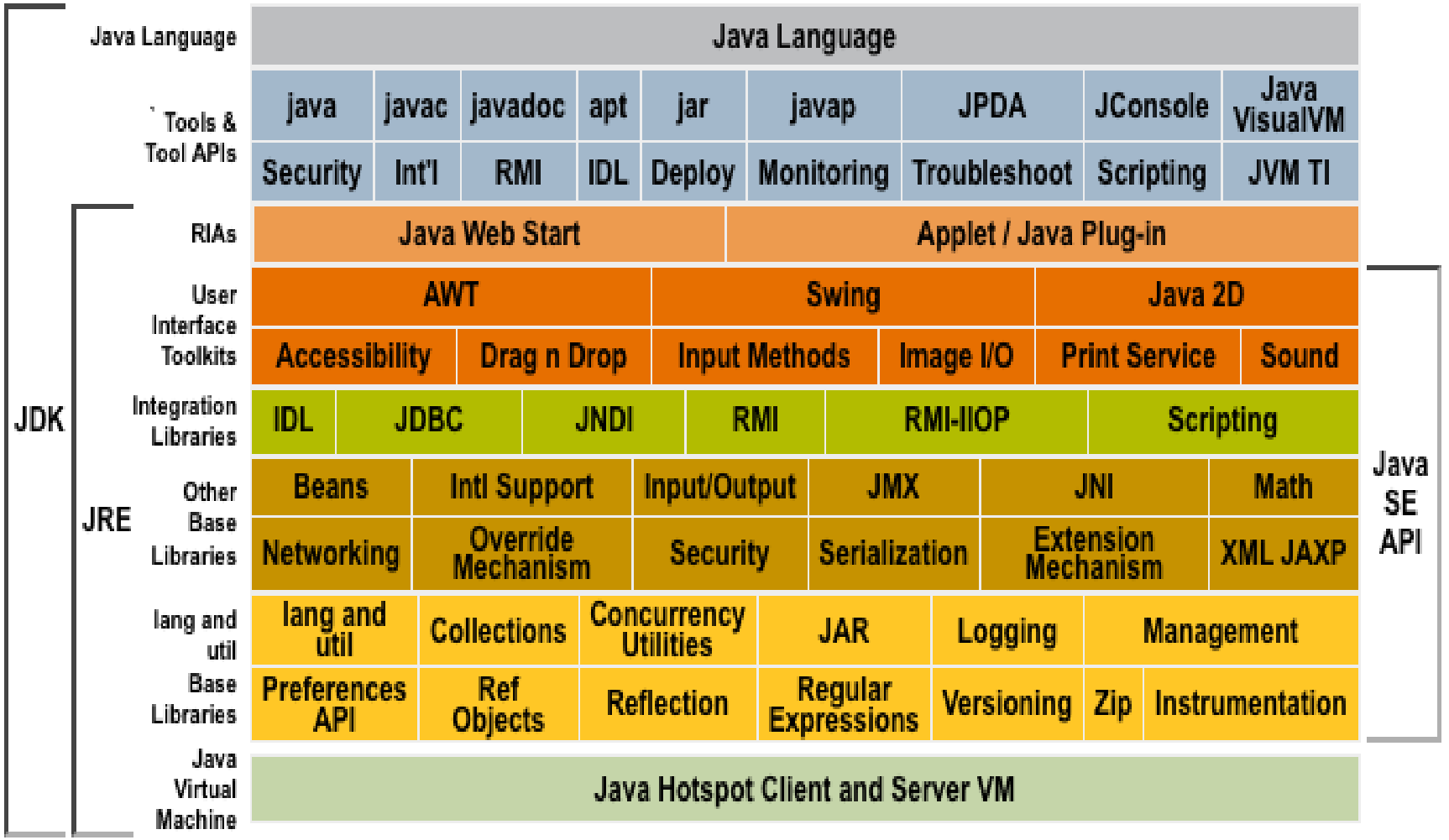
21) Netzverfeinerung (von UML-Assoziationen) mit dem Java-2 Collection Framework

Prof. Dr. rer. nat. Uwe Aßmann
Institut für Software- und Multimediatechnik
Lehrstuhl Softwaretechnologie
Fakultät für Informatik
TU Dresden
Version 20-1.1, 22.05.20
some bugs corrected

Obligatorische Literatur

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



Empfohlene Literatur

- Im Wesentlichen sind die Dokumentationen aller Versionen von Java 6 an nutzbar:
  - http://download.oracle.com/javase/6/docs/
  - http://download.oracle.com/javase/8/docs
  - https://docs.oracle.com/en/java/javase/12/
  - https://docs.oracle.com/en/java/javase/12/ docs/api/java.base/module-summary.html
- Tutorials http://download.oracle.com/javase/tutorial/
- https://docs.oracle.com/javase/tutorial/tutorialLearningPaths.html
- Generics Tutorial:
- http://download.oracle.com/javase/tutorial/extra/generics/index.html

https://de.wikipedia.org/wiki/Tom_Dooley_(Lied)

Video für Gitarrelernen mit Tom Dooley
https://www.youtube.com/watch?v=_J8fFFSVp3Y

http://www.magistrix.de/lyrics/The%20Kingston%20Trio/Tom-Dooley-228080.html

“Stay hungry, stay foolish” (Steve Jobs)

- http://news.stanford.edu/2005/06/14/jobs-061505/ (English)
- http://www.mac-history.de/apple-people/steve-jobs/2008-10-05/ubersetzung-der-rede-von-steve-jobs-vor-den-absolventen-der-stanford-universitat-2005
- Last words:
  - https://www.youtube.com/watch?v=J3dXK4UvAaE

Hinweis: Online-Ressourcen

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

  `21-Bestellung-Listen/Bestellung0.java`

  `..`

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

  `21-Warengruppe-Mengen/Warengruppe0.java`

  `..`

  `21-Warengruppe-Mengen/Warengruppe3.java`
- Beispiel “Kataloge mit Maps”

  `21-Katalog-Mit-Abbildung/Katalog.java`

  `21-Katalog-Mit-Abbildung/Katalog2.java`

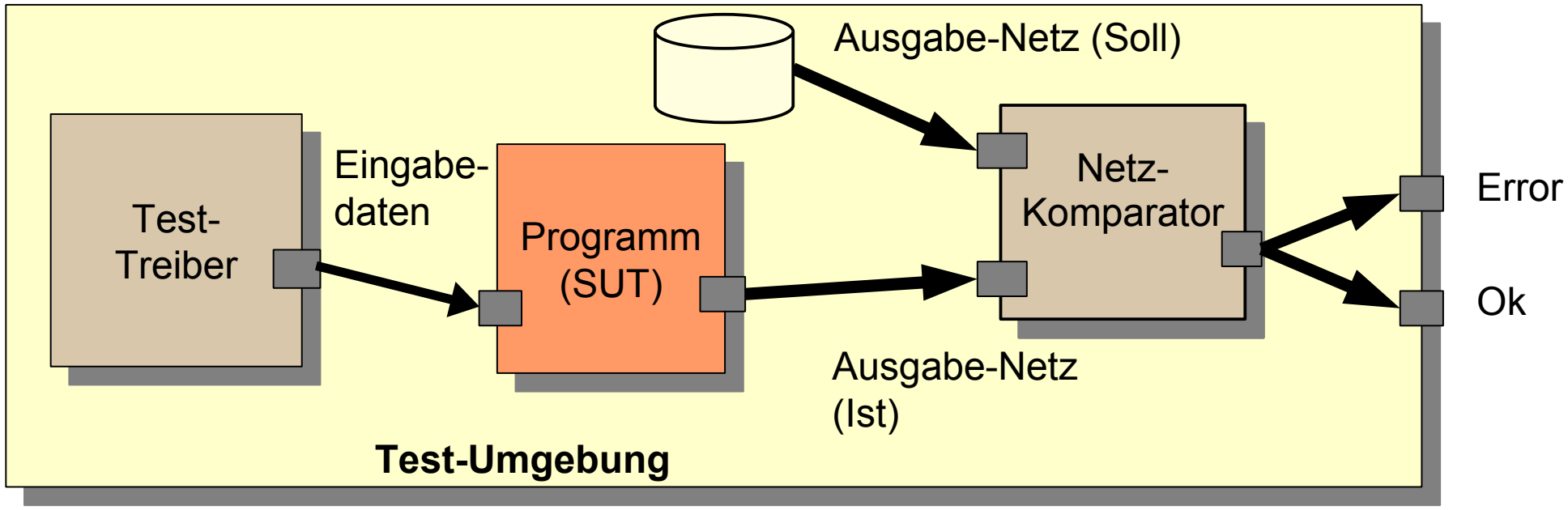
21.1 Verfeinern von Assoziationen

- Die bekannteste Art, Objektnetze zu realisieren, besteht darin, sie in Collection-Datentypen (Nachbarlisten, Nachbarmengen) zu überführen.
- Das hat aber auch seine Tücken und ist schwer zu testen.

Wdh.: Objektorientierte Software hat eine test-getriebene Architektur für Objektnetze

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

> Solange ein Programm keine Tests hat, ist es keine Software



Warum ist Objektnetz-Test wichtig?

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

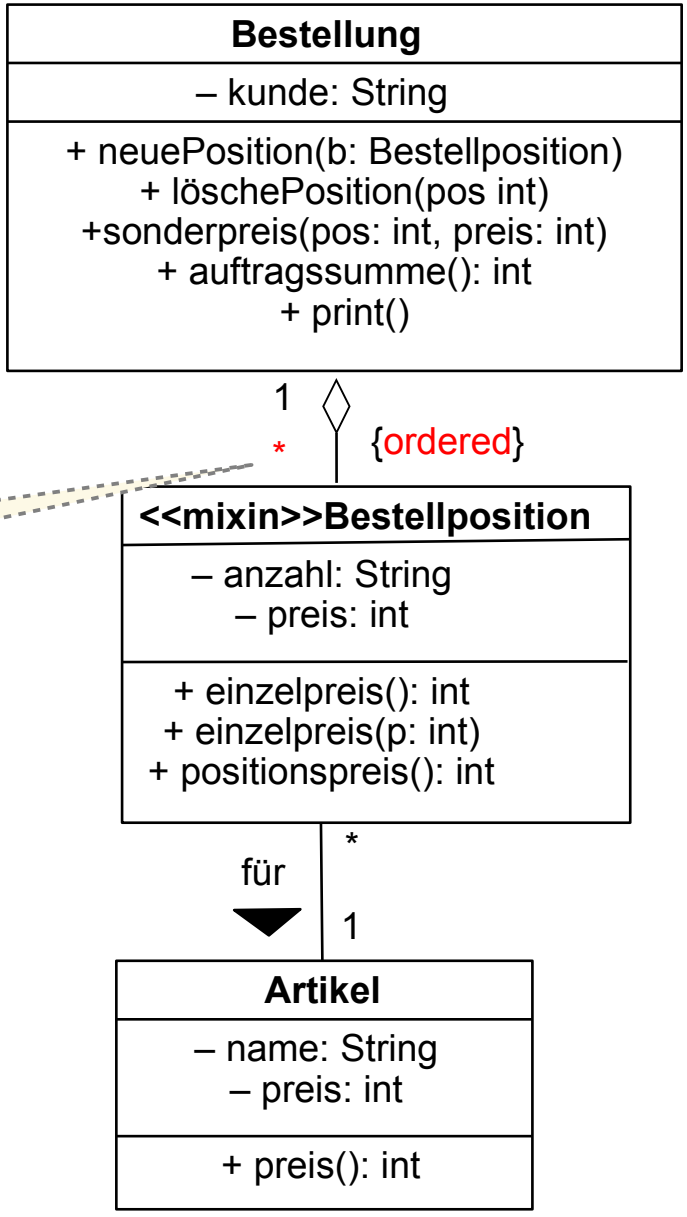
- Bitte mal nach “strange null pointer exception” suchen:
- https://forums.oracle.com/forums/thread.jspa?threadID=2056540
- http://stackoverflow.com/questions/8089798/strange-java-string-array-null-pointer-exception

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

Komplexe Objekte

- Eine `Bestellung` ist ein komplexes Objekt mit vielen `Bestellpositionen` (Mixins) von `Artikeln` (Nachbarn).
- Daher können wir es nicht in einem physischen Objekt im Speicher repräsentieren.
-

"*" führt zu *dynamischen Datenstrukturen*, d.h. Behälterklassen

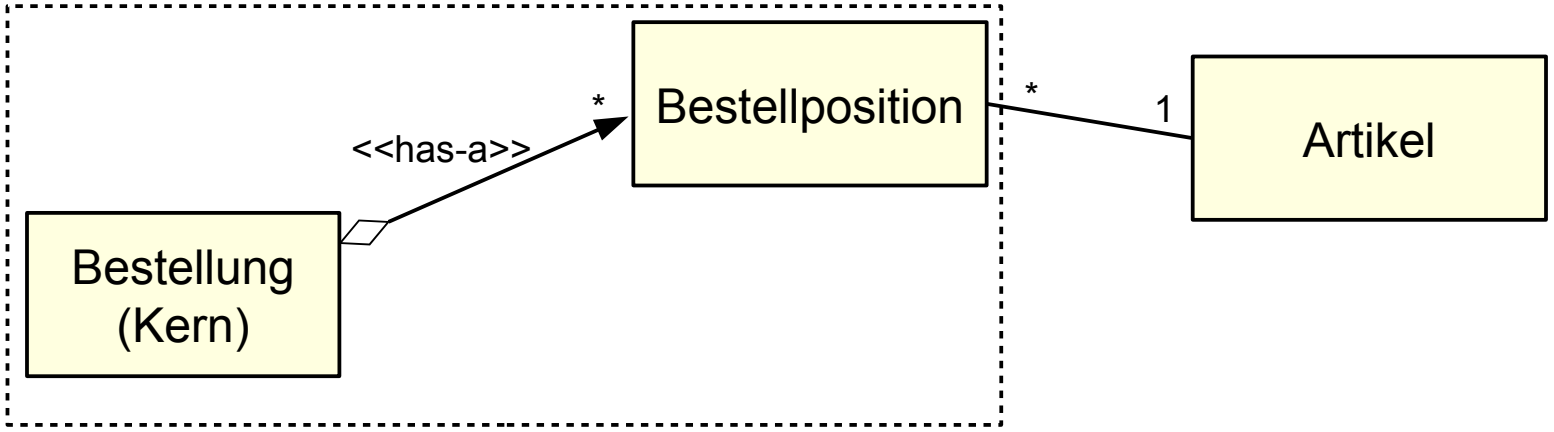


Komplexe Objekte

Ein **komplexes Objekt (Subjekt, big object)** ist ein Objekt,
das in einem Programm und im Speicher wegen seiner Komplexität durch
ein Kernobjekt und mehrere *Unterobjekte (Teilobjekte, Mixin, Satellit)* dargestellt wird.

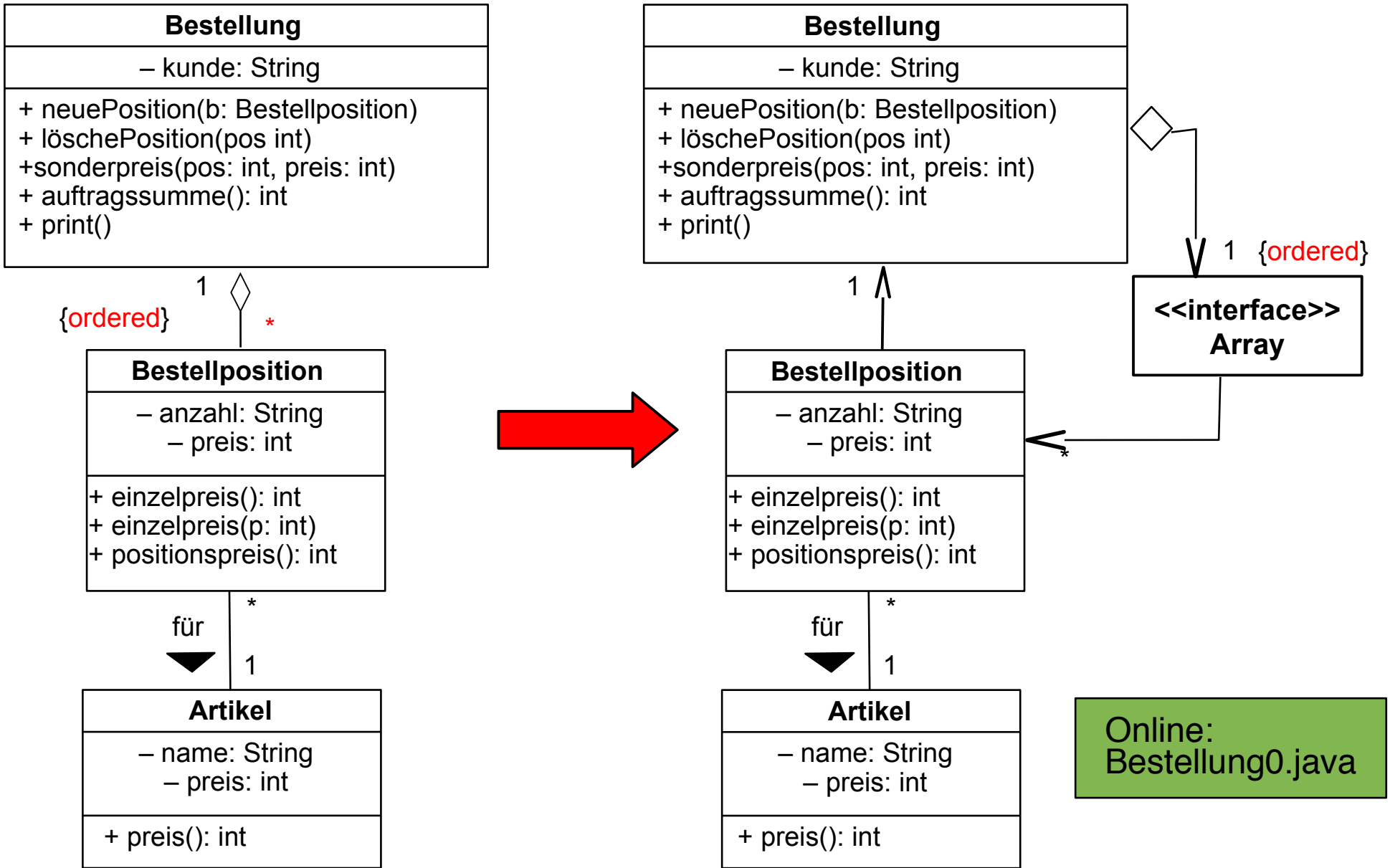
Seine innere Struktur besteht aus einem meist hierarchischen Objektnetz,
dem **Endo-Netz**.

- Ein **Unterobjekt** ist ein **Kernobjekt** angelagert und bildet mit ihm ein integriertes **komplexes Objekt**
  - Das Unterobjekt hat also keine eigene Identität, sondern teilt seine Identität mit dem Kernobjekt (logische Einheit)
  - Es repräsentiert einen Teil des komplexen Objekts
- Das **Endo-Netz** kann beliebig groß werden. Dann werden Behälterklassen benötigt



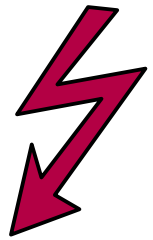
Verfeinern von Assoziationen in komplexen Objekten mit Endo-Netz (Verfeinerung des Endo-Netzes)

- Modell einer Bestellung, eines komplexen Objekts mit einfachem Endonetz (Hierarchie):



Einfache Realisierung des Endo-Netzes mit Arrays – Was ist problematisch?

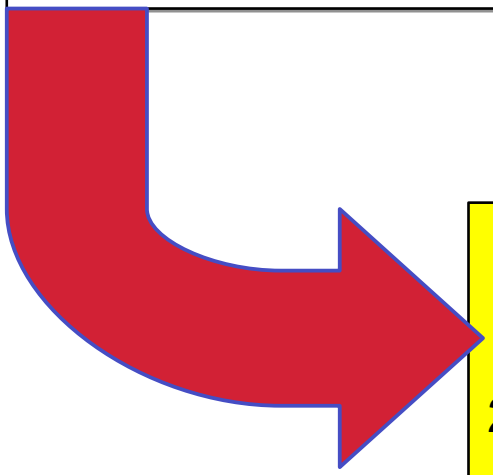

```
class Bestellung {
  private String kunde;
  private Bestellposition[] liste;
  private int anzahl = 0;

  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(); ...}
```

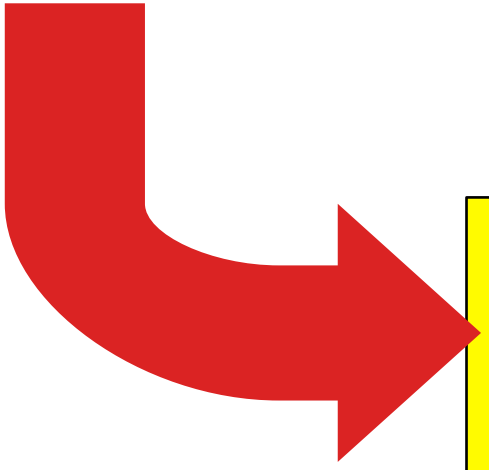


Bestellung fuer Kunde TUD
0. 1 x Tisch Einzelpreis: 200 Summe: 200
1. 4 x Stuhl Einzelpreis: 100 Summe: 400
2. 2 x Schrank Einzelpreis: 300 Summe: 600
Auftragssumme: 1200

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

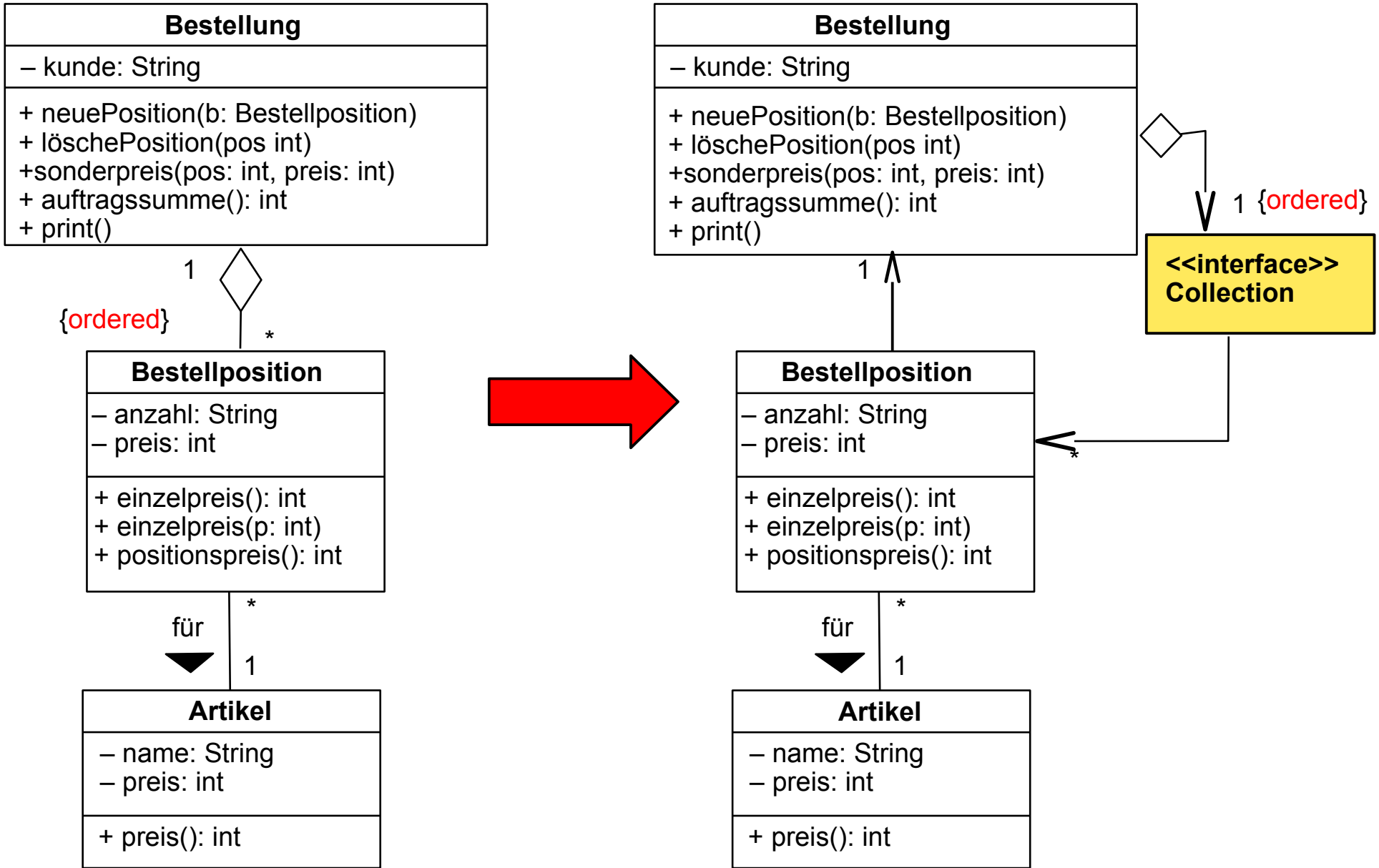
Collections (Behälterklassen)

- Probleme werden durch das Java-Collection-Framework (JCF) 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 **flexibel** 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



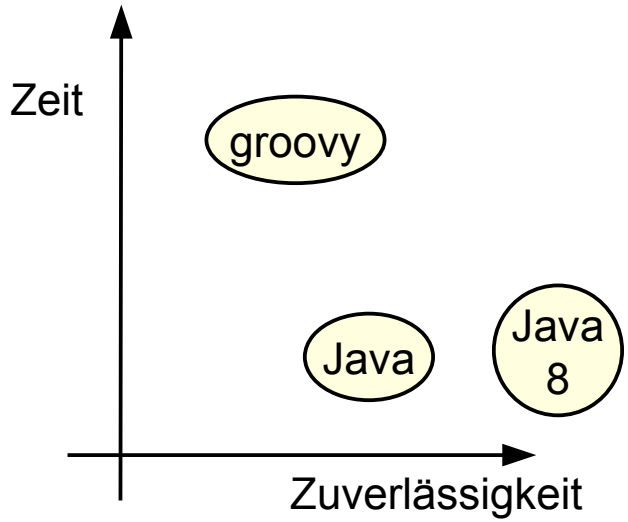
Einfache Realisierung mit Collection-Klasse “List”


```
class Bestellung {
  private String kunde;
  private List<Bestellposition> liste;
  private int anzahl = 0;

  public Bestellung(String kunde) {
    this.kunde = kunde;
    liste = new LinkedList<Bestellposition>(); // Erklärung später
  }
  public void neuePosition (Bestellposition b) {
     liste.set(anzahl,b);
    anzahl++;    // was passiert jetzt bei mehr als 20 Positionen ?
  }
  public void loeschePosition (int pos) {
       liste.remove(pos);
  }
  public void sonderpreis (int pos, int preis) {
    liste.get(pos).einzelpreis(preis);
  }
  public int auftragssumme() {
    int s = 0;
    for(int i=0; i<anzahl; i++) s += liste.get(i).positionspreis();
    return s;
  }
}
```

Online:
Bestellung1LinkedList.java

Orthogonale Trends in der Softwareentwicklung

- **Rapid Application Development (RAD)**
  - *Schneller viel* Code schreiben
  - Hilfsmittel:
    - Typisierung weglassen
    - Ev. *dynamische Typisierung*, damit Fehler zur Laufzeit identifiziert werden können
    - Mächtige Operationen einer Skriptsprache

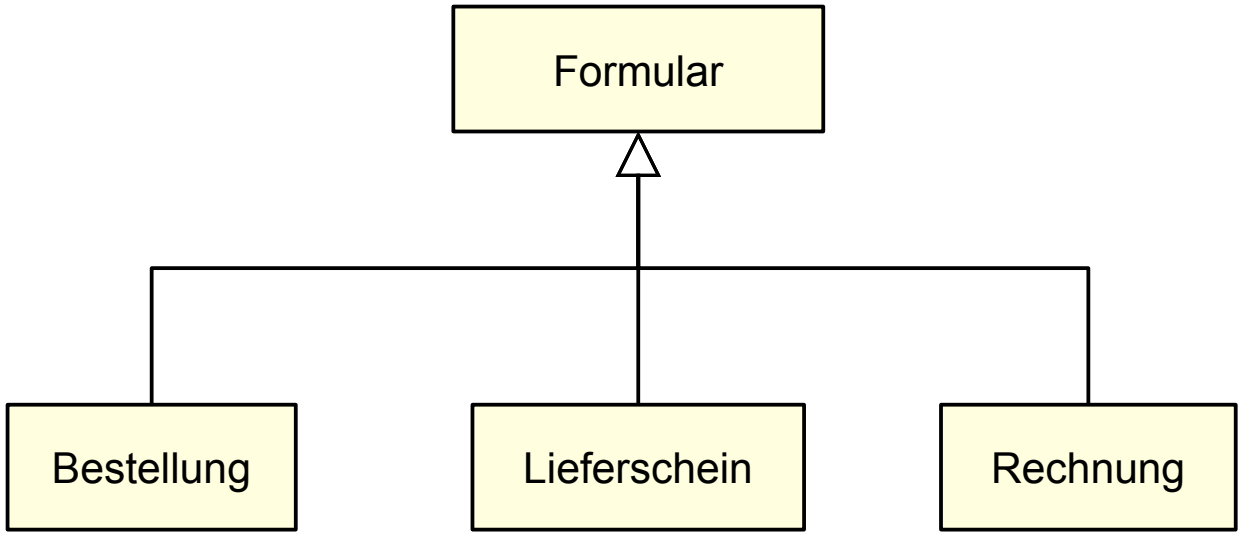


- **Safe Application Development (SAD)**
  - *Guten*, *stabilen*, *wartbaren* Code schreiben
  - Mehr *Entwurfswissen* aus dem Entwurf in die Implementierung übertragen
  - Hilfsmittel:
    - *Statische Typisierung*, damit der Übersetzer viele Fehler entdeckt
    - Generische Klassen
  - Aus der Definition einer Datenstruktur können Bedingungen für ihre Anwendung abgeleitet werden
  - *Generische Collections* für typsicheres Aufbauen von Objektnetzen (Java)

