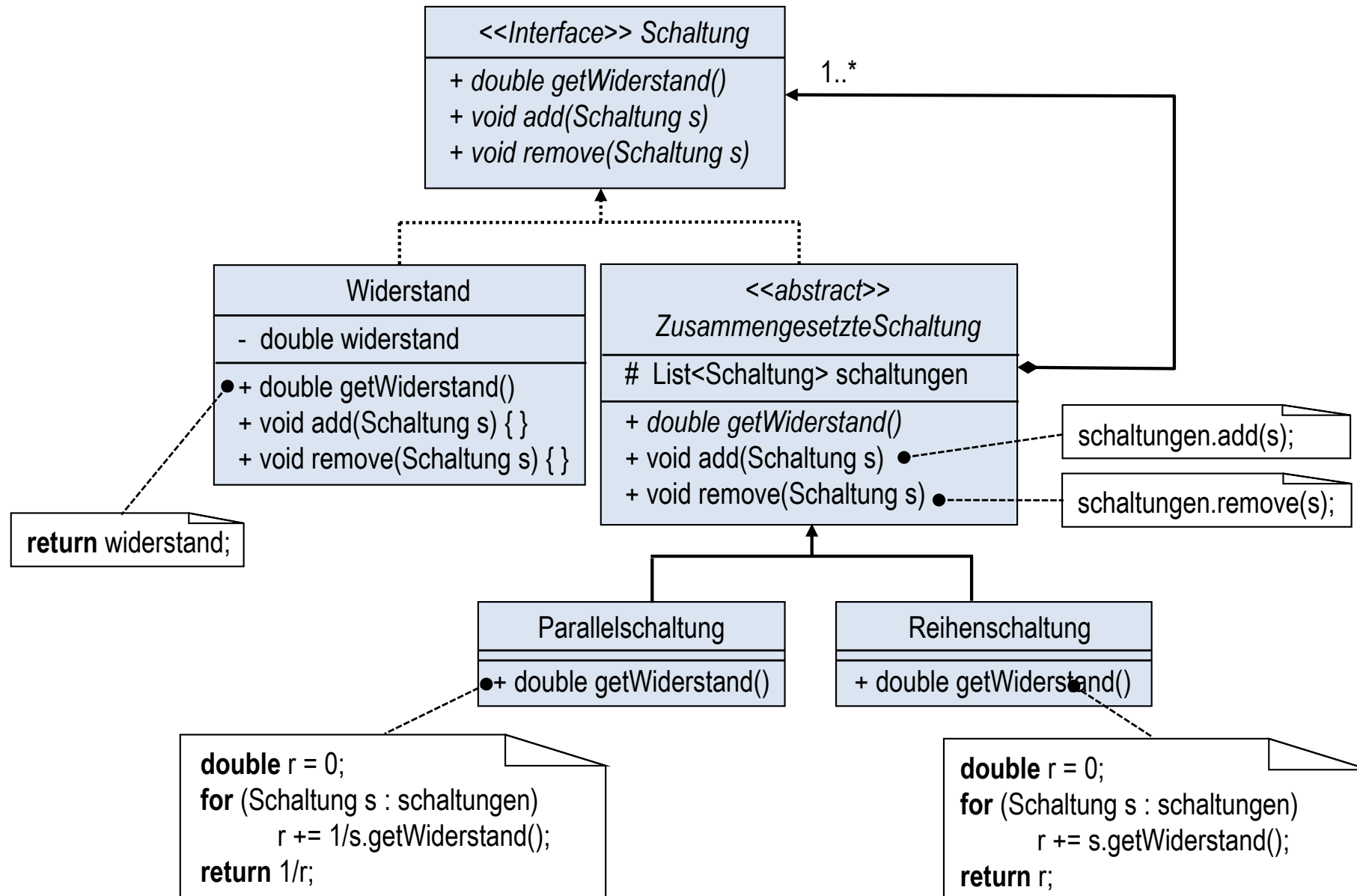
- Eine Komponente ist entweder ein Blatt (elementarer Baustein) oder ein Kompositum (zusammengesetzter Baustein).
- Ein Kompositum kann aus beliebig vielen Komponenten bestehen.
- Nur für ein Kompositum sind die Methoden `add` und `remove` relevant. Für ein Blatt sind diese Methoden leer implementiert.
- Die Methode `operation()` wird rekursiv auf die Komponenten zurückgeführt.

