RUHR-UNIVERSITÄT BOCHUM
Lehrstuhl für Informatik im Bauwesen

RUB

Programmierung und Programmiersprachen

Sommersemester 2023

Observer-Muster

Entwurfsmuster – Observer-Muster

Probleme bei Laufzeiterweiterung

Gegeben sind folgende Klassen für eine Wetterstation-Anwendung

Wetterstation
– id: String – luftfeuchtigkeit: double – temperatur: double – luftdruck: double
+ Wetterstation(id: String): void + messwerteGeaendert(): void *// Getter & Setter* ...

WetterAnzeige
– id: String
+ WetterAnzeige(id: String): void + aktualisiereAnzeige(luftfeuchtigkeit: double, temperatur: double, luftdruck: double): void

Wie können alle Anzeigen aktuell gehalten werden?

Entwurfsmuster – Observer-Muster

Probleme bei Laufzeiterweiterung

Naiver Ansatz: Wait-Loop

```
while (1) {
   for (WetterAnzeige anzeige: wetteranzeigen) {
      anzeige.aktualisiereAnzeige(
         station.getLuftfeuchtigkeit(),
         station.getTemperatur(),
         station.getLuftdruck()
      );
   }
   Thread.sleep(1000);
}
```

Entwurfsmuster – Observer-Muster

Probleme bei Laufzeiterweiterung

Naiver Ansatz: Wait-Loop

```
while (1) {
   for (WetterAnzeige anzeige: wetteranzeigen) {
      anzeige.aktualisiereAnzeige(
         station.getLuftfeuchtigkeit(),
         station.getTemperatur(),
         station.getLuftdruck()
      );
   }
   Thread.sleep(1000);
}
```

- Anzeigen werden nicht aktualisiert zwischen Intervallen
- Thread ist blockiert
- Unnötige Updates auch wenn sich keine Daten ändern

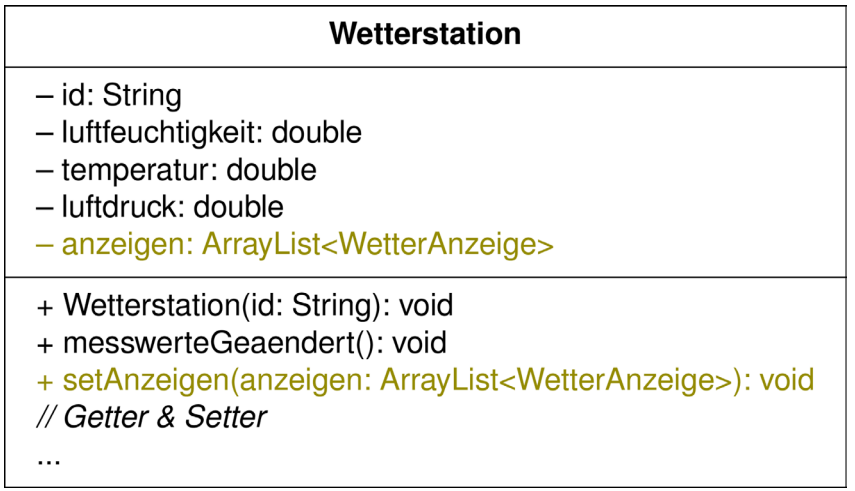
Entwurfsmuster – Observer-Muster

Probleme bei Laufzeiterweiterung

Besserer Ansatz: Die Wetterstation benachrichtigt die Anzeigegeräte wenn sich die Werte ändern.

Dazu muss die Wetterstation die Anzeigegeräte kennen, damit die Methoden zur Aktualisierung der Anzeige aufgerufen werden können

Welche Erweiterungen müssen vorgenommen werden?

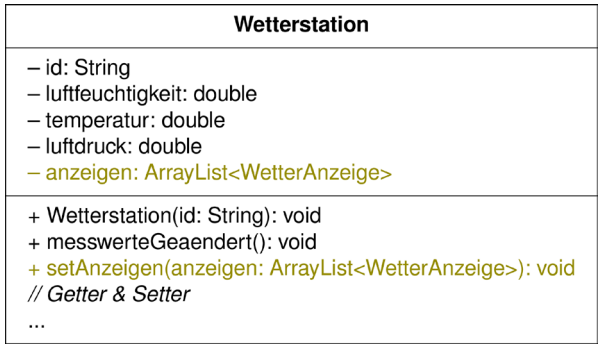


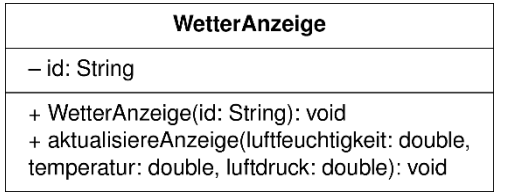
Entwurfsmuster – Observer-Muster

Probleme bei Laufzeiterweiterung

Besserer Ansatz: Die Wetterstation benachrichtigt die Anzeigegeräte wenn sich die Werte ändern.

```
public class Wetterstation {

    private ArrayList<WetterAnzeige> anzeigen
        = new ArrayList<WetterAnzeige>();

    ...

    public void messwerteGeaendert() {

        for (WetterAnzeige anzeige : this.anzeigen) {

            anzeige.aktualisiereAnzeige(this.getLuftfeuchtigkeit(),
                this.getTemperatur(), this.getLuftdruck());

        }
    }
}
```