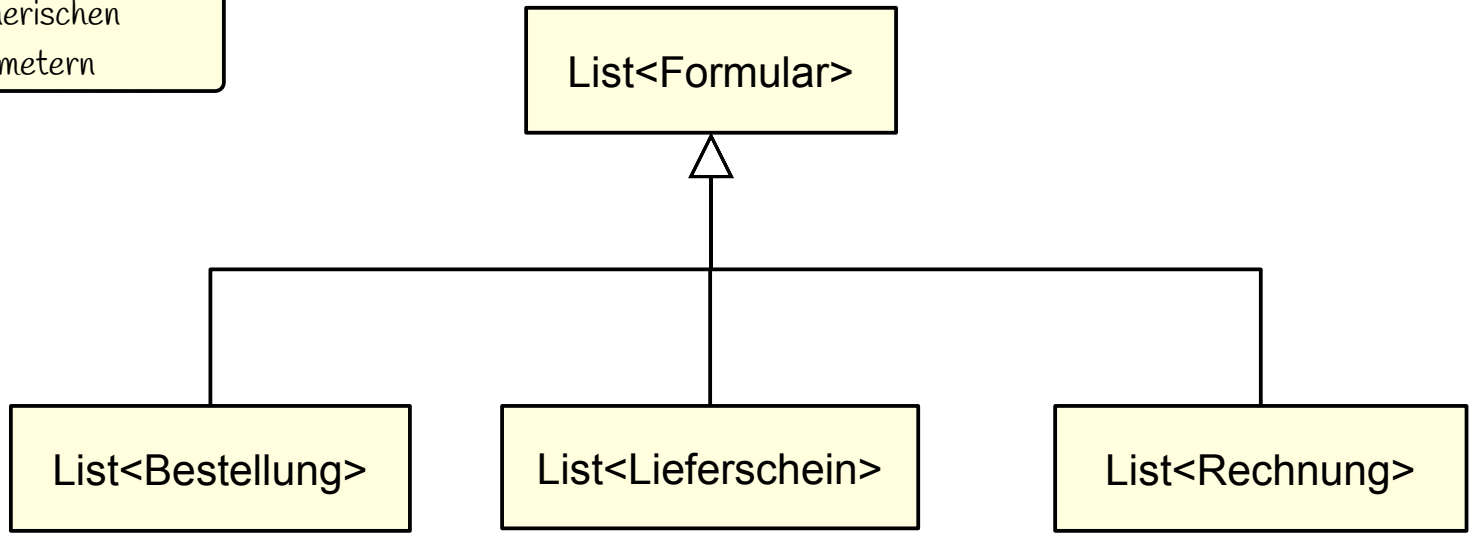
21.2 Die Collection-Bibliothek Java Collection Framework (JCF) (Behälterklassen)

- Ungetypte Behälterklassen für RAD
- Generische Behälterklassen für SAD

Bsp.: Klassen einer Hierarchie und ihre Behälter-Hierarchie



mit generischen Parametern

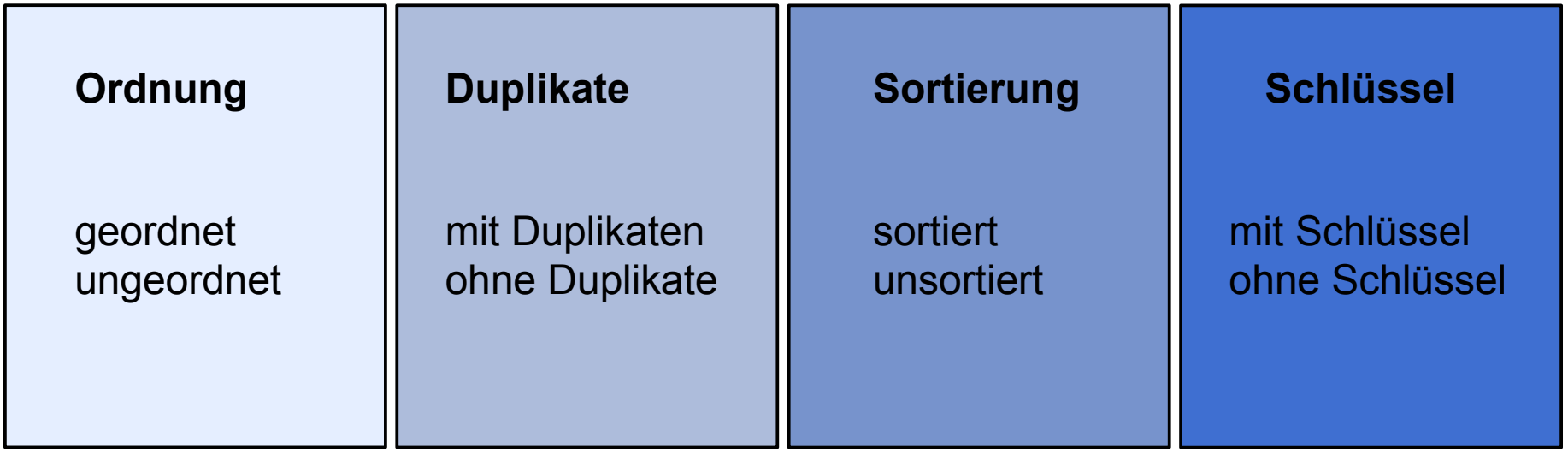


21.2.1 Die einfache Collection-Bibliothek ohne Element-Typen

- Behälterklassen: `Collection<Object>`

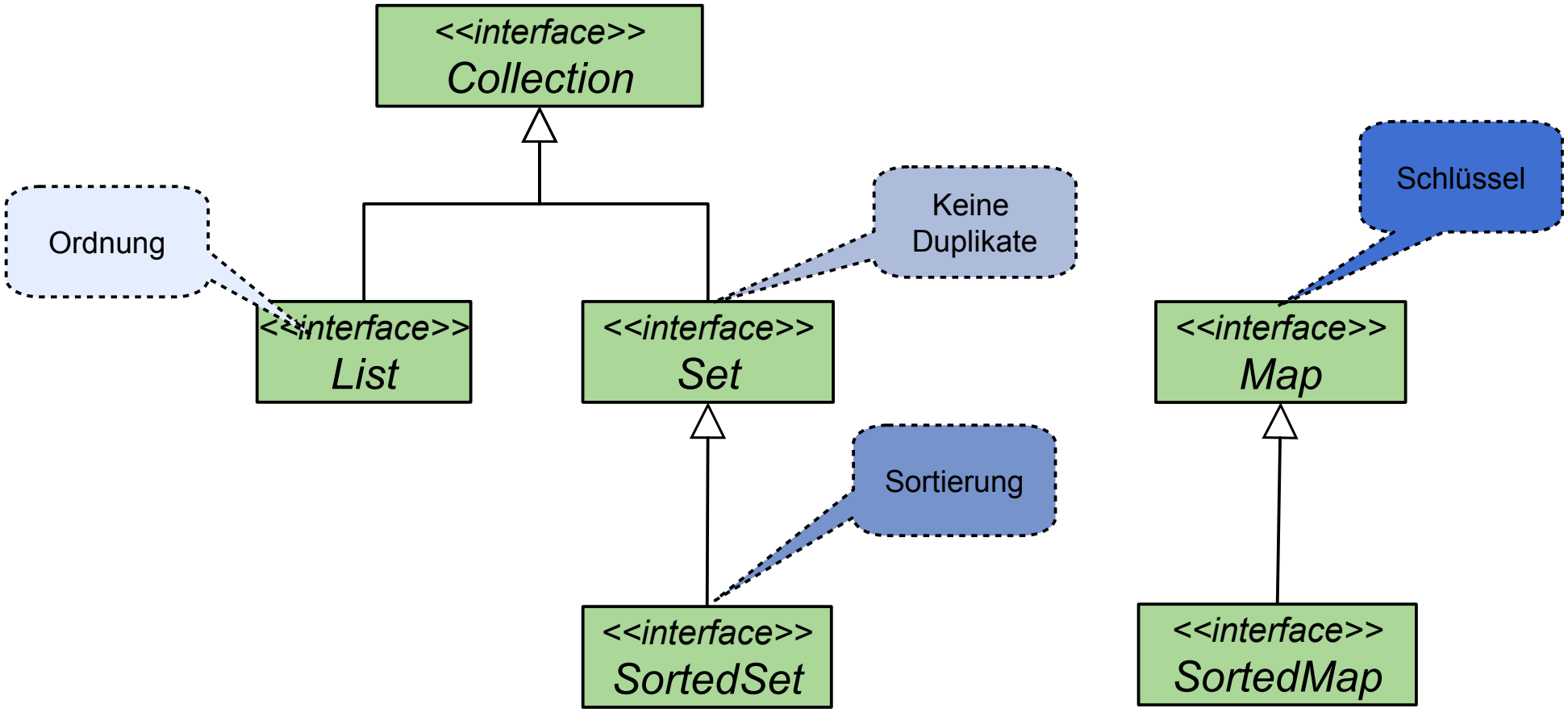
Facetten von Behälterklassen (Collections)

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



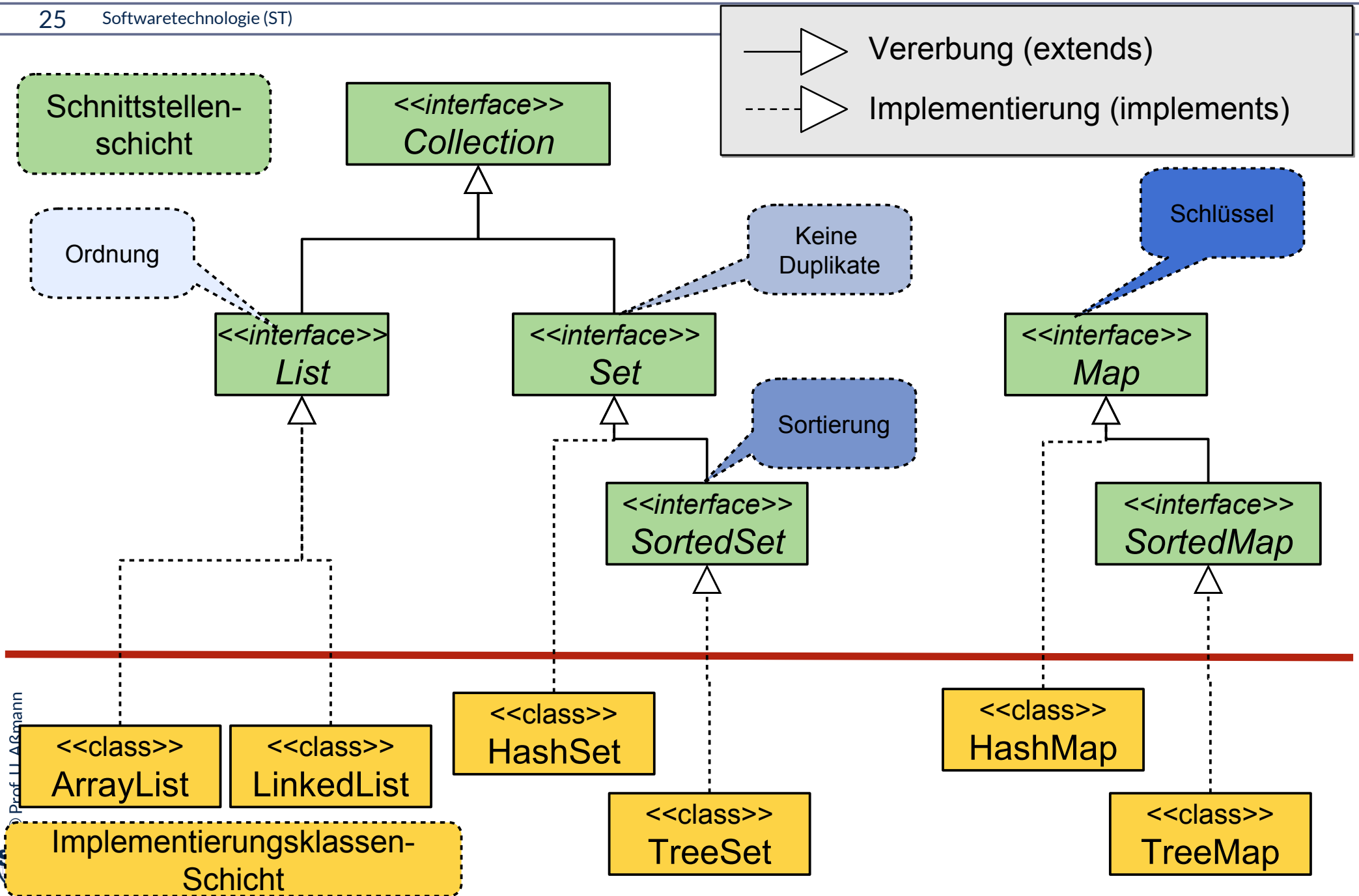
Java Collection Framework: Prinzipielle Struktur

- Die Schnittstellen-Hierarchie der Collections (*vor* Java 1.5)
- Eine Collection kann beliebige Objekte enthalten (Element == `Object`)



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

JCF: Schnittstellen-Schicht vs Implementierungsschicht



Problem 1 ungetypter Schnittstellen in Behälterklassen: 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!!

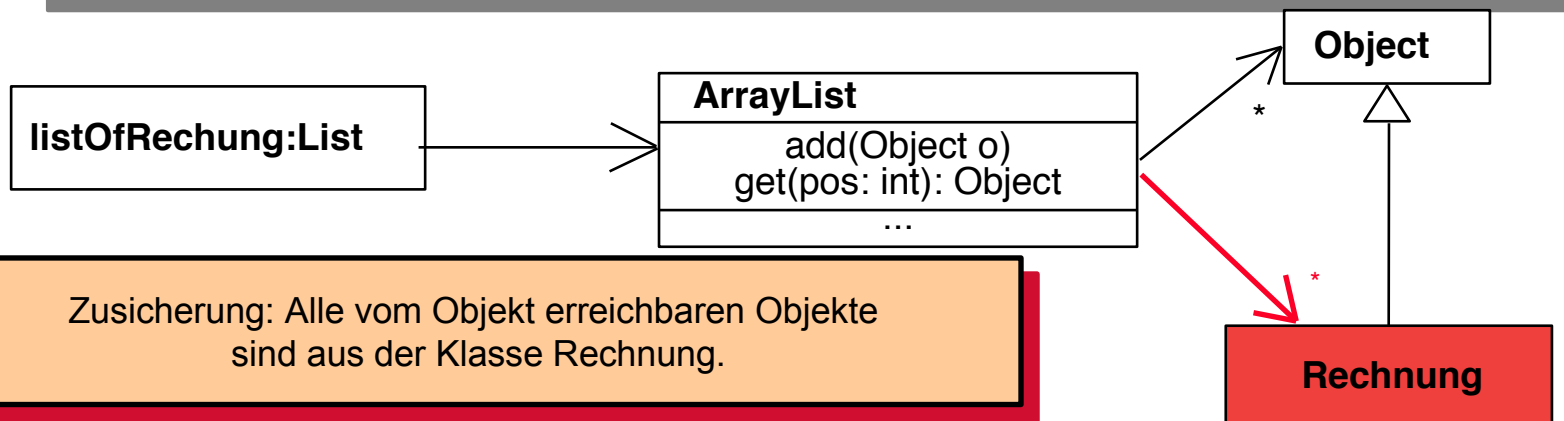
Problem 2 ungetypter Schnittstellen in Behälterklassen: Unnötige Casts

- 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



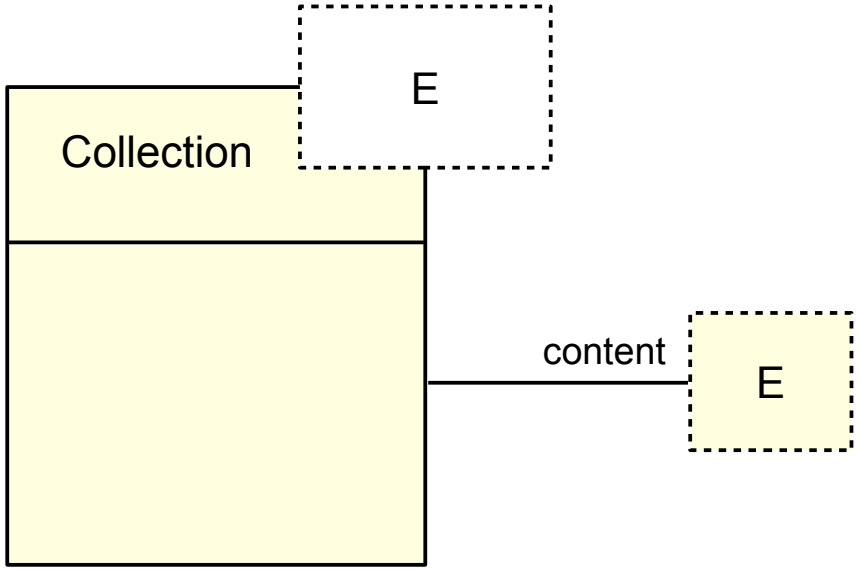
21.2.2 Die Collection-Bibliothek mit Element-Typen

- Generische Behälterklassen erlauben SAD für Objektnetze:
- `Collection<E>` ist die oberste Schnittstelle

Generische Behälterklassen

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

- In UML



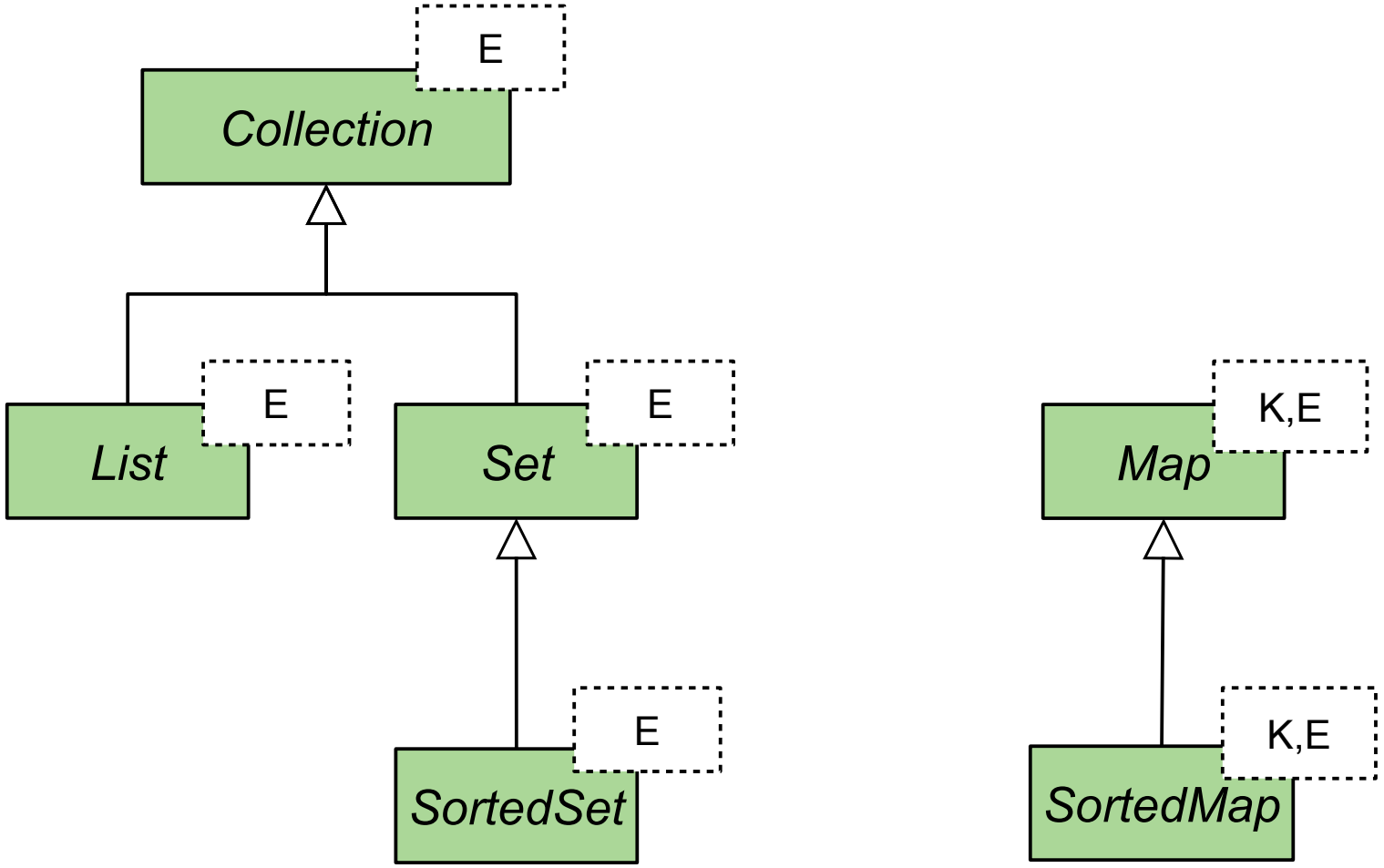
Collection<E>

- In Java
  - Sprachregelung: “Collection of E”

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

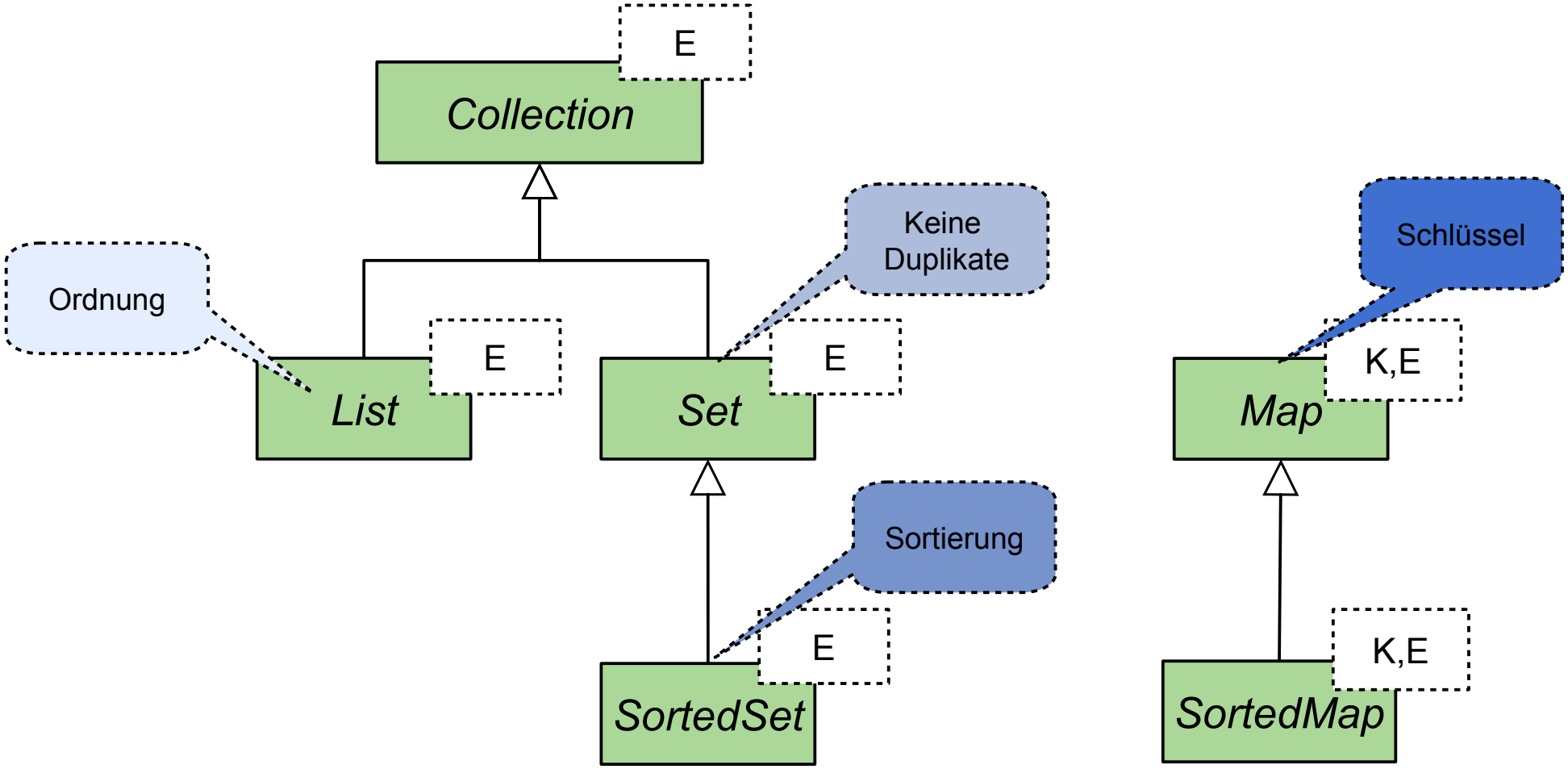
Collection-Hierarchie mit generischen Schnittstellen

- Die generische Schnittstellen-Hierarchie der Collections (seit Java 5)
- Eine Collection enthält nur Elemente vom Typ E, bzw. Schlüssel vom Typ K
  - E: Element, K: Key



Collection-Hierarchie mit generischen Schnittstellen

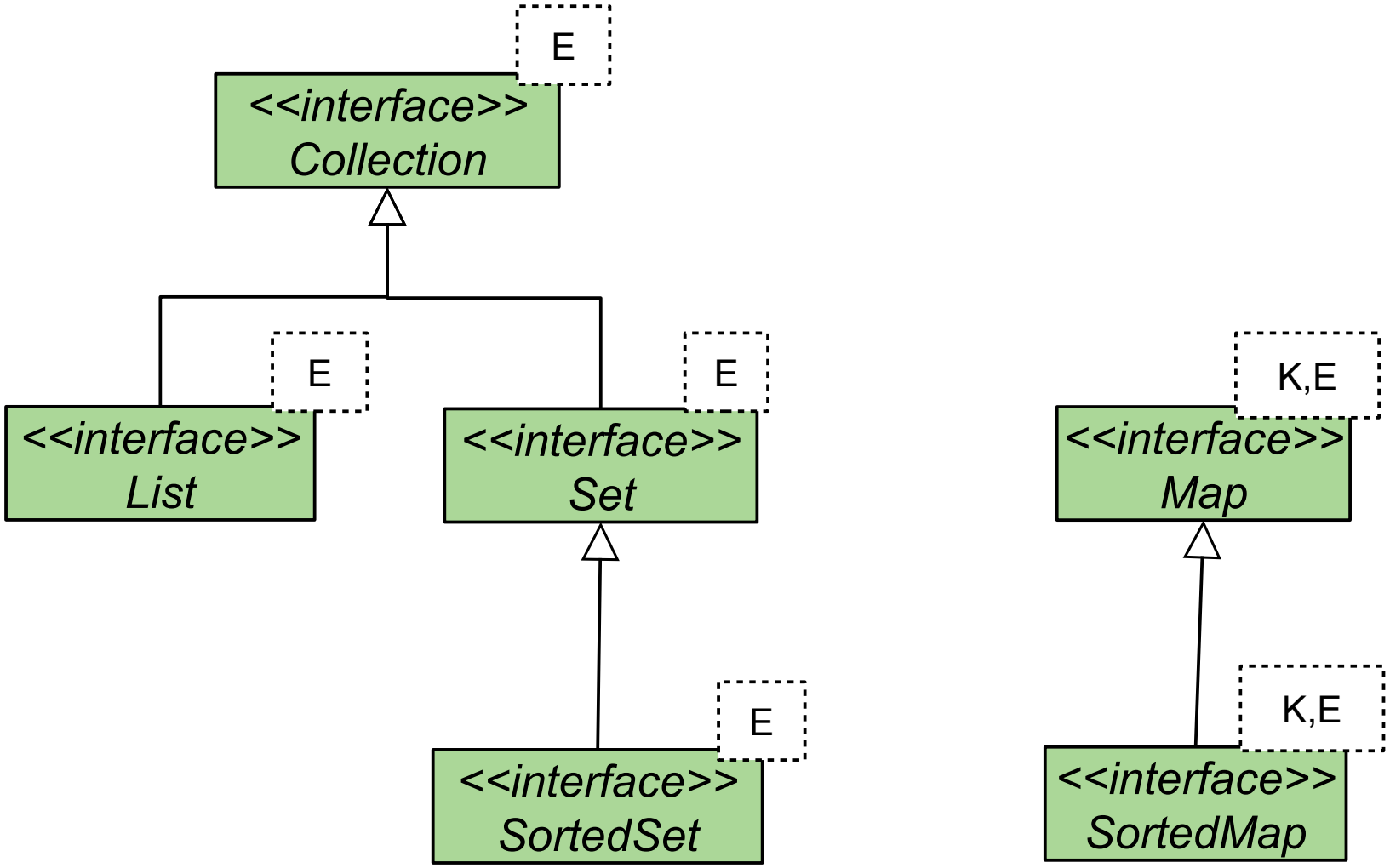
- Die generische Schnittstellen-Hierarchie der Collections (seit Java 5)
- Eine Collection enthält nur Elemente vom Typ E, bzw. Schlüssel vom Typ K
  - E: Element, K: Key



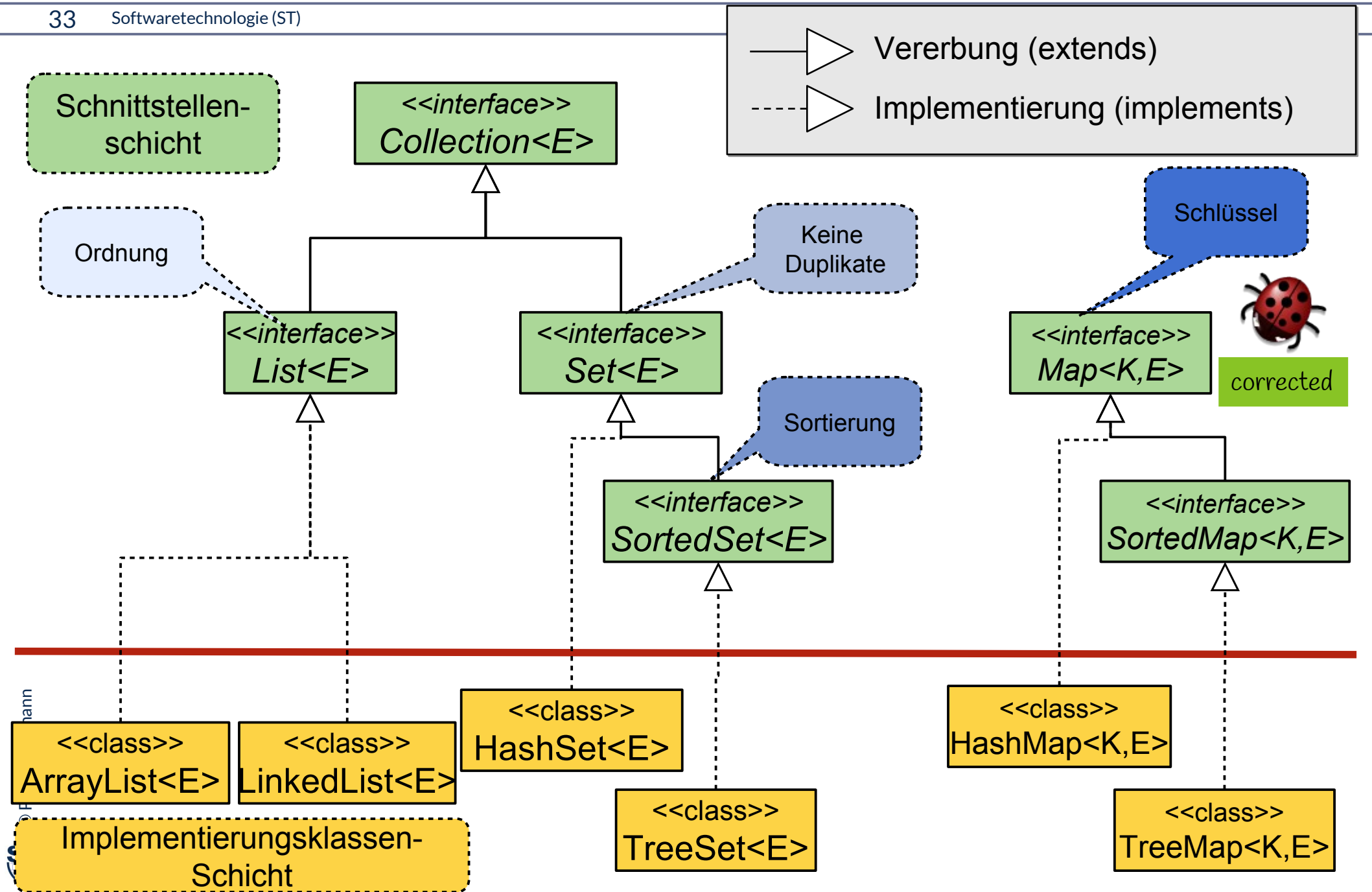
/home/ua1/Courses/ST1/Material/jdk-docs/index.html

