21) Verfeinern von UML-Assoziationen mit dem Java-2 Collection Framework

Prof. Dr. rer. nat. Uwe Aßmann

Institut für Software- und Multimediatechnik

Lehrstuhl Softwaretechnologie

Fakultät für Informatik

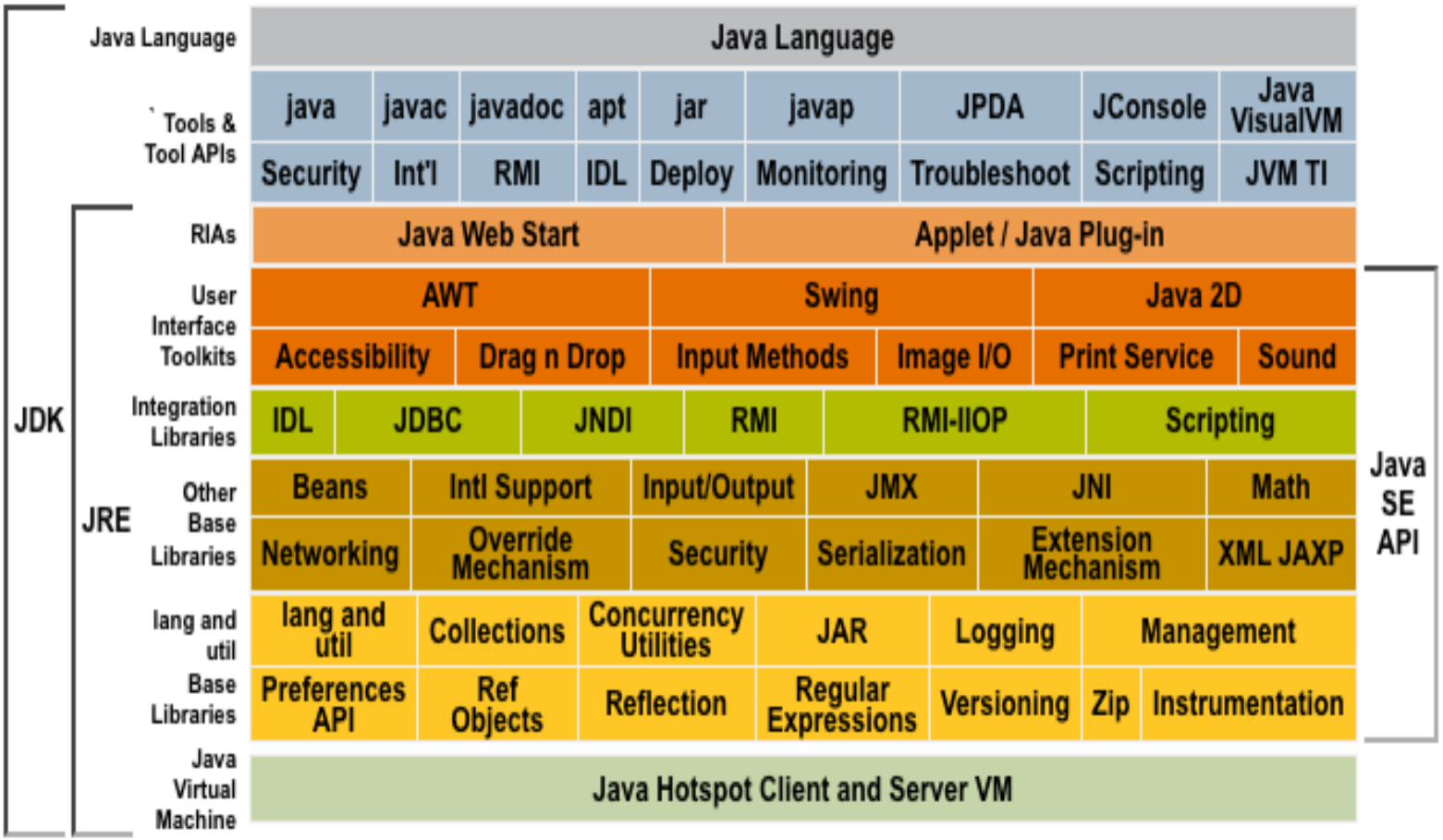
TU Dresden

Version 13-1.1, 13.05.13

1) Verfeinerung von Assoziationen
2) Generische Datenstrukturen
3) Polymorphe Container
4) Weitere Arten von Klassen
5) Ungeordnete Collections
6) Optimierte Auswahl von Implementierungen für Datenstrukturen

Obligatorische Literatur

- JDK Tutorial für J2SE oder J2EE, Abteilung Collections
- http://www.oracle.com/technetwork/java/javase/documentation/index.html



Empfohlene Literatur

- http://download.oracle.com/javase/6/docs/
- Tutorials http://download.oracle.com/javase/tutorial/
- Generics Tutorial: http://download.oracle.com/javase/tutorial/extra/generics/index.html

Hinweis: Online-Ressourcen

- Über die Homepage der Lehrveranstaltung finden Sie verschiedene Java-Dateien dieser Vorlesung.
- Beispiel "Bestellung mit Listen":

  `Bestellung-Collections/Bestellung0.java`

  `..`

  `Bestellung-Collections/Bestellung4.java`
- Beispiel “Warengruppen mit Mengen”

  `Warengruppe-Mengen/Warengruppe0.java`

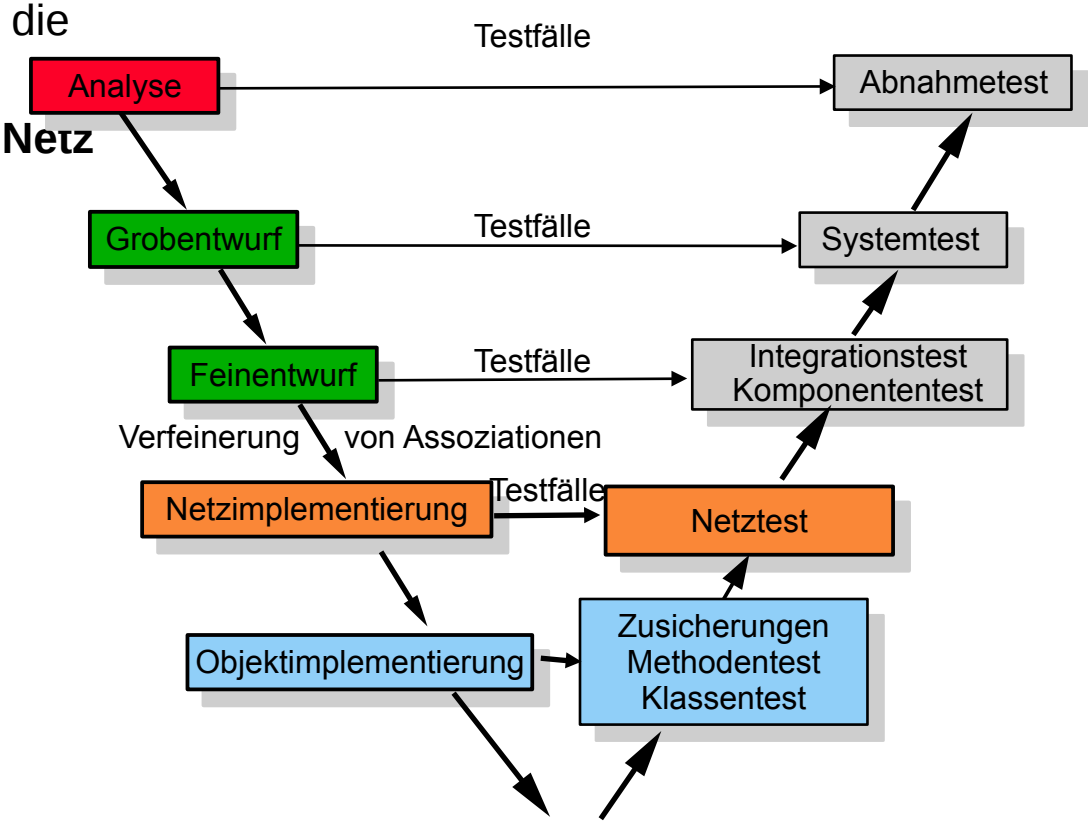
  `..`

  `Warengruppe-Mengen/Warengruppe4.java`

21.1 Verfeinern von Assoziationen

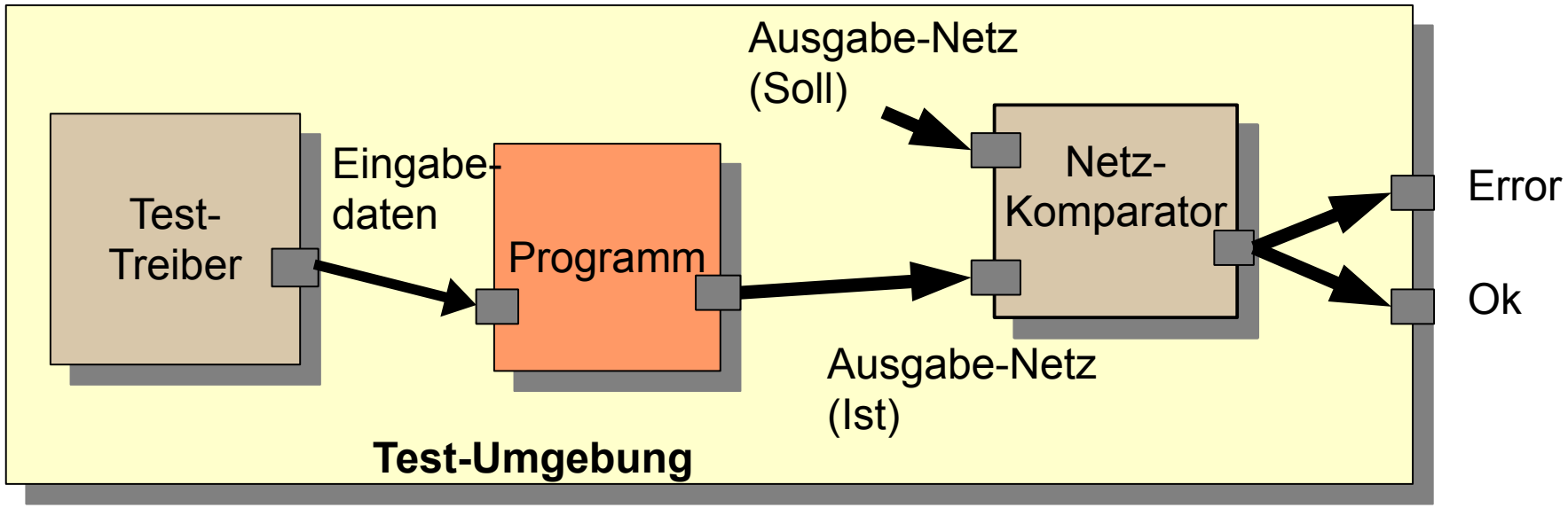
V-Modell mit Standard-Testprozess mit Netz-Entwurf und -Test

- Netze werden im Grob- und Feinentwurf in UML modelliert
- In der Phase “Netzimplementierung” in Java umgesetzt
- Die Tests werden *bottom-up* erledigt:
  - Zuerst Verträge und Testfälle für die Klasse bilden
  - **Verträge und Testfälle für das Netz entwerfen**
  - Dann die einzelne Klasse testen
  - **Dann das Netz testen**
  - Dann die Komponente
  - Dann das System
  - Dann der beta-Test
  - Zum Schluss der Akzeptanztest (Abnahmetest)



Objektorientierte Software hat eine test-getriebene Architektur für Objektnetze

- Testen beinhaltet die **Ist-Soll-Analyse** für Objektnetze
- Stimmt mein Netz mit meinem Entwurf überein?



Warum ist das wichtig?

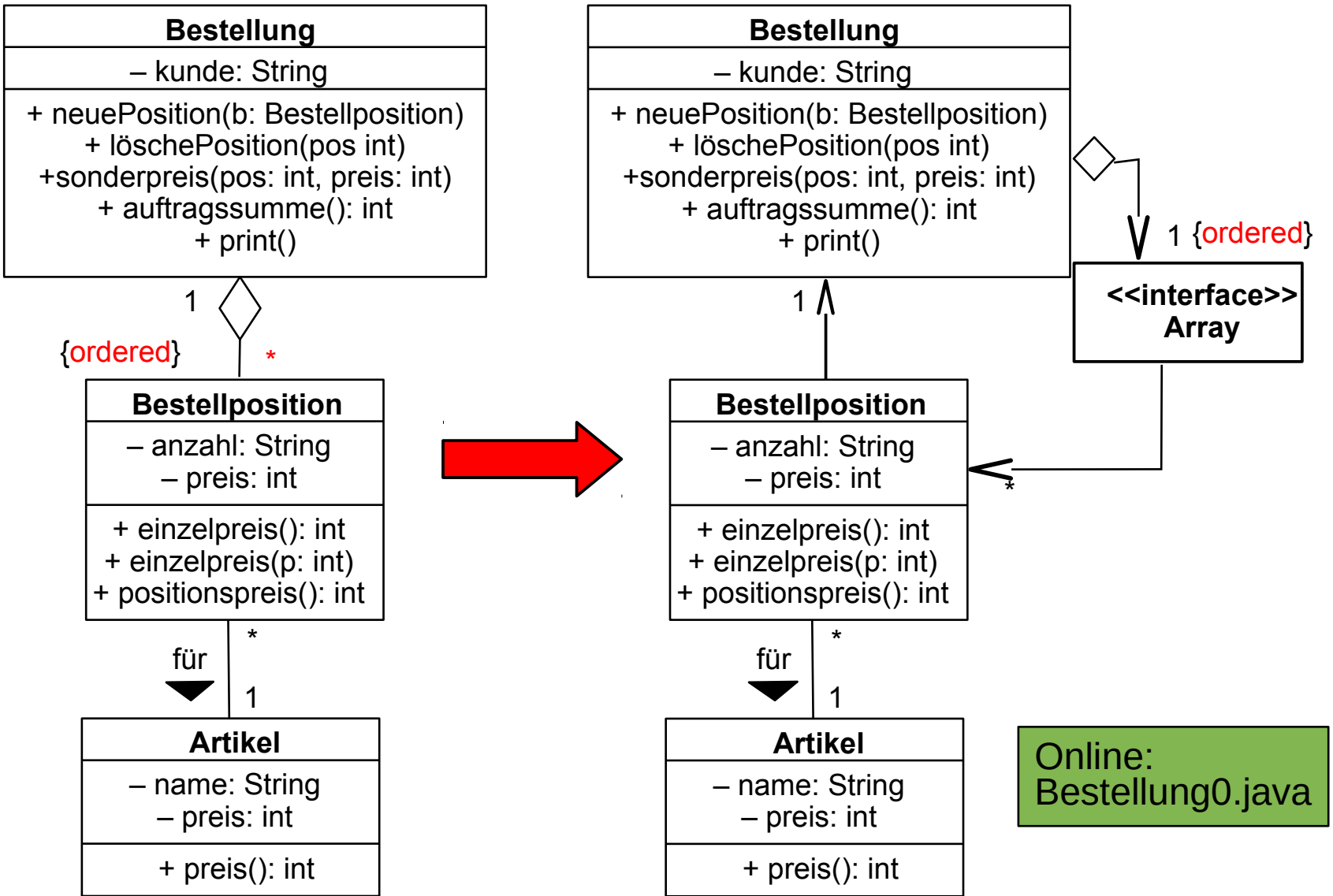
- Schon mal 3 Tage nach einem Zeiger-Fehler (pointer error) gesucht?

- Bitte mal nach “strange null pointer exception” suchen:
- https://forums.oracle.com/forums/thread.jspa?threadID=2056540
- http://stackoverflow.com/questions/8089798/strange-java-string-array-null-pointer-exception

Strange-null-pointer-exception-The-Official-Microsoft-ASP.pdf

Verfeinern von Assoziationen (Netzverfeinerung)

Modell einer Bestellungsabwicklung, eines einfachen Objektnetzes (Hierarchie):



Online: Bestellung0.java

Zentrale Frage:

- Wie bilde ich einseitige Assoziationen aus UML auf Java ab?
- Einfache Antwort 1: durch Abbildung auf Java Arrays

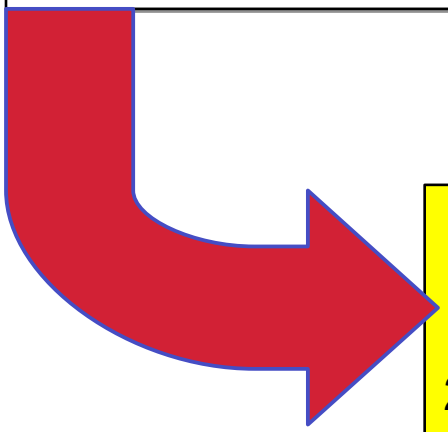
Einfache Realisierung mit Arrays


```
class Bestellung {
  private String kunde;
  private Bestellposition[] liste;
  private int anzahl = 0;

  public Bestellung(String kunde) {
    this.kunde = kunde;
    liste = new Bestellposition[20];
  }
  public void neuePosition (Bestellposition b) {
     liste[anzahl] = b;
    anzahl++;    // was passiert bei mehr als 20 Positionen ?
  }
  public void loeschePosition (int pos) {
       // geht mit Arrays nicht einfach zu realisieren !
  }
  public void sonderpreis (int pos, int preis) {
    liste[pos].einzelpreis(preis);
  }
  public int auftragssumme() {
    int s = 0;
    for(int i=0; i<anzahl; i++) s += liste[i].positionspreis();
    return s;
  }
}
```

Online:
Bestellung0.java

Testprogramm für Anwendungsbeispiel (1)


```
public static void main (String[] args) {
    Artikel tisch = new Artikel("Tisch",200);
    Artikel stuhl = new Artikel("Stuhl",100);
    Artikel schrank = new Artikel("Schrank",300);

    Bestellung b1 = new Bestellung("TUD");
    b1.neuePosition(new Bestellposition(tisch,1));
    b1.neuePosition(new Bestellposition(stuhl,4));
    b1.neuePosition(new Bestellposition(schrank,2));
    b1.print(); ...}
```

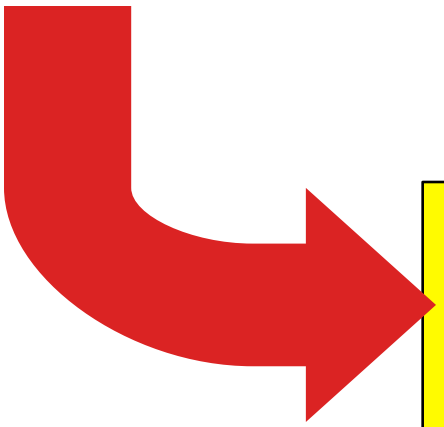


Bestellung fuer Kunde TUD
0. 1 x Tisch Einzelpreis: 200 Summe: 200
1. 4 x Stuhl Einzelpreis: 100 Summe: 400
2. 2 x Schrank Einzelpreis: 300 Summe: 600
Auftragssumme: 1200