Beispiel Elektrische Schaltung (1)

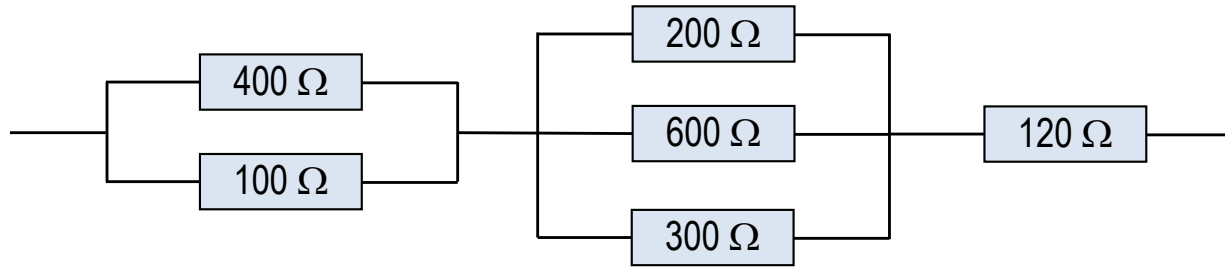
- Eine elektrische Schaltung besteht aus einer Menge von Widerständen, die entweder parallel oder in Reihe geschaltet sind.
- Bei in Reihe geschalteten Widerständen addieren sich die Widerstandswerte.
- Bei einer Parallelschaltung ergibt sich der Kehrwert des Gesamtwiderstands aus der Summe der Kehrwerte der einzelnen Widerstände.
- Im Beispiel:
$$\text{Gesamtwiderstand} = 1/(1/400 + 1/100) + 1/(1/200 + 1/600 + 1/300) + 120 = 300\Omega$$



Beispiel Elektrische Schaltung (2)



Beispiel Elektrische Schaltung (3)



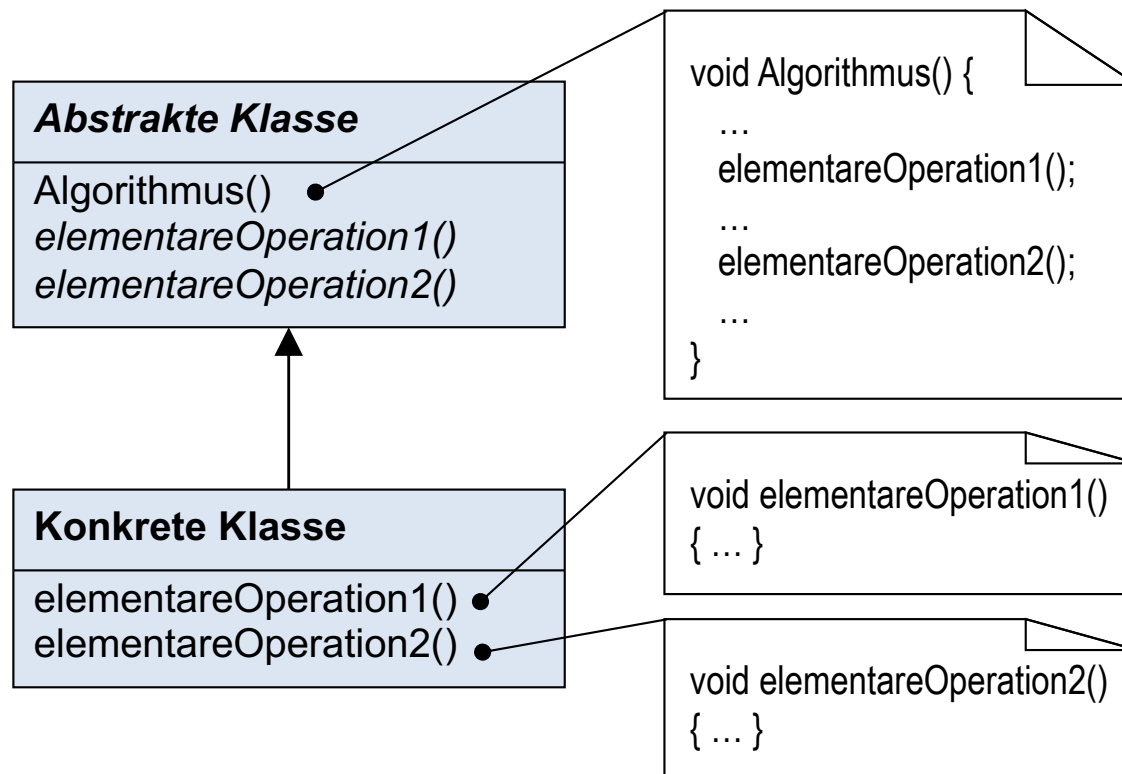
```
Schaltung ps1 = new Parallelschaltung();  
ps1.add(new Widerstand(400));  
ps1.add(new Widerstand(100));  
  
Schaltung ps2 = new Parallelschaltung();  
ps2.add(new Widerstand(200));  
ps2.add(new Widerstand(600));  
ps2.add(new Widerstand(300));  
  
Schaltung rs = new Reihenschaltung();  
rs.add(ps1);  
rs.add(ps2);  
rs.add(new Widerstand(120));  
  
System.out.println(rs.getWiderstand()); // 300
```


Kapitel 15: Entwurfsmuster

- Adapter
- Assoziationen
- Beobachter (Observer)
- Dekorierer (Decorator)
- Kompositum (Composite)
- Schablonenmethode (Template Method)
- Strategie (Strategy)

Zweck und Struktur

- Es soll das Skelett eines Algorithmus (**Schablonenmethode**) implementiert werden, wobei noch einige elementare Operationen offen d.h. abstrakt bleiben sollen.
- Die elementare Operationen (**Einschubmethoden**) werden dann in den Unterklassen konkretisiert.