Collection-Hierarchie mit generischen Schnittstellen

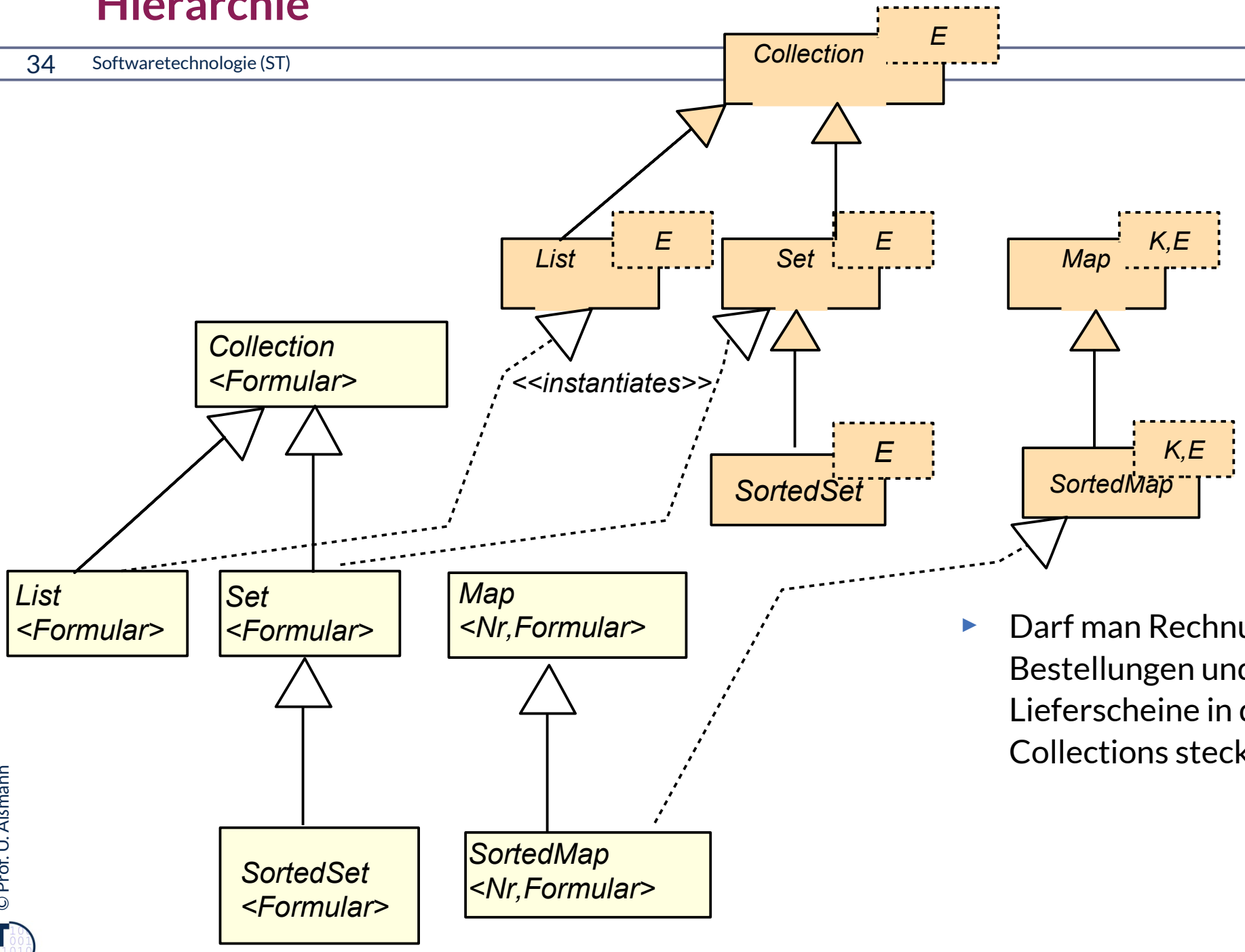
- Die generische Schnittstellen-Hierarchie der Collections (seit Java 5)
  - E: Element, K: Key



JCF: Schnittstellen Schicht vs Implementierungsschicht



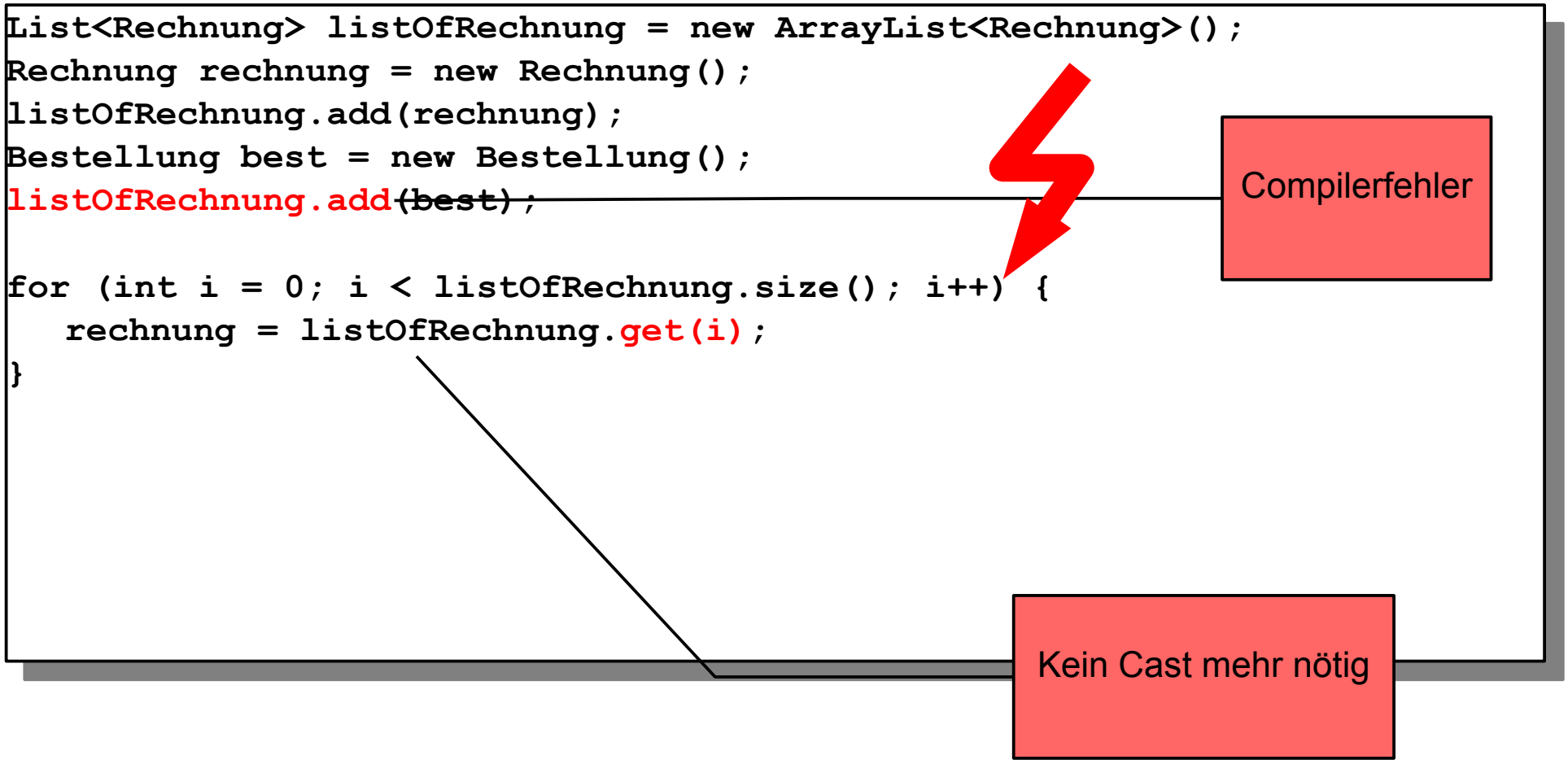
Beispiel: Instanziierung der generischen JCF Schnittstellen-Hierarchie



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

SAD löst Probleme

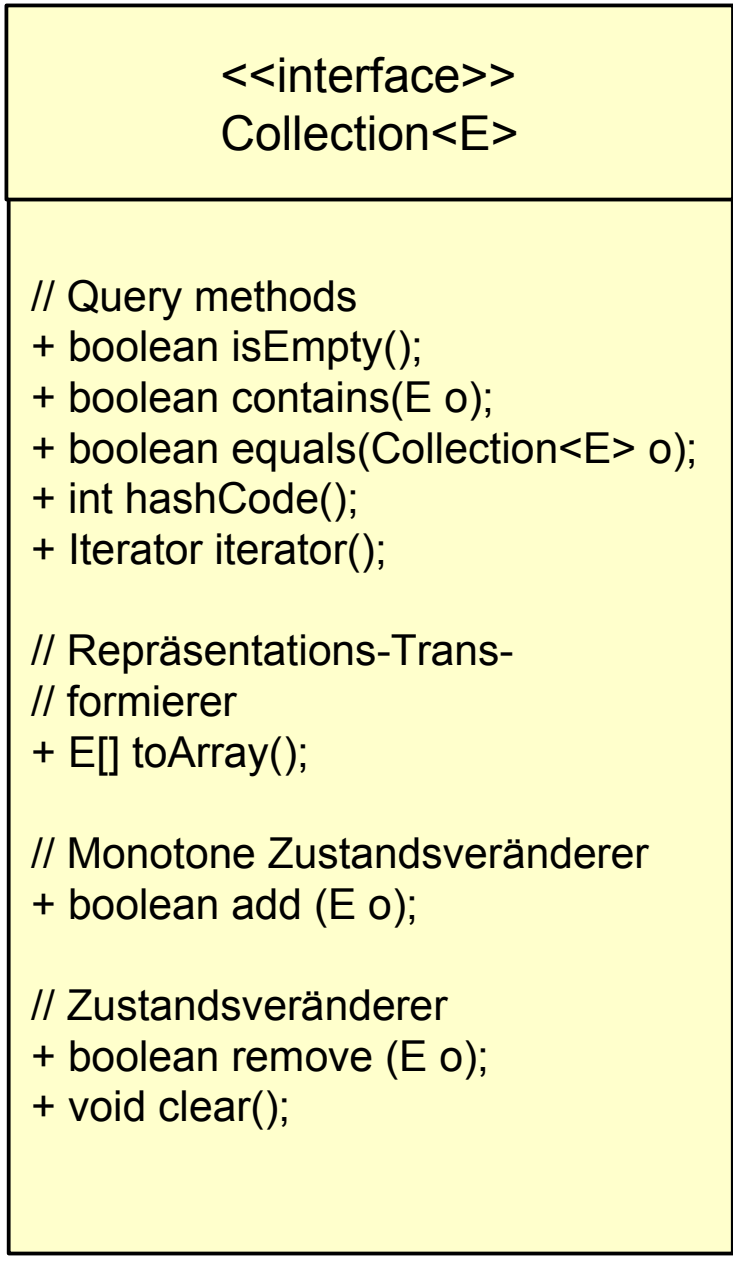
- Bei der Konstruktion von Collections werden jetzt Rechnungen von Bestellungen unterschieden
- Casts sind nicht nötig, der Übersetzer kennt den feineren Typ
- Das ist Safe Application Development (SAD)



21.2.2 Schnittstellen im Detail

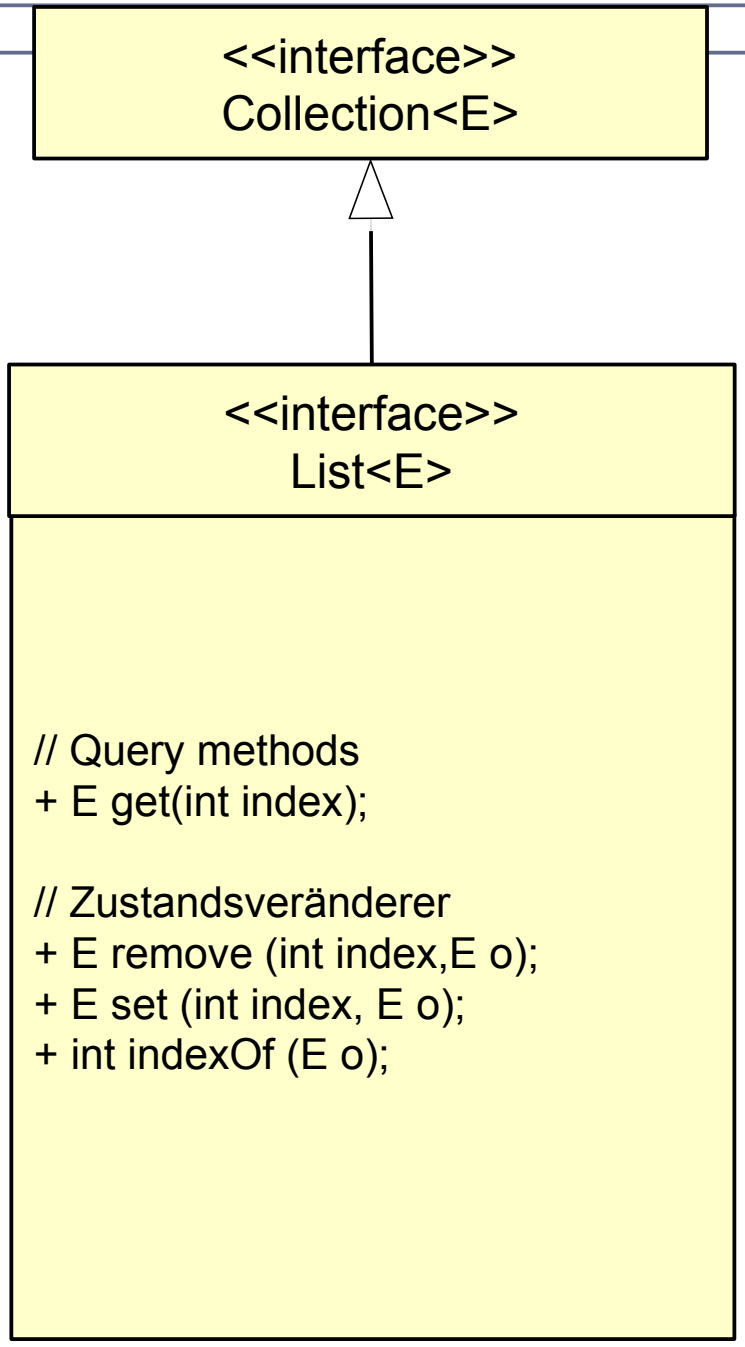
Schnittstelle java.util.Collection (Auszug)


```
public interface Collection<E> {
  // Anfragen (Queries)
  public boolean isEmpty();
  public boolean contains (E o);
  public boolean equals(Collection<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();
  …
}
```



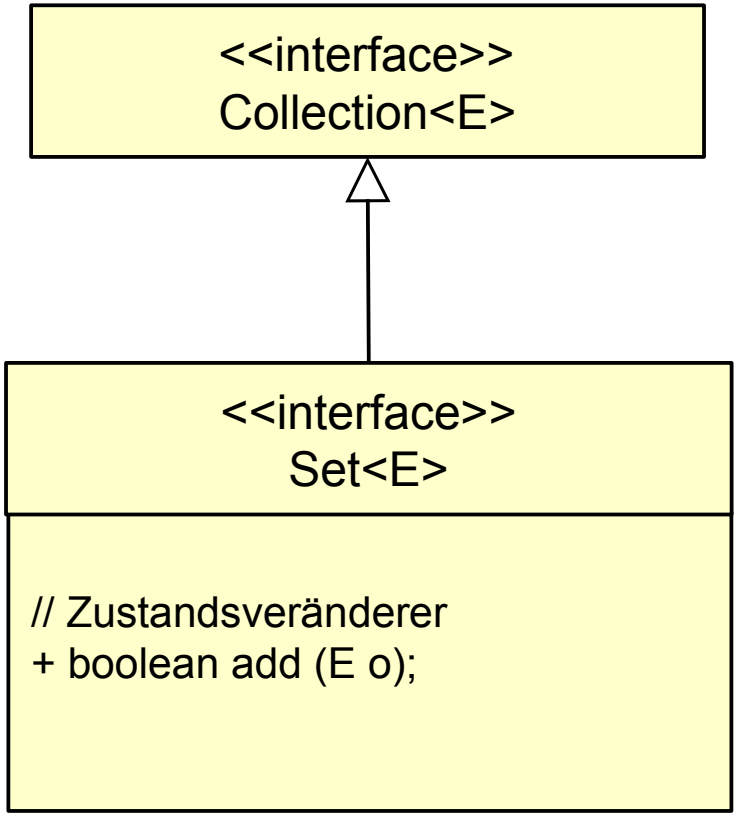
Unterschnittstelle java.util.List (Auszug)


```
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);
  ...
}
```



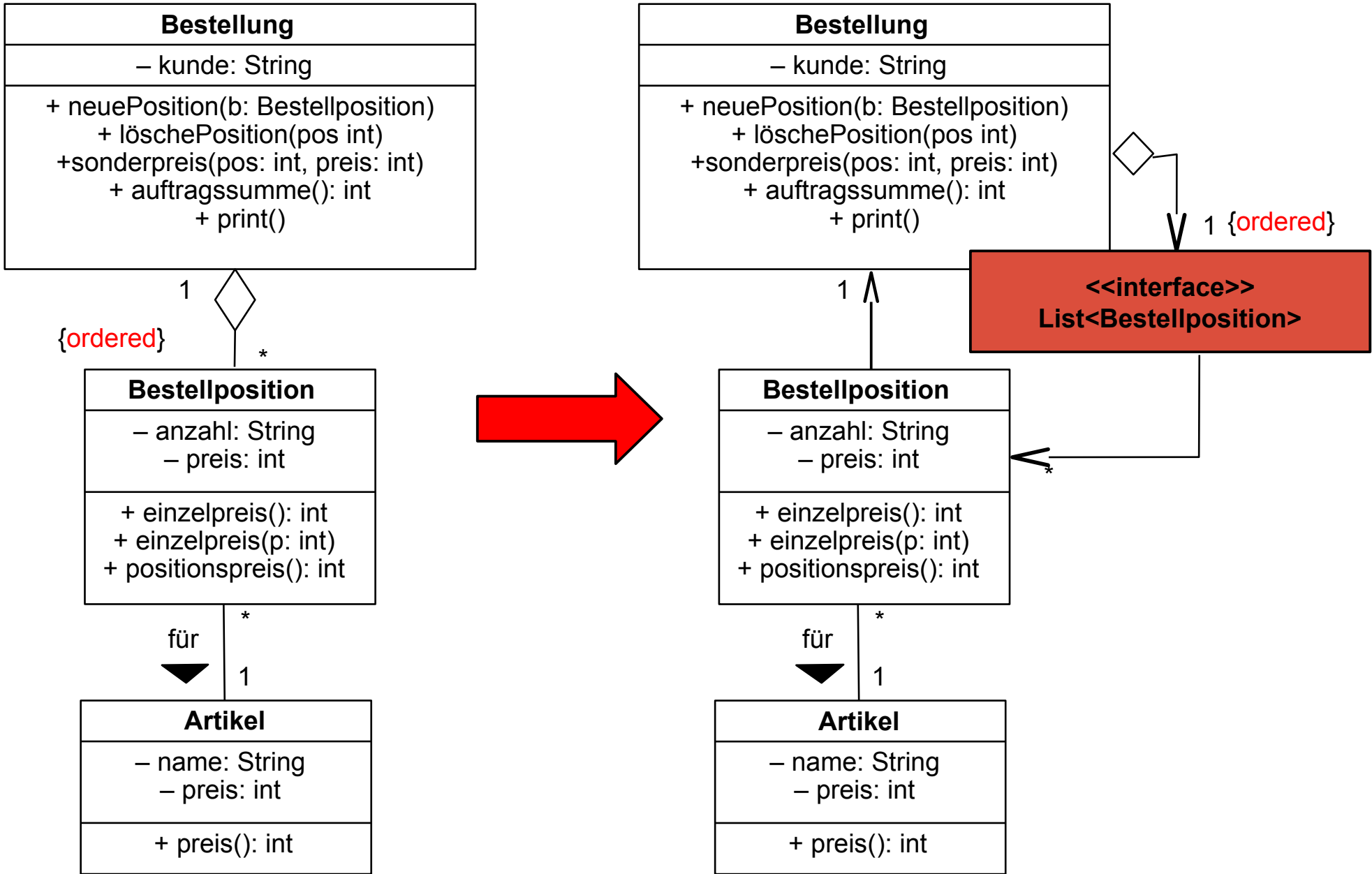
Unterschnittstelle java.util.Set (Auszug)

- 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



Wdh. Einfache Realisierung mit Schnittstellen-Klasse “List” und Implementierungsklasse LinkedList

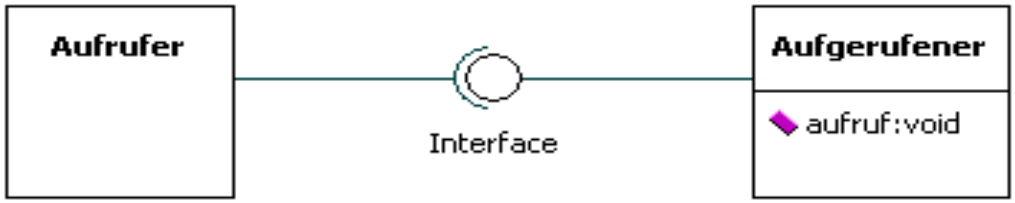

```
class Bestellung {
  private String kunde;
  private List<Bestellposition> liste;
  private int anzahl = 0;

  public Bestellung(String kunde) {
    this.kunde = kunde;
    liste = new LinkedList<Bestellposition>();
    // Konkrete Implementierungsklasse
  }
  public void neuePosition (Bestellposition b) {
     liste.set(anzahl,b);
    anzahl++;    // was passiert jetzt bei mehr als 20 Positionen ?
  }
  public void loeschePosition (int pos) {
       liste.remove(pos);
  }
  public void sonderpreis (int pos, int preis) {
    liste.get(pos).einzelpreis(preis);
  }
  public int auftragssumme() {
    int s = 0;
    for(int i=0; i<anzahl; i++) s += liste.get(i).positionspreis();
    return s;
  }
}
```

Beachte Konstruktor!

Online:
Bestellung1LinkedList.java

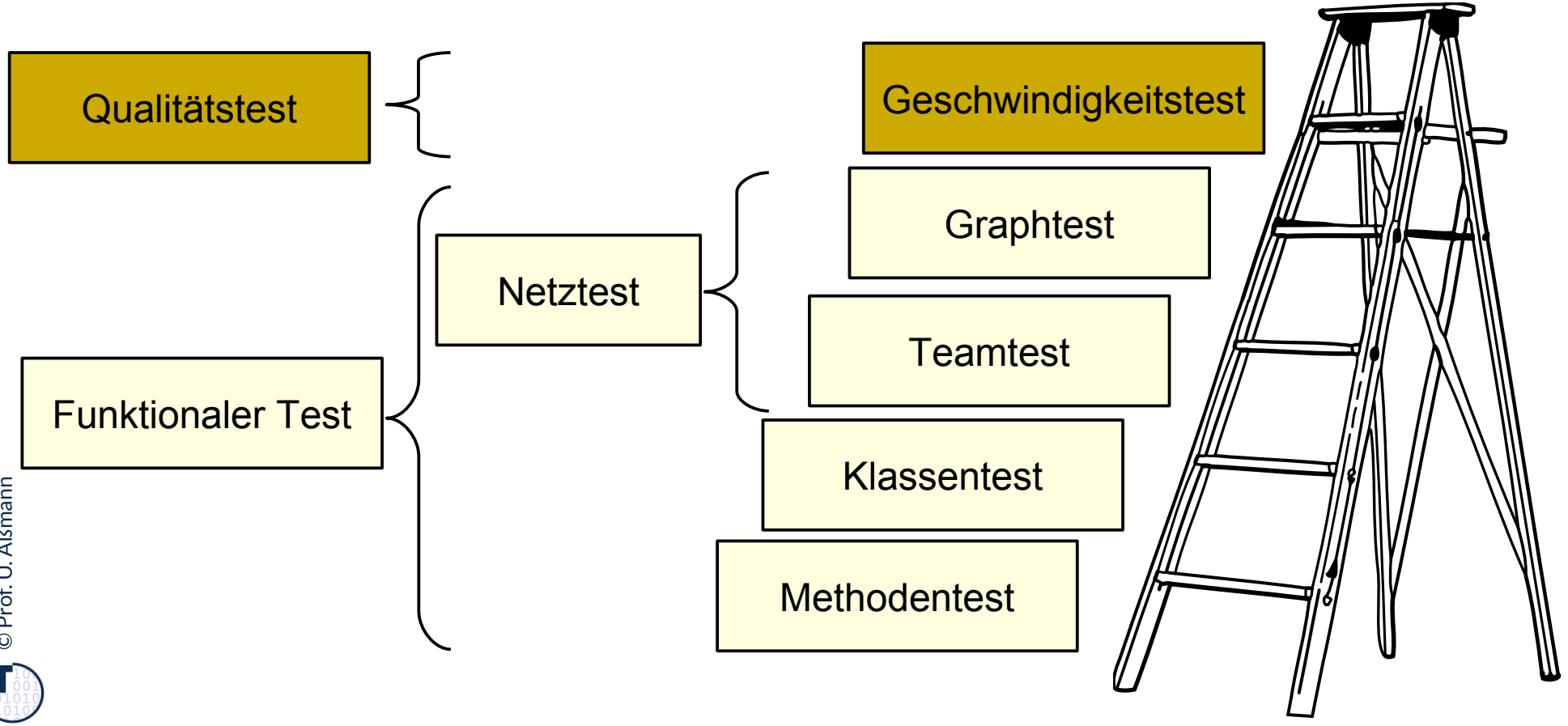
21.3 Programmieren gegen Schnittstellen von polymorphen Containern

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

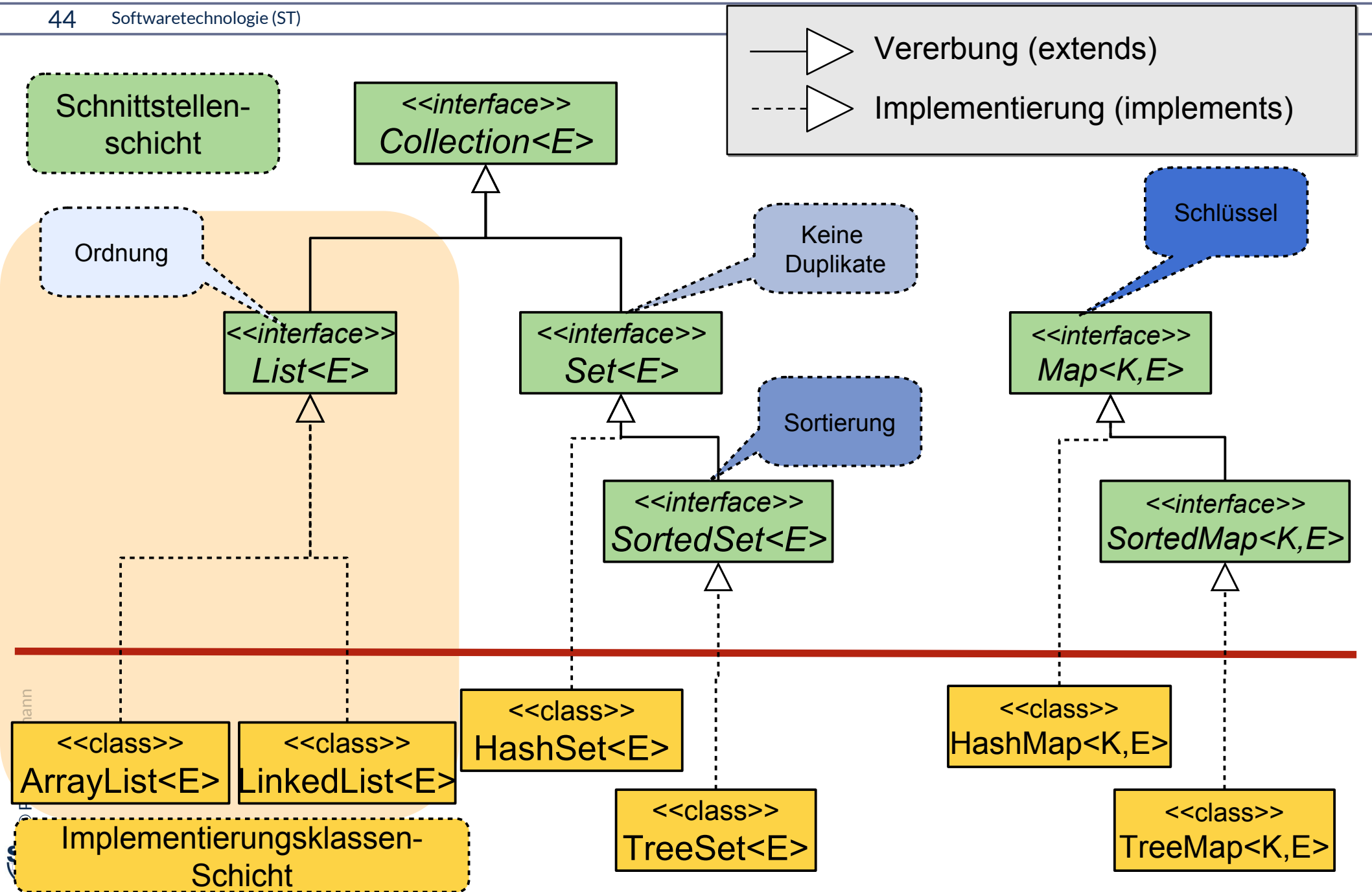


Geschwindigkeitstests (Performance testing)

- Geschwindigkeit ist keine Funktionalität, sondern eine Qualität des Programms
- Die Geschwindigkeit eines Programms hängt wesentlich von der Implementierung des Objektnetzes und der Netzdatenstrukturen ab (Teams, Graphen)
- Geschwindigkeitstests werden mit ***Benchmarks*** durchgeführt
- Schnittstellen erlauben den Austausch von Implementierungen zur Beschleunigung



Schnittstellen und Implementierungen im Collection-Framework bilden generische Behälterklassen

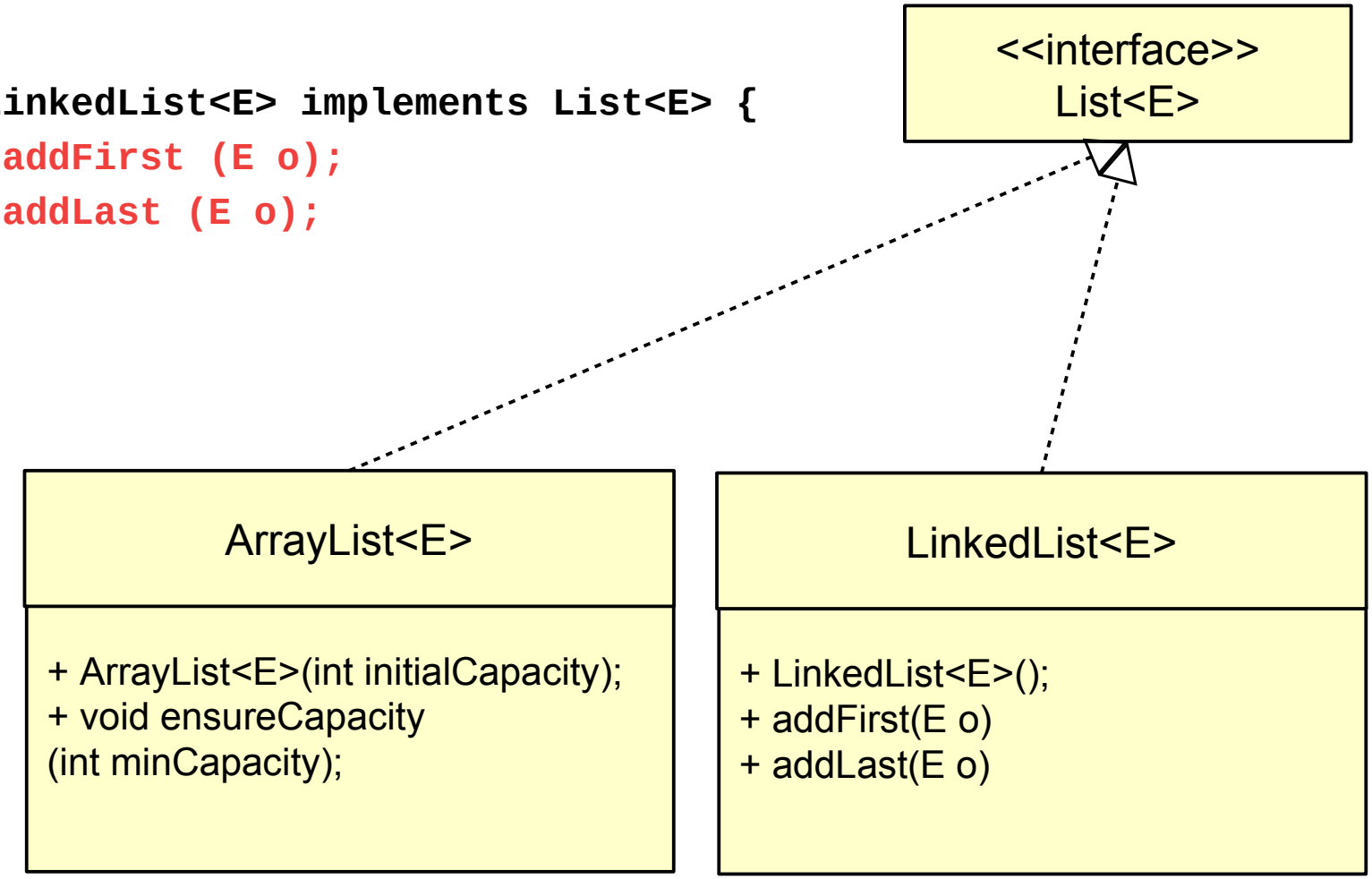


Polymorphie – zwischen abstrakten und konkreten Datentypen

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

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!