Online:
Bestellung0.java

Testprogramm für Anwendungsbeispiel (2)


```
public static void main (String[] args) {
    ...
    b1.sonderpreis(1,50);
    b1.print();
}
```



Bestellung fuer Kunde TUD
0. 1 x Tisch Einzelpreis: 200 Summe: 200
1. 4 x Stuhl Einzelpreis: 50 Summe: 200
2. 2 x Schrank Einzelpreis: 300 Summe: 600
Auftragssumme: 1000

Probleme der Realisierung von Assoziationen mit Arrays

- Java Arrays besitzen eine feste Obergrenze für die Zahl der enthaltenen Elemente
  - Fest zur Übersetzungszeit
  - Fest zur Allokationszeit
- *Dynamische Arrays* sind dynamisch erweiterbar:
  - http://en.wikipedia.org/wiki/Dynamic_array
  - Automatisches Verschieben bei Löschen und Mitten-Einfügen
- Was passiert, wenn keine Ordnung benötigt wird?
- Kann das Array sortiert werden?
  - Viele Algorithmen laufen auf sortierten Universen wesentlich schneller als auf unsortierten (z.B. Anfragen in Datenbanken)
- Wie bilde ich einseitige Assoziationen aus UML auf Java ab?
- Antwort 2: durch Abbildung auf die Schnittstelle Collection

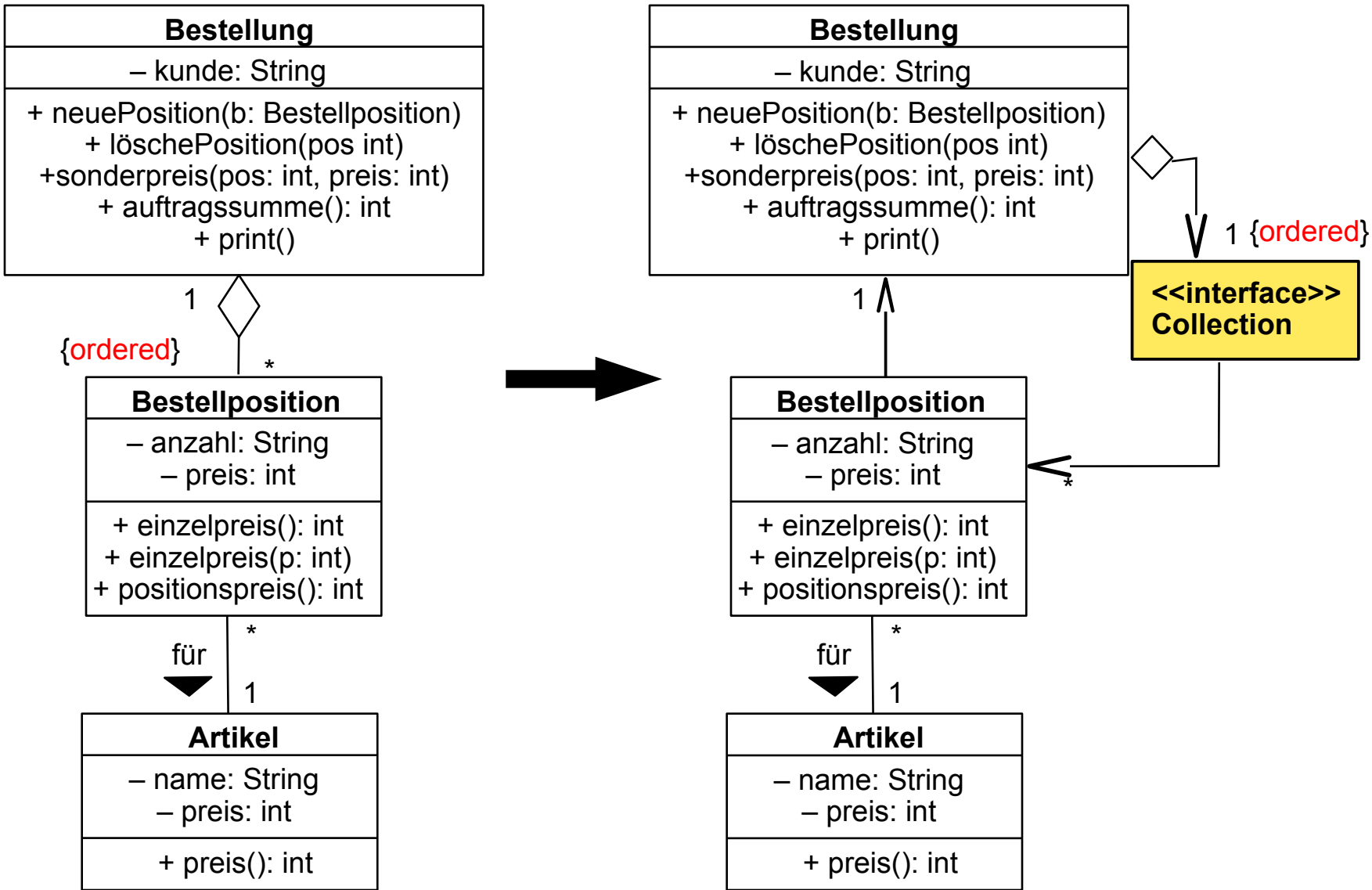
Collections

- Probleme werden durch das Java-Collection-Framework gelöst, eine objektorientierte Datenstrukturbibliothek für Java
  - Meiste Standard-Datenstrukturen abgedeckt
  - Verwendung von Vererbung zur Strukturierung
  - Flexibel auch zur eigenen Erweiterung

- Zentrale Frage: Wie bilde ich einseitige Assoziationen aus UML auf Java ab?
  - Antwort: Einziehen von Behälterklassen (*collections*) aus dem Collection-Framework
  - *Flachklopfen (lowering)* von Sprachkonstrukten: Wir klopfen Assoziationen zu Java-Behälterklassen flach.

Bsp.: Verfeinern von bidir. Assoziationen durch Behälterklassen

Ersetzen von “*”-Assoziationen durch Behälterklassen

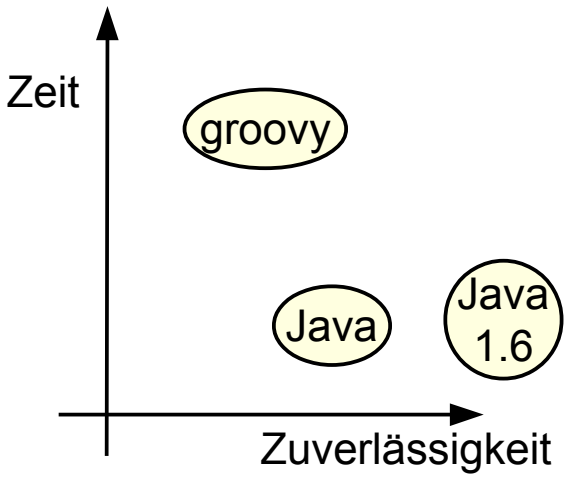


21.2 Die Collection-Bibliothek

- Generische Behälterklassen
- Collection<Object>

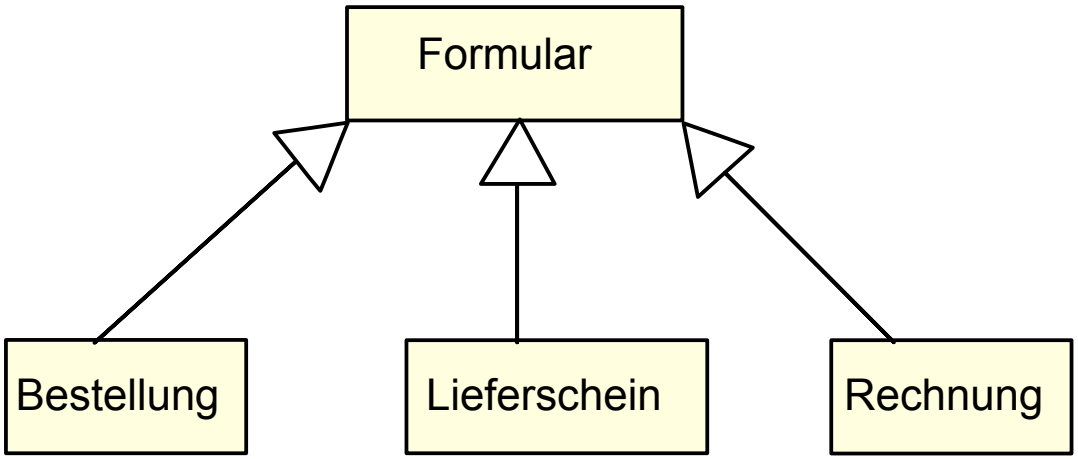
Drei Trends in der Softwareentwicklung

- **Rapid Application Development (RAD)**
  - *Schneller viel* Code schreiben
  - Typisierung weglassen
    - Bei den Assoziationen
    - Beim Programmieren gegen Schnittstellen
  - Ev. *dynamische Typisierung,* damit Fehler zur Laufzeit identifiziert werden können
  - Mächtige Operationen, die schnell zu schreiben sind



- **Safe Application Development (SAD)**
  - *Guten, stabilen, wartbaren* Code schreiben
  - *Statische Typisierung*, damit der Übersetzer viele Fehler entdeckt
  - Mehr Entwurfswissen aus dem Entwurf in die Implementierung übertragen
  - Aus der Definition einer Datenstruktur können Bedingungen für ihre Anwendung abgeleitet werden
  - *Generische Collections* für typsicheres Aufbauen von Objektnetzen (Java 1.6)
- **Gradual Typing** für beides
  - Typen werden Schritt für Schritt annotiert
  - http://ecee.colorado.edu/~siek/gradual-obj.pdf

Bsp.: Elemente einer Hierarchie

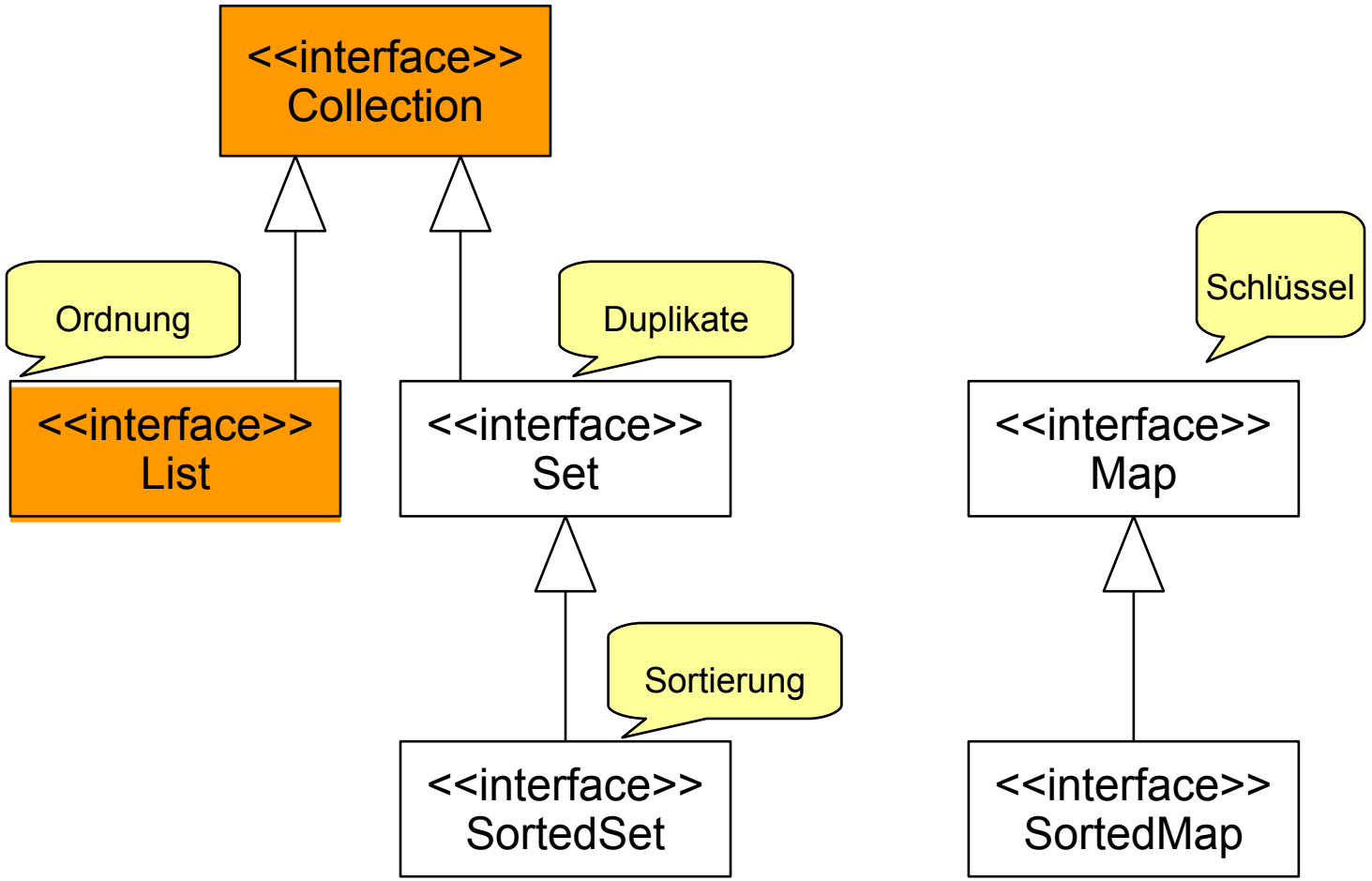


Facetten von Behälterklassen (Collections)

- Behälterklassen können anhand von verschiedenen *Facetten* klassifiziert werden
  - **Facetten** sind orthogonale Dimensionen einer Klassifikation oder eines Modells

Ordnung	**Duplikate**	**Sortierung**	**Schlüssel**
geordnet ungeordnet	mit Duplikaten ohne Duplikate	sortiert unsortiert	mit Schlüssel ohne Schlüssel

Java Collection Framework: Prinzipielle Struktur

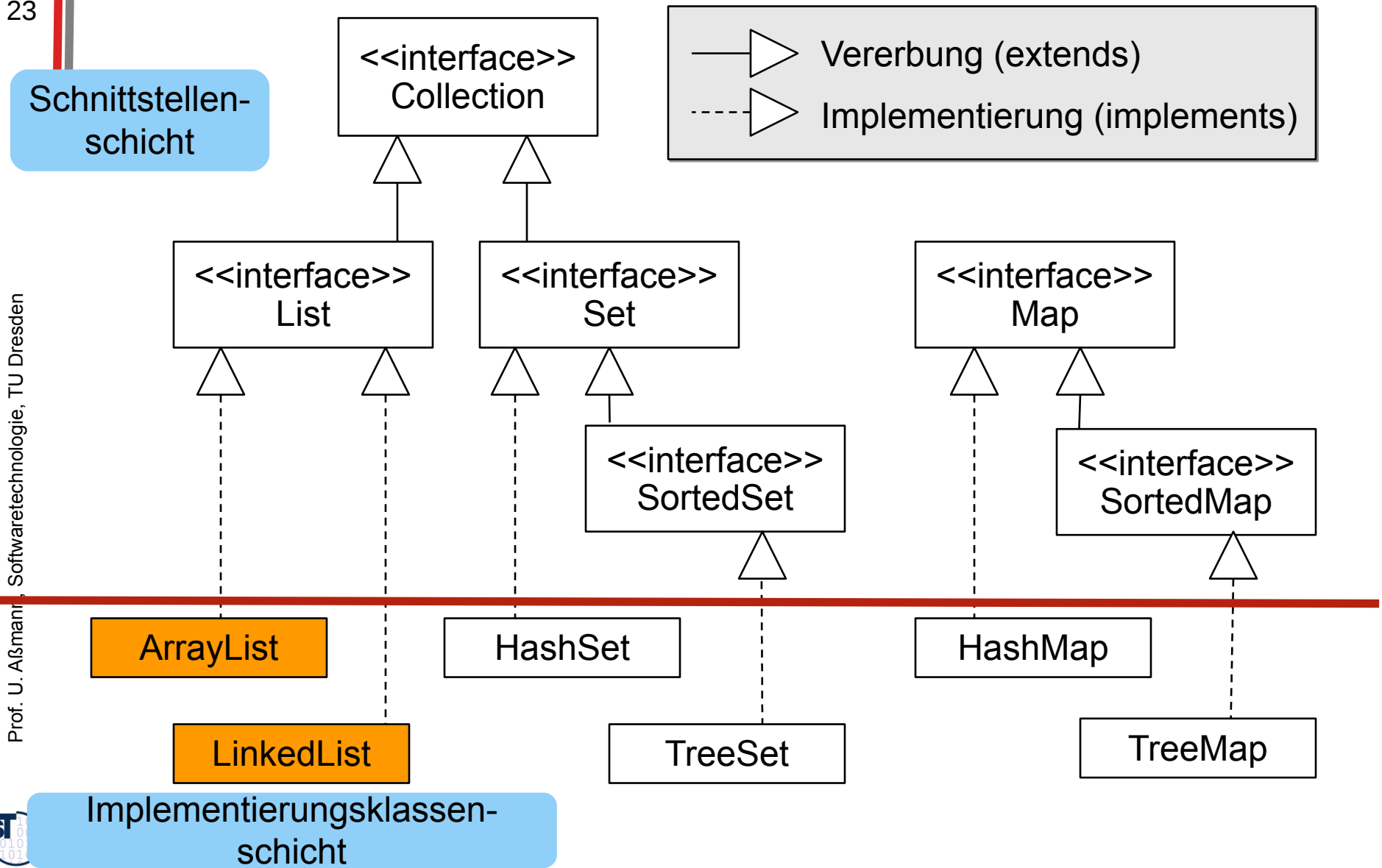


/home/ua1/Courses/ST1/Material/jdk-docs/index.html

Klassifikation der Schnittstellen der Datenstrukturen

- Collection (Behälter, Kollektion):
  - Ansammlung von Datenelementen
  - Hinzufügen, Entfernen, Suchen, Durchlaufen
- Set (Menge):
  - Reihenfolge des Einfügens, Mehrfachvorkommen spielen keine Rolle
  - SortedSet (geordnete Menge): Ordnung auf den Elementen + Sortierung
- List (Liste):
  - Mehrfachvorkommen werden separat abgelegt
  - Reihenfolge des Einfügens bleibt erhalten
- Map (Abbildung, mapping, associative array):
  - Zuordnung von Schlüsselwerten auf Eintragswerte
  - Menge von Tupeln: Mehrfachvorkommen bei Schlüsseln verboten, bei Einträgen erlaubt
  - SortedMap (geordnete Abbildung): Ordnung auf den Schlüsseln + Sortierung danach

Schnittstellen und Implementierungen im unsicheren, ungetypten Collection-Framework (vor Java 1.5)



Problem 1 ungetypter Schnittstellen: Laufzeitfehler

- Bei der Konstruktion von Collections werden oft Fehler programmiert, die bei der Dekonstruktion zu Laufzeitfehlern führen
- Kann in Java < 1.5 nicht durch den Übersetzer entdeckt werden

```
List listOfRechnung = new ArrayList();
Rechnung rechnung = new Rechnung();
listOfRechnung.add(rechnung);
Bestellung best = new Bestellung();
listOfRechnung.add(best);

for (int i = 0; i < listOfRechnung.size(); i++) {
   rechnung = (Rechnung)listOfRechnung.get(i);
}
```

Programmierfehler!