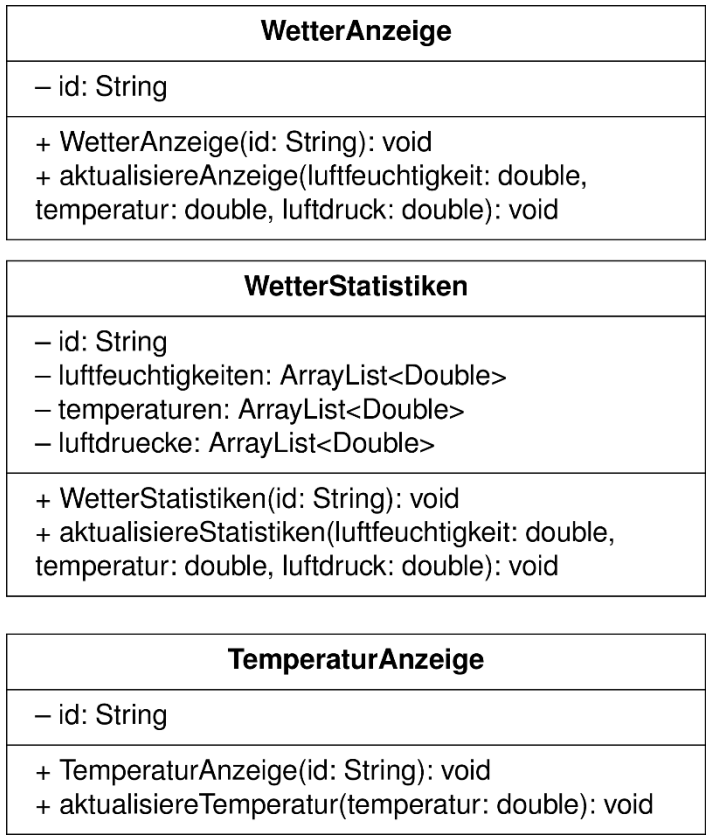


Entwurfsmuster – Observer-Muster

Probleme bei Laufzeiterweiterung

Können neue Anzeigegeräte zur Laufzeit hinzugefügt oder entfernt werden?

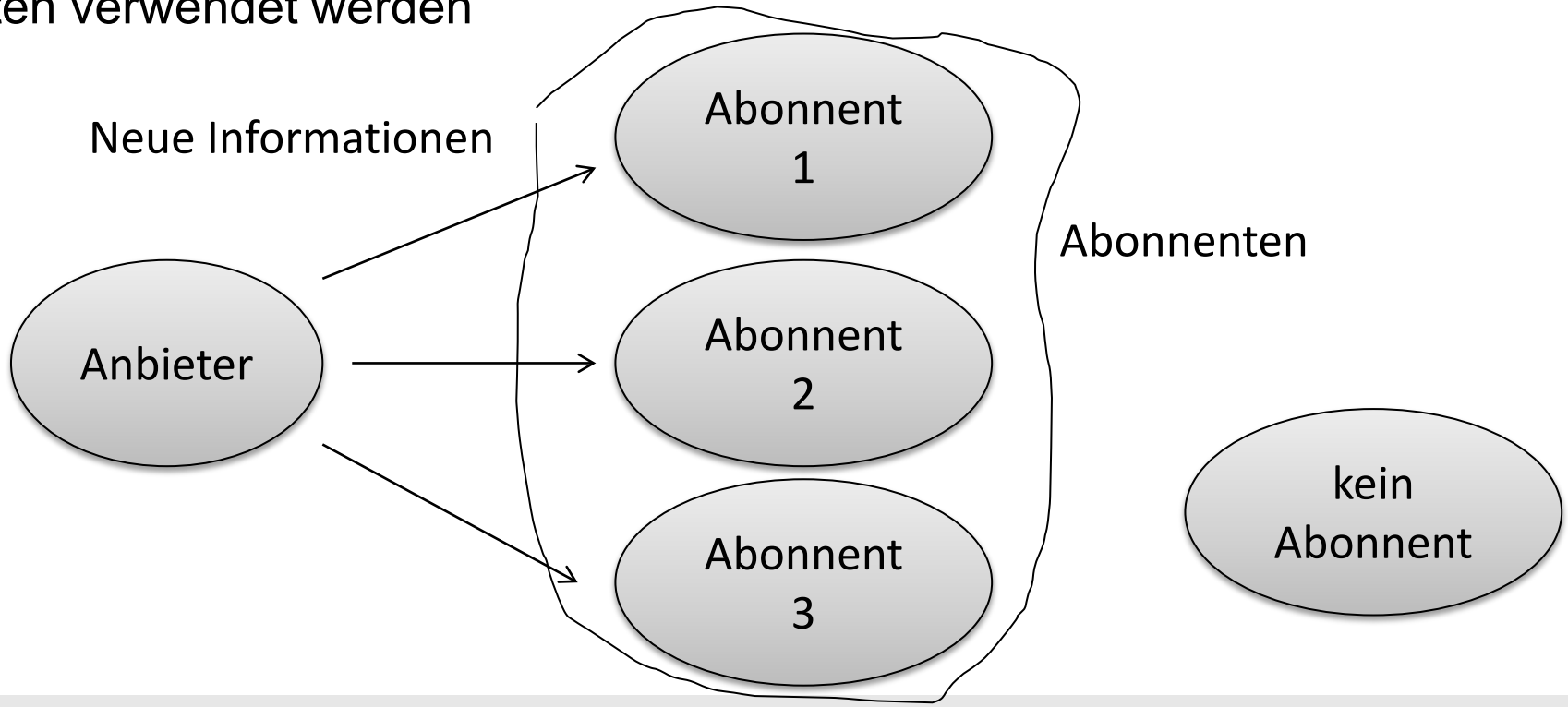
Können andere Anzeigegeräte (z.B. nur Temperatur, Statistiken, etc.) einfach integriert werden?