Wdh. Einfache Realisierung mit Schnittstellen-Klasse “List” und Implementierungsklasse LinkedList


```
class Bestellung {
  private String kunde;
  private List<Bestellposition> liste;
  private int anzahl = 0;

  public Bestellung(String kunde) {
    this.kunde = kunde;
    liste = new ArrayList<Bestellposition>();
    // Konkrete Implementierungsklasse
  }
  public void neuePosition (Bestellposition b) {
     liste.set(anzahl,b);
    anzahl++;    // was passiert jetzt bei mehr als 20 Positionen ?
  }
  public void loeschePosition (int pos) {
       liste.remove(pos);
  }
  public void sonderpreis (int pos, int preis) {
    liste.get(pos).einzelpreis(preis);
  }
  public int auftragssumme() {
    int s = 0;
    for(int i=0; i<anzahl; i++) s += liste.get(i).positionspreis();
    return s;
  }
}
```

Beachte Konstruktor!

Online:
Bestellung1ArrayList.java

Welche Listen-Implementierung soll man wählen, um das Programm schnell und ressourcenschonend zu machen?

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

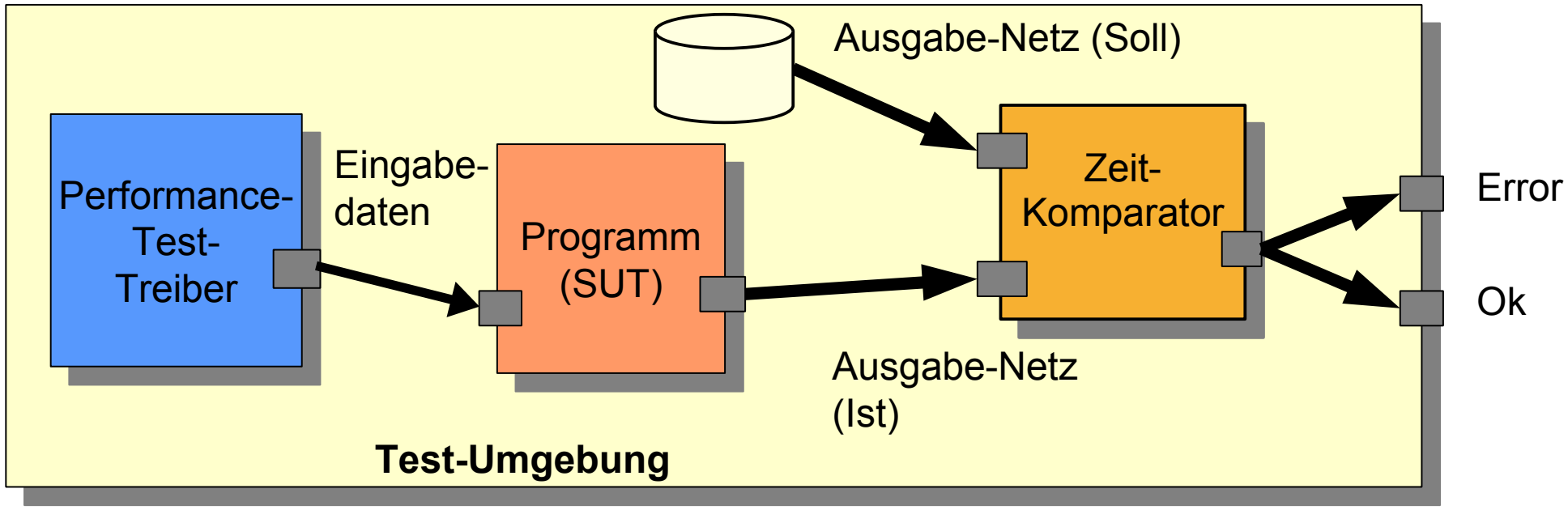
- Gemessener relativer Aufwand für Operationen auf Listen:
  (aus Eckel, Thinking in Java, 2nd ed., 2000)

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

Performance-Tests für polymorphe Objektnetze

- Performance-Test misst die Geschwindigkeit der Implementierung eines Programms
- Die Varianten Implementierungen von Behälterklassen können einfach verglichen werden

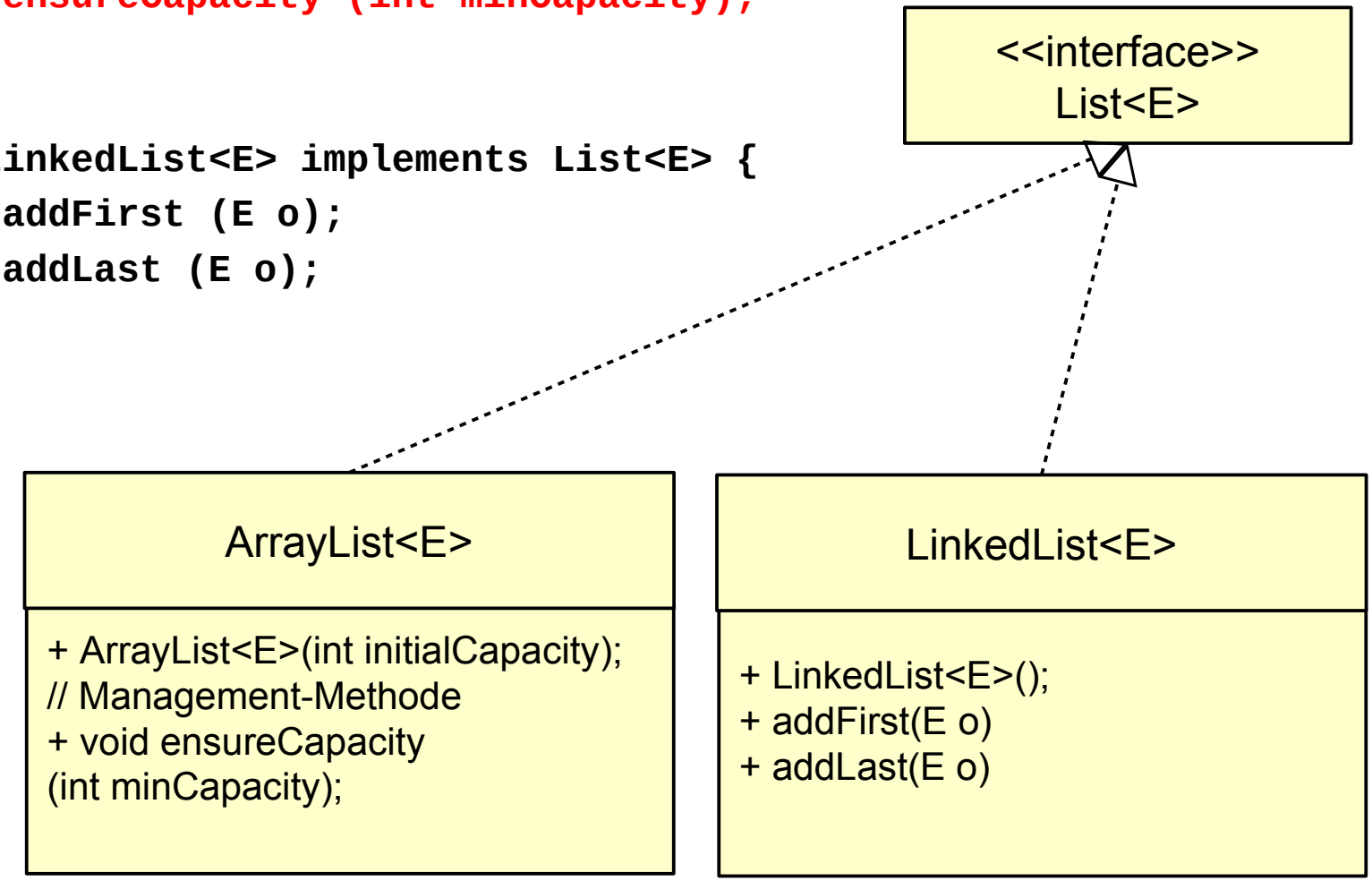
Solange ein Programm keine Performance-Tests hat, ist es keine Software



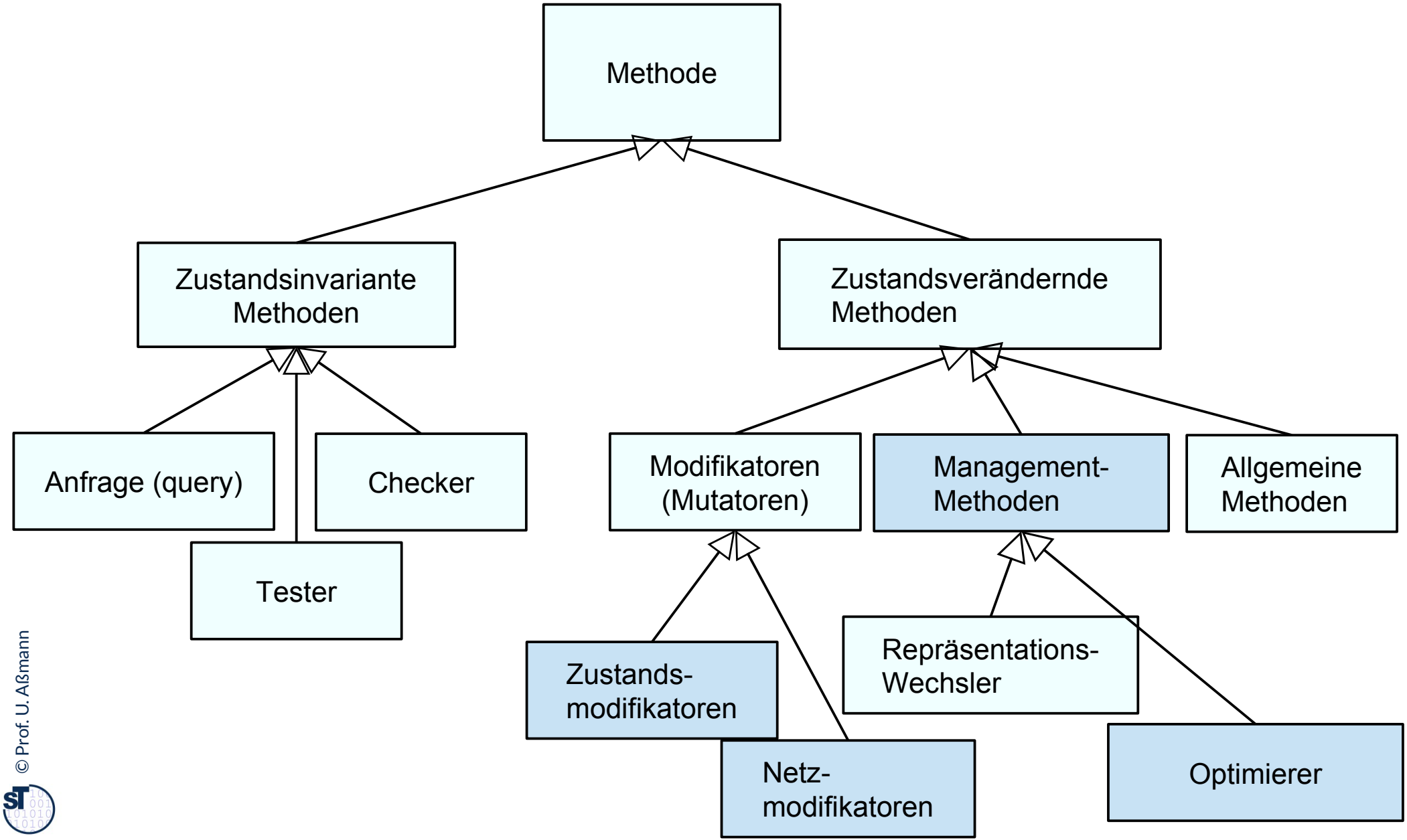
21.4. Weitere Arten von Klassen und Methoden

Management-Methoden in java.util.ArrayList, LinkedList


```
public class ArrayList<E> implements List<E> {
  public ArrayList<E> (int initialCapacity);
  // Management-Methode (Optimierer-Methode)
  public void ensureCapacity (int minCapacity);
  ...
}
public class LinkedList<E> implements List<E> {
  public void addFirst (E o);
  public void addLast (E o);
  ...
}
```



Erweiterung Q3: Taxonomie der Methodenarten



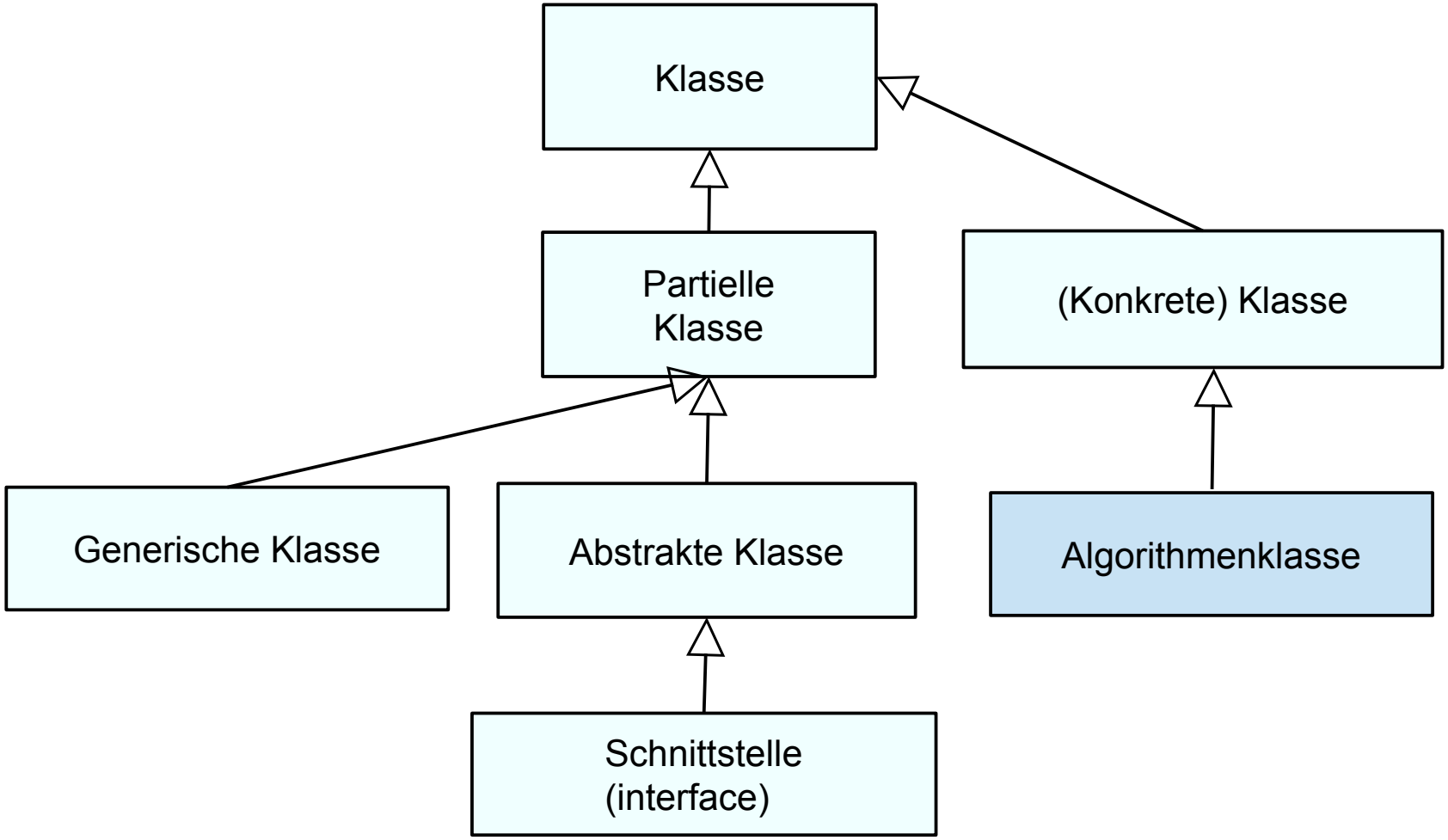
Standardalgorithmen in der *Algorithmenklasse* java.util.Collections

Def.: Algorithmenklassen (Hilfsklassen) enthalten Algorithmen, die auf einer Familie von anderen Klassen arbeiten

```
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

Erweiterung Q2: Begriffshierarchie von Klassen



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);
}
```

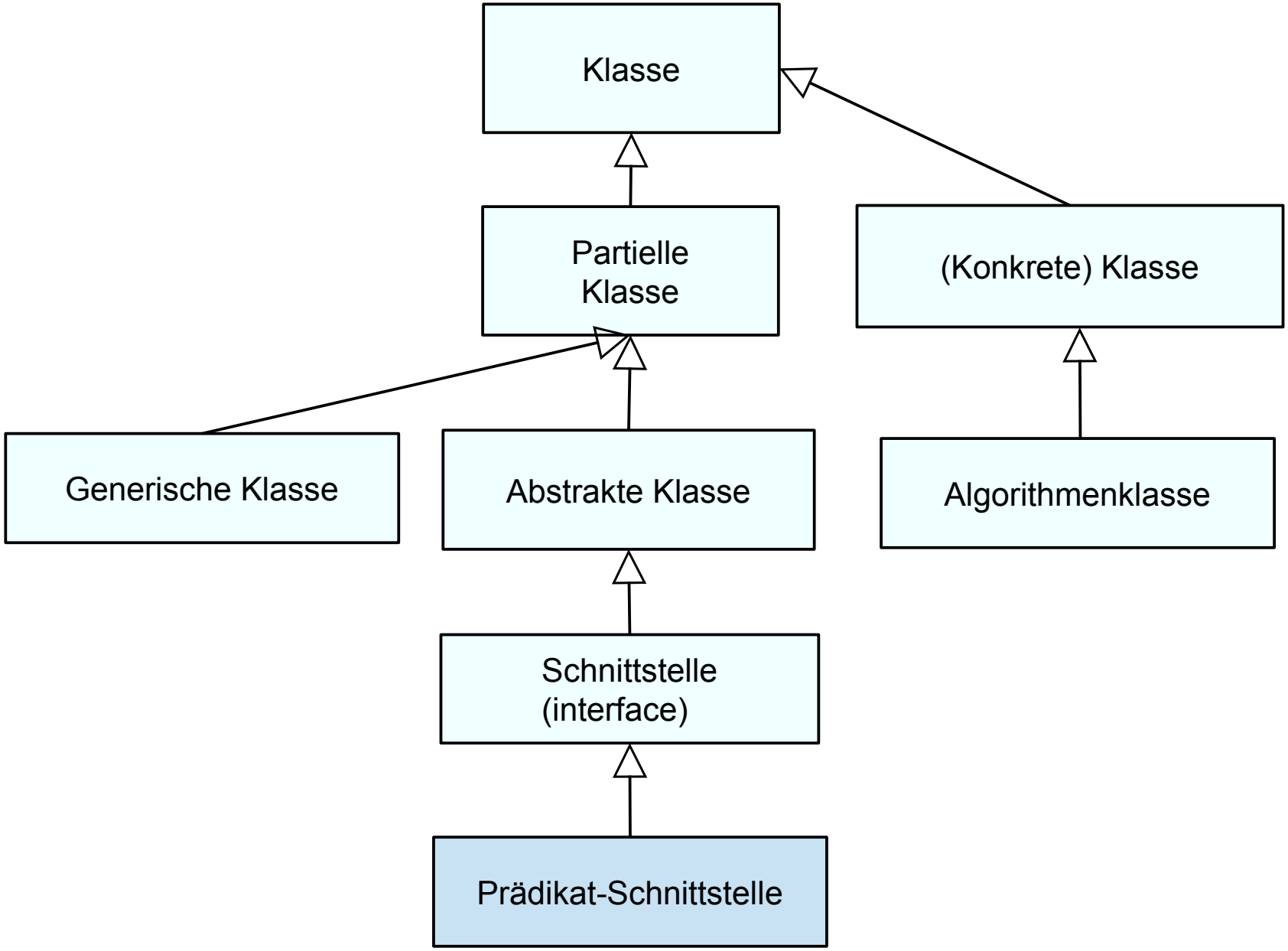
- Resultat ist kleiner/gleich/größer 0:
  genau dann wenn "this" kleiner/gleich/größer als Objekt o

Typschranken generischer Parameter (type bounds), mit Prädikatsschnittstellen

- Prädikatschnittstellen können für einen Typparameter einfache Prädikate ausdrücken
- Beispiel: Comparable<E> als Return-typ in der Collections-Klasse sichert zu, dass die Methode compareTo() existiert

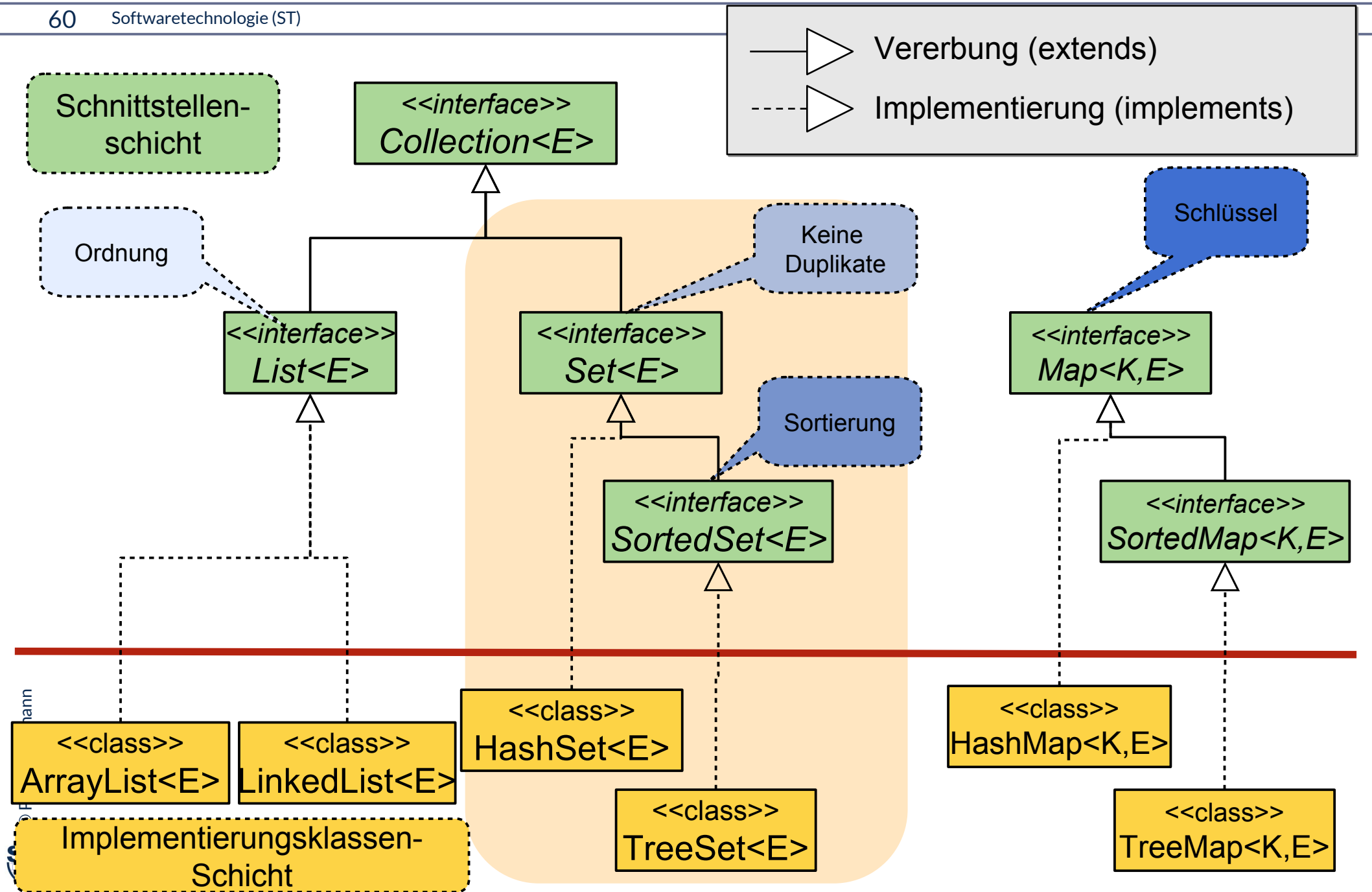
```
class Collections {
/** minimum function for a Collection. Return value is typed
  * with a generic type with a type bound */

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;
}
```

Erweiterung Q2: Begriffshierarchie von Klassen

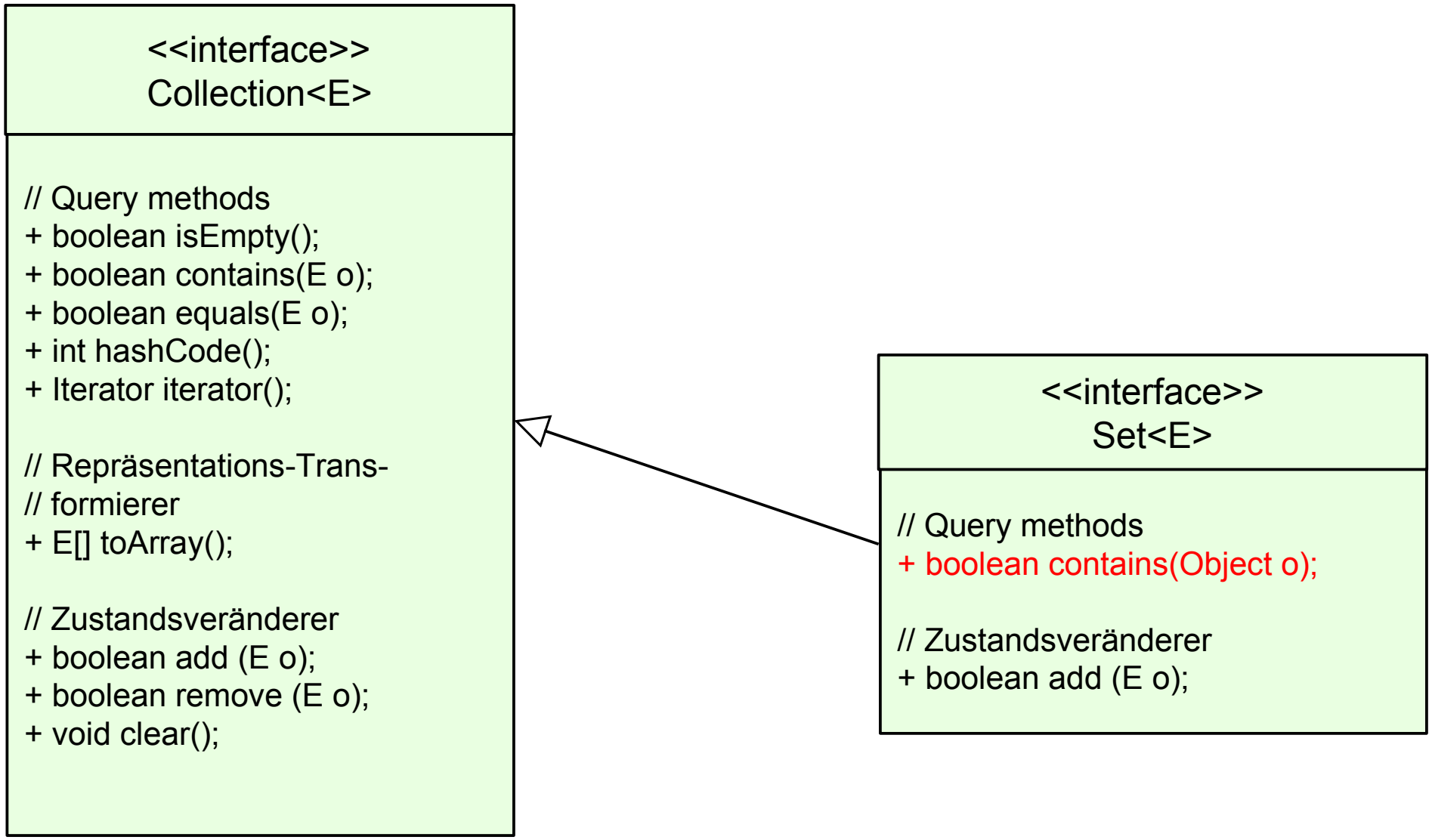


21.5 Ungeordnete Collections mit Set

Schnittstellen und Implementierungen im Collection-Framework bilden generische Behälterklassen