Laufzeitfehler!!

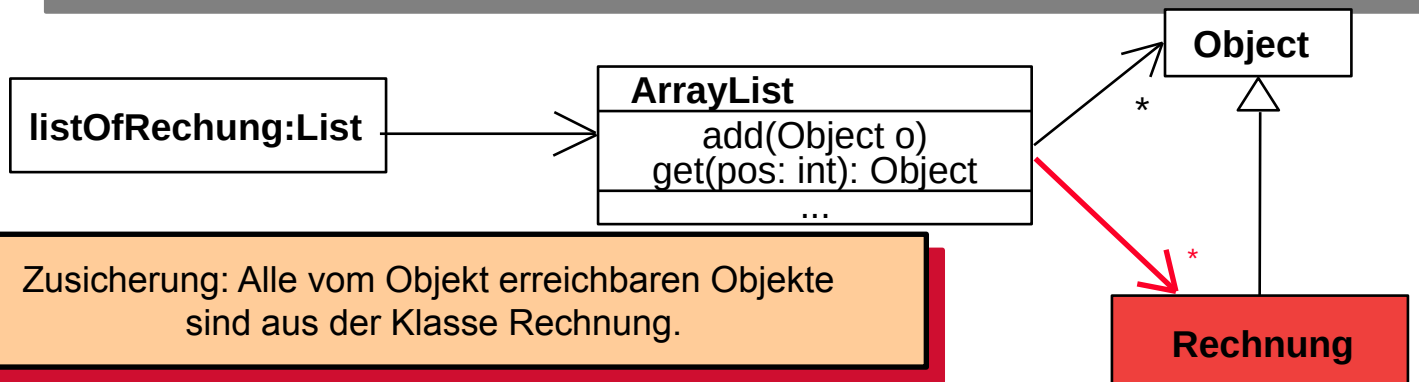
Problem 2 ungetypter Schnittstellen: Unnötige Casts

- Bei der Dekonstruktion von Collections müssen unnötig **Typumwandlungen (Casts)** spezifiziert werden
- Typisierte Collections erhöhen die Lesbarkeit, da sie mehr Information geben

```
List listOfRechnung = new ArrayList();
Rechnung rechnung = new Rechnung();
listOfRechnung.add(rechnung);
Rechnung rechnung2 = new Rechnung();
listOfRechnung.add(rechnung2);

for (int i = 0; i < listOfRechnung.size(); i++) {
   rechnung = (Rechnung)listOfRechnung.get(i);
}
```

Diesmal ok

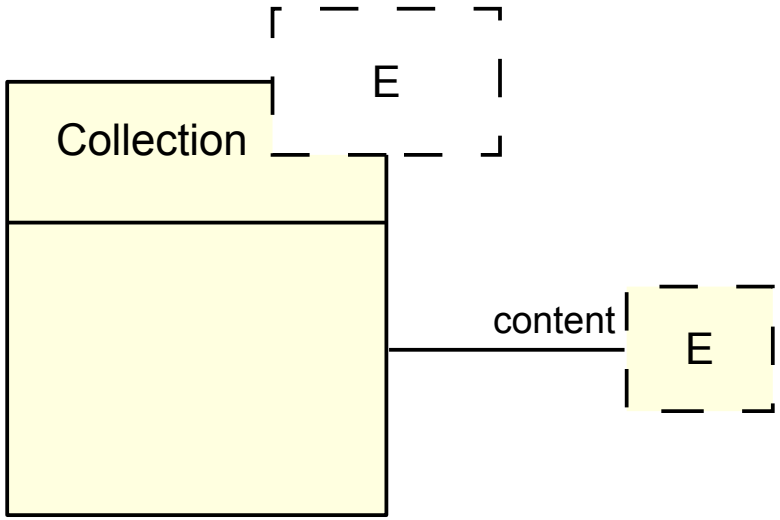
Cast nötig, obwohl alles Rechnungen



Generische Behälterklassen

Eine **generische Behälterklasse** ist eine Klassenschablone einer Behälterklasse, die mit einem Typparameter für den Typ der Elemente versehen ist.

- In UML



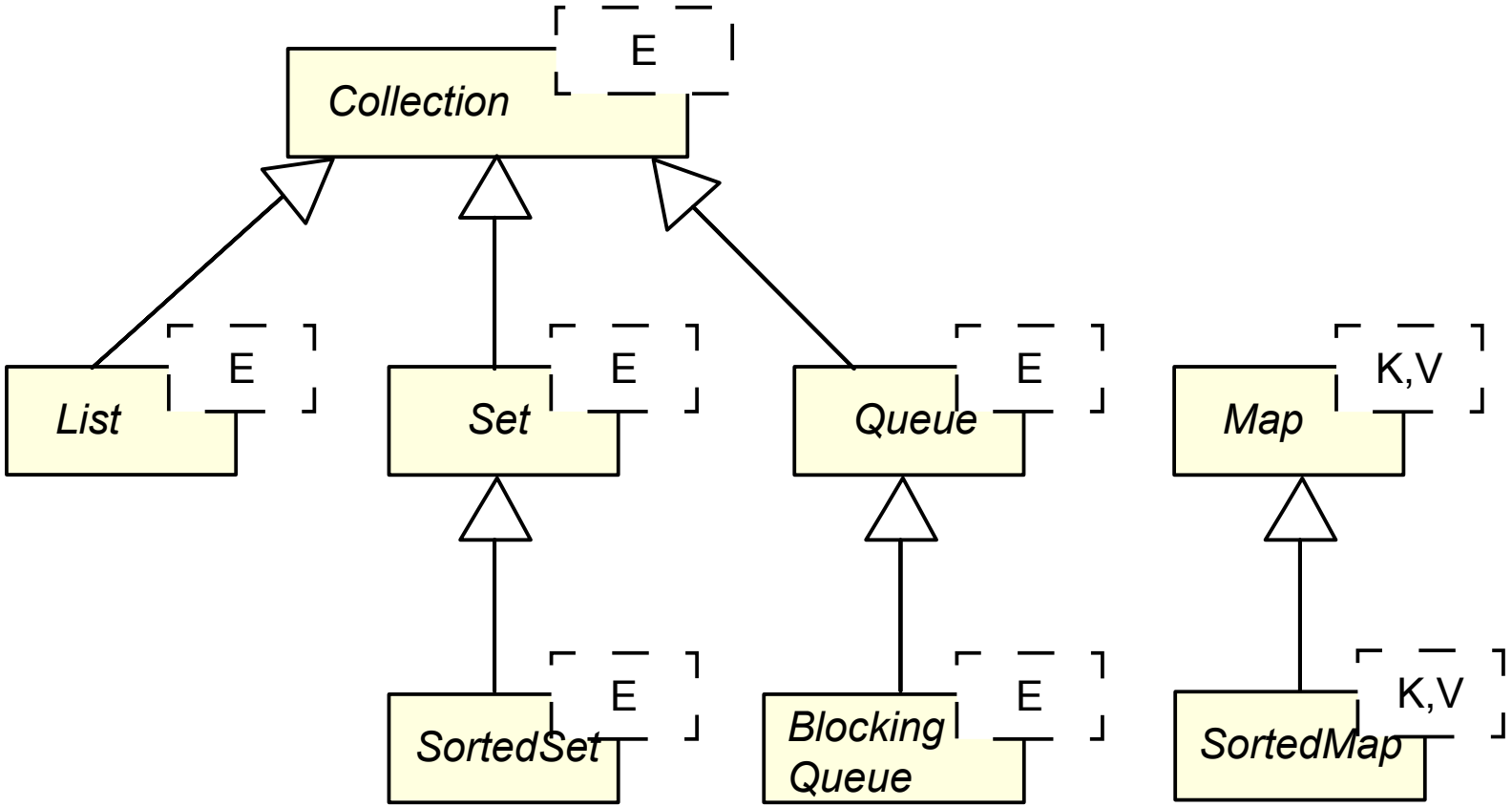
Collection<E>

- In Java
  - Sprachregelung: “Collection of E”

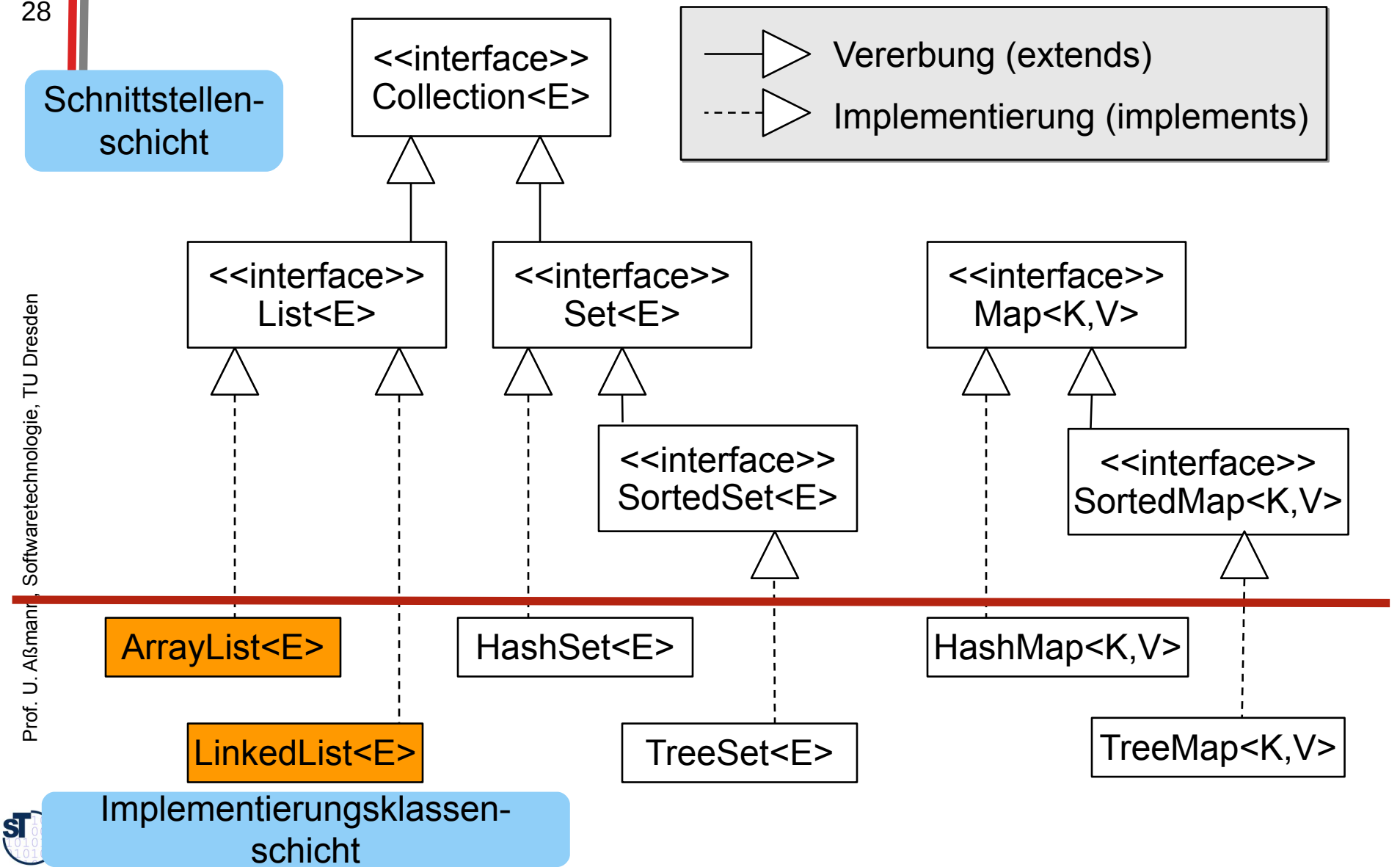
```
class Collection<E> {
  E content[];
}
```

Collection-Hierarchie mit generischen Schnittstellen

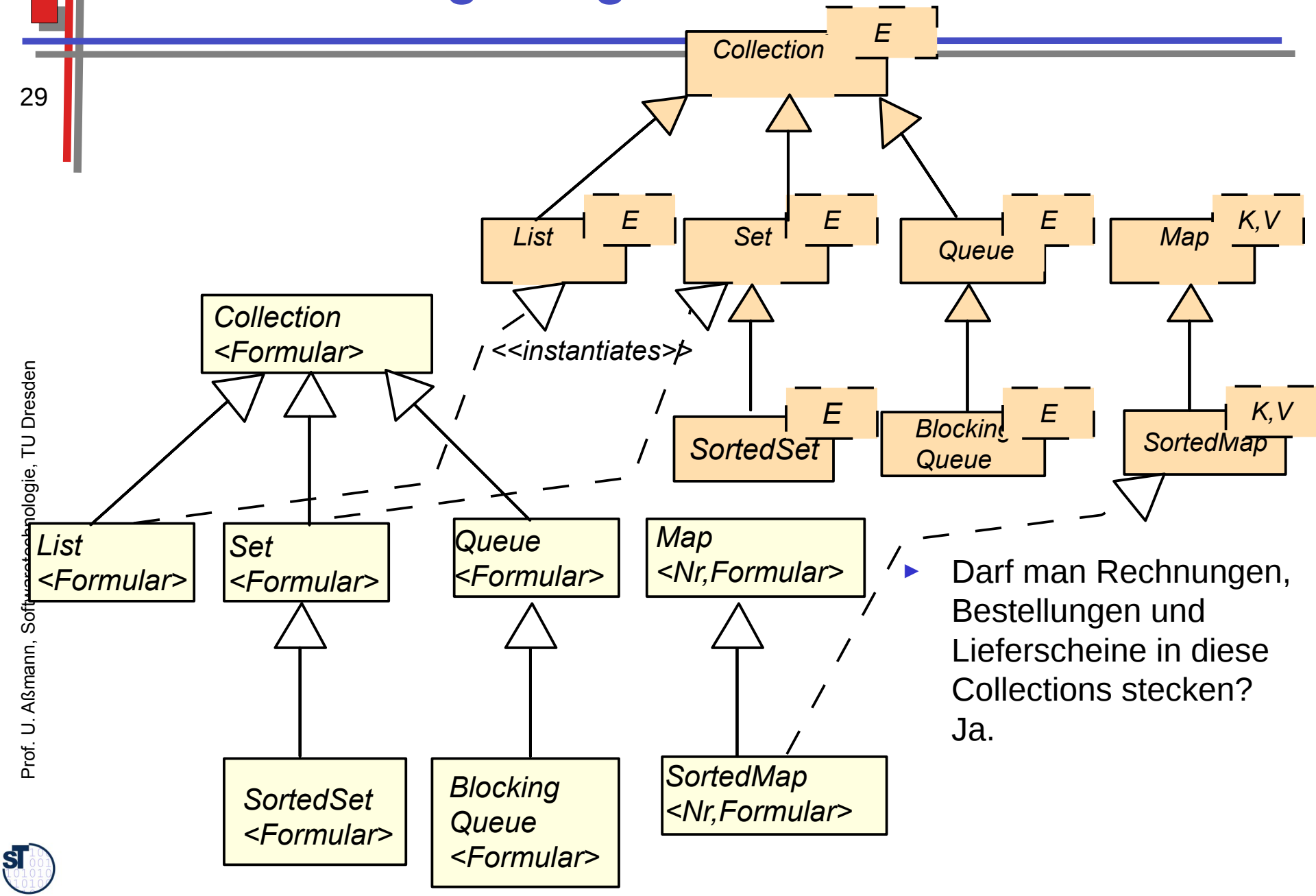
- Die generische Schnittstellen-Hierarchie der Collections (seit Java 1.5)
  - E: Element, K: Key, V: Value



Schnittstellen und Implementierungen im Collection-Framework bilden generische Behälterklassen



Instanziierung der generischen Hierarchie

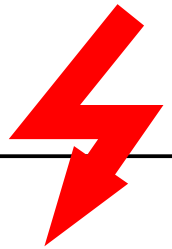


- Darf man Rechnungen, Bestellungen und Lieferscheine in diese Collections stecken?
  Ja.