Entwurfsmuster – Observer-Muster

Anbieter + Abonnenten

Prinzipiell ermittelt eine Wetterstation verschiedene Wetterdaten und bietet diese verschiedenen Systemen zur eigenen Nutzung an. Die Systeme sollten jedoch selber entscheiden können, ob und wie diese Daten verwendet werden



Entwurfsmuster – Observer-Muster

Anbieter + Abonnenten

Anbieter werden häufig *Subjekt* oder *Observable* genannt. Die Abonnenten hingegen *Beobachter* oder *Observer*.

Welche Funktionalitäten werden benötigt?

- Anmeldung beim Anbieter
- Abmeldung beim Anbieter
- Informiere Abonnenten

An dieser Stelle sollten am besten Schnittstellen definiert werden, um einen standardisierten Zugriff zu ermöglich

Das **Observer-Muster** definiert eine 1:n Abhängigkeit zwischen Objekten in der Art, dass alle abhängigen Objekte benachrichtigt werden, wenn sich der Zustand des einen Objektes ändert.

Entwurfsmuster – Observer-Muster

Klasse für Wetterdaten

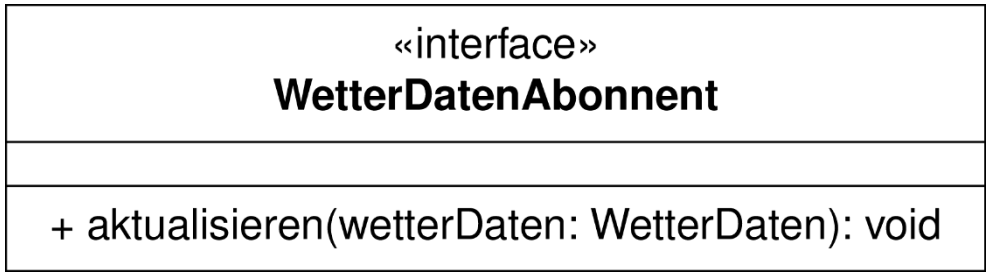
Zur Übermittlung von komplexen Daten sollten in der Regel eigene Klasse verwendet werden. Es wird eine Klasse für Wetterdaten eingeführt. Die Objekte speichern Informationen zur Luftfeuchtigkeit, Temperatur und Luftdruck

WetterDaten
– luftfeuchtigkeit: double – temperatur: double – luftdruck: double
+ WetterDaten(luftfeuchtigkeit: double, temperatur: double, luftdruck: double): void + getLuftfeuchtigkeit(): double + getTemperatur(): double + getLuftdruck(): double

Entwurfsmuster – Observer-Muster

Schnittstelle Abonnent

Die Abonnenten sollen informiert werden, wenn neue Wetterdaten vorliegen. Es soll eine entsprechende Methode aufgerufen werden. Der Methode werden die neuen Wetterdaten übergeben



Entwurfsmuster – Observer-Muster

Schnittstelle Anbieter

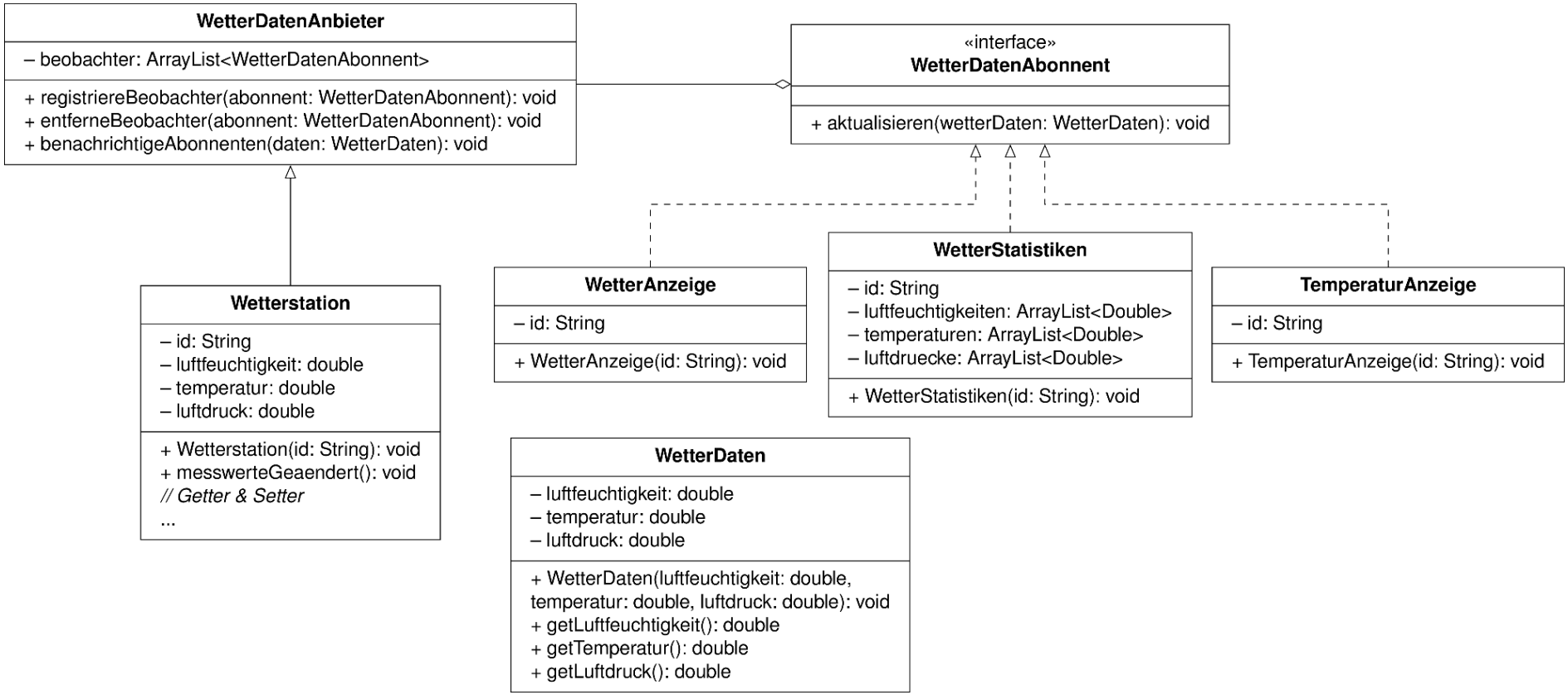
Der Anbieter verwaltet seine Abonnenten, welche die entsprechende Schnittstelle implementieren müssen. Bei neuen Wetterdaten werden die entsprechenden Abonnenten benachrichtigt