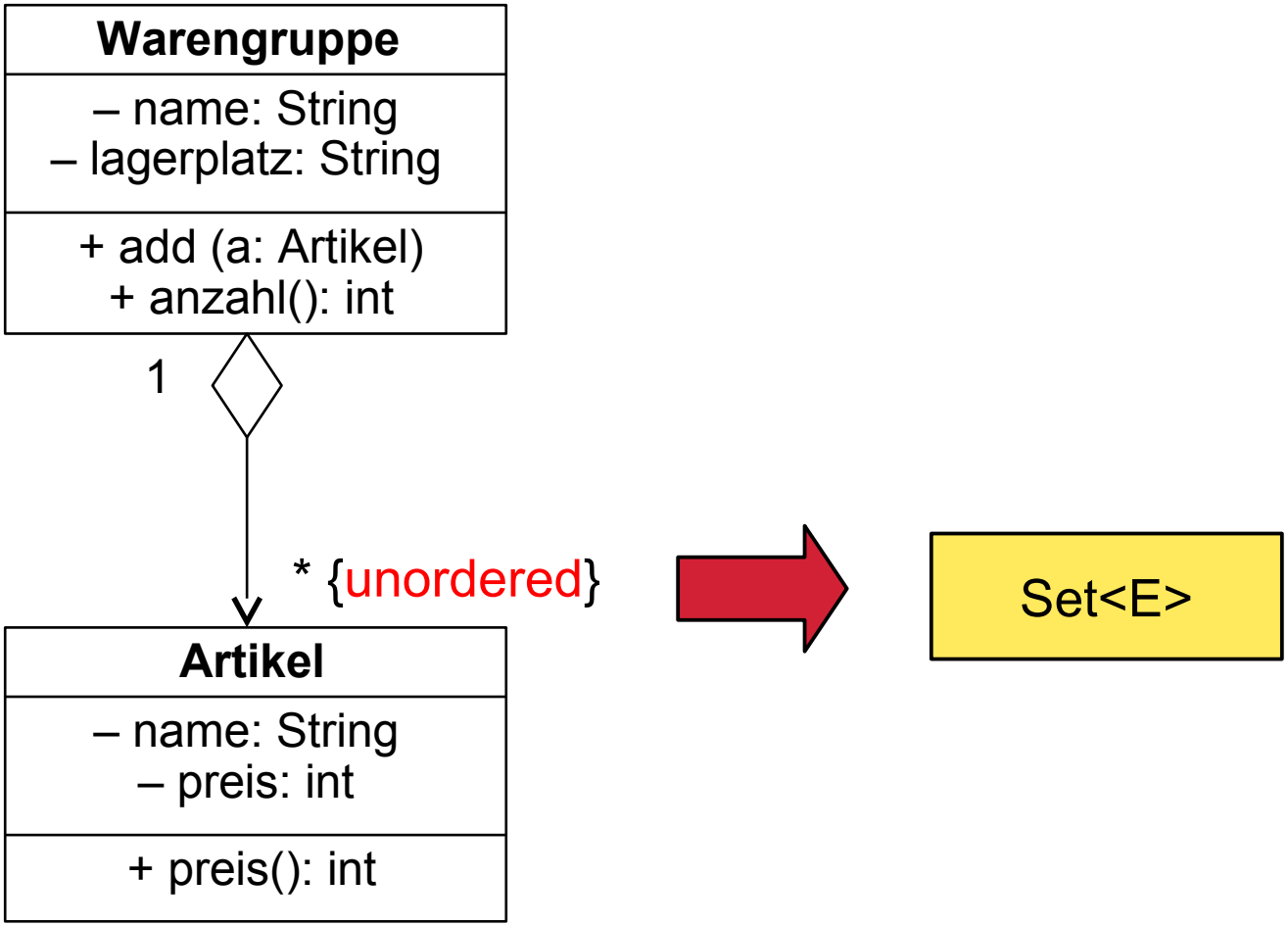


Ungeordnete Mengen: java.util.Set<E> (Auszug)

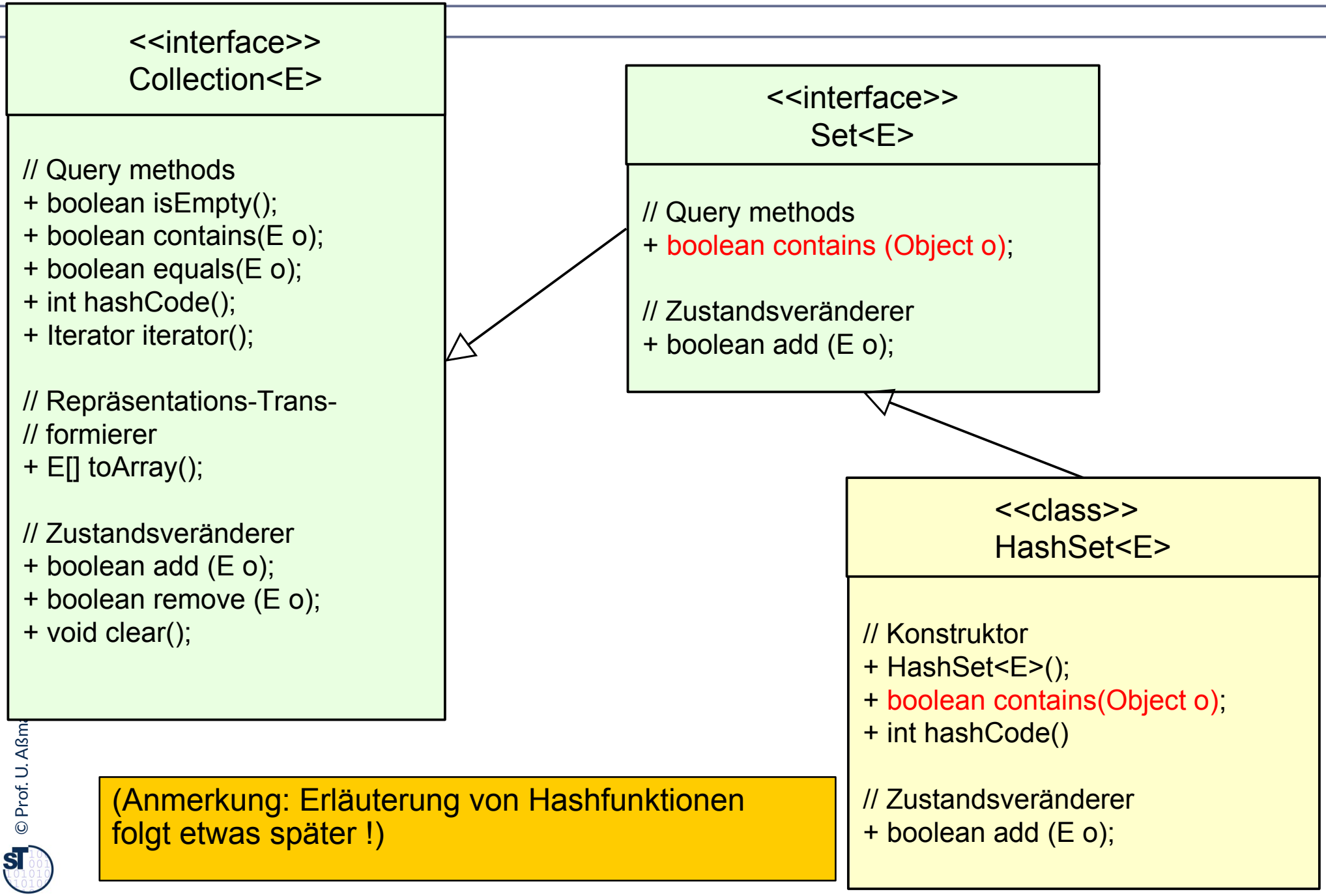


Anwendungsbeispiel für Set<E>

- 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

21.5.2 Re-Definition der Gleichheit von Elmenten in Set

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

- Die Operation **==** prüft auf *Referenz- bzw. Pointer-Gleichheit*, d.h. physische Identität der Objekte in der Halde
  - 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
    - String s.equals(String r) prüft auf *strukturelle Gleichheit*, d.h. auf eine gleiche Folge von Zeichen
  - für selbstdefinierte Klassen genau überlegen, was man möchte:
    - Standardbedeutung Referenzgleichheit
    - oder strukturelle Gleichheit
  - 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:

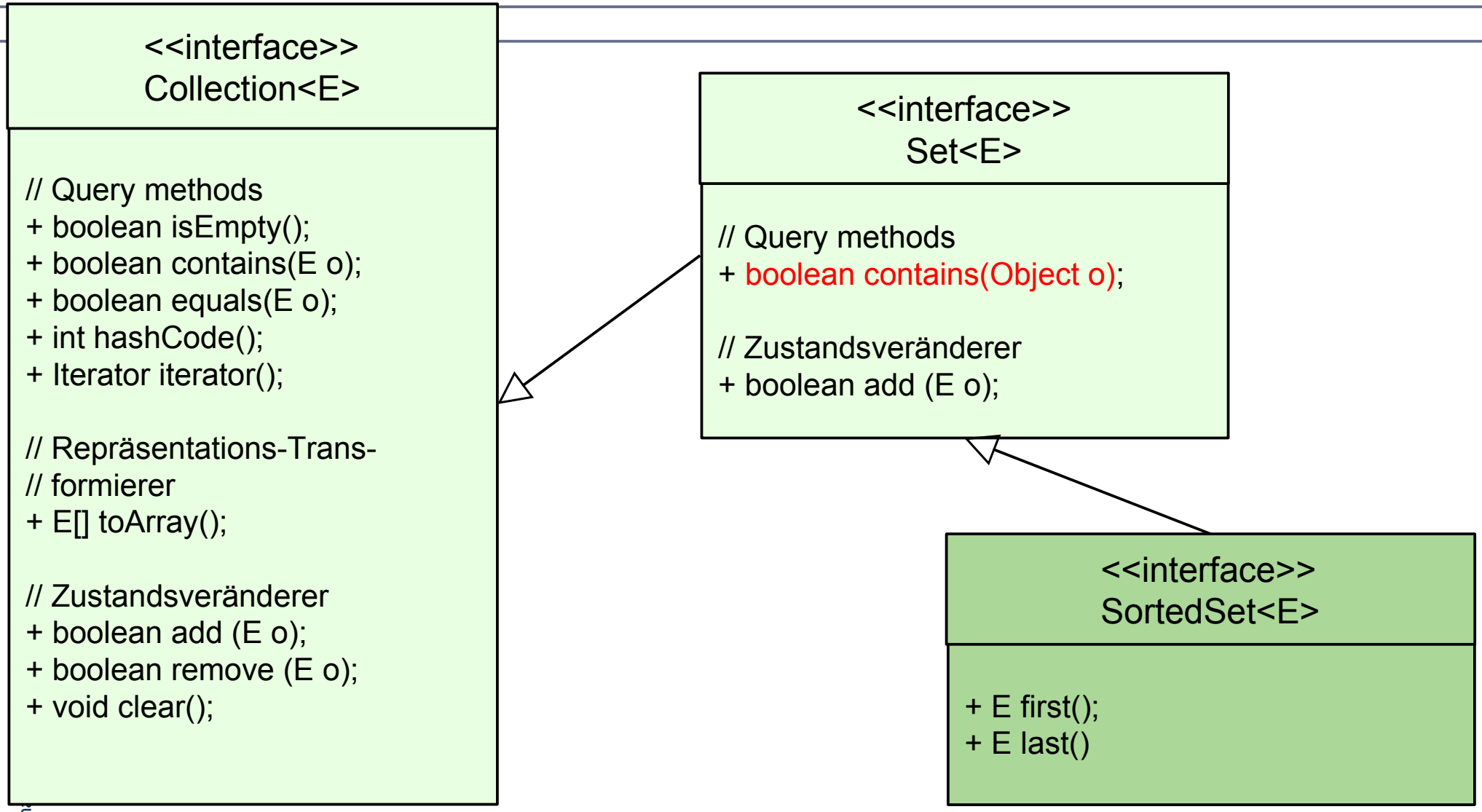
```
Warengruppe Moebel
Schrank(300) Tisch(200) Stuhl(100)
```

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>`
- Modifikation 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.TreeSet<E>.compareTo(TreeSet<E>.java, Compiled Code)

corrected

- 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)

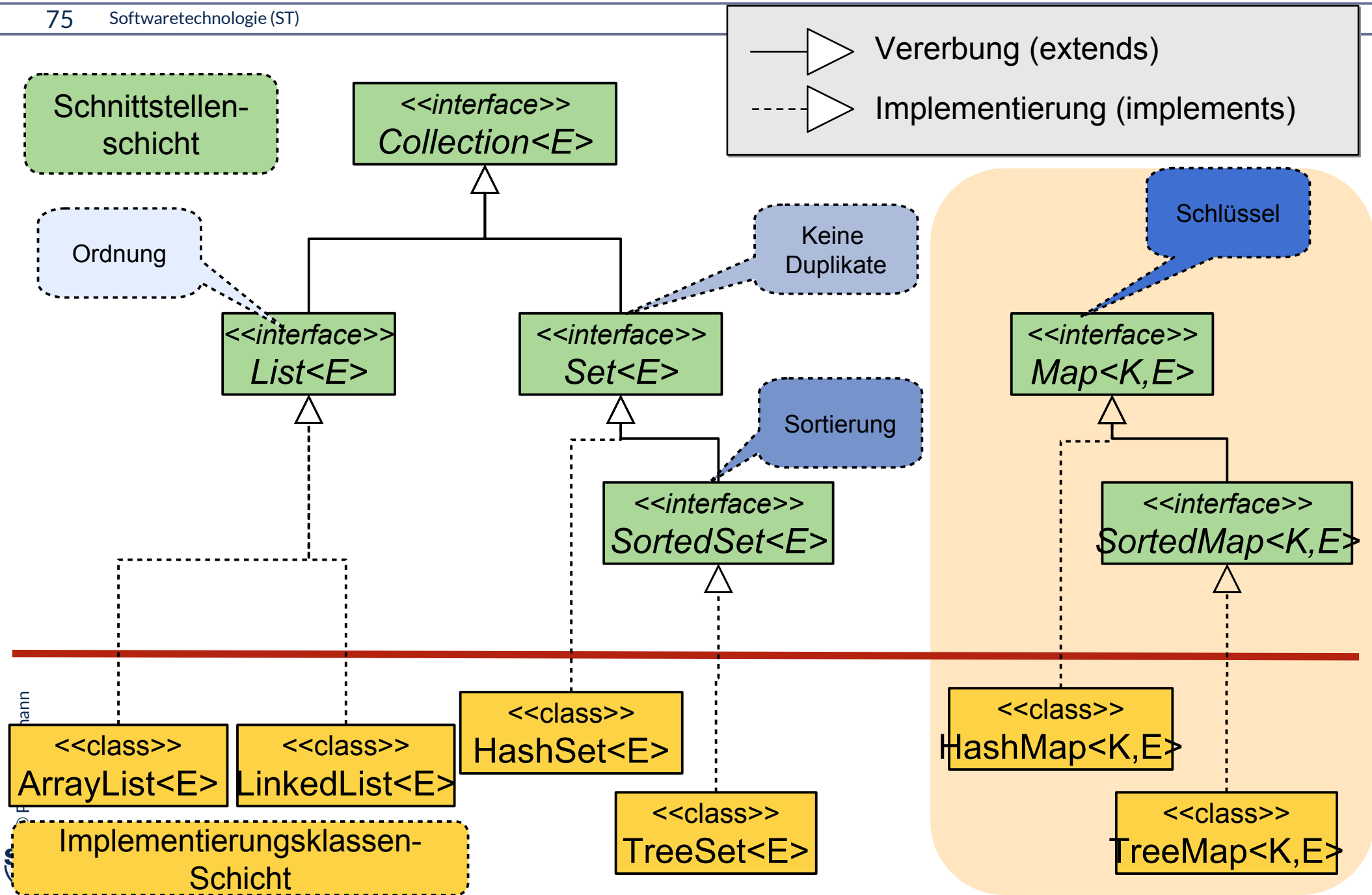
Typ	Einfügen	Enthalten	Iteration
HashSet	36,14	106,5	39,39
TreeSet	150,6	177,4	40,04

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

21.6 Kataloge mit Map

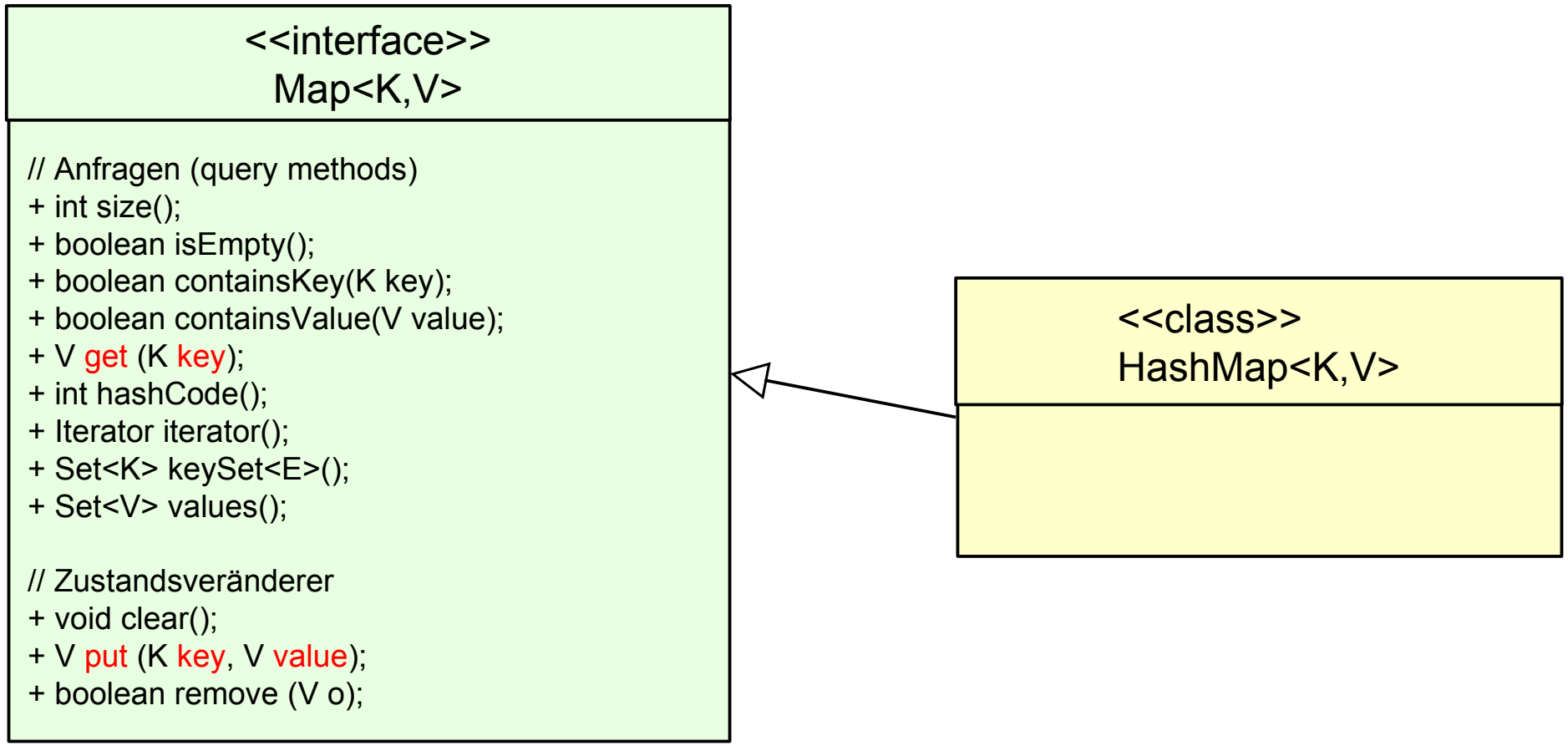
- Ein **Katalog (Wörterbuch, dictionary, map)** ist eine Abbildung eines Schlüssel-Ausgangsbereiches in einen Wertebereich.
- Achtung: Im Collection-Hierarchie der Map wird “Element” als “Value” bezeichnet

Schnittstellen und Implementierungen im Collection-Framework bilden generische Behälterklassen

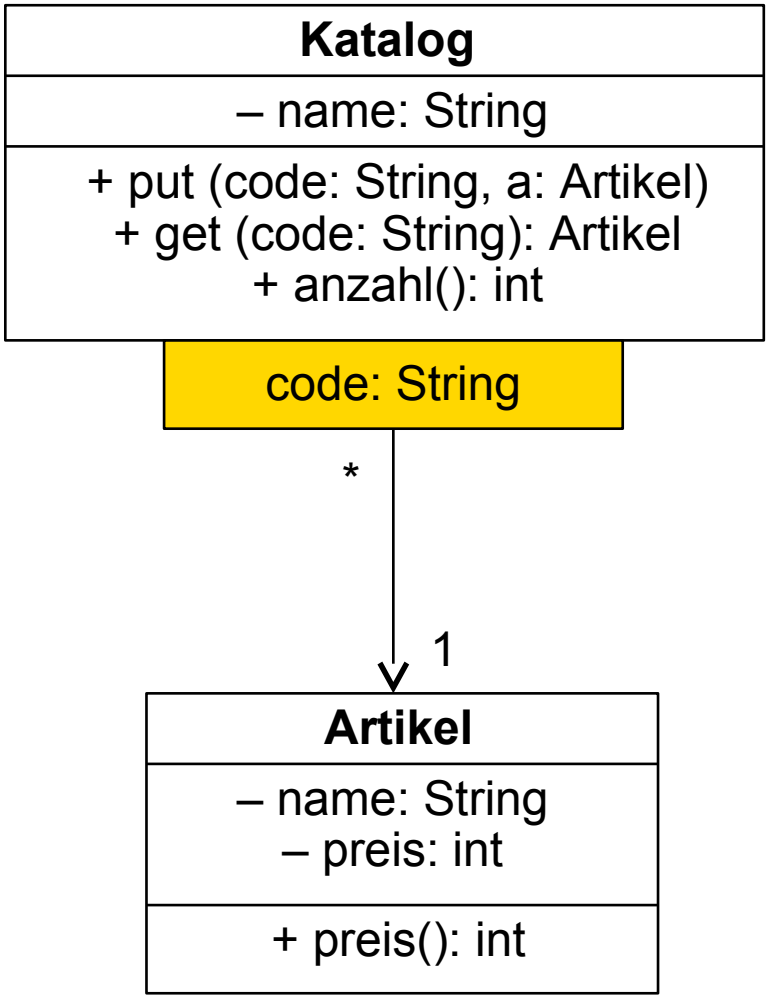


java.util.Map<K,V> (Auszug)

- 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
- Achtung: wir nennen nun im Folgenden “E – Element” “V – Value”, wie in der JCF



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 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:

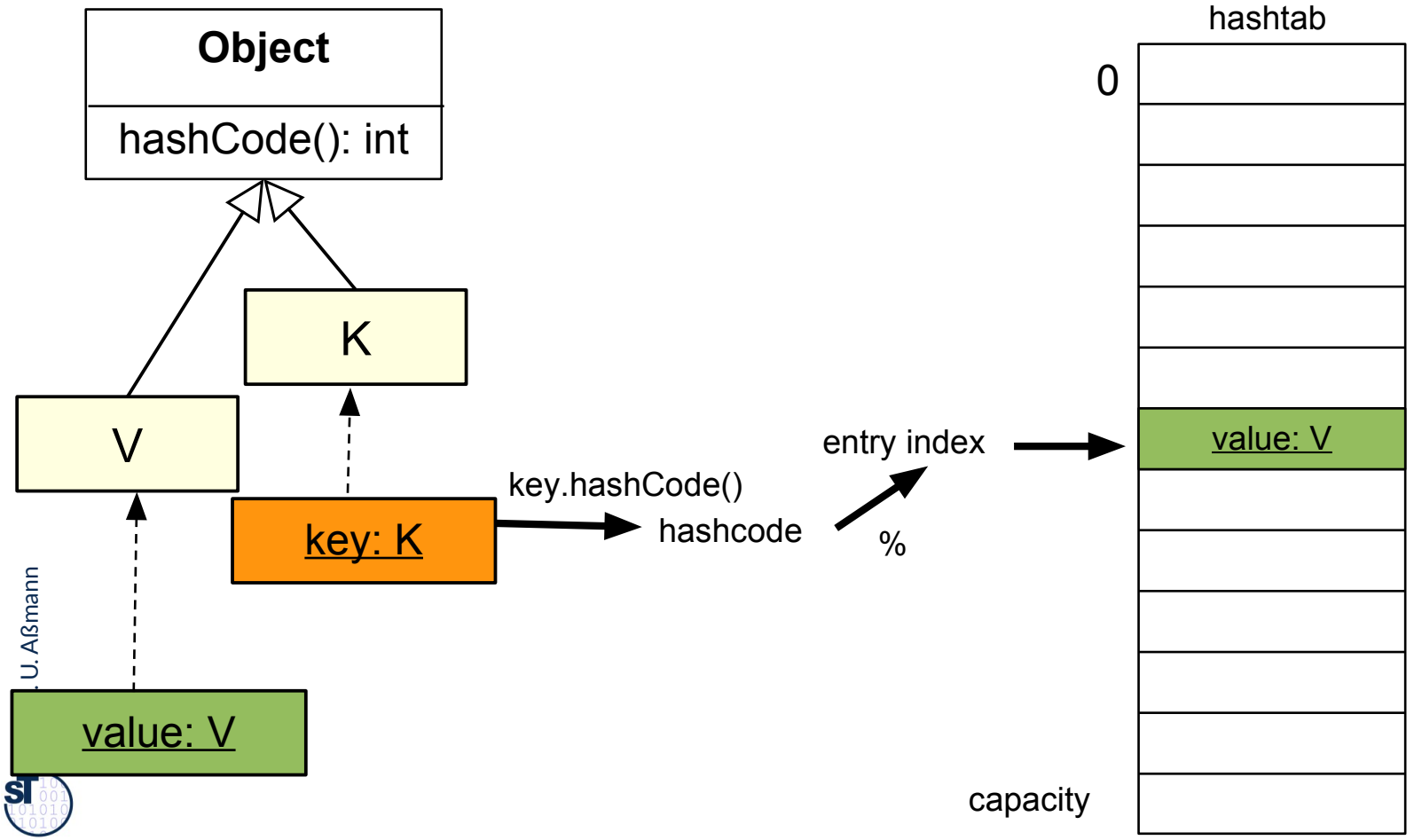
```
Katalog Katalog1
 M03 -> Schrank(300)
 M02 -> Stuhl(100)
 M01 -> Tisch(200)

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).

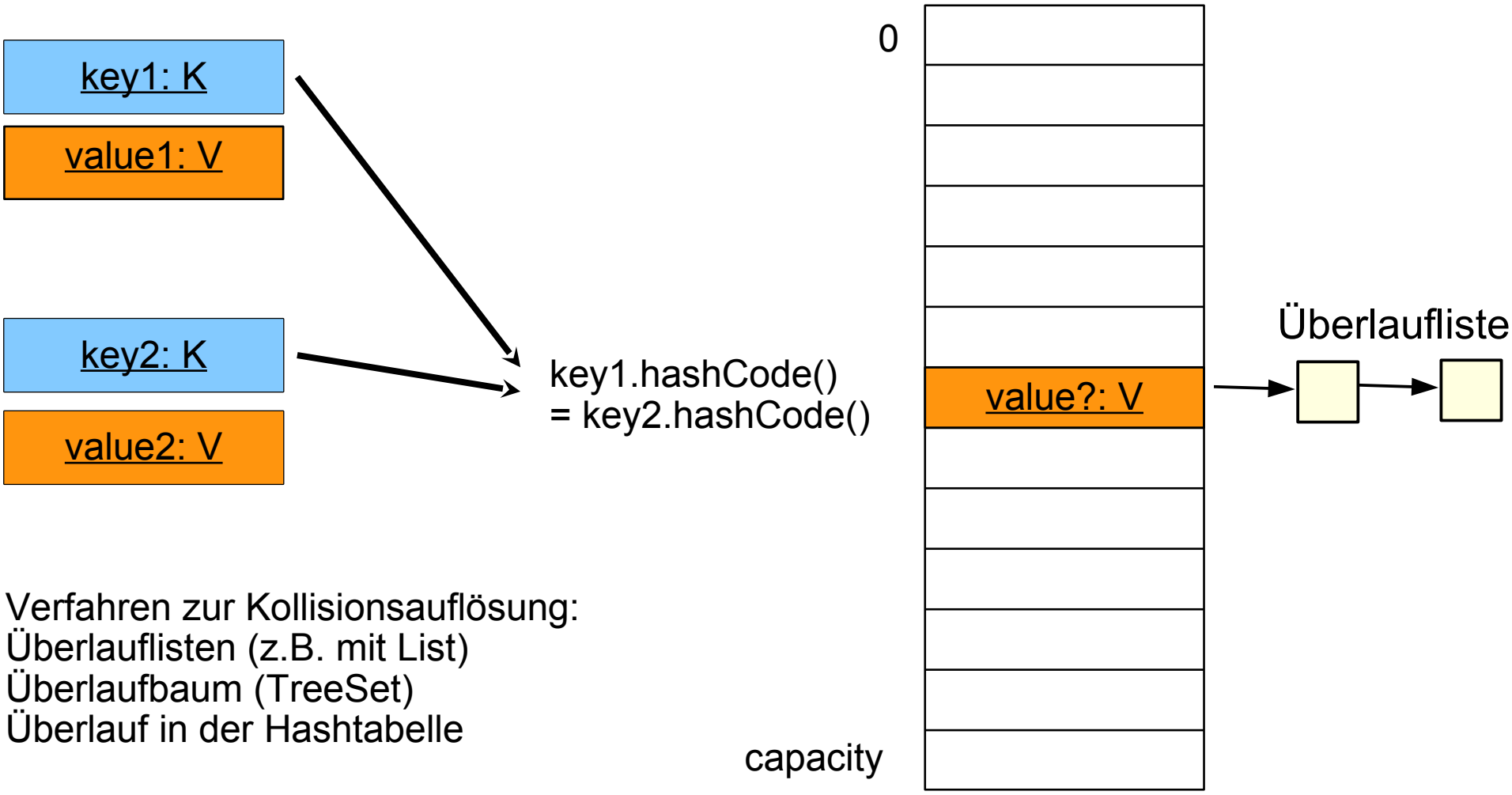
Prinzip der Hashtabelle
Effekt von hashtab.put(key:K,value:V)

- 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



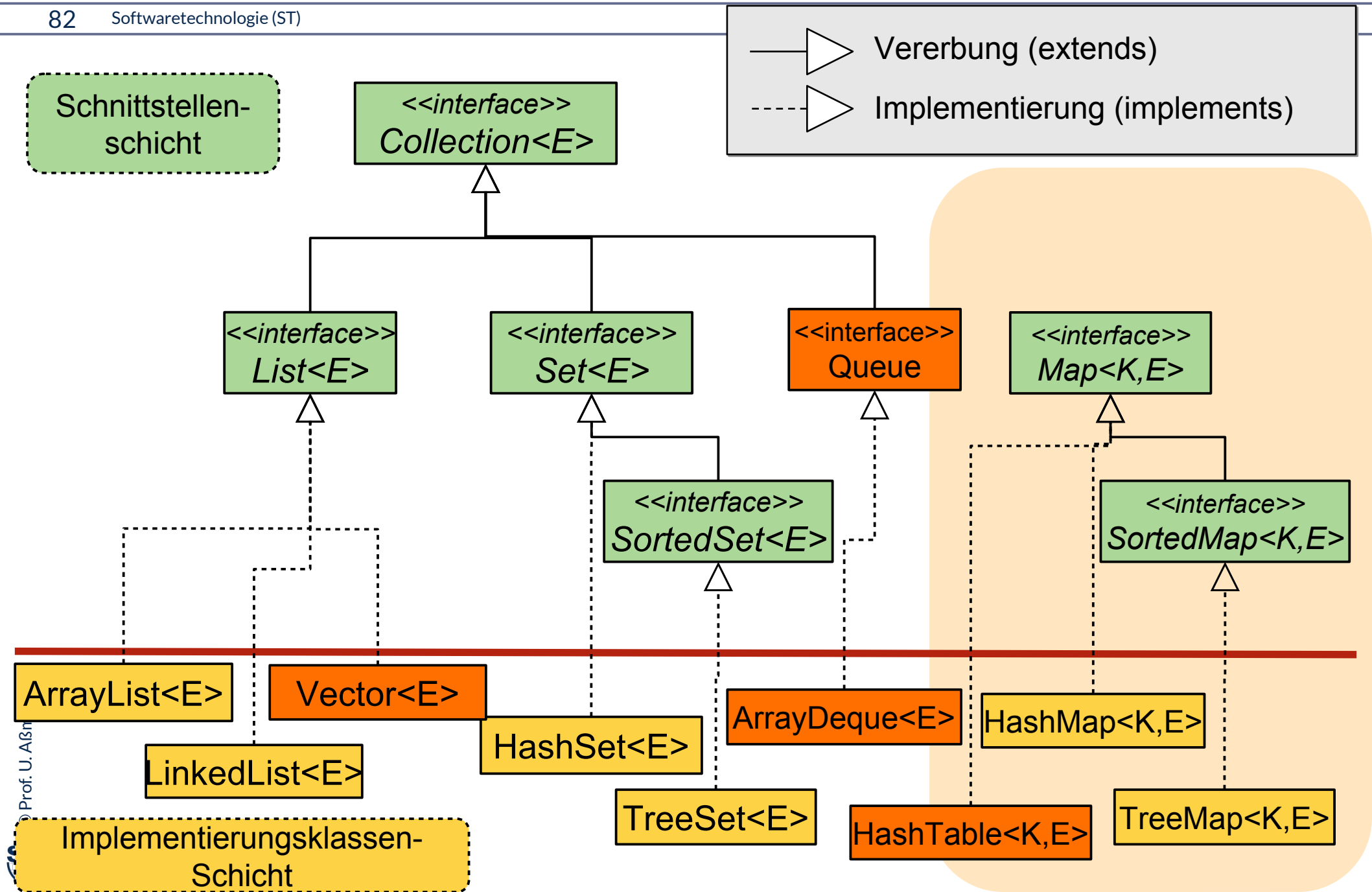
Kollision beim Einstechen

- 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 Schnittstellen und Implementierungen im JCF



21.7 Optimierte Auswahl von Implementierungen von Datenstrukturen

- zur Implementierung verschiedener Arten von UML Assoziationen

Ermittlung der benötigten Assoziationsarten und Facetten der Implementierungsklassen

- An das Ende einer UML-Assoziation können folgende *Bedingungen* notiert werden:
  - Ordnung: {ordered} {unordered}
  - Eindeutigkeit: {unique} {non-unique}
  - Kollektionsart: {set} {bag} {sequence}

- 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 (Ermittlung Benutzungsprofil)

Funktion

Identifikation der funktionalen Anforderungen an die Datenstruktur:
Ermittlung der Facetten der Funktionalität

Abstraktion auf die wesentlichen Eigenschaften

Suche nach vorgefertigten Lösungen
(Nutzung der Collection-Bibliothek)

Effizienz

Ermittlung der häufig benutzten Operationen (Benutzungsprofil)

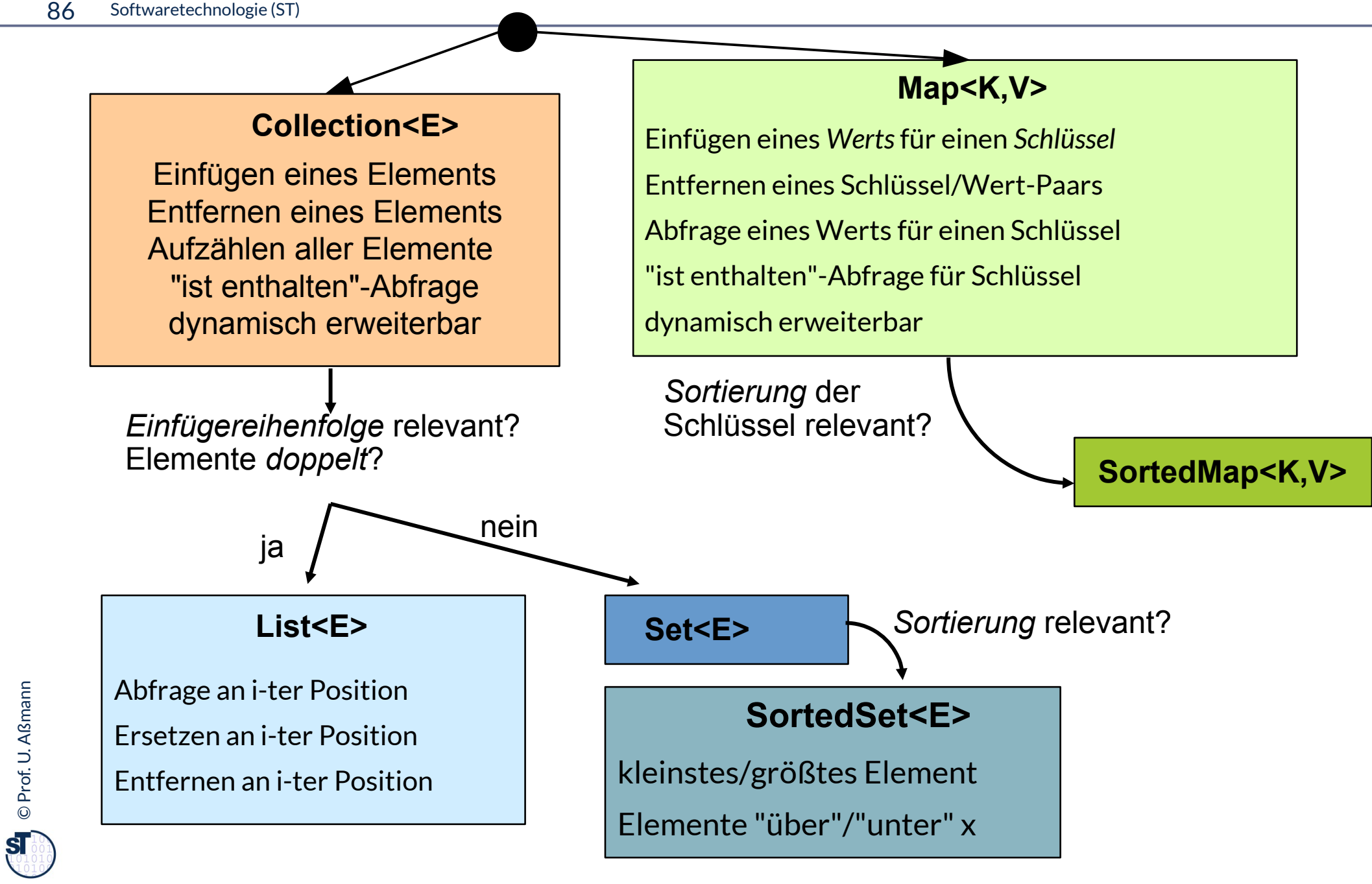
Effizienz-Verbesserung: Ggf. Experimente mit Laufzeiten,
Speicherverbrauch, Energieverbrauch

Anpassung an
vorgefertigte Lösung

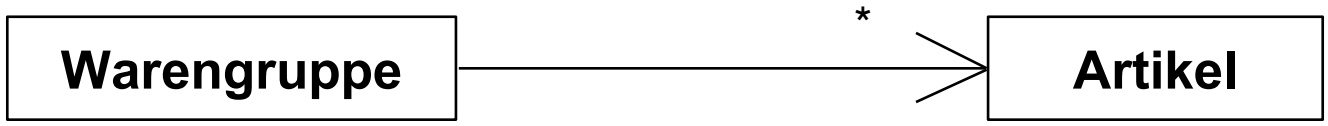
corrected

Entwicklung einer
eigenen, neuartigen Lösung

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

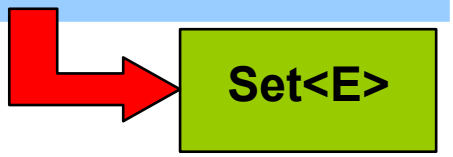


Beispiel: Realisierung von unidirektionalen Assoziationen

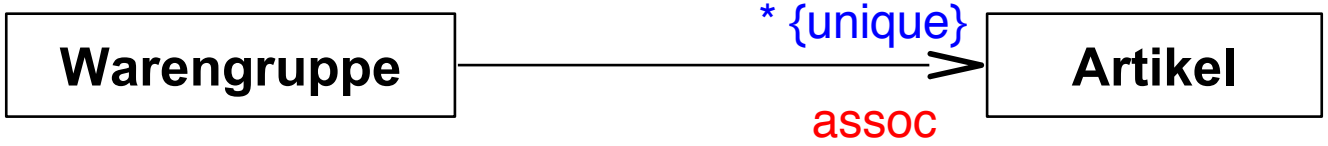


Datenstruktur im Warengruppe-Objekt für Artikel-Referenzen

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

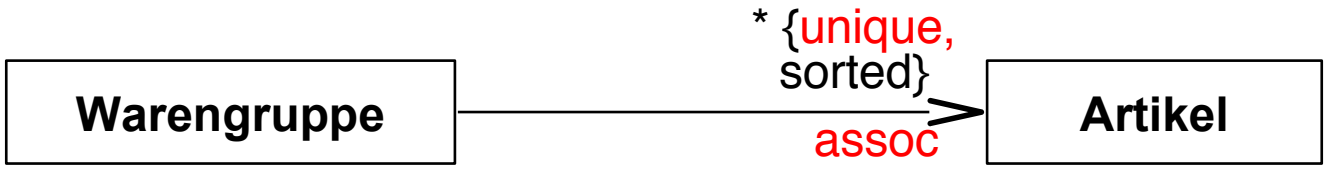


Beispiel:
Realisierung von ungeordneten Assoziationen mit Set<E>




```
class Warengruppe {
  private Set<Artikel> assoc;
  ...
  public void addAssoc (Artikel ziel) {
    assoc.add(ziel);
  }

  public boolean testAssoc (Artikel ziel) {
    return assoc.contains(ziel);
  }

  public Warengruppe {
    assoc = new HashSet<Artikel>();
  }
}
```

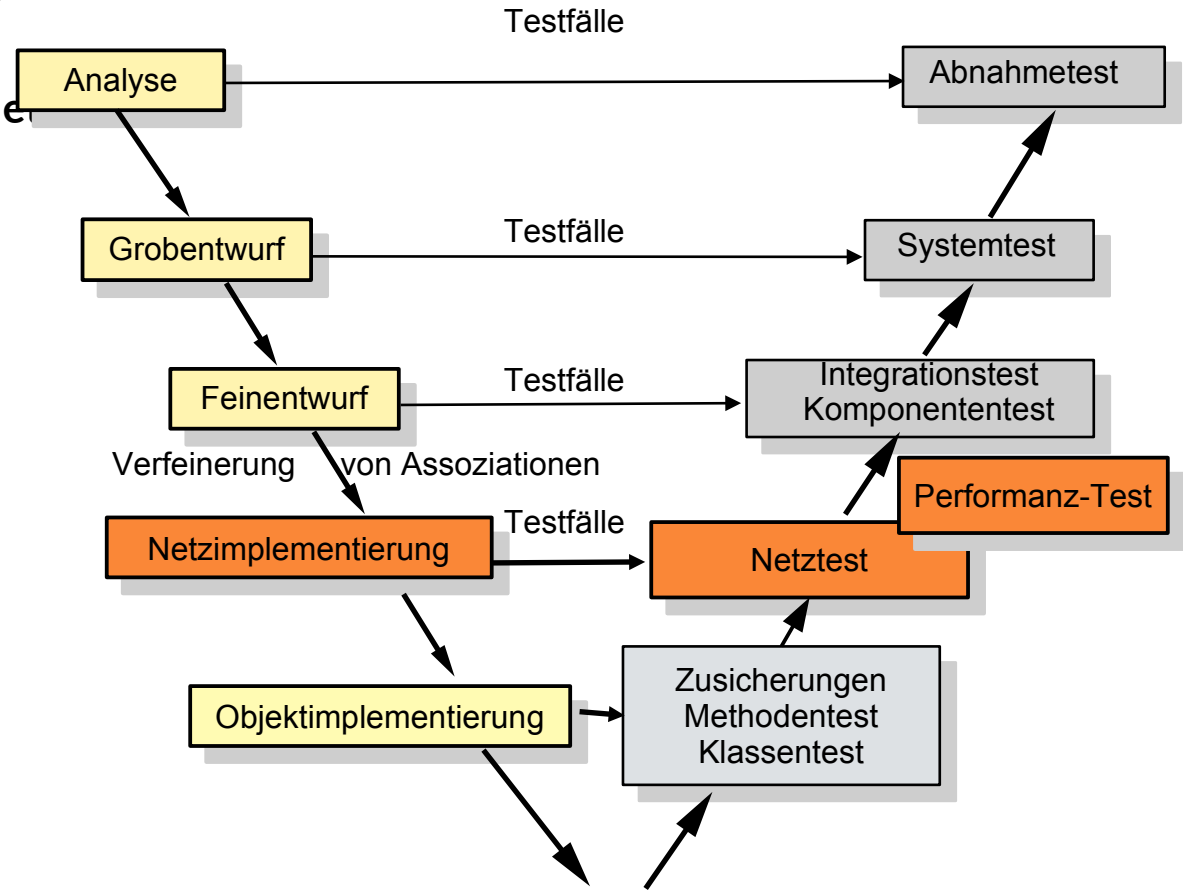
Beispiel:
Realisierung von sortierten Assoziationen mit Set<E>




```
class Warengruppe {
  private SortedSet<Artikel> assoc;
  ...
  public void addAssoc (Artikel ziel) {
    assoc.add(ziel);
  }

  public boolean testAssoc (Artikel ziel) {
    return assoc.contains(ziel);
  }

  public Warengruppe {
    assoc = new TreeSet<Artikel>();
  }
}
```

Ziel: 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 Ne entwerfen**
  - Dann die einzelne Klasse testen
  - **Dann das Netz testen**
  - Dann die Komponente
  - Dann das System
  - Dann der beta-Test
  - Dann der Geschwindigkeitstest
  - Zum Schluss der Akzeptanztest (Abnahmetest)



Was haben wir gelernt

- Wann wende ich welchen Entwicklungsprozess, SAD oder RAD, an?
  - Unterscheide statische vs. dynamische vs. keine Typisierung
  - Safe Application Development (SAD) ist nur mit statischen Typisierung möglich
  - Rapid Application Development (RAD) benötigt dynamische Typisierung
- Testen:
  - Test von Objektnetzen ist wichtig für die Qualität von Software
  - Performance-Test von Objektnetzen ist einfach mit Schnittstellen und verschiedenen Implementierungen (z.B. polymorphen Behälterklassen)
- Generische Collections besitzen den Element-Typ als Typ-Parameter
  - Element-Typ verfeinert Object
  - Weniger Casts, mehr Typsicherheit
- Das Java Collection Framework (JCF) bietet geordnete, ungeordnete Collections sowie Kataloge

The End

- Diese Folien bauen auf der Vorlesung Softwaretechnologie auf von © Prof. H. Hussmann, 2002. Used by permission.
- Warum ist das Lesen in einer ArrayList I.d.R. schneller als in der LinkedList?
- Warum ist das Löschen auf Index 0 in der ArrayList langsamer als in der LinkedList?
- Erklären Sie den Unterschied der 4 Facetten der Collections
- TreeSet verwendet eine baumartige Datenstruktur. Erklären Sie die Vorteile eines Baums für den Insert und das Suchen von Elementen.
- Warum sollte man sich in der Anforderungsanalyse mit aUML um die tagged values von Multiplizitäten kümmern?
- Wieso ist die Hörsaalübung wichtig?
- Welche Rolle spielen Prädikat-Schnittstellenklassen im JCF? Erklären Sie die Comparable-Schnittstelle.

Appendix A
Generische Command Objekte

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);
   }
}
```

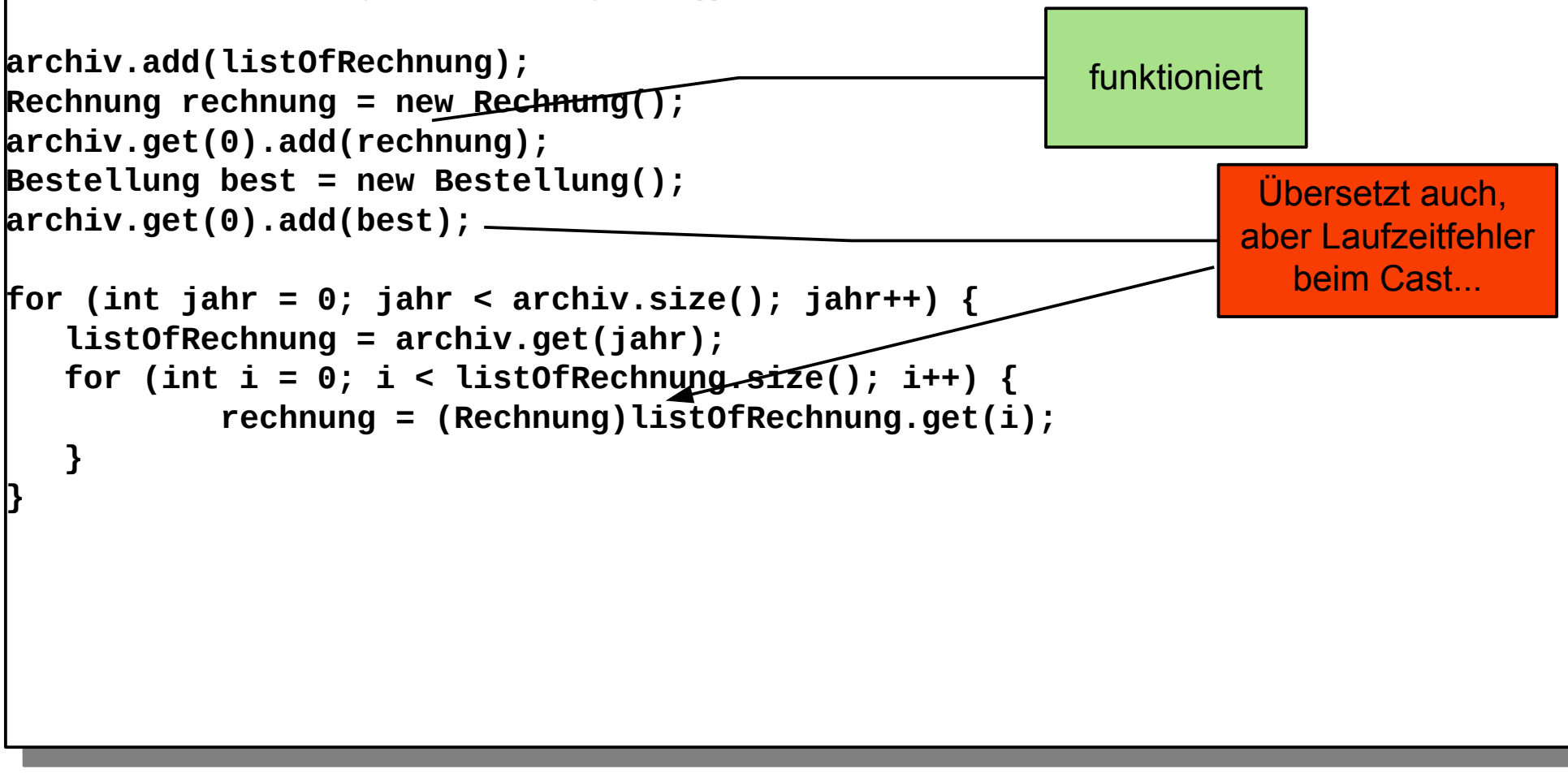
funktioniert

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);
   }
}
```



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

Implementierungsmuster Command: Generische Methoden als Funktionale Objekte

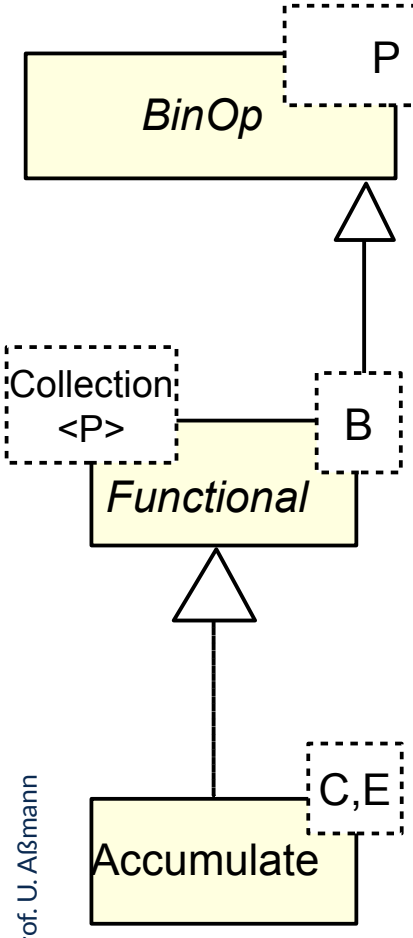
Ein **Funktionalobjekt (Kommandoobjekt)** ist ein Objekt, das eine Funktion darstellt (reifiziert).

- **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

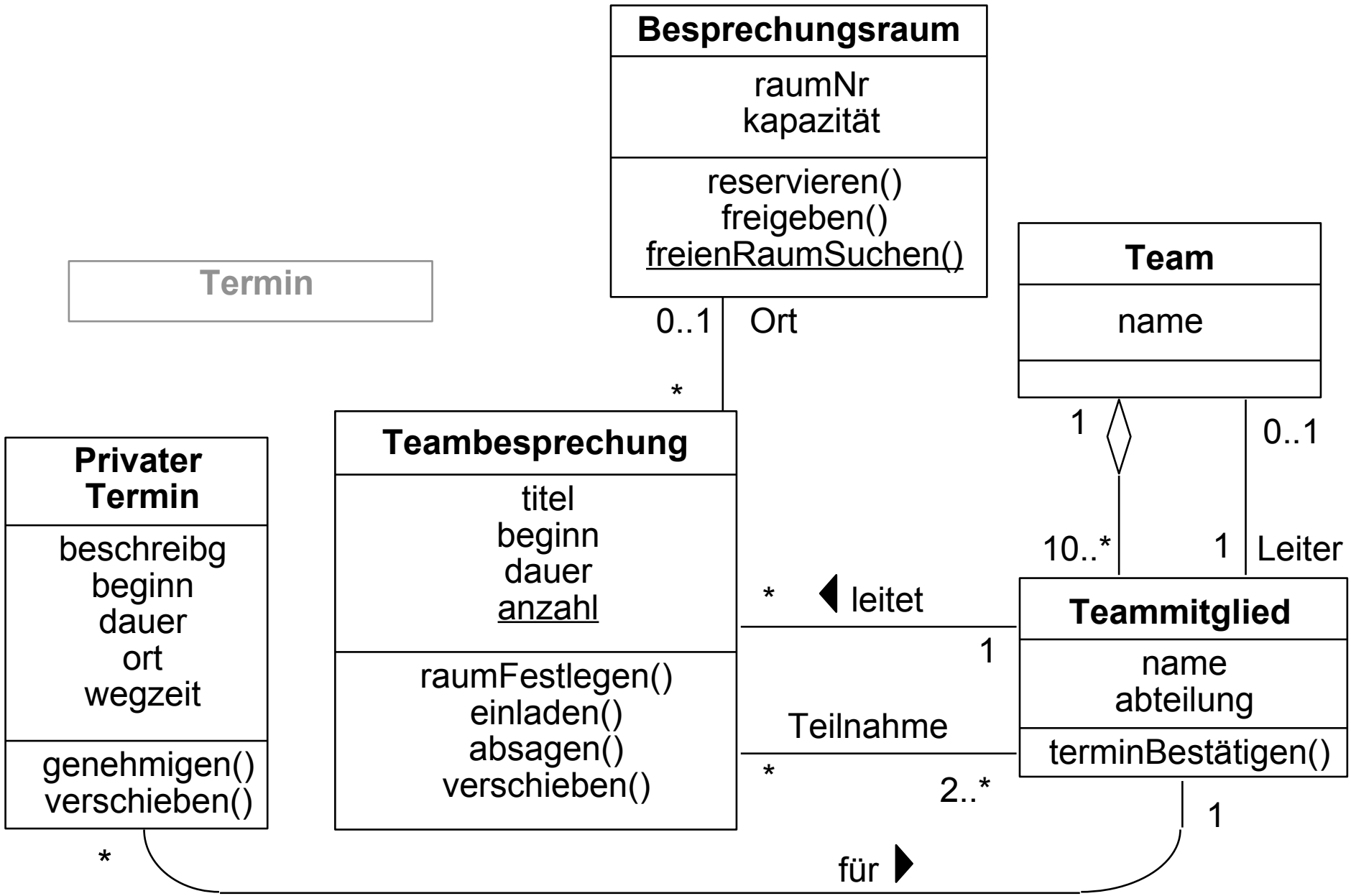
- Anwendung: Akkumulatoren und andere generische Listenoperationen




```
// 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;
  }
}
```

Appendix B
Bestimmung von konkreten Datentypen

Beispiel 2: Analysemodell der Terminverwaltung



Beispiel 2: Sortierte Liste von Räumen in der 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)

Online: TerminvTreeSet.java

Raumverwaltung: Freien Raum suchen

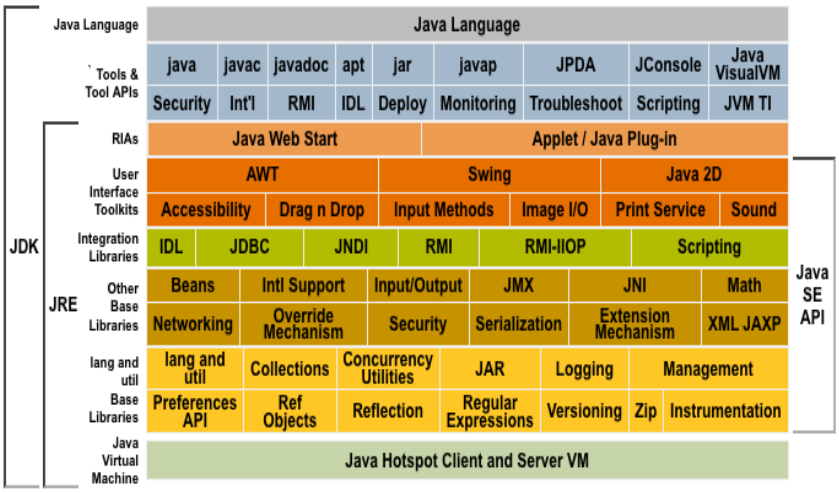

```
class Raumverwaltung {
  // Vorhandene Raeume, aufsteigend nach Größe sortiert
  // statisches Klassenattribut und -methode
  private static SortedSet<E> vorhandeneRaeume
      = new TreeSet<Besprechungsraum>();