SAD löst Probleme

- Bei der Konstruktion von Collections werden jetzt Äpfel von Birnen unterschieden
- Casts sind nicht nötig, der Übersetzer kennt den feineren Typ
- Das ist Safe Application Development (SAD)

```
List<Rechnung> listOfRechnung = new ArrayList<Rechnung>();
Rechnung rechnung = new Rechnung();
listOfRechnung.add(rechnung);
Bestellung best = new Bestellung();
listOfRechnung.add(best);

for (int i = 0; i < listOfRechnung.size(); i++) {
   rechnung = listOfRechnung.get(i);
}
```



Compilerfehler

Kein Cast mehr nötig

Schnittstelle java.util.Collection (Auszug)

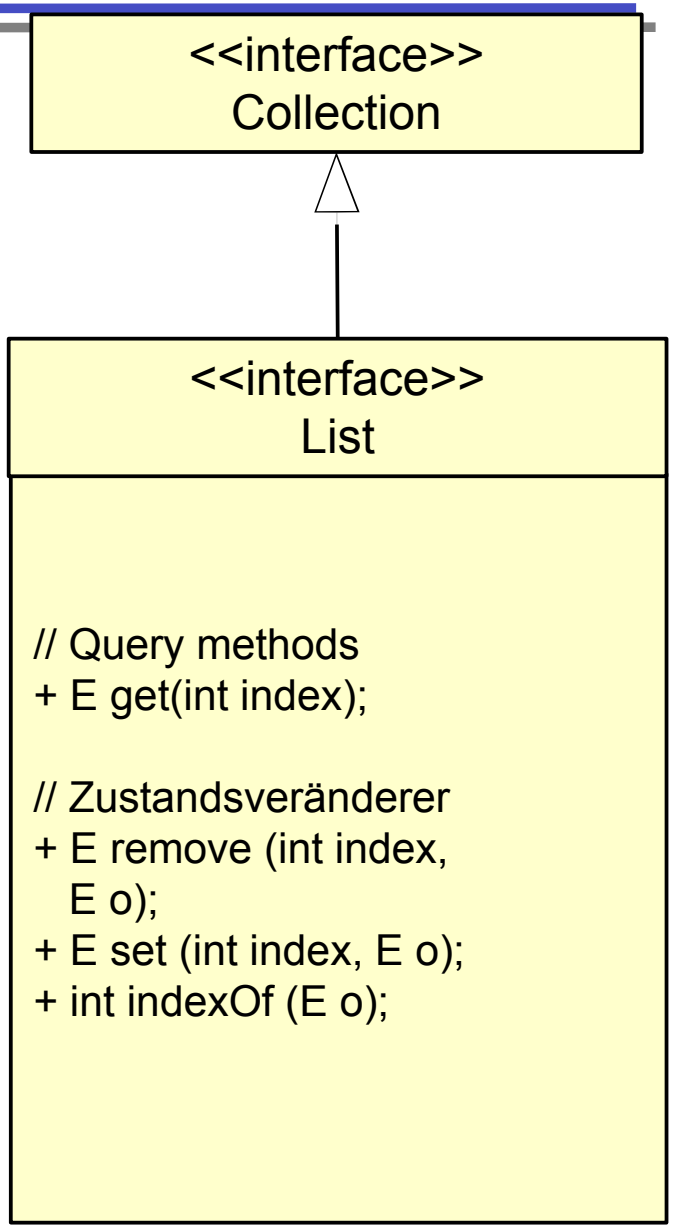

```
public interface Collection<E> {
  // Anfragen (Queries)
  public boolean isEmpty();
  public boolean contains (E o);
  public boolean equals(E o);
  public int size();
  public int hashCode();
  public Iterator iterator();
  // Repräsentations-Transformierer
  public E[] toArray();
  // Zustandsveränderer
  // Monotone Zustandserweiterer
  public boolean add (E o);
  // Nicht-Monotone Zustandsveränderer
  public boolean remove (E o);
  public void clear();
  …
}
```

<<interface>>
Collection<E>

```
// Query methods
+ boolean isEmpty();
+ boolean contains(E o);
+ boolean equals(E o);
+ int hashCode();
+ Iterator iterator();

// Repräsentations-Trans-
// formierer
+ E[] toArray();

// Monotone Zustandsveränderer
+ boolean add (E o);

// Zustandsveränderer
+ boolean remove (E o);
+ void clear();
```

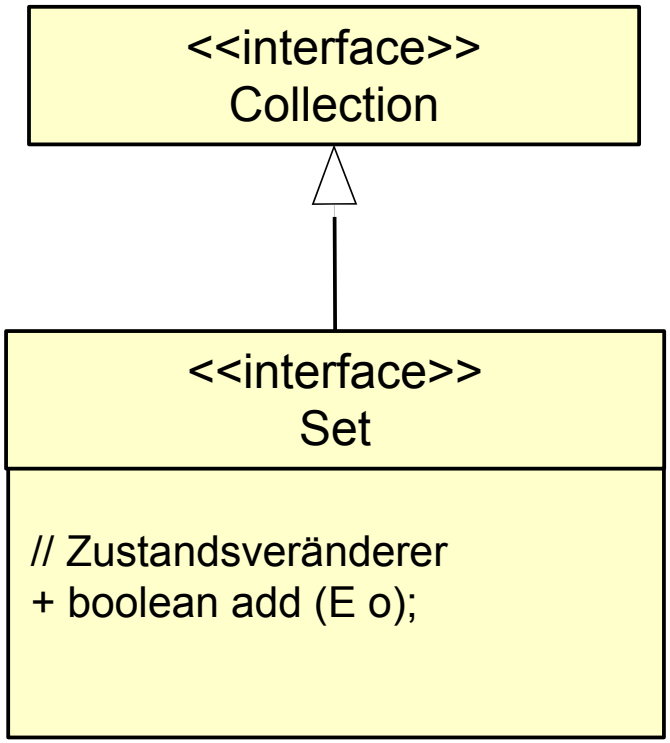
Unterschnittstelle java.util.List (Auszug)


```
public interface List extends Collection {

  public boolean isEmpty();
  public boolean contains (E o);
  public int size();
  public boolean add (E o);
  public boolean remove (E o);
  public void clear();
  public E get (int index);
  public E set (int index, E element);
  public E remove (int index);
  public int indexOf (E o);
  ...
}
```



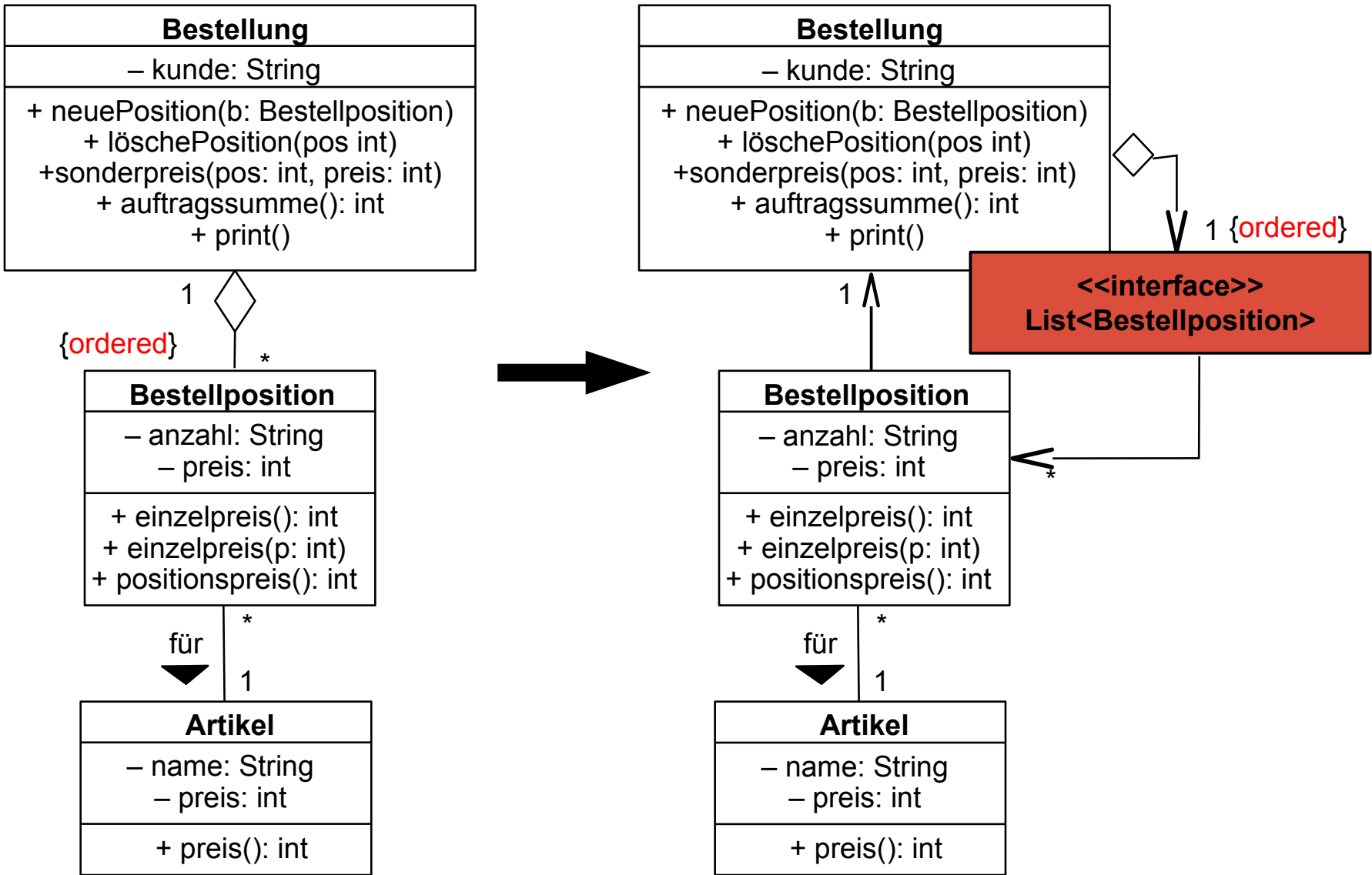
Unterschnittstelle java.util.Set (Auszug)

- Spezifische Schnittstellen sowie konkrete Implementierungen in der Collection-Hierarchie können spezialisiertes Verhalten aufweisen.
- Bsp.: Was ändert sich bei Set im Vergleich zu List in der Semantik von `add()`?

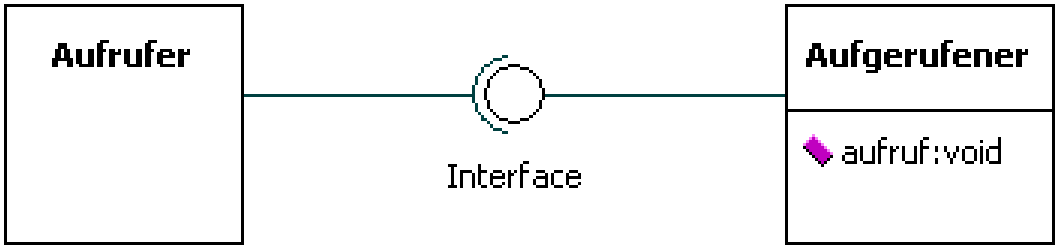


Verfeinern von bidir. Assoziationen durch getypte Behälterklassen

Ersetzen von “*”-Assoziationen durch Behälterklassen

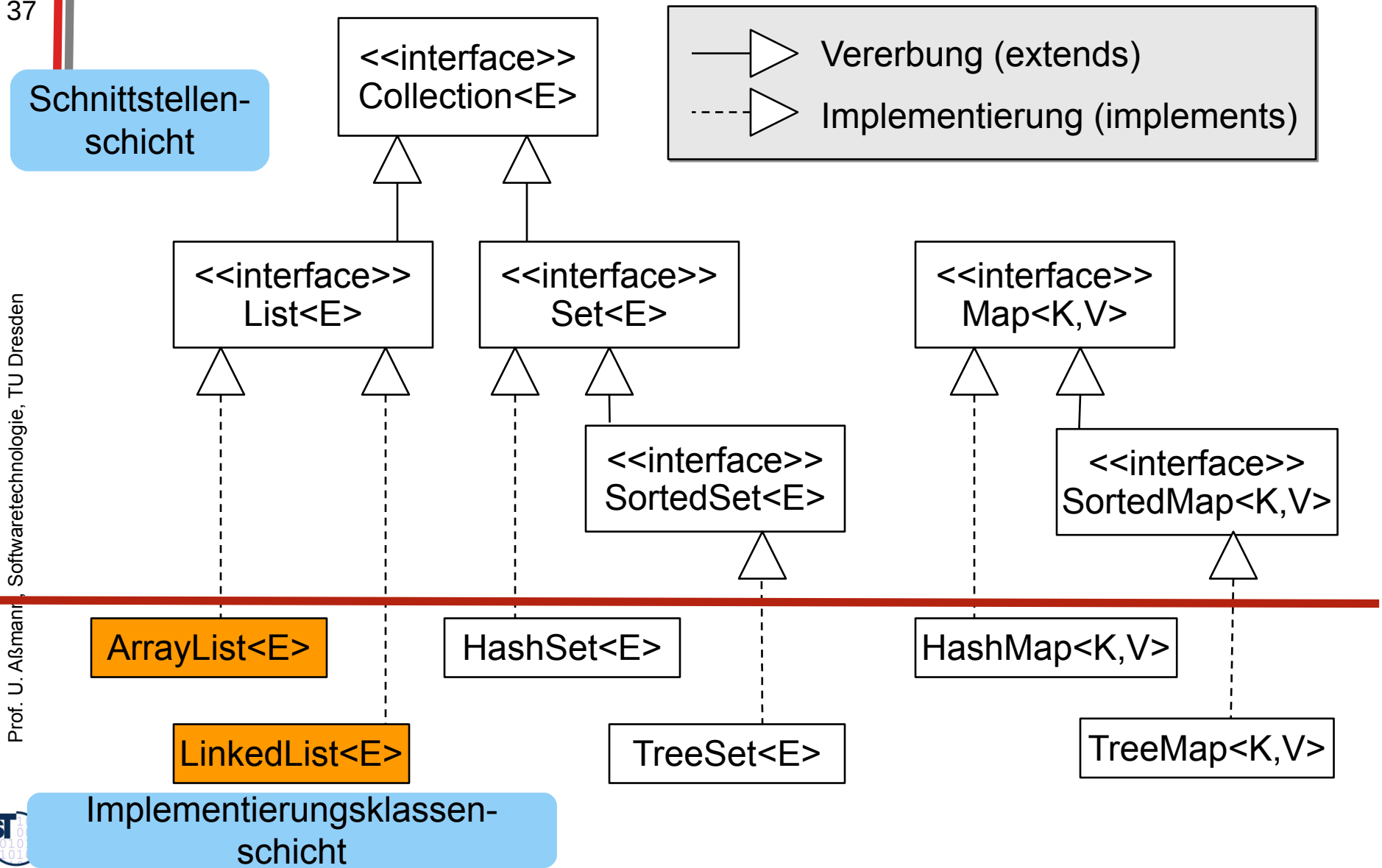


21.3 Programmieren gegen Schnittstellen mit polymorphen Containern



"Der Aufrufer programmiert *gegen* die Schnittstelle,
er befindet sich sozusagen im luftleeren Raum."
Siedersleben/Denert,
Wie baut man Informationssysteme,
Informatik-Spektrum, August 2000

Schnittstellen und Implementierungen im Collection-Framework bilden generische Behälterklassen

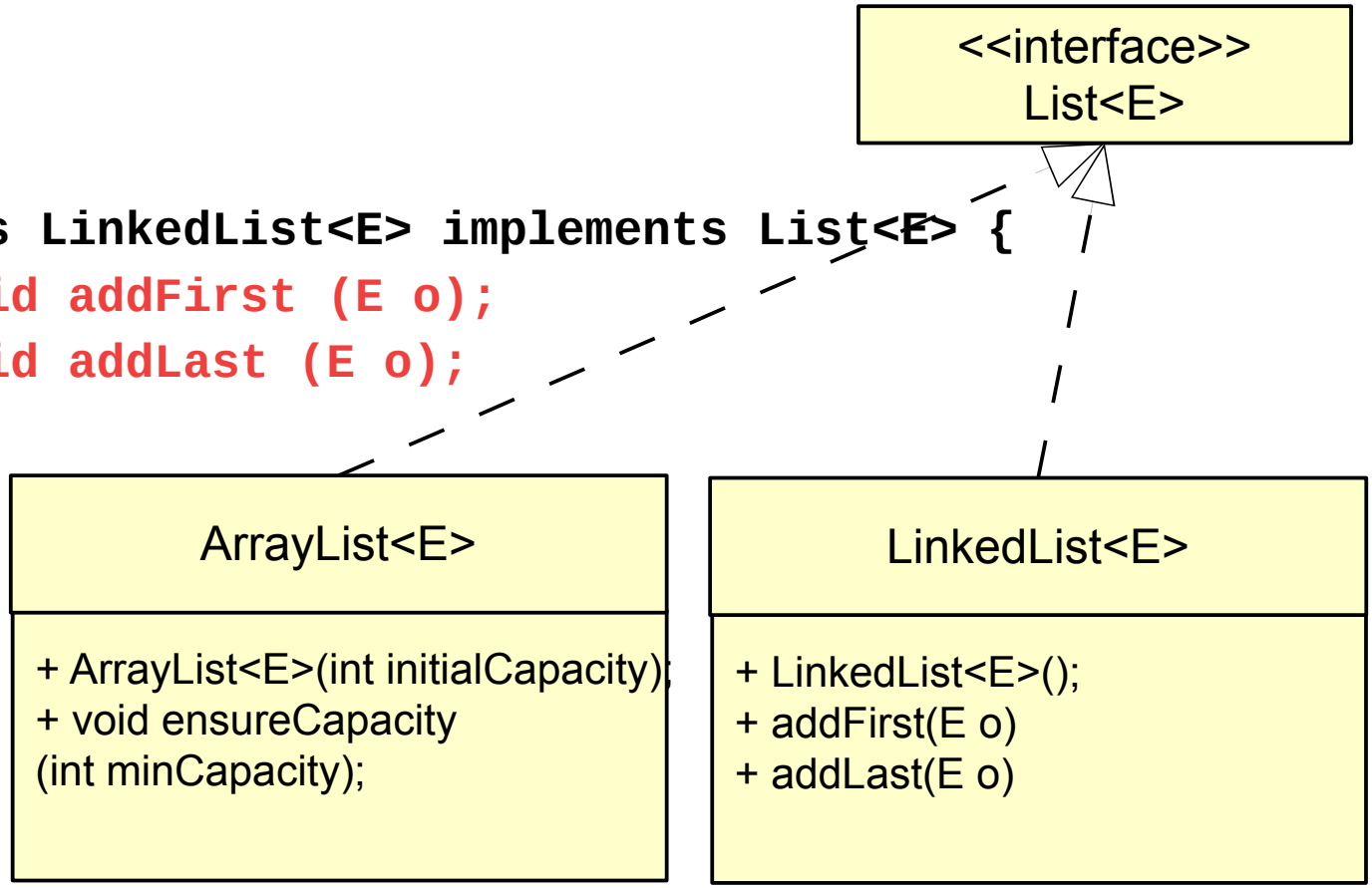


Polymorphie – zwischen abstrakten und konkreten Datentypen

Abstrakter Datentyp (Schnittstelle)	***Konkreter Datentyp (Implementierung)***
▸ Abstraktion: – Operationen – Verhalten der Operationen	▸ Konkretisierung: – Instantiierbare Klassen – Ausführbare Operationen
▸ *Theorie:* – Algebraische Spezifikationen • Axiomensysteme	▸ *Theorie:* – Datenstrukturen – Komplexität
▸ *Praxis:* – Abstrakte Klassen – Interfaces	▸ *Praxis:* – Alternativen
▸ *Beispiel:* – List	▸ *Beispiel:* – Verkettete Liste – Liste durch Feld

Beispiel: Implementierungsklassen java.util.ArrayList, LinkedList


```
public class ArrayList<E> implements List<E> {
  public ArrayList<E> (int initialCapacity);
  public void ensureCapacity (int minCapacity);
  ...
}

public class LinkedList<E> implements List<E> {
  public void addFirst (E o);
  public void addLast (E o);
  ...
}
```



Programmieren gegen Schnittstellen
-- Polymorphe Container


```
class Bestellung {
  private String kunde;
  private List<Bestellposition> liste;
  ... // Konstruktor s. u.
  public void neuePosition (Bestellposition b) {
    liste.add(b);    }
  public void loeschePosition (int pos) {
    liste.remove(pos);    }
  public void sonderpreis (int pos, int preis) {
    liste.get(pos).einzelpreis(preis);    }
```

```
public Bestellung(String kunde) {
  this.kunde = kunde;
  this.liste = new ArrayList<
        Bestellposition>();
} ...
```

```
public Bestellung(String kunde) {
  this.kunde = kunde;
  this.liste = new LinkedList<
        Bestellposition>();
} ...
```

Bei polymorphen Containern muß der Code bei Wechsel der Datenstruktur nur an einer Stelle im Konstruktor geändert werden!

Welche Listen-Implementierung soll man wählen?