WetterDatenAnbieter
– beobachter: ArrayList<WetterDatenAbonnent>
+ registriereBeobachter(abonnent: WetterDatenAbonnent): void + entferneBeobachter(abonnent: WetterDatenAbonnent): void + benachrichtigeAbonnenten(daten: WetterDaten): void

Entwurfsmuster – Observer-Muster

Neue Klassenhierarchie

Jede Wetterstation verwaltet seine eigenen Abonnenten



Entwurfsmuster – Observer-Muster

Implementierung (Auszug)

```
public class WetterDatenAnbieter {
    ...
    // Alle Abonnenten - gleich eine leere Liste erstellen
    private ArrayList<WetterDatenAbonnent> abonnenten
       = new ArrayList<WetterDatenAbonnent>();

    // Einen neuen Abonnenten eintragen
    public void registriereBeobachter(WetterDatenAbonnent abonnent) {
       // ArrayList kann doppelte Elemente haben
       if (!this.abonnenten.contains(abonnent)) {
          this.abonnenten.add(abonnent);
       }
    }
    ...
}
```

Entwurfsmuster – Observer-Muster

Implementierung (Auszug)

```
public class WetterDatenAnbieter {
    ...
    // Abonnenten benachrichtigen
    public void benachrichtigeAbonnenten(WetterDaten wetterDaten) {
        // ArrayList kann doppelte Elemente haben
        for (WetterDatenAbonnent abonnent : this.abonnenten) {
            abonnent.aktualisieren(wetterDaten);
        }
    }
}
```

Entwurfsmuster – Observer-Muster

Implementierung (Auszug)

```
public class Wetterstation extends WetterDatenAnbieter {
    ...
    // Messwerte geändert
    public void messwerteGeaendert() {
        // Neue Wetterdaten erzeugen
        benachrichtigeAbonnenten(new WetterDaten(
            getLuftfeuchtigkeit(), getTemperatur(), getLuftdruck()));
    }
    ...
}
```

Die Funktion **`benachrichtigeAbonnenten(...)`** sollte immer aufgerufen werden, wenn sich Daten ändern, z.B. auch in Setter-Methoden.

Entwurfsmuster – Observer-Muster

Implementierung (Auszug)

```
public class WetterAnzeige implements WetterDatenAbonnent {
   ...
   // Aktualisierung vornehmen
   public void aktualisieren(WetterDaten wetterDaten) {
      // Einfach in der Konsole ausgeben
      System.out.println("Wetteranzeige " +getId());
      System.out.println("Luftfeuchtigkeit: " +
         wetterDaten.getLuftfeuchtigkeit());
      System.out.println("Temperatur: " +
         wetterDaten.getTemperatur());
      System.out.println("Luftdruck: " +
         wetterDaten.getLuftdruck());
      System.out.println("---------------------------------");
   }
   ...
}
```