  // 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;
  } ...
}
```

21) Netzverfeinerung (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 20-1.1, 22.05.20
some bugs corrected

Obligatorische Literatur

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



Empfohlene Literatur

- Im Wesentlichen sind die Dokumentationen aller Versionen von Java 6 an nutzbar:
  - http://download.oracle.com/javase/6/docs/
  - http://download.oracle.com/javase/8/docs
  - https://docs.oracle.com/en/java/javase/12/
  - https://docs.oracle.com/en/java/javase/12/docs/api/java.base/module-summary.html
- Tutorials http://download.oracle.com/javase/tutorial/
- https://docs.oracle.com/javase/tutorial/tutorialLearningPaths.html
- Generics Tutorial:
- http://download.oracle.com/javase/tutorial/extra/generics/index.html

https://de.wikipedia.org/wiki/Tom_Dooley_(Lied)

Video für Gitarrelernen mit Tom Dooley
https://www.youtube.com/watch?v=_J8fFFSVp3Y

http://www.magistrix.de/lyrics/The%20Kingston%20Trio/Tom-Dooley-228080.html

“Stay hungry, stay foolish” (Steve Jobs)

- http://news.stanford.edu/2005/06/14/jobs-061505/ (English)
- http://www.mac-history.de/apple-people/steve-jobs/2008-10-05/ubersetzung-der-rede-von-steve-jobs-vor-den-absolventen-der-stanford-universitat-2005
- Last words:
  - https://www.youtube.com/watch?v=J3dXK4UvAaE

Hinweis: Online-Ressourcen

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

  `21-Bestellung-Listen/Bestellung0.java`

  `..`

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

  `21-Warengruppe-Mengen/Warengruppe0.java`

  `..`

  `21-Warengruppe-Mengen/Warengruppe3.java`
- Beispiel “Kataloge mit Maps”

  `21-Katalog-Mit-Abbildung/Katalog.java`

  `21-Katalog-Mit-Abbildung/Katalog2.java`

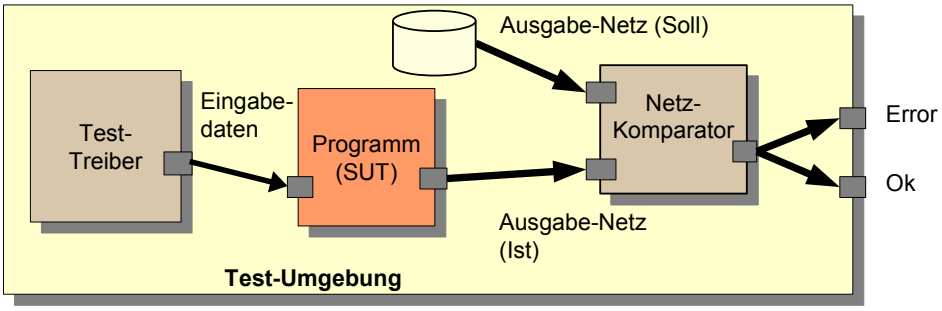
21.1 Verfeinern von Assoziationen

- Die bekannteste Art, Objektnetze zu realisieren, besteht darin, sie in Collection-Datentypen (Nachbarlisten, Nachbarmengen) zu überführen.
- Das hat aber auch seine Tücken und ist schwer zu testen.

Wdh.: Objektorientierte Software hat eine test-getriebene Architektur für Objektnetze

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

> Solange ein Programm keine Tests hat, ist es keine Software



Warum ist Objektnetz-Test wichtig?

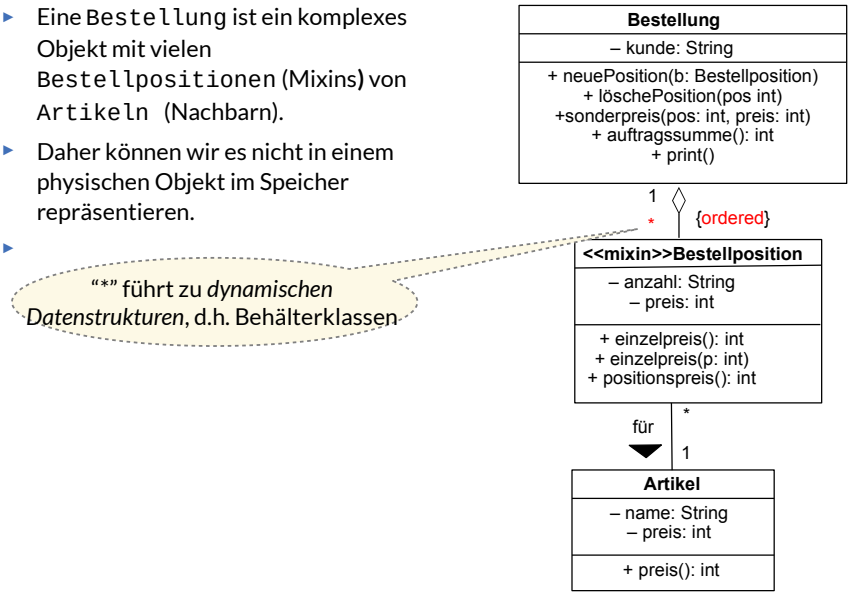
- Schon mal 3 Tage nach einem Zeiger-Fehler (pointer error) in einem Objektnetz gesucht?

- 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

Komplexe Objekte

- Eine `Bestellung` ist ein komplexes Objekt mit vielen `Bestellpositionen` (Mixins) von `Artikeln` (Nachbarn).
- Daher können wir es nicht in einem physischen Objekt im Speicher repräsentieren.
-



•Zentrale Frage: Wie bilde ich einseitige Assoziationen aus UML auf Java ab?

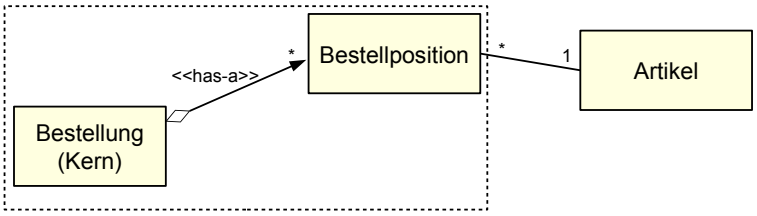
•Bestellung ist ein typisches fachliches Konzept aus dem Bereich Informationssysteme für Betriebe.

Komplexe Objekte

Ein **komplexes Objekt (Subjekt, big object)** ist ein Objekt,
das in einem Programm und im Speicher wegen seiner Komplexität durch
ein Kernobjekt und mehrere *Unterobjekte (Teilobjekte, Mixin, Satellit)* dargestellt wird.

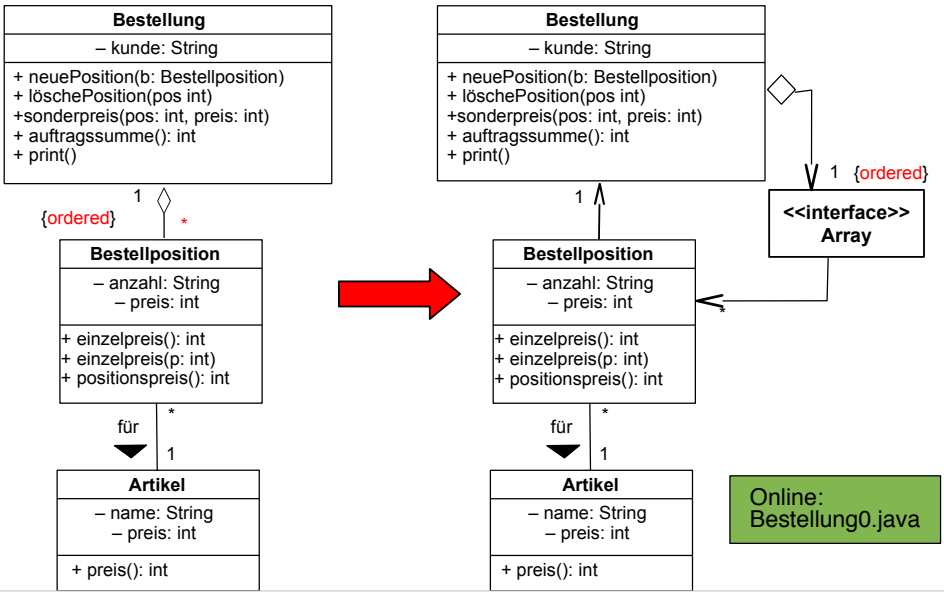
Seine innere Struktur besteht aus einem meist hierarchischen Objektnetz,
dem **Endo-Netz**.

- Ein **Unterobjekt** ist ein **Kernobjekt** angelagert und bildet mit ihm ein integriertes **komplexes Objekt**
  - Das Unterobjekt hat also keine eigene Identität, sondern teilt seine Identität mit dem Kernobjekt (logische Einheit)
  - Es repräsentiert einen Teil des komplexen Objekts
- Das **Endo-Netz** kann beliebig groß werden. Dann werden Behälterklassen benötigt



Verfeinern von Assoziationen in komplexen Objekten mit Endo-Netz (Verfeinerung des Endo-Netzes)

- Modell einer Bestellung, eines komplexen Objekts mit einfachem Endonetz (Hierarchie):



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

Einfache Realisierung des Endo-Netzes mit Arrays – Was ist problematisch?

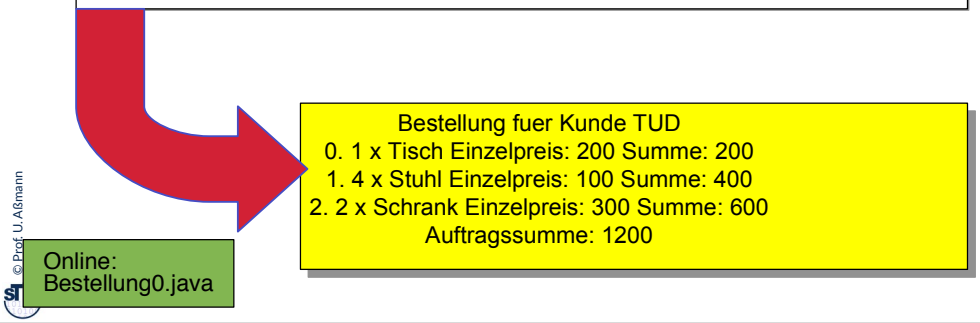

```
class Bestellung {
  private String kunde;
  private Bestellposition[] liste;
  private int anzahl = 0;

  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

Problematisch ist hier, dass Java-Arrays eine fixe Obergrenze haben. Das Programm stürzt also ab. Wo genau?

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(); ...}
```



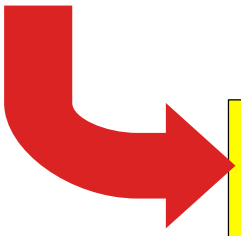
```
Bestellung fuer Kunde TUD
 0. 1 x Tisch Einzelpreis: 200 Summe: 200
 1. 4 x Stuhl Einzelpreis: 100 Summe: 400
 2. 2 x Schrank Einzelpreis: 300 Summe: 600
          Auftragssumme: 1200
```

Online:
Bestellung0.java

In der Regel muss mann für Realisierung mit fixen Collections oder Arrays Testprogramme selbst entwickeln, sie gibt es nicht vorgefertigt. Dadurch steigt der Testaufwand.

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

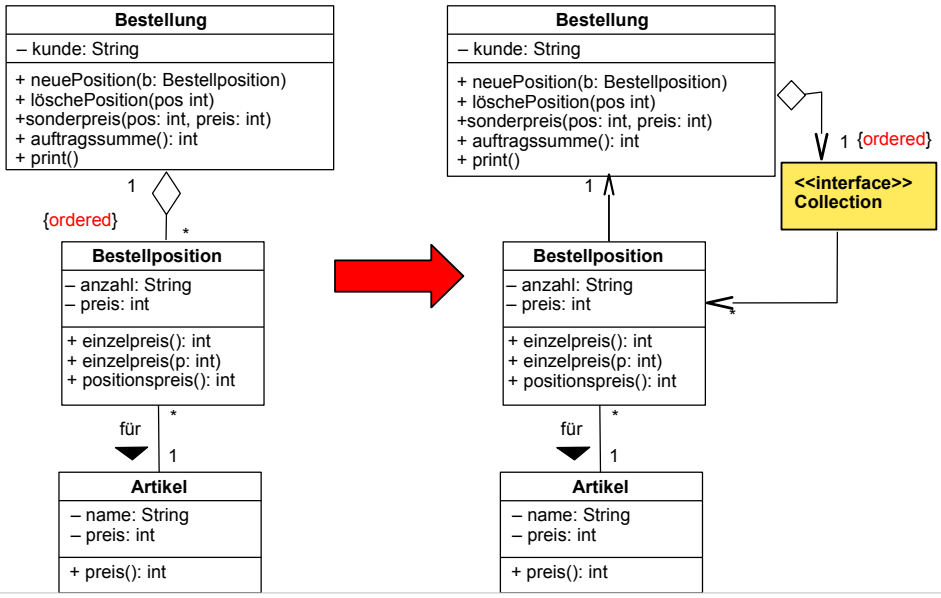
Collections (Behälterklassen)

- Probleme werden durch das Java-Collection-Framework (JCF) 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 **flexibel** 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



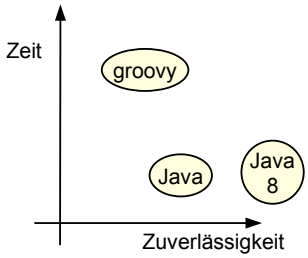
Einfache Realisierung mit Collection-Klasse “List”


```
class Bestellung {
  private String kunde;
  private List<Bestellposition> liste;
  private int anzahl = 0;

  public Bestellung(String kunde) {
    this.kunde = kunde;
    liste = new LinkedList<Bestellposition>(); // Erklärung später
  }
  public void neuePosition (Bestellposition b) {
     liste.set(anzahl,b);
    anzahl++;    // was passiert jetzt bei mehr als 20 Positionen ?
  }
  public void loeschePosition (int pos) {
       liste.remove(pos);
  }
  public void sonderpreis (int pos, int preis) {
    liste.get(pos).einzelpreis(preis);
  }
  public int auftragssumme() {
    int s = 0;
    for(int i=0; i<anzahl; i++) s += liste.get(i).positionspreis();
    return s;
  }
}
```

Online:
Bestellung1LinkedList.java

Orthogonale Trends in der Softwareentwicklung

- **Rapid Application Development (RAD)**
  - *Schneller viel* Code schreiben
  - Hilfsmittel:
    - Typisierung weglassen
    - Ev. *dynamische Typisierung*, damit Fehler zur Laufzeit identifiziert werden können
    - Mächtige Operationen einer Skriptsprache



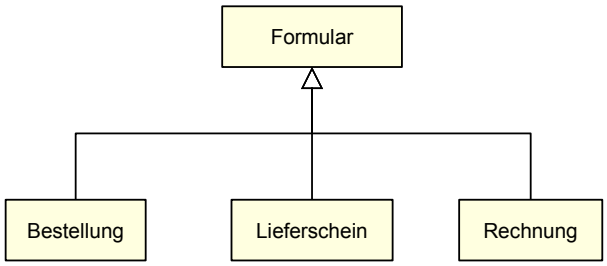
- **Safe Application Development (SAD)**
  - *Guten, stabilen, wartbaren* Code schreiben
  - Mehr *Entwurfswissen* aus dem Entwurf in die Implementierung übertragen
  - Hilfsmittel:
    - *Statische Typisierung*, damit der Übersetzer viele Fehler entdeckt
    - Generische Klassen
  - Aus der Definition einer Datenstruktur können Bedingungen für ihre Anwendung abgeleitet werden
  - *Generische Collections* für typsicheres Aufbauen von Objektnetzen (Java)

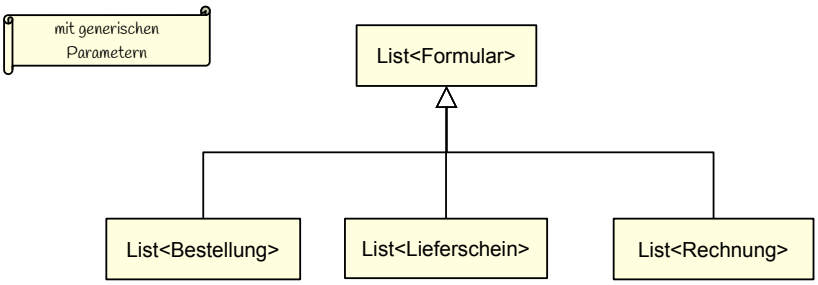
- Gradual Typing für beides
  - Typen werden Schritt für Schritt annotiert
  - http://ecee.colorado.edu/~siek/gradual-obj.pdf

21.2 Die Collection-Bibliothek Java Collection Framework (JCF) (Behälterklassen)

- Ungetypte Behälterklassen für RAD
- Generische Behälterklassen für SAD

Bsp.: Klassen einer Hierarchie und ihre Behälter-Hierarchie





21.2.1 Die einfache Collection-Bibliothek ohne Element-Typen

- Behälterklassen: `Collection<Object>`

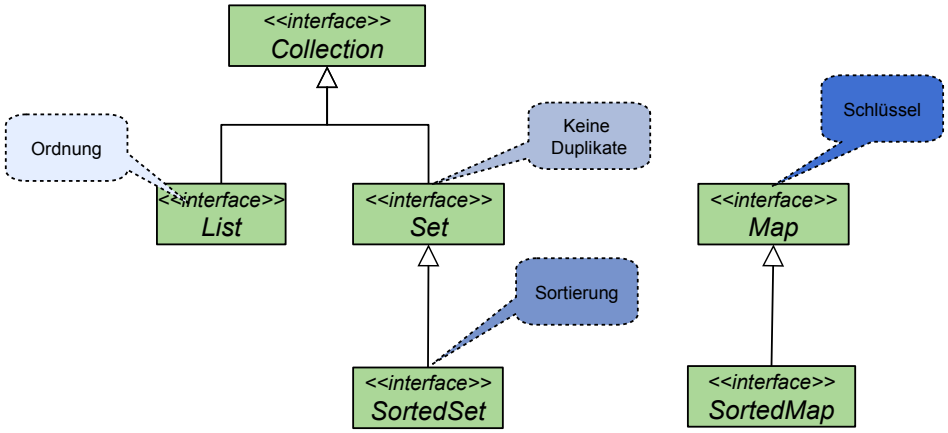
Facetten von Behälterklassen (Collections)

- 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

- Die Schnittstellen-Hierarchie der Collections (*vor* Java 1.5)
- Eine Collection kann beliebige Objekte enthalten (Element == `Object`)



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

Dieses Klassendiagramm enthält noch keine Generics, sondern ist über dem Elementtyp Object definiert.

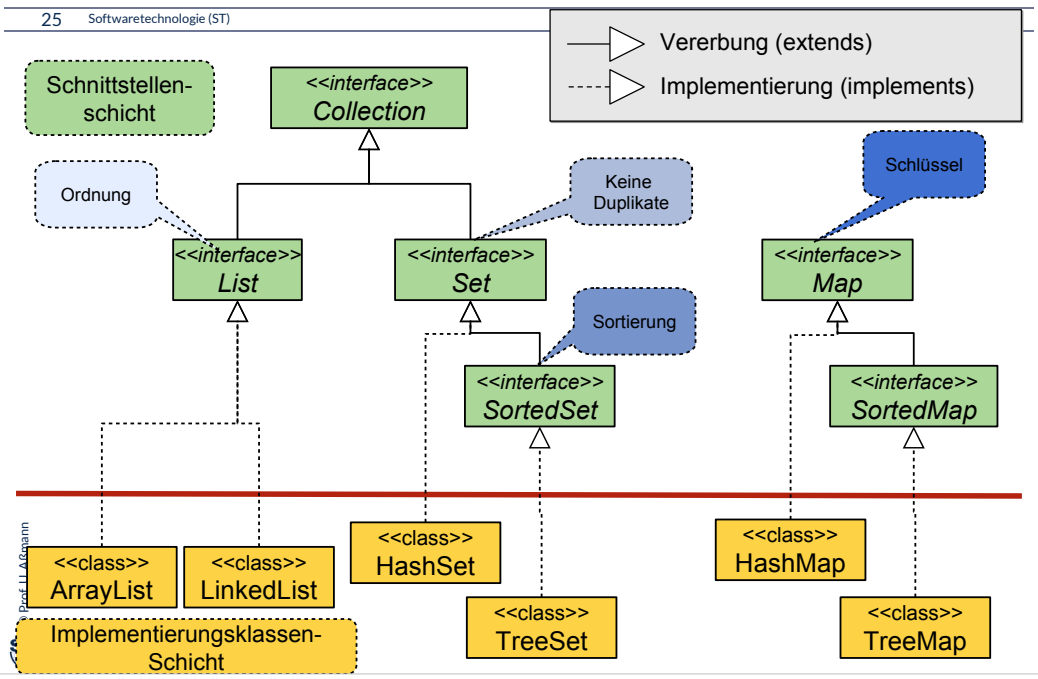
Damit kann man den Elementtyp nicht festlegen.

Generics dienen der Festsetzung von Nachbartypen. Hier können sie eingesetzt werden, um Elementtypen von Collections festzulegen.

Ordnung heißt: die Integer sind als Index in die Collection verfügbar; ein Element ist einem Index zugeordnet

Sortierung heißt: die Werte der Collection sind sortiert

JCF: Schnittstellen-Schicht vs Implementierungsschicht



Im JDK gibt es sehr viele Implementierungsklassen für die Schnittstellen der JCF Collections.

Problem 1 ungetypter Schnittstellen in Behälterklassen: 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!!

Behälterklassen sollten genau qualifizieren, was sie beinhalten.

Klassische Behälterklassen arbeiten auf “Object”, können also nicht näher qualifizieren, was sie beinhalten. Das führt leicht zu Programmierfehlern.

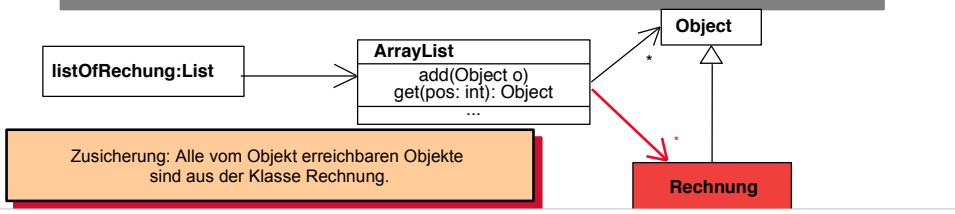
Problem 2 ungetypter Schnittstellen in Behälterklassen: Unnötige Casts

- 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



Wenn man aus einer ungetypten Behälterklasse ein Element herausholt, muss es von “Object” auf einen spezielleren Typ gewandelt (gecastet) werden.

Das führt zu “Codeverschmutzung”, d.h. unübersichtlichem Code.

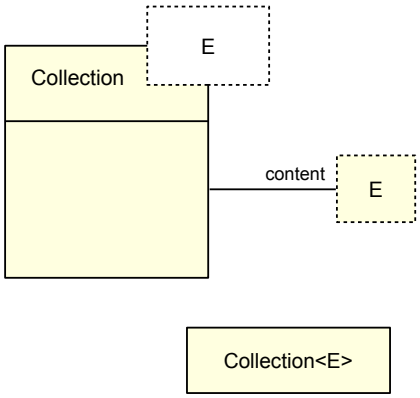
21.2.2 Die Collection-Bibliothek mit Element-Typen

- Generische Behälterklassen erlauben SAD für Objektnetze:
- `Collection<E>` ist die oberste Schnittstelle

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>

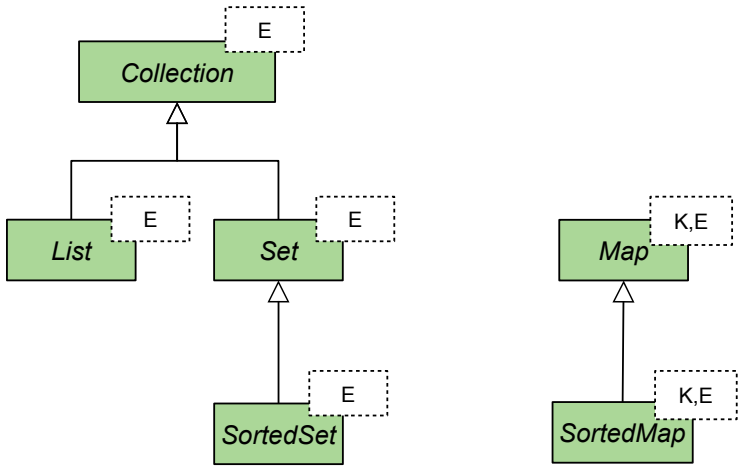
- In Java
  - Sprachregelung: “Collection of E”

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


Auch Collection-Klassen können mit einem Typparameter versehen werden.

Collection-Hierarchie mit generischen Schnittstellen

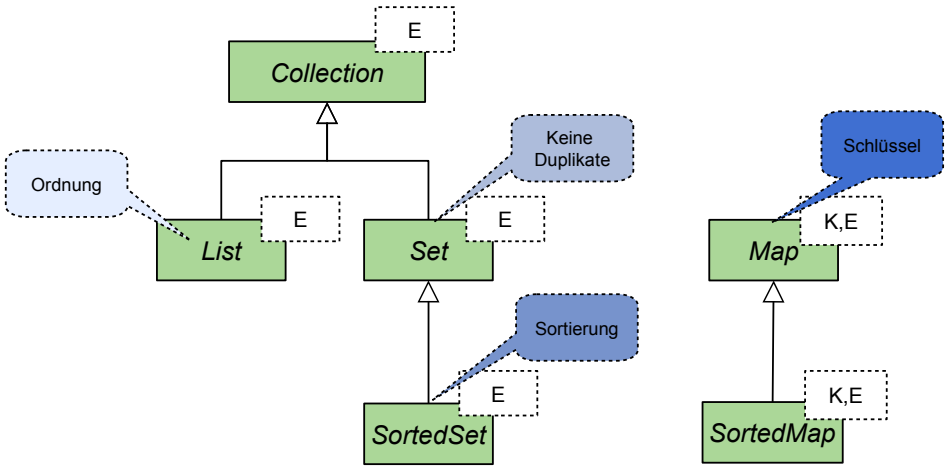
- Die generische Schnittstellen-Hierarchie der Collections (seit Java 5)
- Eine Collection enthält nur Elemente vom Typ E, bzw. Schlüssel vom Typ K
  - E: Element, K: Key



Die generische Schnittstellenhierarchie kann nun benutzt werden, um die Elementtypen (E und K) festzulegen.

Collection-Hierarchie mit generischen Schnittstellen

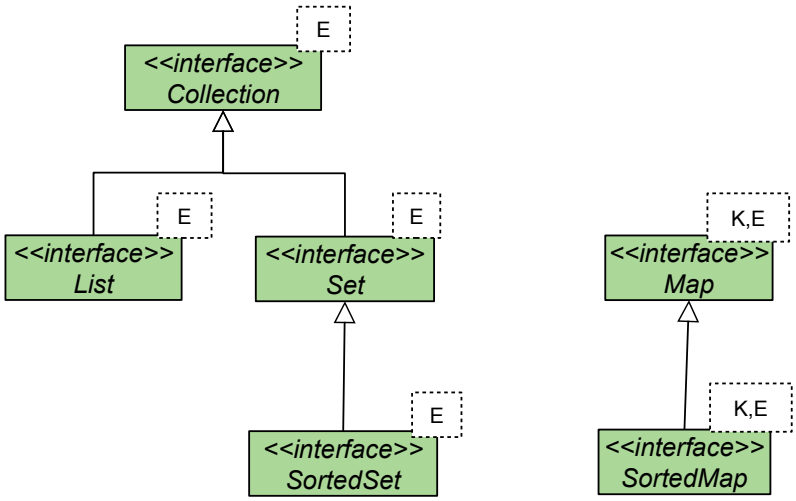
- Die generische Schnittstellen-Hierarchie der Collections (seit Java 5)
- Eine Collection enthält nur Elemente vom Typ E, bzw. Schlüssel vom Typ K
  - E: Element, K: Key



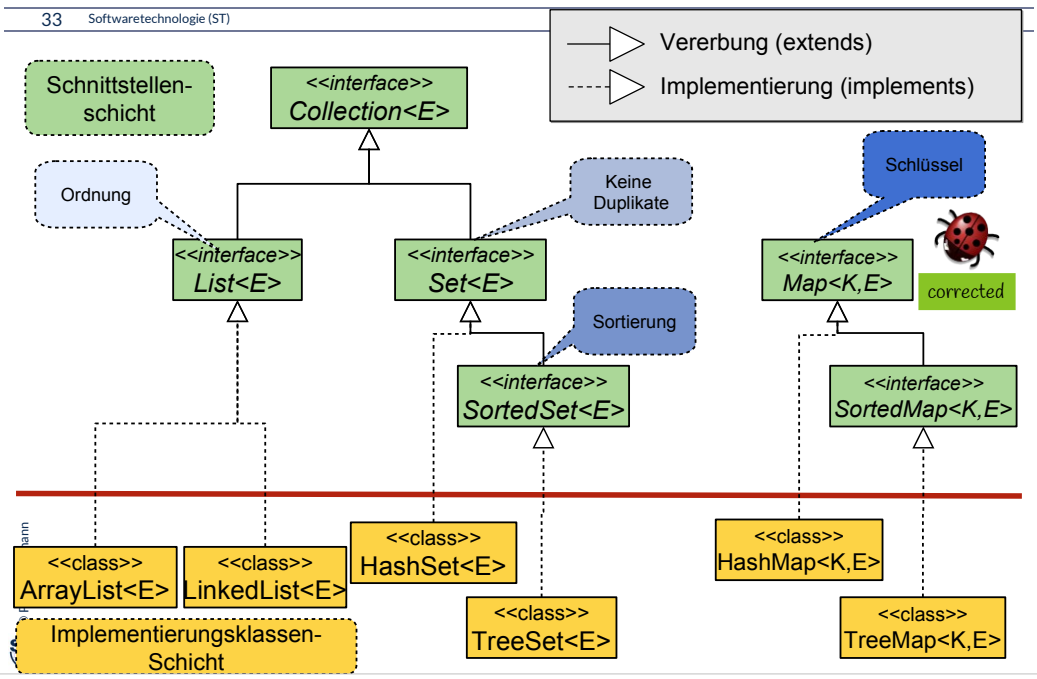
/home/ua1/Courses/ST1/Material/jdk-docs/index.html

Collection-Hierarchie mit generischen Schnittstellen

- Die generische Schnittstellen-Hierarchie der Collections (seit Java 5)
  - E: Element, K: Key

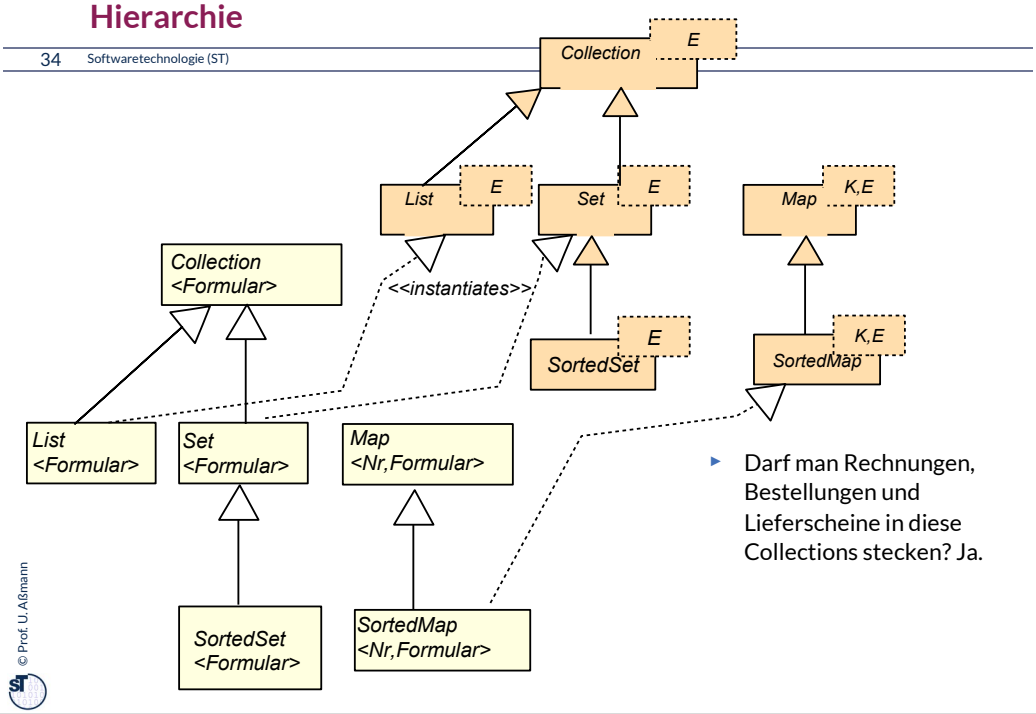


JCF: Schnittstellen Schicht vs Implementierungsschicht



Dieses Diagramm zeigt, wie sich die generische Nachbartypschranke aus den Schnittstellen in die Implementierungsklassen hin fortsetzt.

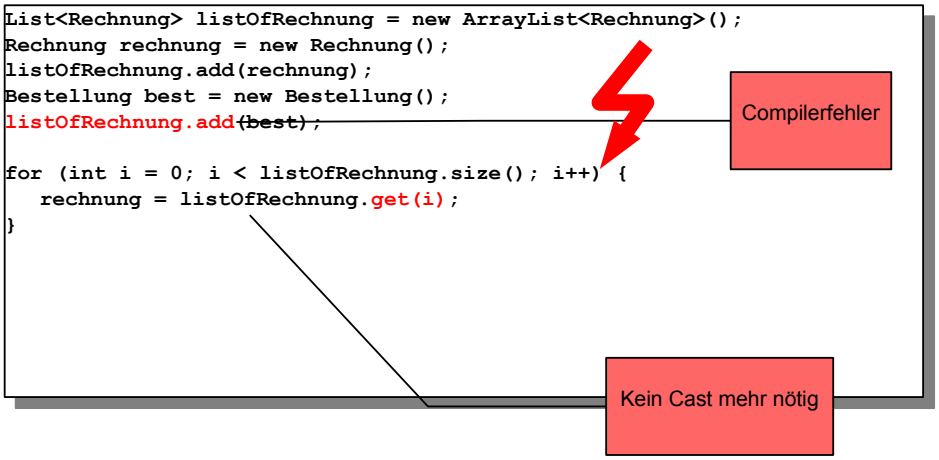
Beispiel: Instanziierung der generischen JCF Schnittstellen-Hierarchie



Man kann die Hierarchie von generischen Behälterklassen insgesamt für einen konkreten Elementtyp instanziieren.

SAD löst Probleme

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

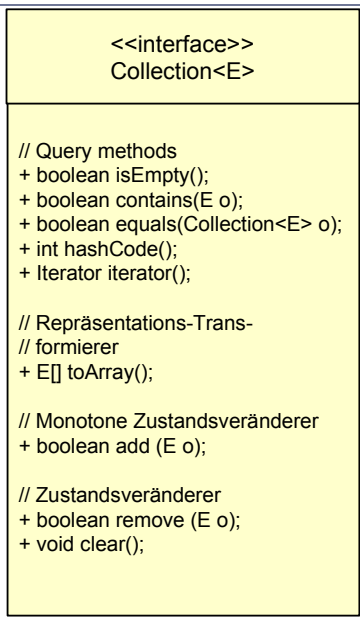


Mit generischen Behälterklassen können jetzt die Elemente unterschieden werden, und es braucht keine “Casts”.

21.2.2 Schnittstellen im Detail

Schnittstelle java.util.Collection (Auszug)


```
public interface Collection<E> {
  // Anfragen (Queries)
  public boolean isEmpty();
  public boolean contains (E o);
  public boolean equals(Collection<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();
  …
}
```

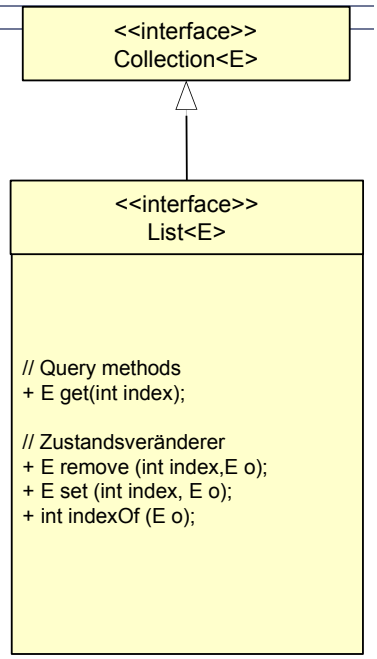


An Behälterklassen sieht man schön, welche Arten von Methoden es gibt (siehe Kapitel “Klassen”).

Man sieht hier: Query-Methoden, Repräsentationswechsler, monotone und nicht-monotone Zustandsveränderer,

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);
  ...
}
```

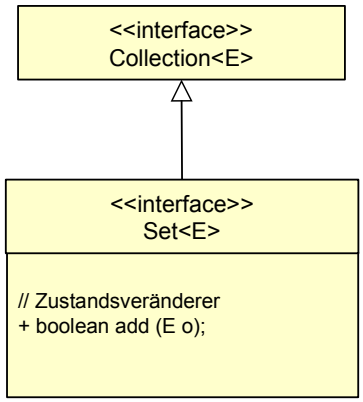


Listen erlauben es, per Index auf die Elemente der Collection zuzugreifen.