Aus alternativen Implementierungen einer Schnittstelle wählt man diejenige, die für das Benutzungsprofil der Operationen die größte Effizienz bereitstellt (Geschwindigkeit, Speicherverbrauch, Energieverbrauch)

- Innere Schleifen bilden die „heißen Punkte“ (hot spots) eines Programms
  - Optimierung von inneren Schleifen durch Auswahl von Implementierungen mit geeignetem Zugriffsprofil
- Gemessener relativer Aufwand für Operationen auf Listen:
  (aus Eckel, Thinking in Java, 2nd ed., 2000)
  - Stärken von ArrayList: wahlfreier Zugriff
  - Stärken von LinkedList: Iteration, Einfügen und Entfernen irgendwo in der Liste
  - Vector (deprecated) ist generell die langsamste Lösung

Typ	Lesen	Iteration	Einfügen	Entfernen
array	1430	3850	--	--
ArrayList	3070	12200	500	46850
LinkedList	16320	9110	110	60
Vector	4890	16250	550	46850

Generizität auf Containern funktioniert auch geschachtelt


```
// Das Archiv listOfRechnung fasst die Rechnungen des
// aktuellen Jahres zusammen
List<Rechnung> listOfRechnung = new ArrayList<Rechnung>();
List<List<Rechnung>> archiv = new ArrayList<List<Rechnung>>;
archiv.add(listOfRechnung);
Rechnung rechnung = new Rechnung();
archiv.get(0).add(rechnung);
Bestellung best = new Bestellung();
archiv.get(0).add(best);

for (int jahr = 0; jahr < archiv.size(); jahr++) {
   listOfRechnung = archiv.get(jahr);
   for (int i = 0; i < listOfRechnung.size(); i++) {
          rechnung = listOfRechnung.get(i);
   }
}
```

funktioniert

Übersetzungs-Fehler

Benutzung von getypten und ungetypten Schnittstellen

- .. ist ab Java 1.5 ohne Probleme nebeneinander möglich

```
// Das Archiv fasst alle Rechnungen aller bisherigen Jahrgänge zusammen
List<List<Rechnung>> archiv = new ArrayList<List<Rechnung>>();
// listOfRechnung fasst die Rechnungen des aktuellen Jahres zusammen
List listOfRechnung = new ArrayList();

archiv.add(listOfRechnung);
Rechnung rechnung = new Rechnung();
archiv.get(0).add(rechnung);
Bestellung best = new Bestellung();
archiv.get(0).add(best);

for (int jahr = 0; jahr < archiv.size(); jahr++) {
   listOfRechnung = archiv.get(jahr);
   for (int i = 0; i < listOfRechnung.size(); i++) {
           rechnung = (Rechnung)listOfRechnung.get(i);
   }
}
```

funktioniert

Übersetzt auch, aber Laufzeitfehler beim Cast...

Unterschiede zu C++

- In Java: einmalige Übersetzung des generischen Datentyps
  - Verliert etwas Effizienz, da der Übersetzer alle Typinformation im generierten Code vergisst und nicht ausnutzt
  - z.B. sind alle Instanzen mit *unboxed objects* als *boxed objects realisiert*
- C++ bietet Code-Templates (snippets, fragments) an, mit denen man mehr parameterisieren kann, z.B. Methoden
- In C++ können Templateparameter Variablen umbenennen:

```
template class C <class T> {
    T attribute<T>
}
```

Templateparameter können Variablen umbenennen

21.4) Weitere Arten von Klassen

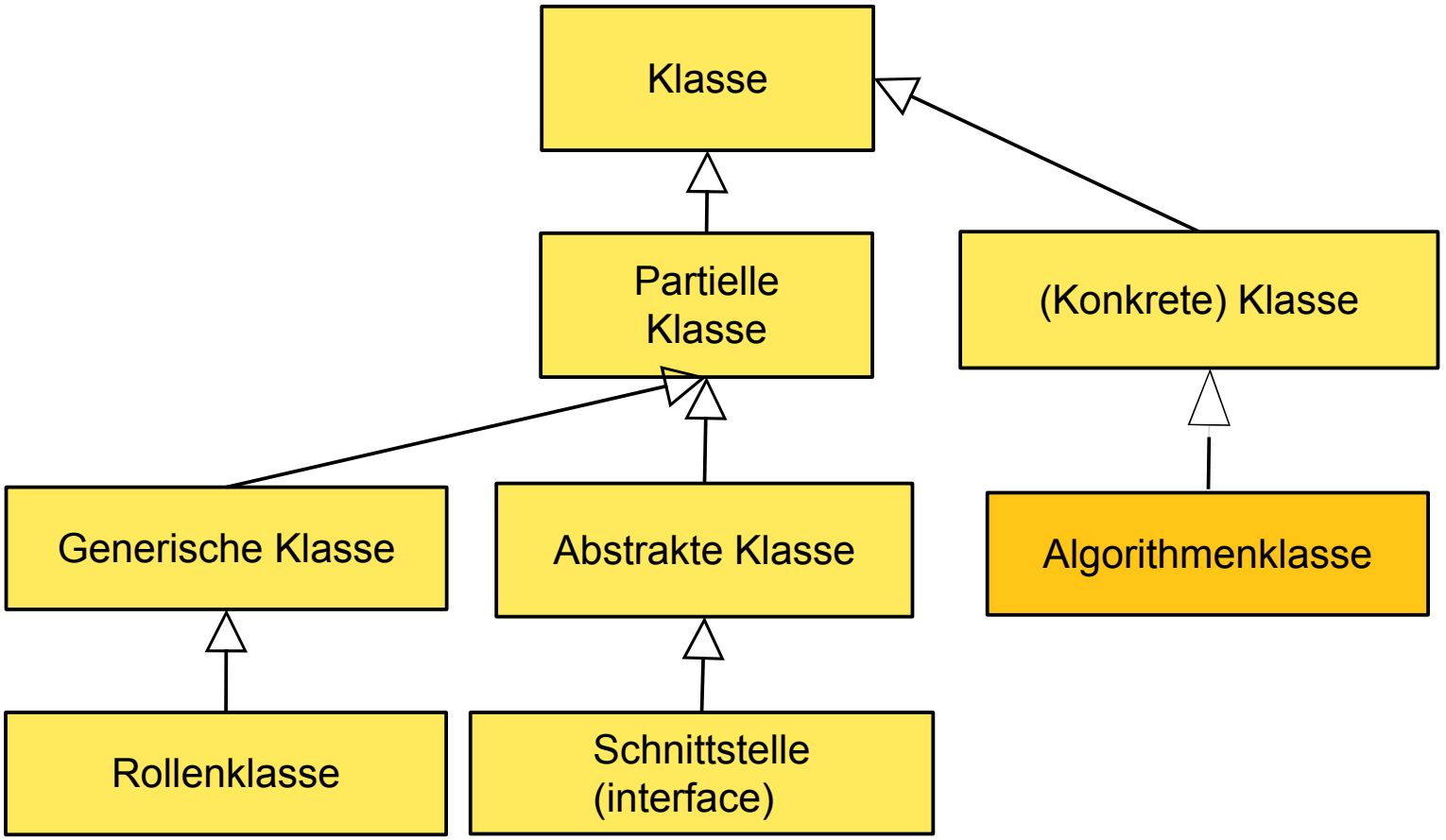
Standardalgorithmen in der *Algorithmenklasse* java.util.Collections

- **Algorithmenklassen (Hilfsklassen)** enthalten Algorithmen, die auf einer Familie von anderen Klassen arbeiten
  - Hier: java.util.Collections enthält Algorithmen auf beliebigen Klassen, die das Collection- bzw. List-Interface implementieren
  - Bei manchen Operationen ist Ordnung auf Elementen vorausgesetzt.
  - Achtung: Statische Klassenoperationen!

```
public class Collections<E> {
  public static E max (Collection<E> coll);
  public static E min (Collection<E> coll);
  public static int binarySearch(List<E> list, E
key);
  public static void reverse (List<E> list);
  public static void sort (List<E> list)
  ...
}
```


file://localhost/Users/ua1/Courses/ST1/Material/jdk-docs/api/java/util/Collections.html

Begriffshierarchie von Klassen (Erweiterung)



Typschranken generischer Parameter (type bounds)

- Beispiel: Comparable<E> als Return-typ in der Collections-Klasse sichert zu, dass die Methode compareTo() existiert

```
class Collections {
/** minimum function for a Collection. Return value is typed
  * with a generic type with a type bound */

public static <E extends Comparable<E>> min(Collection<E> ce) {
   Iterator<E> iter =  ce.iterator();
   E curMin = iter.next;
   if (curMin == null) return curMin;
   for (E element = curMin;
                  iter.hasNext(), element = iter.next) {
          if (element.compareTo(curMin) < 0) {
                  curMin = element;
          }
   }
   return curMin;
}
```

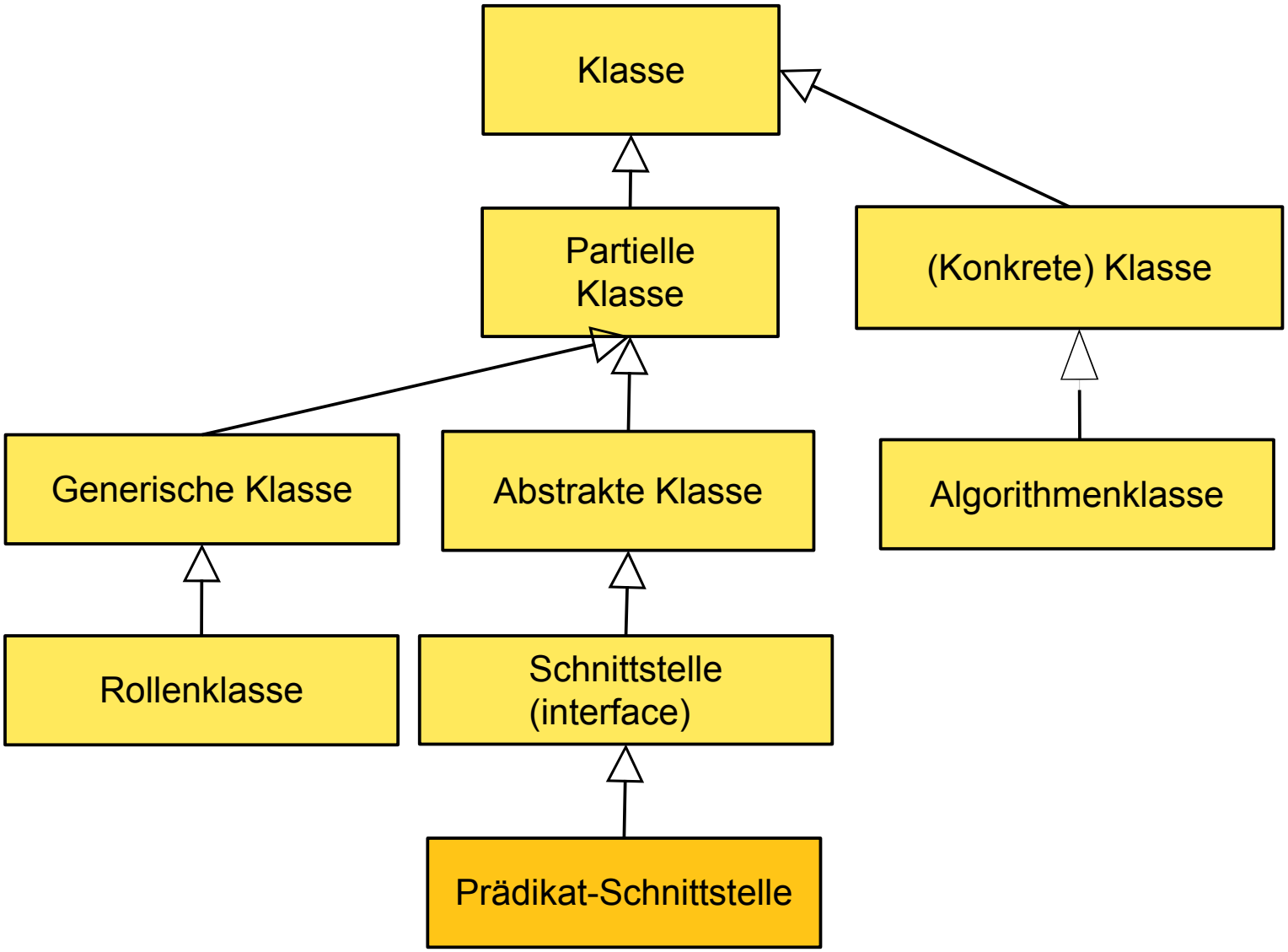
Prädikat-Schnittstellen (...able Schnittstellen)

- **Prädikat-Schnittstellen** drücken bestimmte Eigenschaft einer Klasse aus. Sie werden oft mit dem Suffix “able” benannt:
  - Iterable
  - Clonable
  - Serializable
- Beispiel: geordnete Standarddatentypen (z.B. String oder List) implementieren die Prädikatschnittstelle *Comparable:*

```
public interface Comparable<E> {
  public int compareTo (E o);
}
```

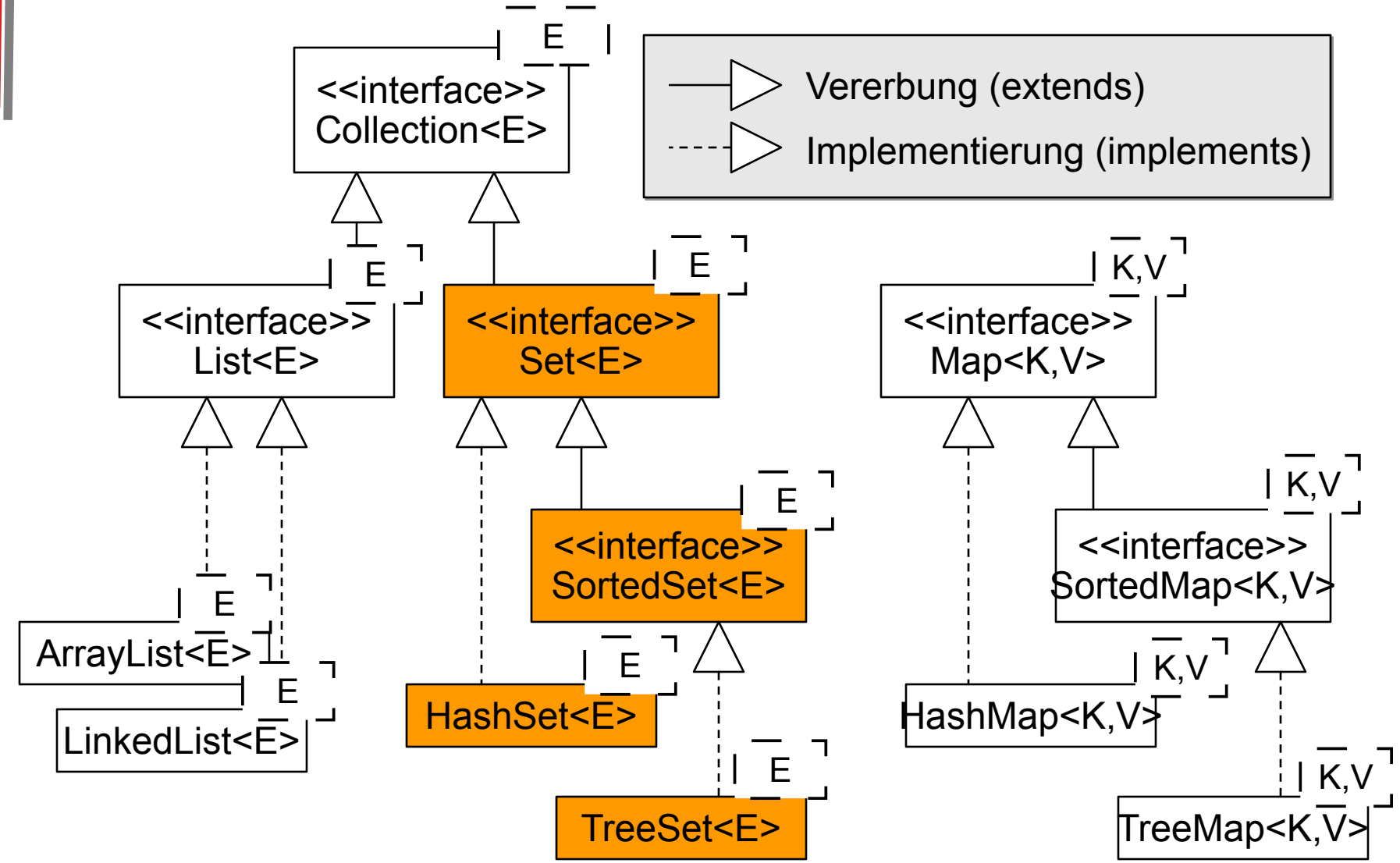
- Resultat ist kleiner/gleich/größer 0:
  genau dann wenn "this" kleiner/gleich/größer als Objekt o

Begriffshierarchie von Klassen (Erweiterung)