Beispiel Benchmark

- Verwendung einer abstrakten Klasse zur Realisierung eines Frameworks für Laufzeitmessungen von Benchmark-Funktionen.
- (aus *The Java Programming Language* von Gosling u.a.; größeres Beispiel in Abschnitt 3.11.1)

```
abstract class Benchmark {  
  
    abstract void benchmark();  
  
    public final long repeat(int n) {  
        long start = System.nanoTime();  
        for (int i = 0; i < n; i++)  
            benchmark();  
        return System.nanoTime() - start;  
    }  
}
```

einfachheitshalber
eine leere Benchmark-
Funktion

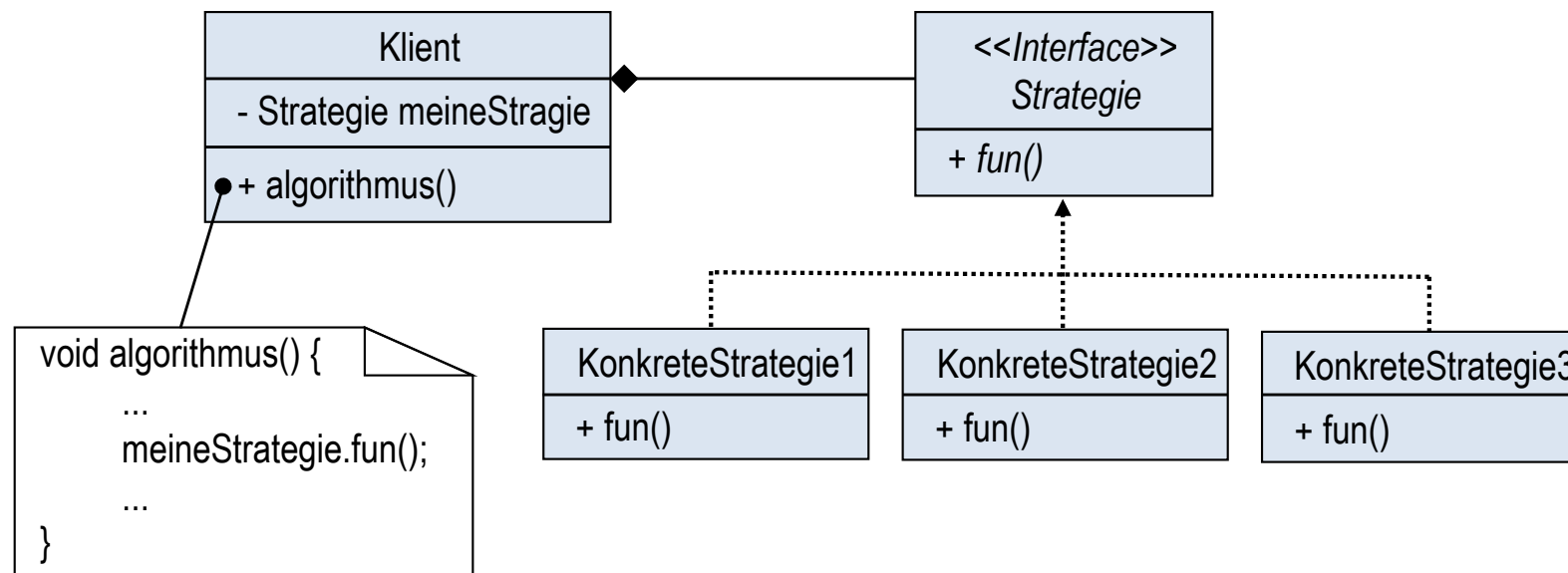
```
class MethodBenchmark extends Benchmark{  
  
    void benchmark() {}  
  
    public static void main (String[] args) {  
        int n = Integer.parseInt(args[0]);  
        long time = new MethodBenchmark().repeat(n);  
        System.out.println(n + " method in " +  
                           time + " nanosec");  
    }  
}
```

Kapitel 15: Entwurfsmuster

- Adapter
- Assoziationen
- Beobachter (Observer)
- Dekorierer (Decorator)
- Kompositum (Composite)
- Schablonenmethode (Template Method)
- Strategie (Strategy)

Zweck und Struktur

- Eine Klientenklasse hat einen Algorithmus, der an einer bestimmten Stelle zur Laufzeit variiert werden soll.
- Die variablen Teile werden jeweils als eigene Klasse in einer Methode gekapselt. All diese Klassen implementieren eine gemeinsame Schnittstelle - Strategie genannt.
- Der Klientenalgorithmus benutzt eine der Varianten über die Strategieschnittstelle.



Beispiel: Textanalyse mit verschiedenen Filtern