Entwurfsmuster – Observer-Muster

Testprogramm

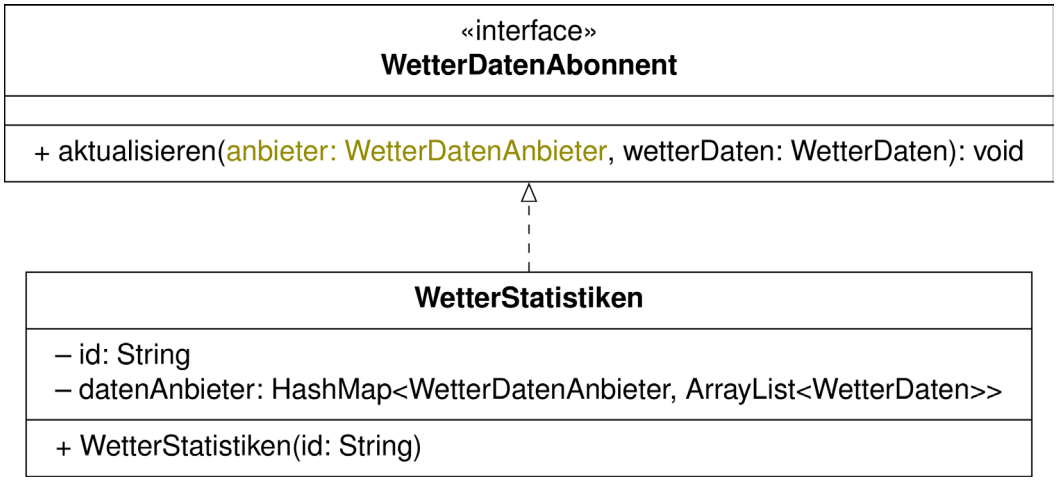
```
public class WetterTestProgramm {
   public static void main(String[] args) {

      WetterStation station = new WetterStation("Station");

      WetterAnzeige anzeige = new WetterAnzeige("Anzeige");
      WetterStatistiken statistiken =
         new WetterStatistiken("Statistiken");

      station.registriereBeobachter(anzeige);
      station.registriereBeobachter(statistiken);

      station.setTemperatur(21.5);

      station.entferneBeobachter(anzeige);

      station.setLuftfeuchtigkeit(52.3);
   }
}
```

Entwurfsmuster – Observer-Muster

Mögliche Erweiterung

Manchmal sollte ein Beobachter wissen, von welchem Anbieter eine bestimmte Information kommt. In solchen Fällen sollte immer auch der Anbieter an die Methode zur Aktualisierung übergeben werden.

Beispiel: Eine Wetterstatistik soll Daten für mehrere Wetterstationen in einer Region einzeln und zusammen auswerten können. D.h. die Wetterdaten sollten je Wetterstation gespeichert werden.



Entwurfsmuster – Observer-Muster

Implementierung (Auszug)

```
public class WetterStatistiken implements WetterDatenAbonnent {
    ...
    // Aktualisierung vornehmen
    public void aktualisieren(WetterDatenAnbieter anbieter,
                                                WetterDaten wetterDaten) {
        // Daten holen
        ArrayList<WetterDaten> daten = this.datenAnbieter.get(anbieter);
        // Noch keine Daten vorhanden?
        if (daten == null) {
            // Speicher erzeugen
            daten = new ArrayList<WetterDaten>();
            // und ablegen
            this.datenAnbieter.put(anbieter, daten)
        }
        // Neue Daten einfügen
        daten.add(wetterDaten)
        // Jetzt können die Statisitken aktualisiert werden
        ...
    }
}
```

Entwurfsmuster – Observer-Muster

Fazit

Es handelt sich beim Observer-Muster um eine lose Kopplung von Objekten zur Laufzeit.

Das Observable (Anbieter) weiß über den Observer (Beobachter) nur, dass er eine bestimmte Schnittstelle implementiert.

Die Observable-Klasse muss nicht erweitert werden, um neue Observer-Klassen zu integrieren, so lange diese die Schnittstelle implementieren.

Observable und Observer können auch unabhängig voneinander wieder verwendet werden.

In der Regel sollte immer eine lose Kopplung angestrebt werden. Eine lose Kopplung ermöglicht die flexible Implementierung von komplexen Softwaresystemen.

RUHR-UNIVERSITÄT BOCHUM
Lehrstuhl für Informatik im Bauwesen

RUB

Programmierung und Programmiersprachen

Sommersemester 2023

Java Collection Framework

Datenstrukturen – Mengen

Java Collection Framework

Das Java Collection Framework ist eine Menge von Klassen und Interfaces für häufig verwendete Datenstrukturen

Das Basis-Interface Collection definiert Methoden, die für alle Datenstrukturen verfügbar sein sollten

Overview Package Class Use Tree Deprecated Index Help
PREV CLASS NEXT CLASS FRAMES NO FRAMES
SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

Java™ Platform Standard Ed. 6

java.util
Interface Collection<E>

All Superinterfaces:
Iterable<E>

All Known Subinterfaces:
BeanContext, BeanContextServices, BlockingDeque<E>, BlockingQueue<E>, Deque<E>, List<E>, NavigableSet<E>, Queue<E>, Set<E>, SortedSet<E>

All Known Implementing Classes:
AbstractCollection, AbstractList, AbstractQueue, AbstractSequentialList, AbstractSet, ArrayBlockingQueue, ArrayDeque, ArrayList, AttributeList, BeanContextServicesSupport, BeanContextSupport, ConcurrentLinkedQueue, ConcurrentSkipListSet, CopyOnWriteArrayList, CopyOnWriteArraySet, DelayQueue, EnumSet, HashSet, JobStateReasons, LinkedBlockingDeque, LinkedBlockingQueue, LinkedHashSet, LinkedList, PriorityBlockingQueue, PriorityQueue, RoleList, RoleUnresolvedList, Stack, SynchronousQueue, TreeSet, Vector

```
public interface Collection<E>
extends Iterable<E>
```

The root interface in the *collection hierarchy*. A collection represents a group of objects, known as its *elements*. Some collections allow duplicate elements and others do not. Some are ordered and others unordered. The JDK does not provide any *direct* implementations of this interface: it provides implementations of more specific subinterfaces like `Set` and `List`. This interface is typically used to pass collections around and manipulate them where maximum generality is desired.