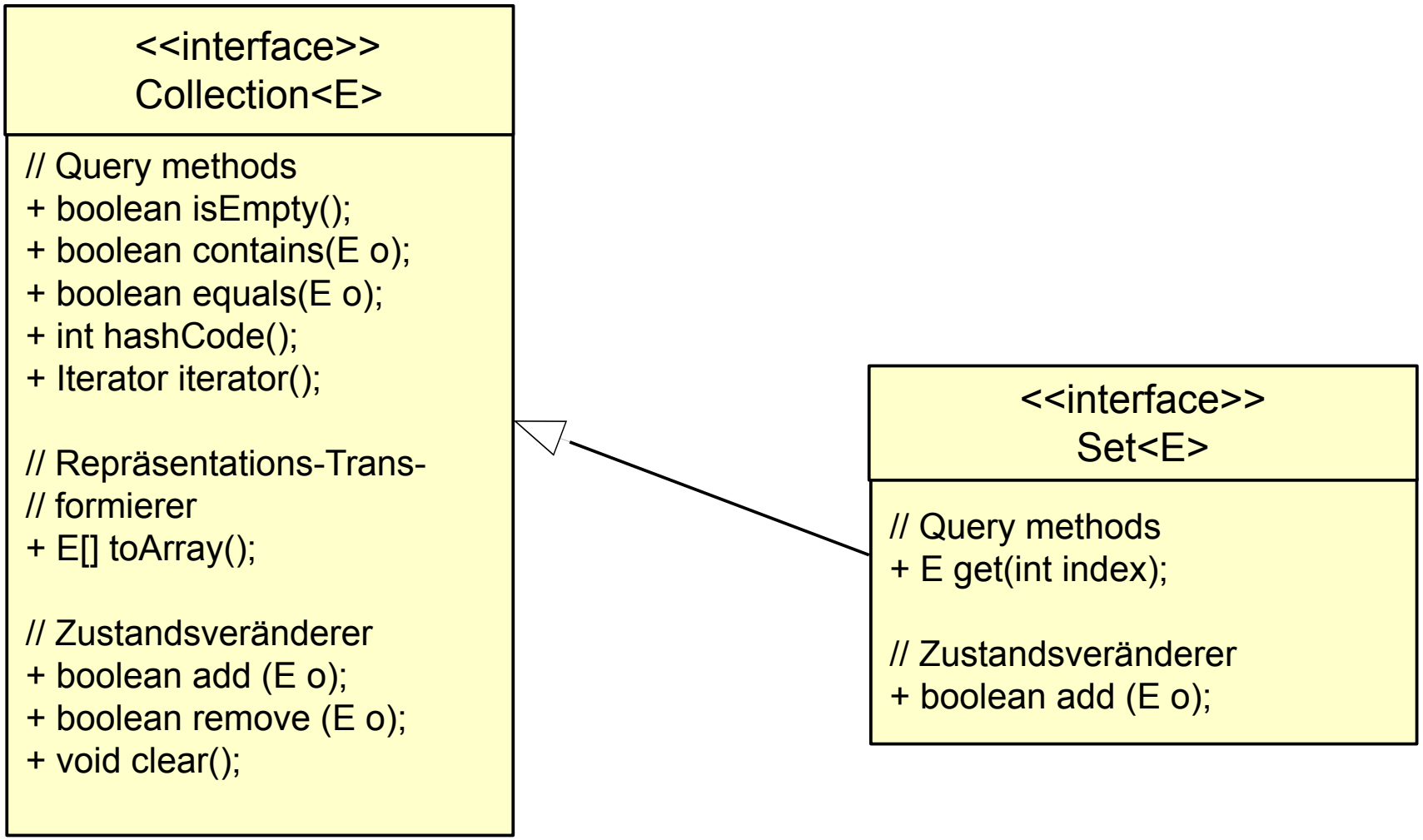


21.5 Ungeordnete Collections: Set und Map

Mengen (ohne Mehrfacheintrag) im Collection Framework

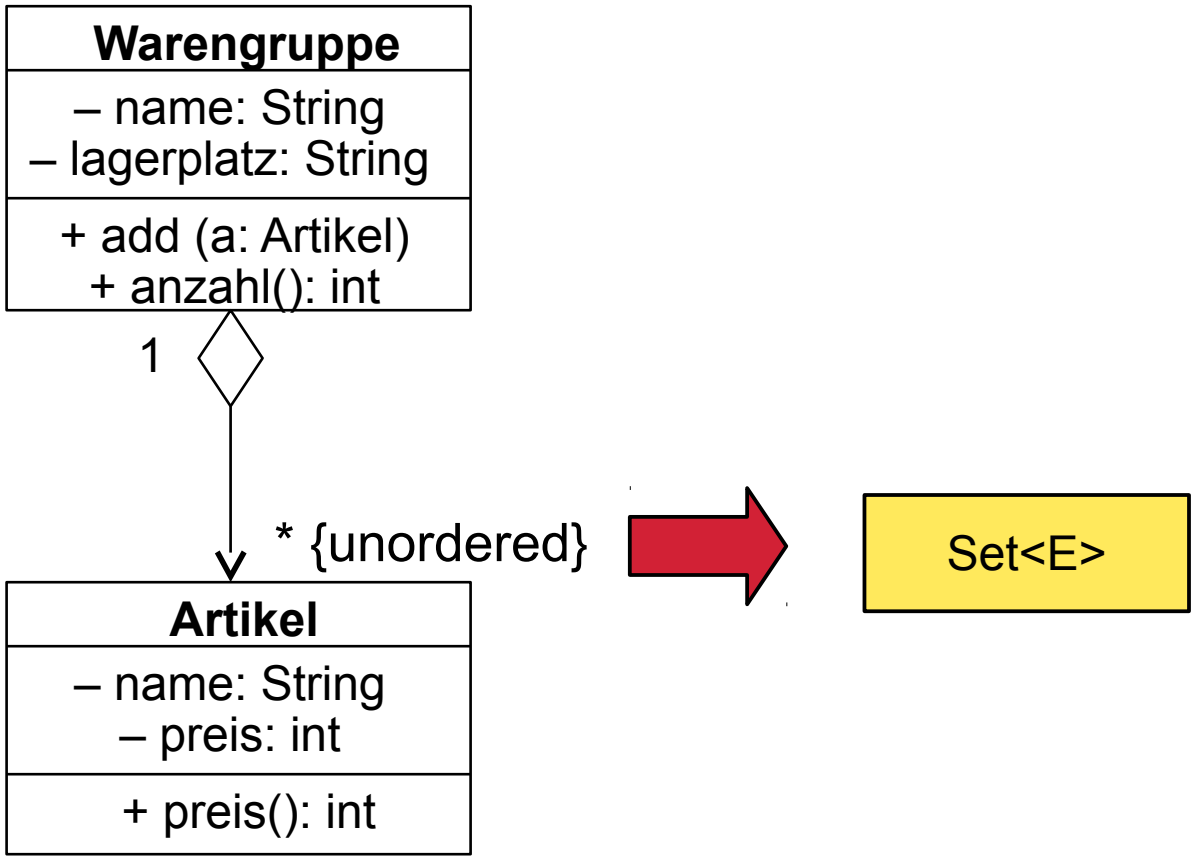


Ungeordnete Mengen: java.util.Set<E> (Auszug)

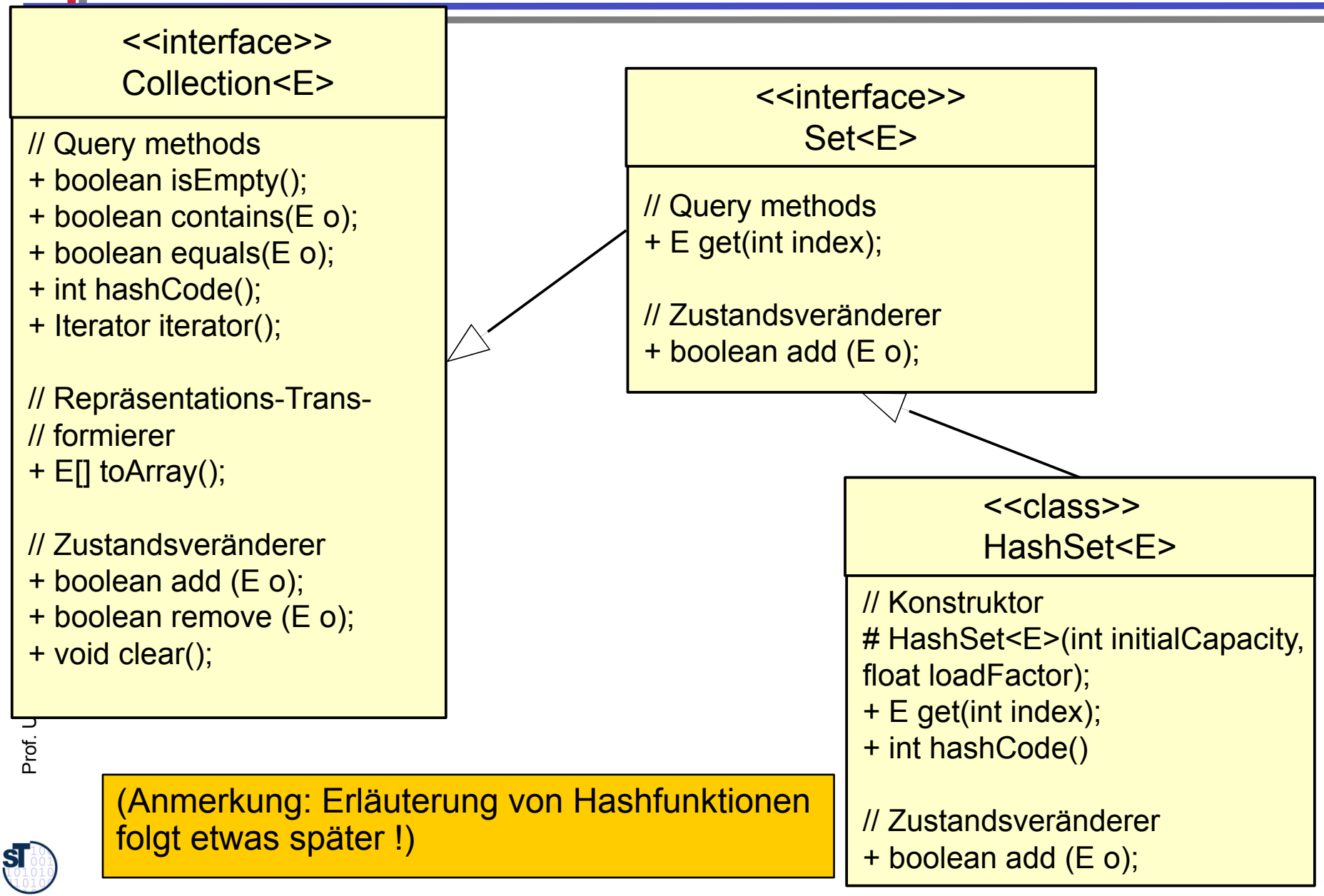


Anwendungsbeispiel für Set<E>

- Eine Assoziation in UML kann als {unordered} gekennzeichnet sein



Konkreter Datentyp java.util.HashSet<E> (Auszug)



(Anmerkung: Erläuterung von Hashfunktionen folgt etwas später !)

Anwendungsbeispiel mit HashSet<E>


```
class Warengruppe {

  private String name;
  private String lagerplatz;
  private Set<Artikel> inhalt;

  public Warengruppe
     (String name, String lagerplatz) {
    this.name = name;
    this.lagerplatz = lagerplatz;
    this.inhalt = new HashSet<Artikel>();
  }

  public void add (Artikel a) { inhalt.add(a); }

  public int anzahl() { return inhalt.size(); }

  public String toString() {
    String s = "Warengruppe "+name+"\n";
    Iterator it = inhalt.iterator();
    while (it.hasNext()) {
      s += " "+(Artikel)it.next();
    };
  }
}
```

Online:
Warengruppe0.java

Duplikatsprüfung für Elemente in Mengen: Wann sind Objekte gleich? (1)

- Der Vergleich mit Operation **==** prüft auf Referenzgleichheit, d.h. physische Identität der Objekte
  - Typischer Fehler: Stringvergleich mit "=="
    (ist nicht korrekt, geht aber meistens gut!)

- Alternative: Vergleich mit Gleichheitsfunktion **`o.equals():`**
  - deklariert in java.lang.Object
  - überdefiniert in vielen Bibliotheksklassen
    - z.B. java.lang.String
  - für selbstdefinierte Klassen
    - Standardbedeutung Referenzgleichheit
    - bei Bedarf selbst überdefinieren !

    (Ggf. für ***kompatible*** Definition der Operation **`o.hashCode()`** aus java.lang.Object sorgen)

Wann sind Objekte gleich? (2) Referenzgleichheit


```
public static void main (String[] args) {
  Warengruppe w1 = new Warengruppe("Moebel","L1");
  w1.add(new Artikel("Tisch",200));
  w1.add(new Artikel("Stuhl",100));
  w1.add(new Artikel("Schrank",300));
  w1.add(new Artikel("Tisch",200));
  System.out.println(w1);
}
```

Systemausgabe beim Benutzen der Standard-Gleichheit:

```
Warengruppe Moebel
Tisch(200) Tisch(200) Schrank(300) Stuhl(100)
```

Online:
Warengruppe0.java

Wann sind Objekte gleich? (3) Referenzgleichheit


```
public static void main (String[] args) {
  Artikel tisch = new Artikel("Tisch",200);
  Artikel stuhl = new Artikel("Stuhl",100);
  Artikel schrank = new Artikel("Schrank",300);

  Warengruppe w2 = new Warengruppe("Moebel","L2");
  w2.add(tisch);
  w2.add(stuhl);
  w2.add(schrank);
  w2.add(tisch);
  System.out.println(w1);
}
```

Systemausgabe bei Referenzgleichheit:

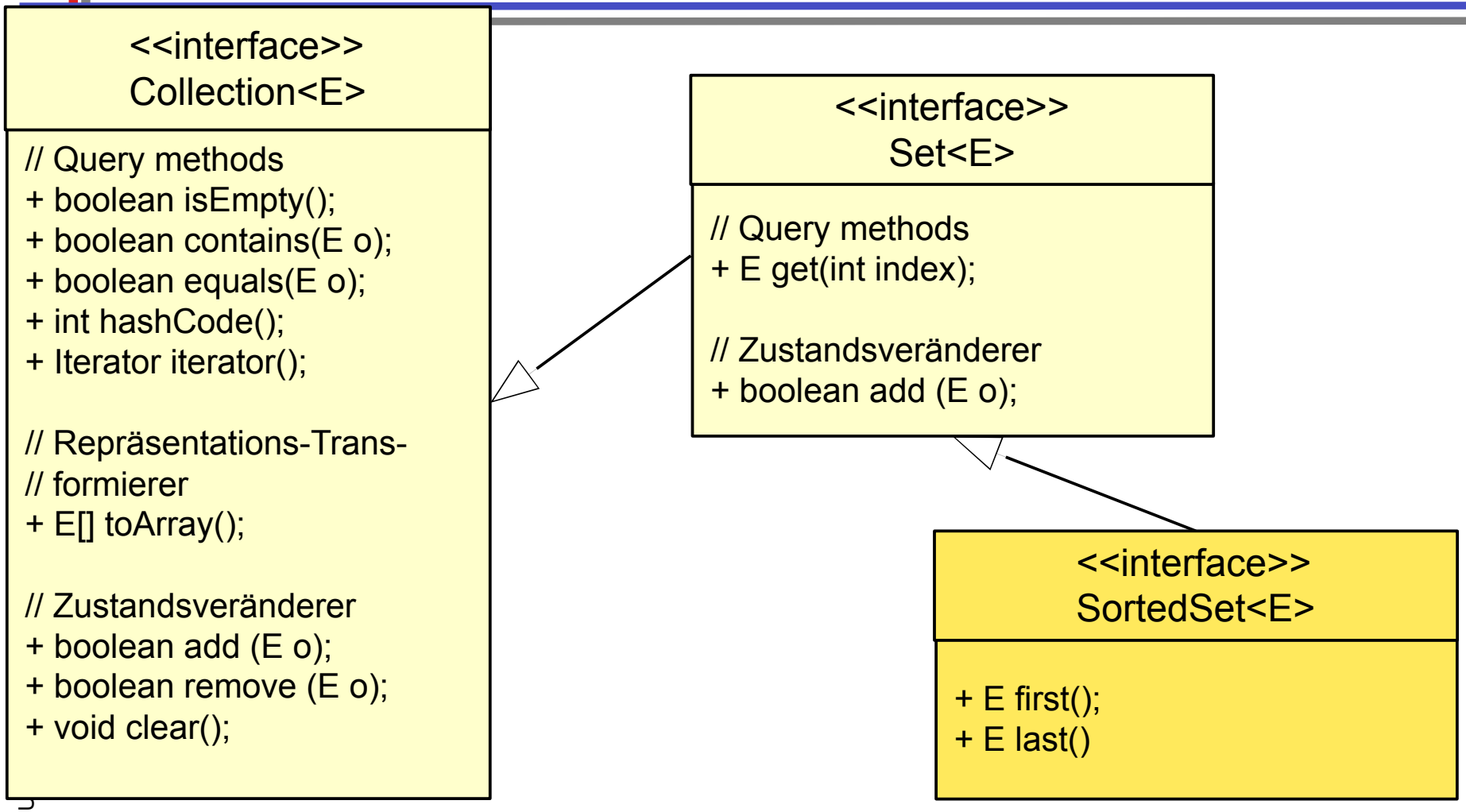
```
Warengruppe Moebel
Schrank(300) Tisch(200) Stuhl(100)
```

Es wurde zweifach dasselbe Tisch-Objekt übergeben !
(Gleiches Verhalten bei Strukturgleichheit, s. Warengruppe1.java)

Online: Warengruppe1.java

Online: Warengruppe2.java

java.util.SortedSet<E> (Auszug)



Sortierung von Mengen mit TreeSet nutzt Vergleichbarkeit von Elementen

- java.util.TreeSet<E> implementiert ein geordnete Menge mit Hilfe eines Baumes und benötigt zum Sortieren dessen die Prädikat-Schnittstelle `Comparable<E>`
- Modif kation der konkreten Klasse Warengruppe:

```
class Warengruppe<E> {
  private Set<E> inhalt;
  public Warengruppe (…) {
     …
     this.inhalt = new TreeSet<E>();
  } …
}
```

- Aber Systemreaktion:

Exception in thread "main" java.lang.ClassCastException: Artikel
at java.util.TreeMap<K,V>.compare(TreeMap<K,V>.java, Compiled Code)

- in java.util.TreeSet<E>:
  `public class TreeSet<E> … implements SortedSet<E> … { … }`

Anwendungsbeispiel mit TreeSet<E>

- Artikel muss von Schnittstelle `Comparable<Artikel>` erben
- Modifikation der Klasse „Artikel“:

```
class Artikel implements Comparable<Artikel> {
...
    public int compareTo (Artikel o) {
      return name.compareTo(o.name);
    }
}
```

Systemausgabe:

```
Warengruppe Moebel
Schrank(300) Stuhl(100) Tisch(200)
```

Online:
Warengruppe3.java

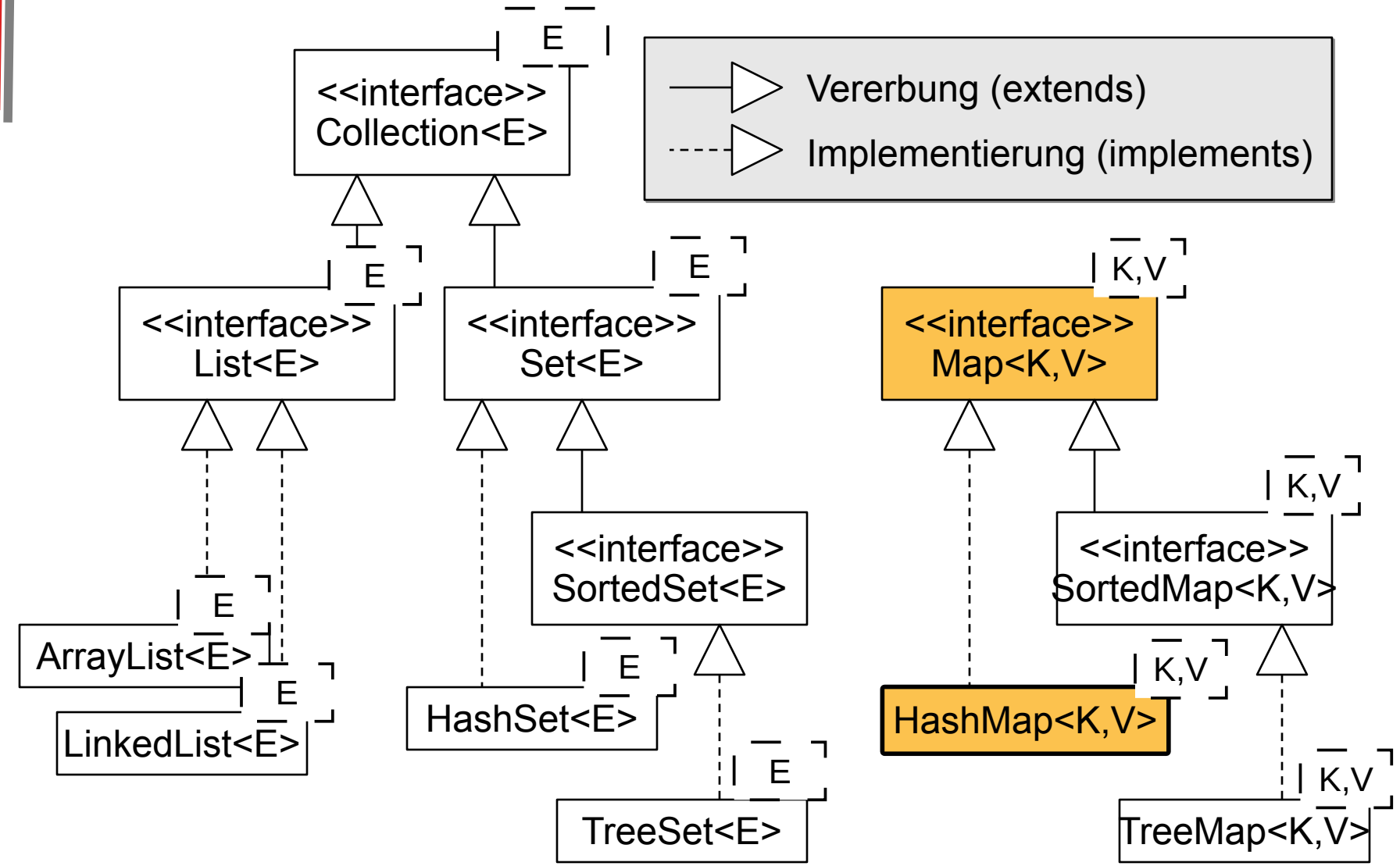
HashSet oder TreeSet?

- Gemessener relativer Aufwand für Operationen auf Mengen:
  (aus Eckel, Thinking in Java, 2nd ed., 2000)

Typ	Einfügen	Enthalten	Iteration
HashSet	36,14	106,5	39,39
TreeSet	150,6	177,4	40,04

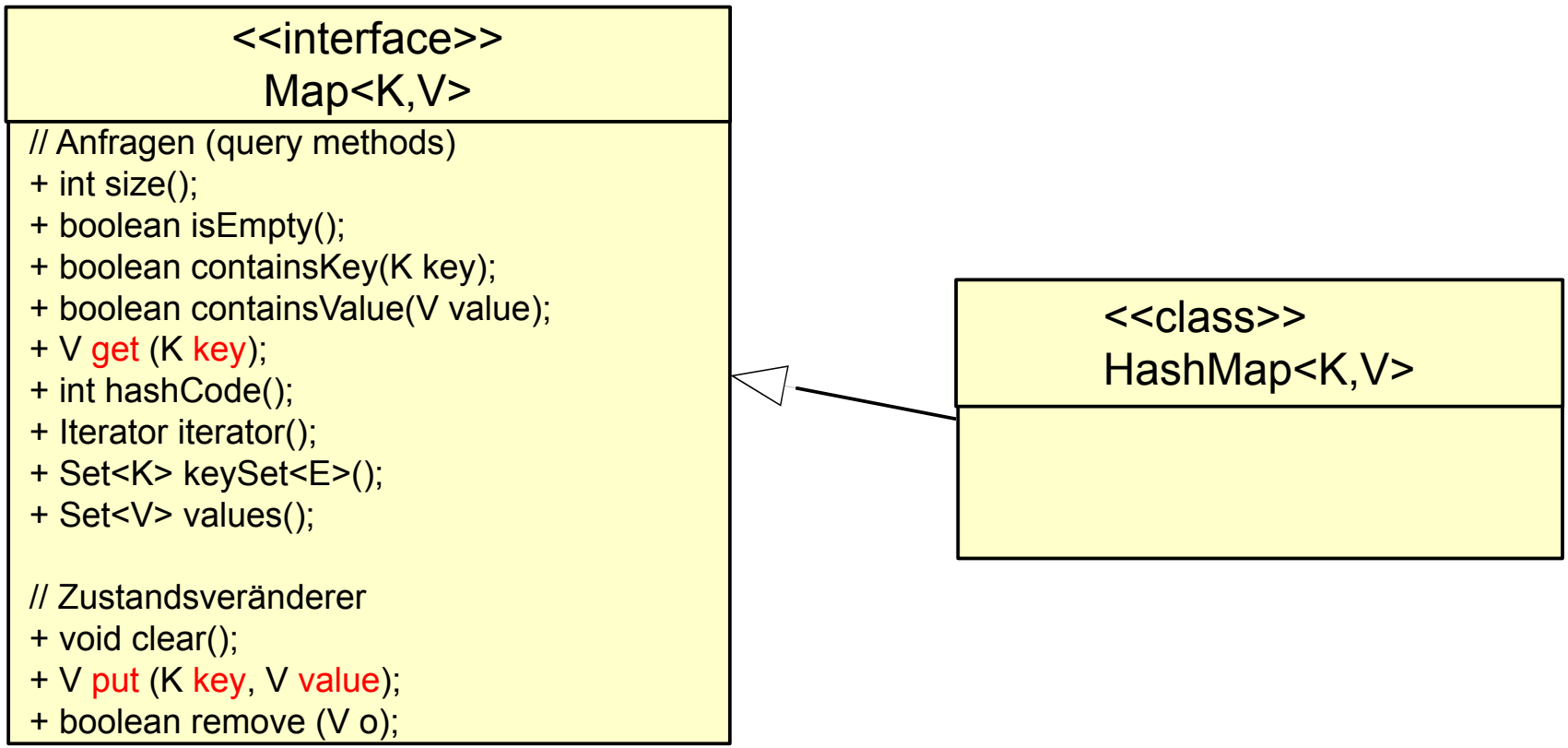
- Stärken von HashSet:
  - in allen Fällen schneller !
- Stärken von TreeSet:
  - erlaubt Operationen für sortierte Mengen

Collection Framework (Überblick) (Maps)

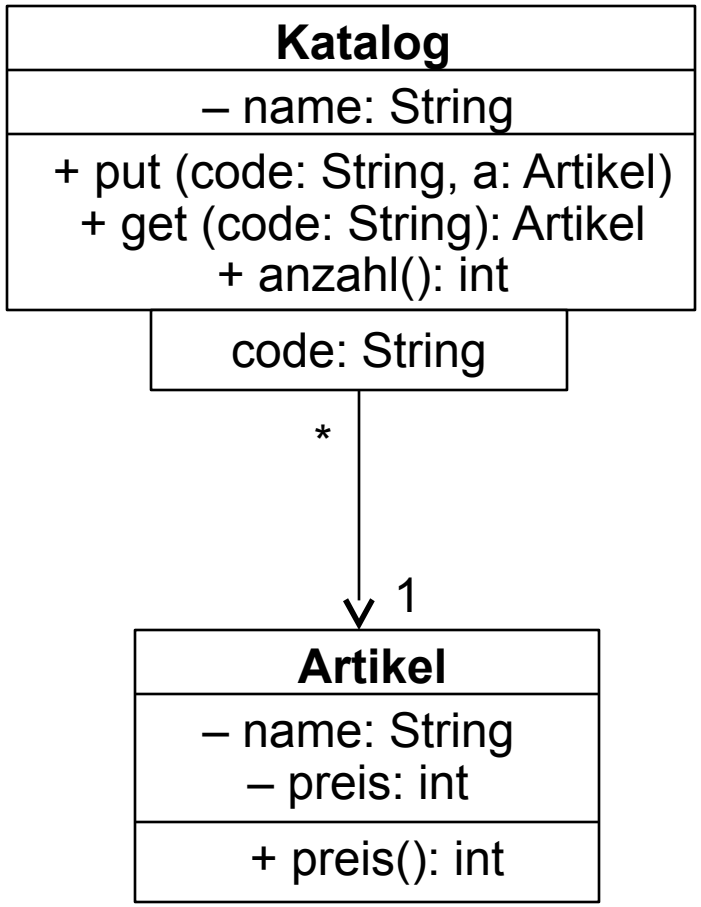


java.util.Map<K,V> (Auszug)

- Eine Map ist ein „assoziativer Speicher“ (associative array), der Objekte als Werte (value) unter Schlüsseln (key) zugreifbar macht
  - Ein Schlüssel liefert einen Wert (Funktion).
  - Map liefert funktionale Abhängigkeit zwischen Schlüssel und Wert



Anwendungsbeispiel: Verfeinerung von qualifizierten Assoziationen in UML



HashMap ist eine sehr günstige Umsetzung für *qualifizierte* Assoziationen:

Der Qualifikator bildet den Schlüssel; die Zielobjeke den Wert

Hier:

- Schlüssel: code:String
- Wert: a:Artikel

Anwendungsbeispiel mit HashMap


```
class Katalog {
  private String name;
  private Map<String,Artikel> inhalt; // Polymorphe Map

  public Katalog (String name) {
    this.name = name;
    this.inhalt = new HashMap<String,Artikel>();
  }

  public void put (String code, Artikel a) {
    inhalt.put(code,a);
  }

  public int anzahl() {
    return inhalt.size();
  }

  public Artikel get (String code) {
    return (Artikel)inhalt.get(code);
  }
  ...
}
```

Online: Katalog.java

Testprogramm für Anwendungsbeispiel: Speicherung der Waren mit Schlüsseln


```
public static void main (String[] args) {
    Artikel tisch = new Artikel("Tisch",200);
    Artikel stuhl = new Artikel("Stuhl",100);
    Artikel schrank = new Artikel("Schrank",300);
    Artikel regal = new Artikel("Regal",200);

    Katalog k = new Katalog("Katalog1");
    k.put("M01",tisch);
    k.put("M02",stuhl);
    k.put("M03",schrank);
    System.out.println(k);

    k.put("M03",regal);
    System.out.println(k);
}
```

Systemausgabe:

```
Katalog Katalog1
 M03 -> Schrank(300)
 M02 -> Stuhl(100)
 M01 -> Tisch(200)

Katalog Katalog1
 M03 -> Regal(200)
 M02 -> Stuhl(100)
 M01 -> Tisch(200)
```

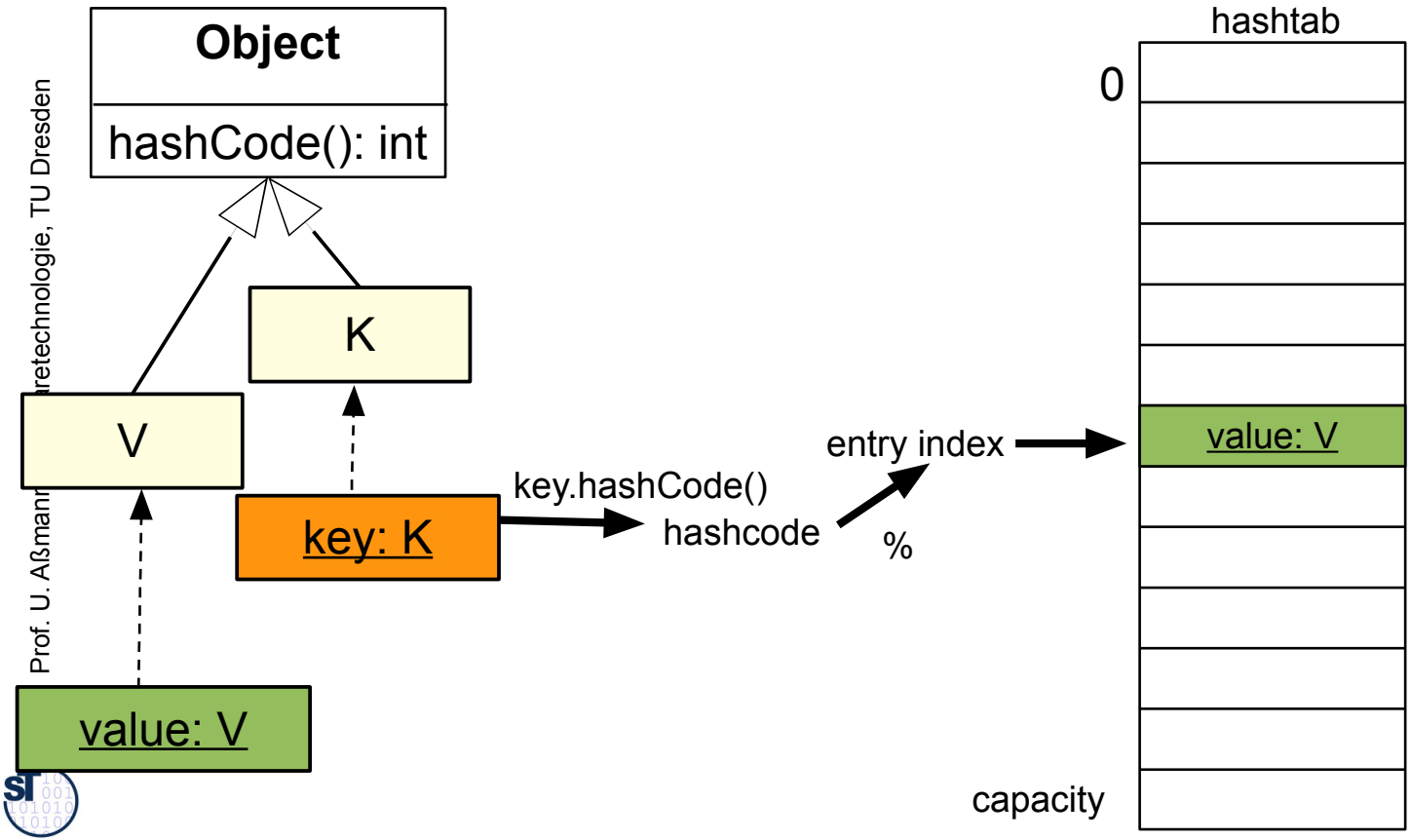
put(...) überschreibt vorhandenen Eintrag (Ergebnis = vorhandener Eintrag).

Ordnung auf den Schlüsseln: SortedMap (Implementierung z.B.TreeMap).

Prinzip der Hashtabelle

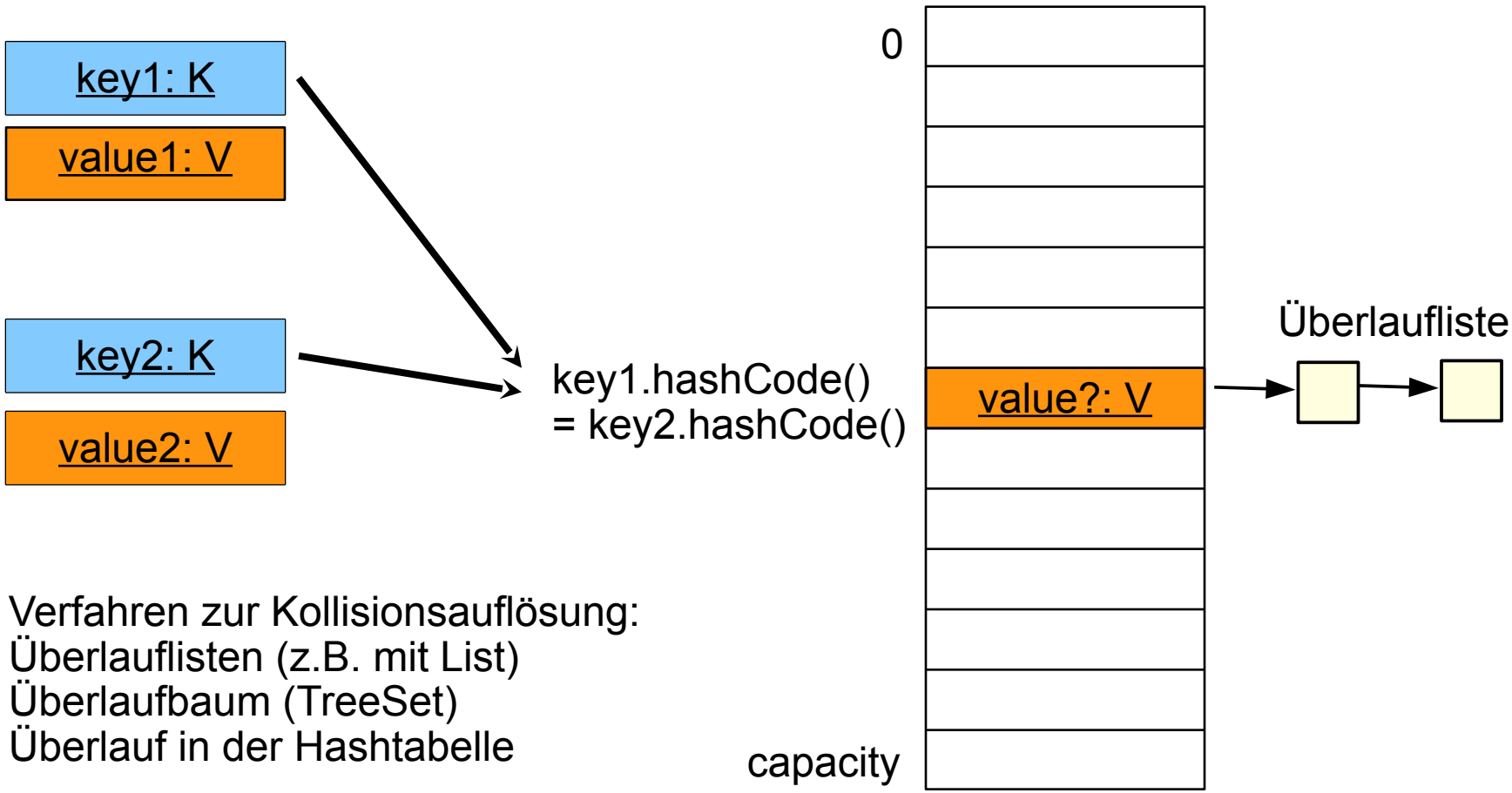
Effekt von hashtab.put(key:K,value:V)

- Typischerweise wird der Schlüssel (key) transformiert:
  - Das Objekt liefert seinen Hashwert mit der Hash-Funktion hashCode()
  - Der Hashwert wird auf einen Zahlenbereich modulo der Kapazität der Hashtabelle abgebildet, d.h., der Hashwert wird auf die Hashtabelle “normiert”
  - Mit dem Eintragswert wird in eine Hashtabelle eingestochen