Bags or *multisets* (unordered collections that may contain duplicate elements) should implement this interface directly.

Datenstrukturen – Mengen

Mengen

Disjunkte Zusammenfassung von eindeutig unterscheidbaren Objekte bzw. Elementen mit gleichem Datentypen:

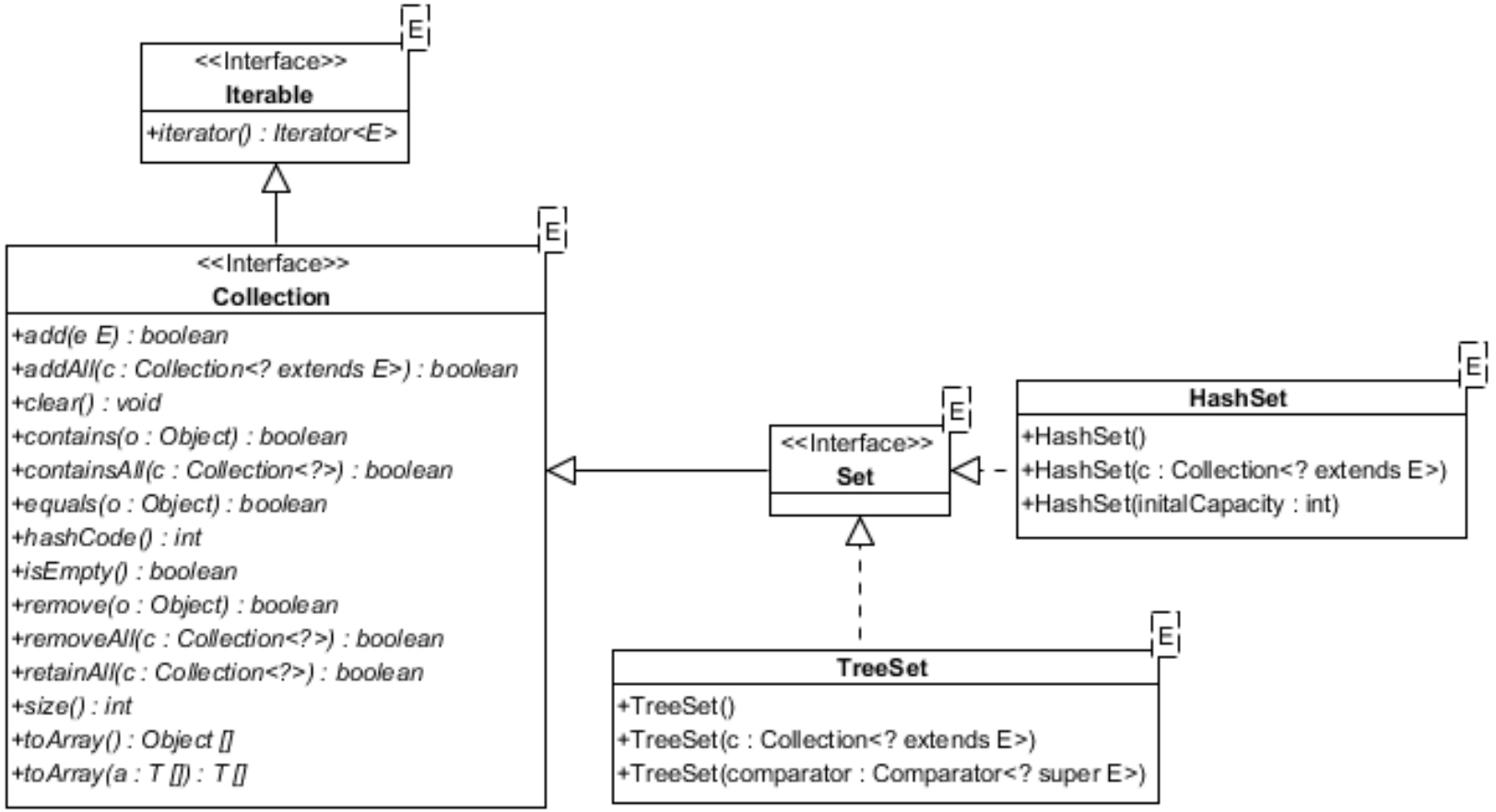
- Jedes Element ist eindeutig identifizierbar (z.B. Speicheradresse)
- Keine doppelten Elemente sind enthalten

Beispiele:

- Menge von Zahlen
- Menge von Studierenden
- Menge von Punkten

Datenstrukturen – Mengen

Schnittstelle für Mengen



Datenstrukturen – Mengen

Klasse HashSet

Die Schnittstelle `Set<E>` kann von verschiedenen Klassen implementiert werden. Ein leistungsfähiges Verfahren zum Speichern und Suchen von Objekten ist das Hash-Verfahren auf Basis eines Feldes.

```
// Klasse für Mengen auf Basis des Hash-Verfahrens
public class HashSet<E> implements Set<E> {

    // Feld mit den Elementen
    private E[] elements;

    // Konstruktur für eine leere Menge
    public HashSet() {
        this.elements = new E[0];
    }
    // ...
}
```

Datenstrukturen – Mengen

Beispiel – Menge von Punkten

```
public class PointSetDemo {

      public static void main(String[] args) {

            // Menge von 100 zufälligen Punkten im Intervall [0, 1.0]
            Set<Point2D> points = new HashSet<Point2D>();
            for (int i=0; i<100; i++) {
                points.add(new Point2D(Math.random(), Math.random());
            }

            // Durchlaufe die Menge und geben jeden Punkt aus
            for (Point2D p : points) {
                System.out.println(p);
            }
      }
}
```

Die Punkte werden in der Regel **nicht** in der Reihenfolge ausgegeben, in der sie hinzugefügt wurden.