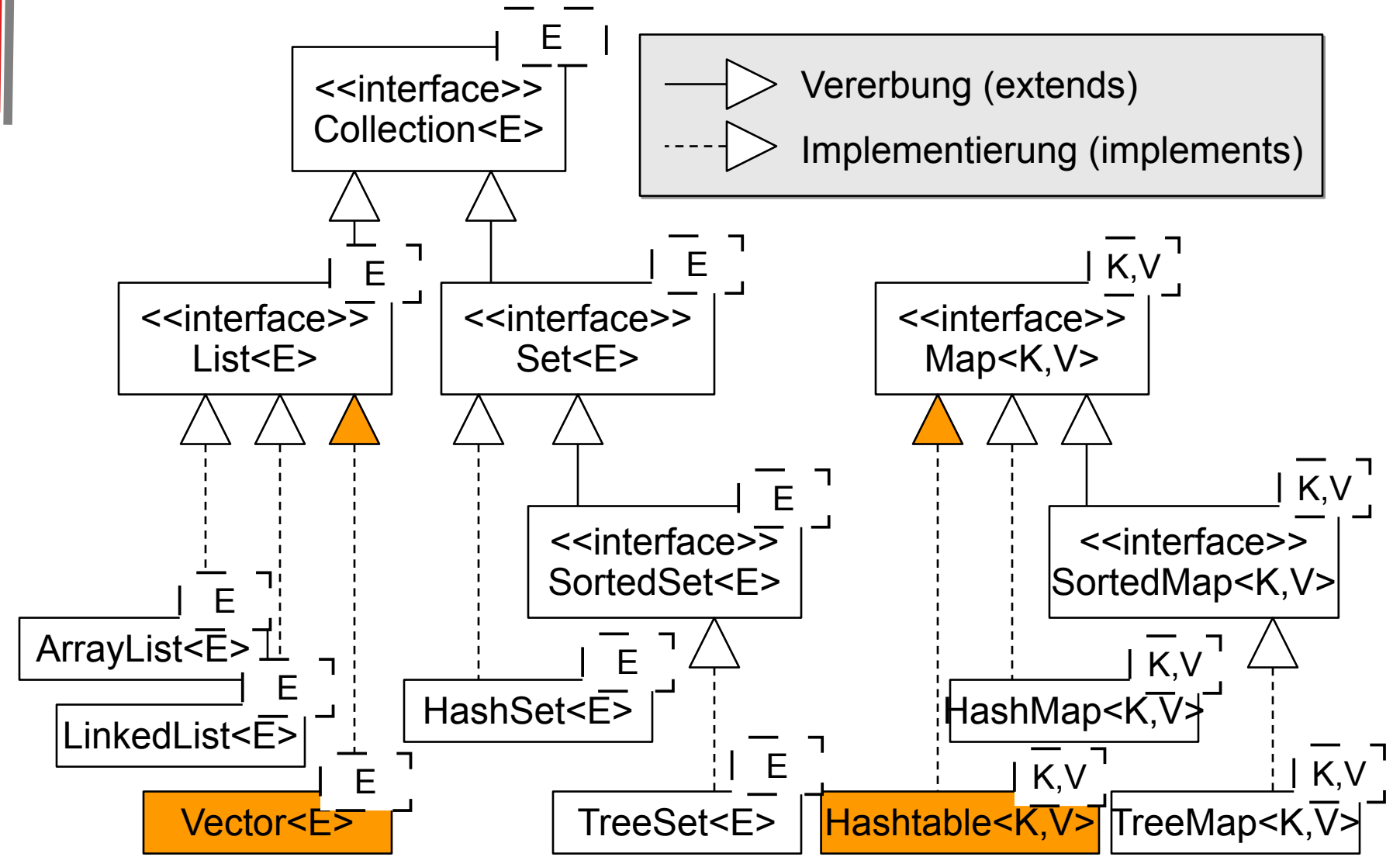
Kollision beim Einstechen

- Die Hashfunktion ist *mehrdeutig (nicht injektiv):*
  - Bei nicht eindeutigen Schlüsseln, oder auch durch die Normierung, werden Einträge doppelt “adressiert” (Kollision)



Verfahren zur Kollisionsauflösung:
Überlauflisten (z.B. mit List)
Überlaufbaum (TreeSet)
Überlauf in der Hashtabelle

Weitere Implementierungen in der Collection-Hierarchie



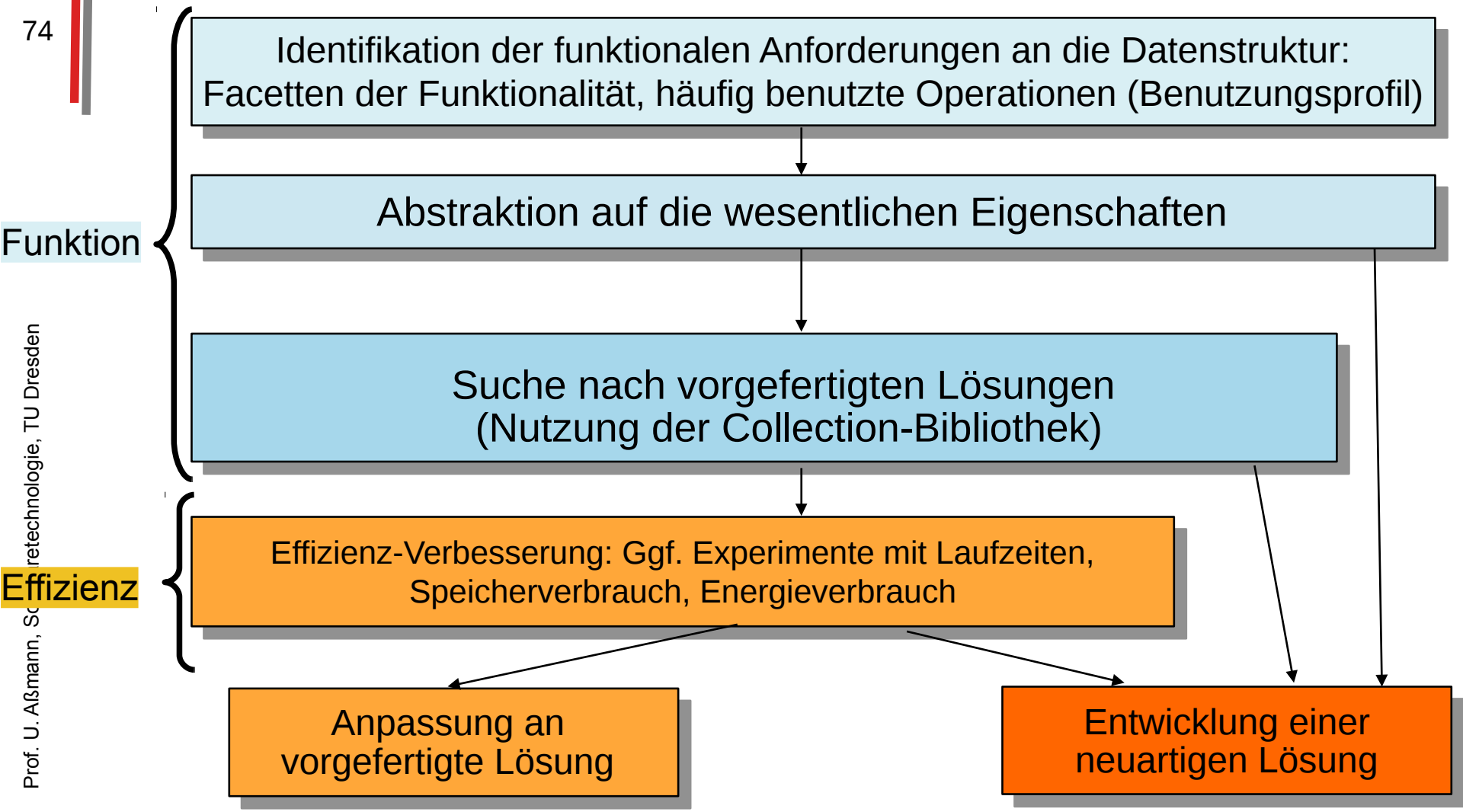
21.6 Optimierte Auswahl von Implementierungen von Datenstrukturen

Facetten und Assoziationsarten

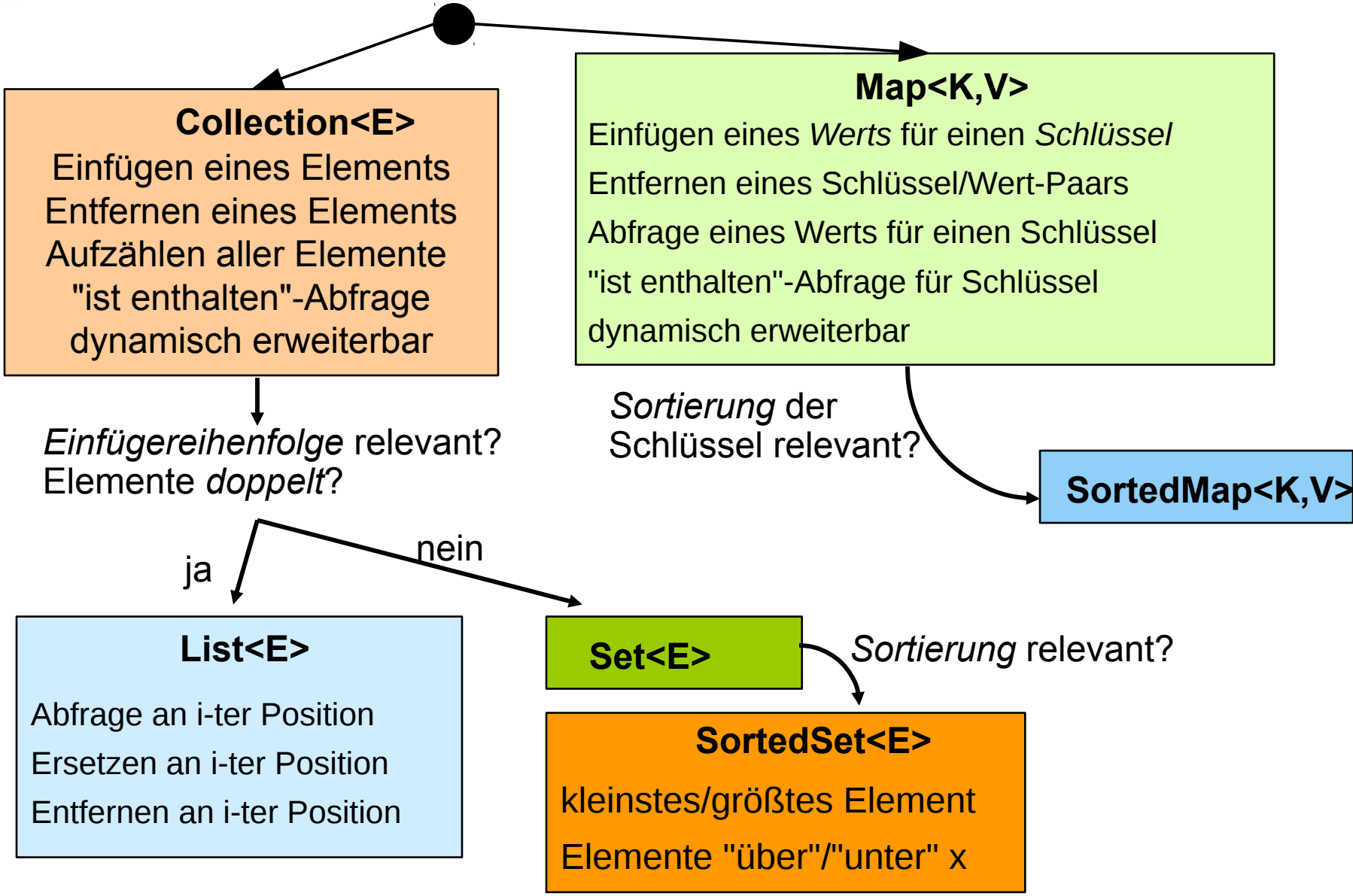
- An das Ende einer UML-Assoziation können folgende *Bedingungen* notiert werden:
  - Ordnung: {ordered} {unordered}
  - Eindeutigkeit: {unique} {non-unique}
  - Kollektionsart: {set} {bag} {sequence}
  - Zusammenhang: {union} {subsets}
- Beim Übergang zum Implementierungsmodell müssen diese Bedingungen auf Unterklassen von Collections abgebildet werden

Ordnung	**Duplikate**	**Sortierung**	**Schlüssel**
geordnet ungeordnet	mit Duplikaten ohne Duplikate	sortiert unsortiert	mit Schlüssel ohne Schlüssel

Vorgehensweise beim funktionalen und effizienzbasierten Datenstruktur-Entwurf



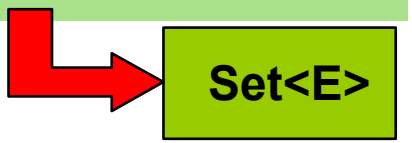
Suche nach vorgefertigten Lösungen (anhand der Facetten der Collection-Klassen)



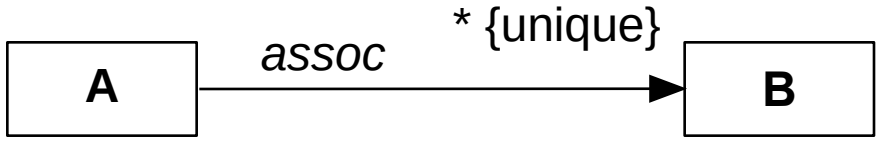
Beispiel: Realisierung von Assoziationen

Datenstruktur im A-Objekt für B-Referenzen

Anforderung	Realisierung
1) Assoziation anlegen	1) Einfügen (ohne Reihenfolge)
2) Assoziation entfernen	2) Entfernen (ohne Reihenfolge)
3) Durchlaufen aller bestehenden Assoziationen zu B-Objekten	3) Aufzählen aller Elemente
4) Manchmal: Abfrage, ob Assoziation zu einem B-Objekt besteht	4) "ist enthalten"-Abfrage
5) Keine Obergrenze der Multiplizität gegeben	5) Maximalanzahl der Elemente unbekannt; dynamisch erweiterbar



Realisierung von ungeordneten Assoziationen mit Set<E>




```
class A {
  private Set<B> assoc;
  ...

  public void addAssoc (B b) {
    assoc.add(b);
  }

  public boolean testAssoc (B b) {
    return assoc.contains(b);
  }

public A {
    ...
    assoc = new HashSet<B>();
}
```

Beispiel: Raumverwaltung


```
static Besprechungsraum freienRaumSuchen
   (int groesse, Hour beginn, int dauer)
```

- Suche unter vorhandenen Räumen nach Raum mit mindestens der Kapazität *groesse*, aber möglichst klein.
  - Datenstruktur für vorhandene Räume in Klasse Raumverwaltung
    - » SortedSet<Besprechungsraum> (Elemente: Besprechungsraum)
- Überprüfung eines Raumes, ob er für die Zeit ab *beginn* für die Länge *dauer* bereits belegt ist.
  - Operation in Klasse Besprechungsraum:

    `boolean frei (Hour beginn, int dauer`)
  - Datenstruktur in Klasse Besprechungsraum für Zeiten (Stunden):
    - » Set<Hour> (Elemente: Hour)
- Zusatzanforderung (Variante): Überprüfung, welcher andere Termin eine bestimmte Stunde belegt.
  - Datenstruktur in Klasse Besprechungsraum:
    - » Map<Hour,Teambesprechung> (Schlüssel: Hour, Wert: Teambesprechung)

Raumverwaltung: Freien Raum suchen


```
class Raumverwaltung {

  // Vorhandene Raeume, aufsteigend nach Größe sortiert
  // statisches Klassenattribut und -methode
  private static SortedSet<E> vorhandeneRaeume
      = new TreeSet<Besprechungsraum>();

  // Suche freien Raum aufsteigend nach Größe
  static Besprechungsraum freienRaumSuchen
   (int groesse, Hour beginn, int dauer) {
    Besprechungsraum r = null;
    boolean gefunden = false;
    Iterator it = vorhandeneRaeume.iterator();
    while (! gefunden && it.hasNext()) {
      r = (Besprechungsraum)it.next();
      if (r.grossGenug(groesse)&& r.frei(beginn,dauer))
        gefunden = true;
    };
    if (gefunden)
      return r;
    else
      return null;
  } ...
}
```

Was haben wir gelernt

- Static vs. dynamic vs. gradual vs. no typing
- Safe Application Development (SAD) nur mit statischen Typisierung möglich
- Rapid Application Development (RAD) benötigt dynamisches oder graduelle Typisierung
- Generische Collections besitzen den Element-Typ als Typ-Parameter
  - Element-Typ verfeinert Object
  - Weniger Casts, mehr Typsicherheit
- Implementierungsmuster Command

The End

- Diese Folien bauen auf der Vorlesung Softwaretechnologie auf von © Prof. H. Hussmann, 2002. Used by permission.

Appendix A
Generische Command Objekte

Implementierungsmuster Command: Generische Methoden als Funktionale Objekte

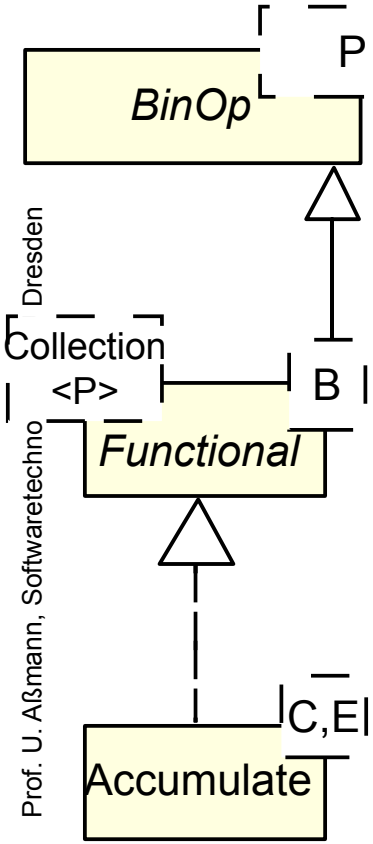
Ein **Funktionalobjekt (Kommandoobjekt)** ist ein Objekt, das eine Funktion darstellt (reifiziert).

- **Funktionalobjekte** können Berechnungen kapseln und später ausführen (laziness) (Entwurfsmuster Command)
  - Es gibt eine Standard-Funktion in der Klasse des Funktionalobjektes, das die Berechnung ausführt (Standard-Name, z.B. *execute()* oder *doIt())*
- Zur Laufzeit kann man das Funktionalobjekt mit Parametern versehen, herumreichen, und zum Schluss ausführen

```
// A functional object that is like a constant
interface NullaryOpCommand { void execute();
   void undo(); }
// A functional object that takes one parameter
interface UnaryOpCommand<P> { P execute(P p1);
   void undo(); }
// A functional object that operates on two parameters
interface BinOp<P> { P execute(P p1, P p2);
   void undo(); }
```

Generische Methoden als Funktionale Objekte

- Anwendung: Akkumulatoren und andere generische Listenoperationen




```
// An interface for a collection of binary operation on
// collections
interface Functional<Collection<P>,B extends BinOp<P>> {
   P compute(Collection<P> p);
}
class Accumulate<C,E> implements Functional<C,BinOp<E>> {
  E curSum;       E element;     BinOp<E> binaryOperation;
  public E compute(C coll) {
    for (int i = 0; i < coll.size(); i++) {
   element = coll.get(i);
   curSum = binaryOperation.execute(curSum,element);
    }
    return curSum;
  }
}
```