Datenstrukturen – Mengen

Hash-Verfahren

- Die Klasse `HashSet` verwendet prinzipiell zur Speicherung ein eindimensionales Feld:

- Die Reihenfolge der Element einer Menge ist nicht fest vorgegeben, daher kann auf ein Element nicht über einen Index zu gegriffen werden.
- Um einen schnellen Zugriff zu ermöglichen, existiert zu jedem Objekt ein Schlüssel (der Hashcode) und ein Verfahren zur Berechnung eines Indizes zur Speicherung.
- Ist der Hashcode nicht explizit definiert bzw. programmiert, wird die Speicheradresse als Schlüssel verwendet.

Datenstrukturen – Mengen

Hash-Verfahren

- Das Feld eines `HashSet`-Objektes besitzt `n` Speicherplätze.
- Jedes Objekt besitzt einen Schlüssel `k`.
- Eine einfache Hash-Funktion `h` wäre:

```
h(k) = k mod n
```

- Jedem Schlüssel wird eindeutig ein Index `a` im Feld zugeordnet.
- Es können jedoch Kollisionen auftreten, die speziell behandelt werden müssen.
- Sollen mehr Elemente eingefügt werden als Speicherplätze vorhanden sind, wird das Feld automatisch vergrößert.

n=8

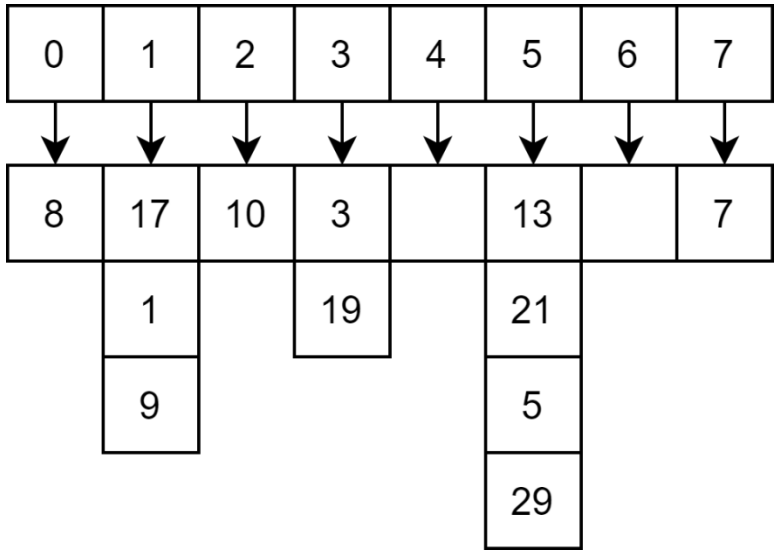
k	h(k)
10	2
12	4
20	4
22	6
52	4
54	6

Datenstrukturen – Mengen

Hash-Verfahren - Kollisionsbewältigung

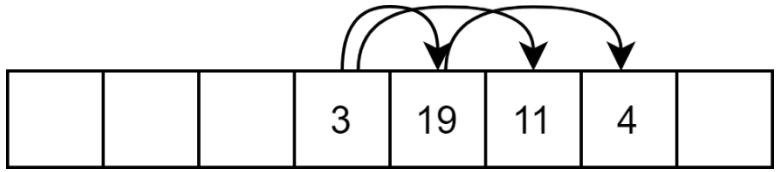
Verkettung

- Jeder Eintrag ist eine Liste. Bei Kollisionen werden Elemente mit gleichem Hash in die selbe Liste gesetzt.



Offene Adressierung

- Kollidierende Elemente werden auf den nächsten freien Platz geschoben.



Datenstrukturen – Mengen

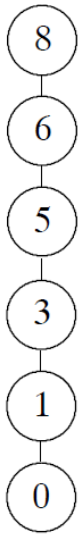
Sortierte Menge

Für einige Fragestellungen sollen die Elemente einer Menge sortiert nach einer bestimmten Ordnungsrelation durchlaufen werden können.

Eine Ordnungsrelation definiert einen relationalen Ausdruck $x \sqsubset y$ für ein Elementpaar x, y, der zum Vergleich der Element verwendet wird.

Beispiel: Menge mit ganzen Zahlen

- M = {3, 8, 5, 0, 6, 1}
- Ordnungsrelation $x \sqsubset y \coloneqq x < y$
- Geordnete Menge O = {0, 1, 3, 5, 6, 8}
- Totale Ordnung
  für $x \neq y$ gilt $x \sqsubset y$ oder $y \sqsubset x$

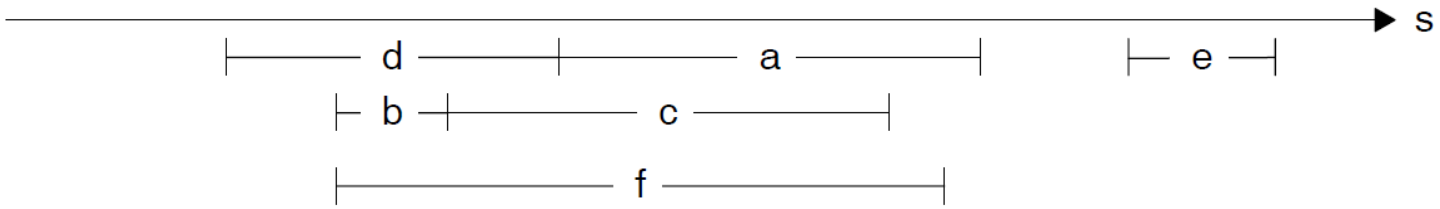


Ordnungs-
diagramm

Datenstrukturen – Mengen

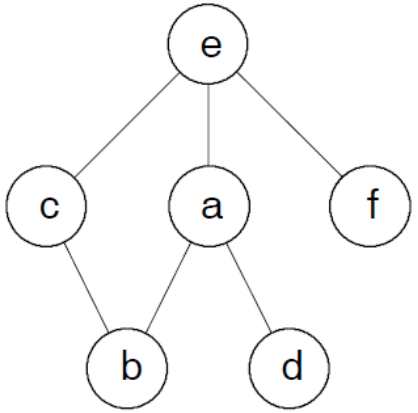
Sortierte Menge

Beispiel: Menge mit Intervallen (Anfang x_a, Ende x_b, mit $x_a < x_b$)



- M = {a, b, c, d, e, f}
- Ordnungsrelation $x \sqsubset y := x_b \leqq y_a$
- Geordnete Mengen
  O = {b, d, f, c, a, e}, O = {d, f, b, a, c, e}, ...
- Strenge Ordnung
  für $x \neq y$ gilt weder $x \sqsubset y$ noch $y \sqsubset x$

Ordnungsdiagramm

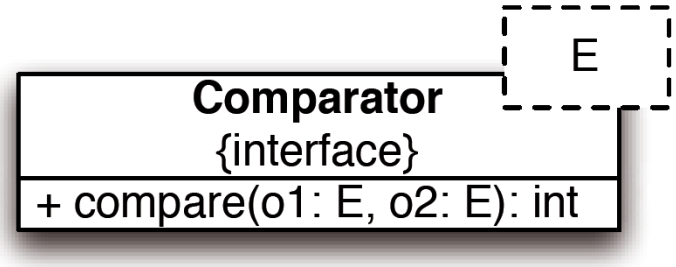


Datenstrukturen – Mengen

Schnittstelle Comparator

Die Schnittstelle `Comparator` definiert eine Vergleichsfunktion für zwei Objekte der gleichen Klasse:

```
// Schnittstelle für Vergleichsklassen
public interface Comparator<E> {

    // Vergleich von zwei Elementen
    public int compare(E o1, E o2);
}
```



Ganzzahliger Rückgabewert wird erwartet

- $x \sqsubset y$ ⇒ compare(x, y) < 0
- $x = y$ ⇒ compare(x, y) = 0
- $\dashv (x \sqsubset y \ \vee \ x = y)$ ⇒ compare(x, y) > 0

Datenstrukturen – Mengen

Beispiel Comparator für Intervalle

```
// Klasse für eindimensionale Intervalle
public class Interval {

    private double a, b;

    public Interval(double a, double b) {
        this.a = a;
        this.b = b;
    }

    public double getBegin() {
        return this.a;
    }

    public double getEnd() {
        return this.b;
    }
}
```

Datenstrukturen – Mengen

Beispiel Comparator für Intervalle

```
// Klasse für den Vergleich von Intervallen
public class IntervalComparator implements Comparator<Interval> {

    public IntervalComparator() {}

    public int compare(Interval i1, Interval i2) {

        if (i1.getEnd() <= i2.getBegin()) {
            return -1;
        }
        if (i1.getBegin() == i2.getBegin()
                && i1.getEnd() == i2.getEnd()) {
            return 0;
        }
        return 1;
    }
}
```

Datenstrukturen – Mengen

Schnittstelle SortedSet

Für sortierte Menge sind spezielle zusätzliche Methoden sinnvoll, die durch die Schnittstelle `SortedSet` beschrieben werden:

```
// Schnittstelle für Vergleichsklassen
public interface SortedSet<E> {

    // Liefert das erste Element zurück
    public E first();

    // Liefert das letzte Element zurück
    public E last();

    // Liefert alle Elemente zwischen zwei Elementen zurück
    // inkl. fromElement, exkl. toElement
    public SortedSet<E> subSet(E fromElement, E toElement);

    // Liefert den zugehörigen Comparator zurück
    public Comparator<E> comparator();
}
```

Datenstrukturen – Mengen

Binäre Bäume

Eine geeignete Möglichkeit zur Umsetzung einer geordneten Menge sind sogenannte binäre Bäume (binäre Suchbäume).

Ein Baum besteht aus Knoten, die mittels Kanten hierarchisch angeordnet sind. Ein Spezieller Knoten ist die Wurzel des Baums.

Eine Kante verbindet zwei Knoten im Sinne einer direkten Vorgänger-Nachfolger Beziehung.

Für jeden binären Baum muss eine Ordnungsrelation definiert sein.