Entsprechend werden alle Kategorien von Methoden mit neuen Methoden erweitert, die Zugriff per Index ermöglichen.

Unterschnittstelle java.util.Set (Auszug)

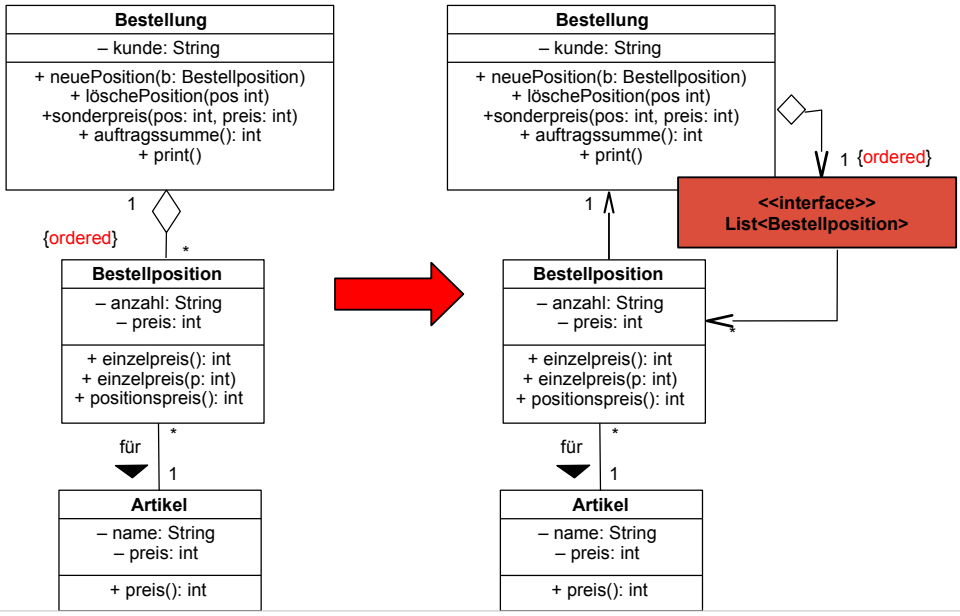
- Spezifische Schnittstellen sowie konkrete Implementierungen in der Collection-Hierarchie können spezialisiertes Verhalten aufweisen.
- Bsp.: Was ändert sich bei Set im Vergleich zu List in der Semantik von `add( )`?



Set besitzt zur Schnittstelle List eine unterschiedliche Semantik von “add()”, denn Mengen dürfen Elemente nur einmal enthalten (Duplikate werden eliminiert).

Verfeinern von bidir. Assoziationen durch getypte Behälterklassen

Ersetzen von "*"-Assoziationen durch Behälterklassen



Wdh. Einfache Realisierung mit Schnittstellen-Klasse “List” und Implementierungsklasse LinkedList

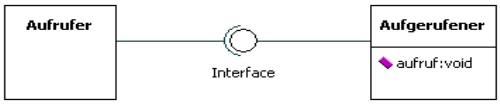

```
class Bestellung {
  private String kunde;
  private List<Bestellposition> liste;
  private int anzahl = 0;

  public Bestellung(String kunde) {
    this.kunde = kunde;
    liste = new LinkedList<Bestellposition>();
    // Konkrete Implementierungsklasse
  }
  public void neuePosition (Bestellposition b) {
     liste.set(anzahl,b);
    anzahl++;    // was passiert jetzt bei mehr als 20 Positionen ?
  }
  public void loeschePosition (int pos) {
       liste.remove(pos);
  }
  public void sonderpreis (int pos, int preis) {
    liste.get(pos).einzelpreis(preis);
  }
  public int auftragssumme() {
    int s = 0;
    for(int i=0; i<anzahl; i++) s += liste.get(i).positionspreis();
    return s;
  }
}
```

Beachte Konstruktor!

Online:
Bestellung1LinkedList.java

21.3 Programmieren gegen Schnittstellen von polymorphen Containern

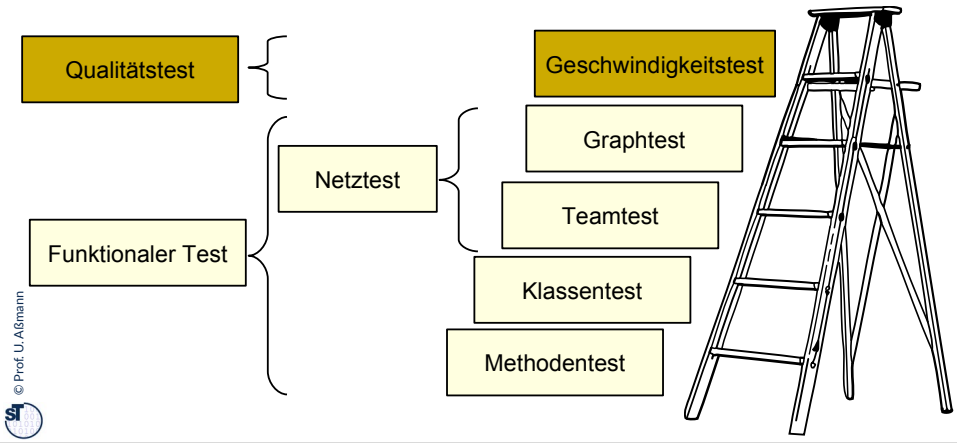
"Der Aufrufer programmiert *gegen* die Schnittstelle,
er befindet sich sozusagen im luftleeren Raum."
Siedersleben/Denert,
Wie baut man Informationssysteme,
Informatik-Spektrum, August 2000



DRESDEN concept
Exzellenz aus Wissenschaft und Kultur

Geschwindigkeitstests (Performance testing)

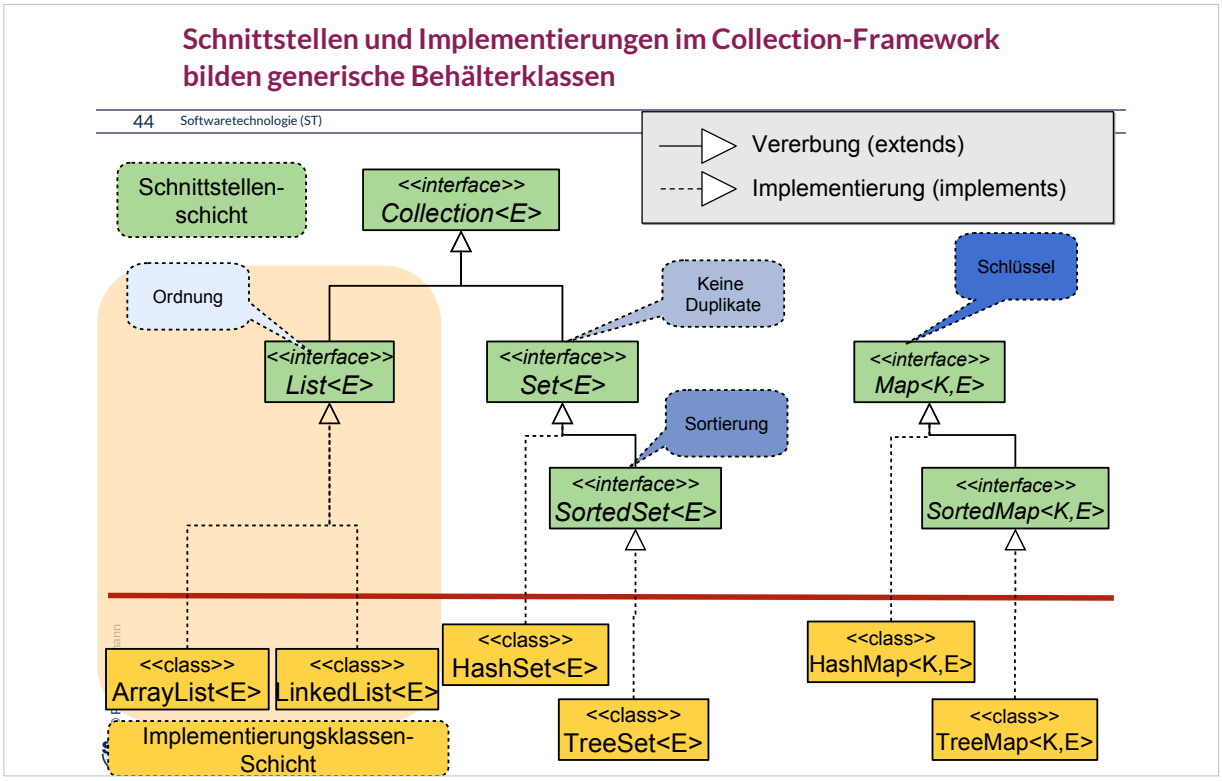
- Geschwindigkeit ist keine Funktionalität, sondern eine Qualität des Programms
- Die Geschwindigkeit eines Programms hängt wesentlich von der Implementierung des Objektnetzes und der Netzdatenstrukturen ab (Teams, Graphen)
- Geschwindigkeitstests werden mit ***Benchmarks*** durchgeführt
- Schnittstellen erlauben den Austausch von Implementierungen zur Beschleunigung



Programmieren gegen Schnittstellen hat verschiedene Vorteile:

•Nutzung für Geschwindigkeitstests: Oft ist nicht klar, mit welchen Implementierungsklassen für ein Objektnetz ein Programm am schnellsten läuft. Ein Geschwindigkeitstest kann daher verschiedene Varianten für die Implementierung des Netzes angeben und die beste auswählen.

•Wiederverwendung: Man kann Pakete für Objektnetze wiederverwenden und in anderen Wiederverwendungskontesten mit anderen Implementierungen versehen.



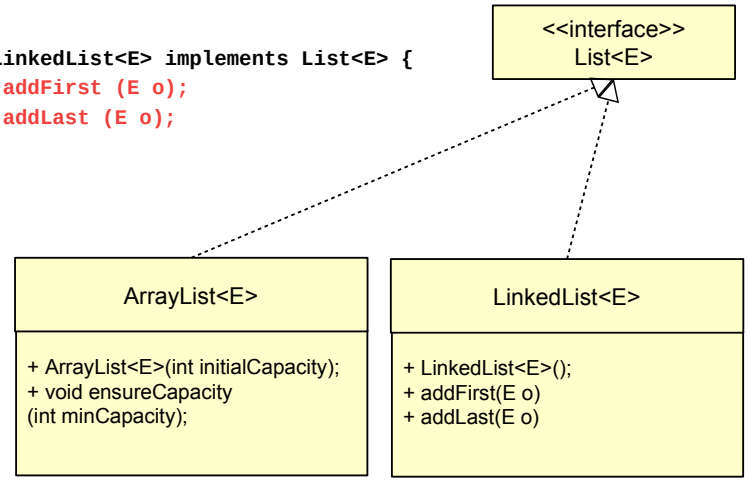
Wir lernen am Beispiel von ArrayList und LinkedList, warum deren Implementierungen bezgl. unterschiedlicher Nutzungsprofile unterschiedliche Geschwindigkeiten bieten.

Polymorphie – zwischen abstrakten und konkreten Datentypen

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

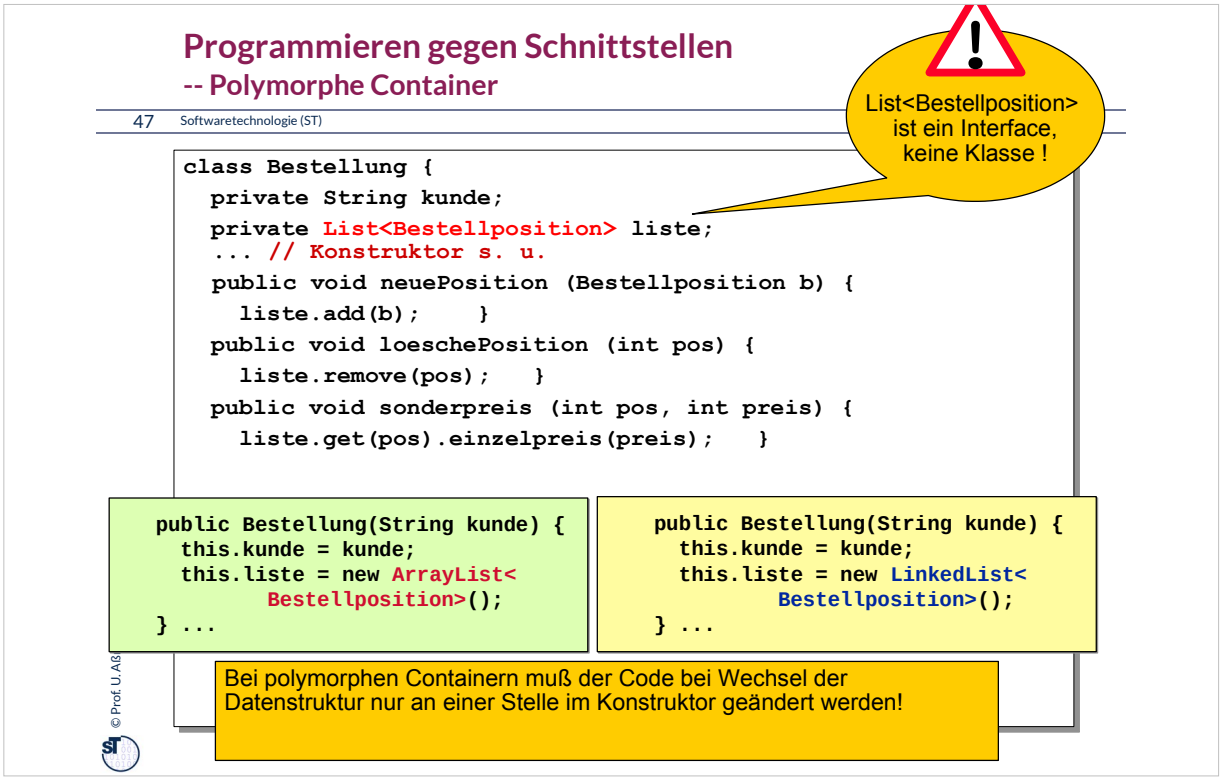
Beispiel: Implementierungsklassen java.util.ArrayList, LinkedList


```
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);
  ...
}
```



Polymorphe Containerklassen erlauben das Programmieren gegen Schnittstellen der gemeinsamen Oberklasse.

Polymorphe generische Containerklassen sind zusätzlich typsicher, was die Verwendung von Elementen betrifft.



Bei diesem Beispiel von generischen Behälterklassen sieht man

- die Typdefinition erlaubt die polymorphe Verwendung
- die Allokationen in den Konstruktoren der konkreten Implementierungsklassen geben die spezifische Implementierungsmethode an

Wdh. Einfache Realisierung mit Schnittstellen-Klasse “List” und Implementierungsklasse LinkedList


```
class Bestellung {
  private String kunde;
  private List<Bestellposition> liste;
  private int anzahl = 0;

  public Bestellung(String kunde) {
    this.kunde = kunde;
    liste = new ArrayList<Bestellposition>();
    // Konkrete Implementierungsklasse
  }
  public void neuePosition (Bestellposition b) {
     liste.set(anzahl,b);
    anzahl++;    // was passiert jetzt bei mehr als 20 Positionen ?
  }
  public void loeschePosition (int pos) {
       liste.remove(pos);
  }
  public void sonderpreis (int pos, int preis) {
    liste.get(pos).einzelpreis(preis);
  }
  public int auftragssumme() {
    int s = 0;
    for(int i=0; i<anzahl; i++) s += liste.get(i).positionspreis();
    return s;
  }
}
```

Beachte Konstruktor!

Online:
Bestellung1ArrayList.java

Welche Listen-Implementierung soll man wählen, um das Programm schnell und ressourcenschonend zu machen?

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

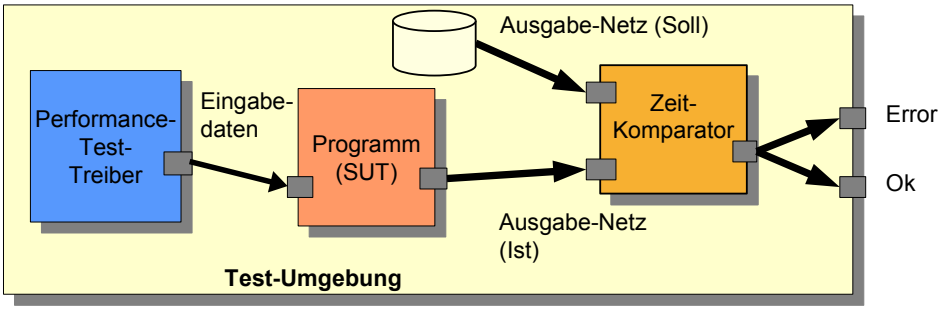
Typ	Lesen	Iteration	Einfügen	Entfernen
array	1430	3850	--	--
ArrayList	3070	12200	500	46850
LinkedList	16320	9110	110	60
Vector	4890	16250	550	46850

- 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

Performance-Tests für polymorphe Objektnetze

- Performance-Test misst die Geschwindigkeit der Implementierung eines Programms
- Die Varianten Implementierungen von Behälterklassen können einfach verglichen werden

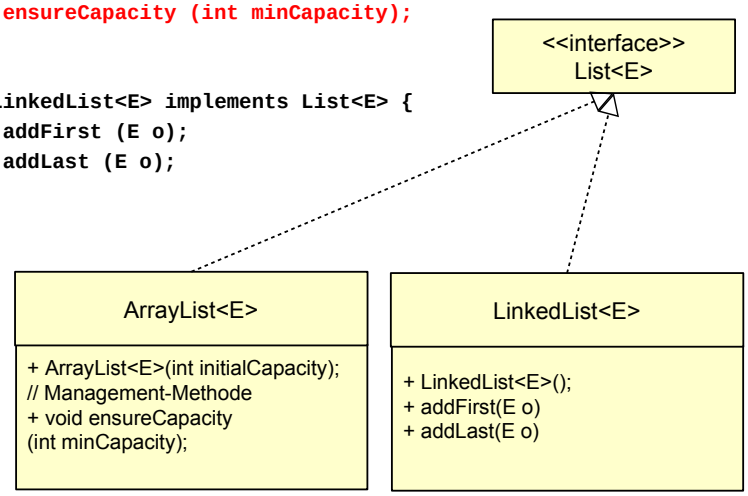
Solange ein Programm keine Performance-Tests hat, ist es keine Software



21.4. Weitere Arten von Klassen und Methoden

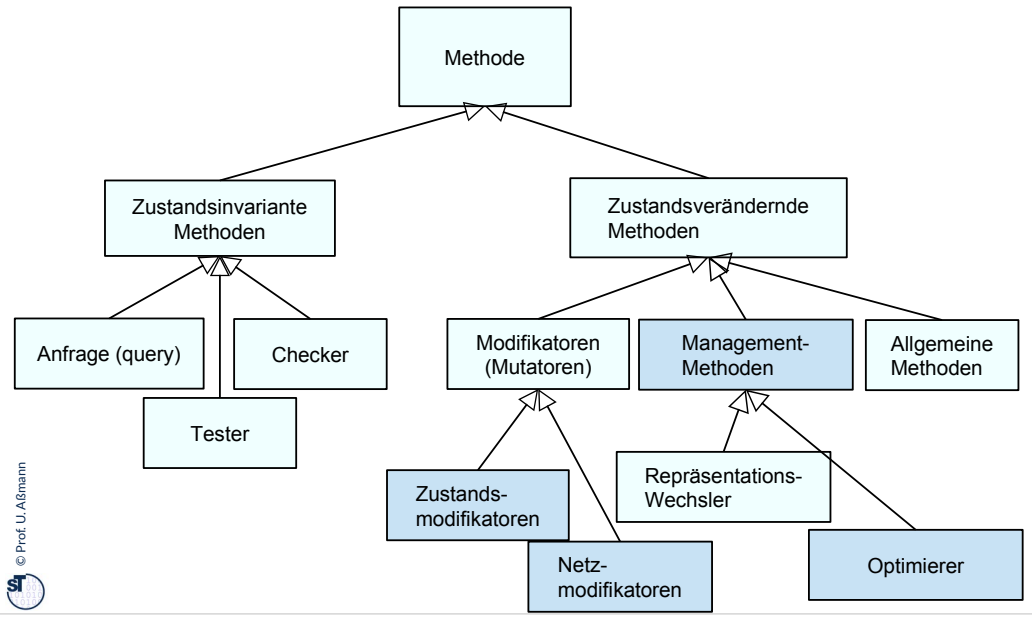
Management-Methoden in java.util.ArrayList, LinkedList


```
public class ArrayList<E> implements List<E> {
  public ArrayList<E> (int initialCapacity);
  // Management-Methode (Optimierer-Methode)
  public void ensureCapacity (int minCapacity);
  ...
}
public class LinkedList<E> implements List<E> {
  public void addFirst (E o);
  public void addLast (E o);
  ...
}
```



Management-Methoden bilden eine neue Kategorie von Methoden: sie erlauben es, Laufzeitparameter ihrer Datentypen zu verändern und zu optimieren.

Erweiterung Q3: Taxonomie der Methodenarten



Management-Methoden erweitern unsere Taxonomie der Methodenarten. Wir setzen eine Mittelklasse ein (Refactoring der Taxonomie).

- **Optimierer**-Methoden versuchen, Parameter der Implementierungsklassen zu verändern
- **Zustands-** können von **Netzmodifikatoren** unterschieden werden. (Ähnliches gilt für Anfrage-, Checker- und Tester-Methoden).

Standardalgorithmen in der *Algorithmenklasse* java.util.Collections

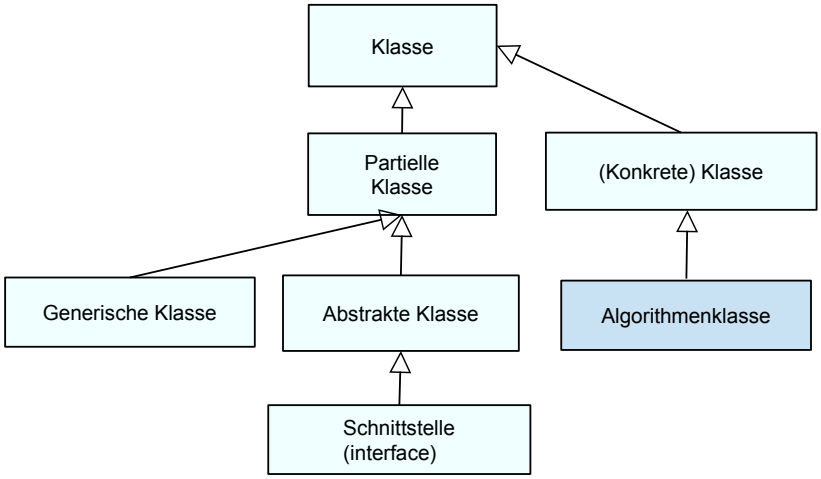
Def.: **Algorithmenklassen (Hilfsklassen)** enthalten Algorithmen, die auf einer Familie von anderen Klassen arbeiten

```
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

- **Algorithmenklassen** sind Hilfsklassen, die Algorithmen enthalten, 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!

Erweiterung Q2: Begriffshierarchie von Klassen



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);
}
```

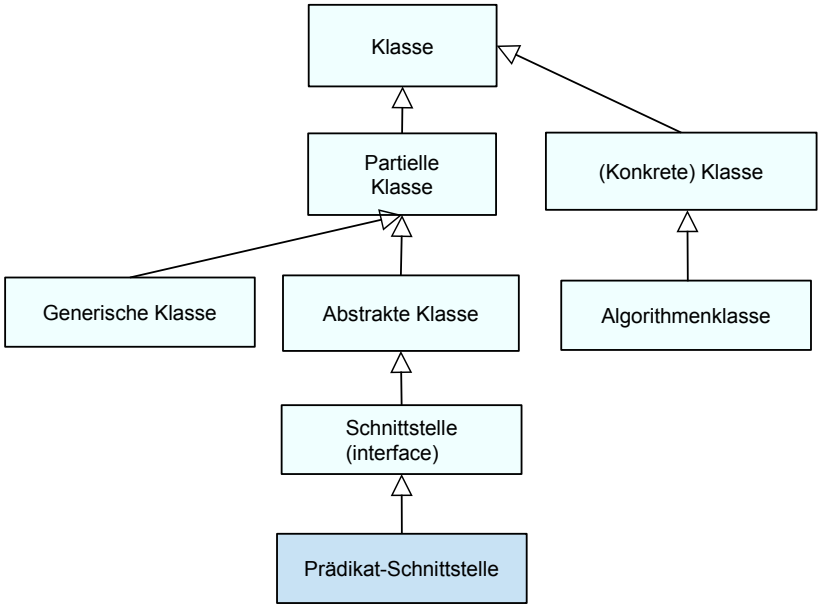
- Resultat ist kleiner/gleich/größer 0:
  genau dann wenn "this" kleiner/gleich/größer als Objekt o

Typschranken generischer Parameter (type bounds), mit Prädikatsschnittstellen

- Prädikatschnittstellen können für einen Typparameter einfache Prädikate ausdrücken
- Beispiel: Comparable<E> als Return-typ in der Collections-Klasse sichert zu, dass die Methode compareTo() existiert

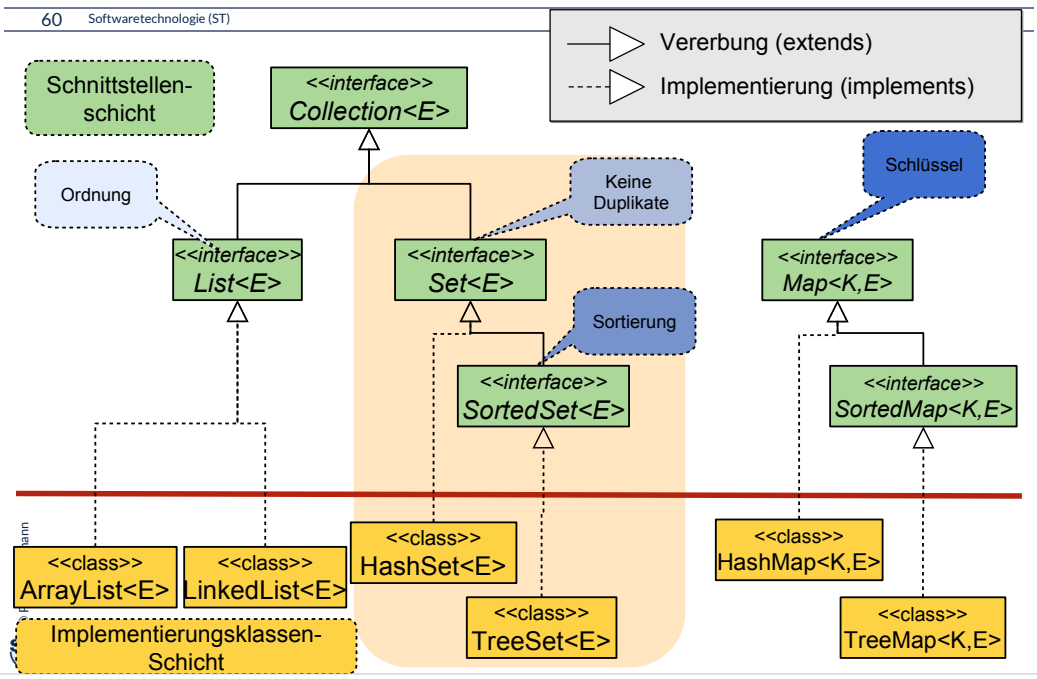
```
class Collections {
/** minimum function for a Collection. Return value is typed
  * with a generic type with a type bound */

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;
}
```

Erweiterung Q2: Begriffshierarchie von Klassen



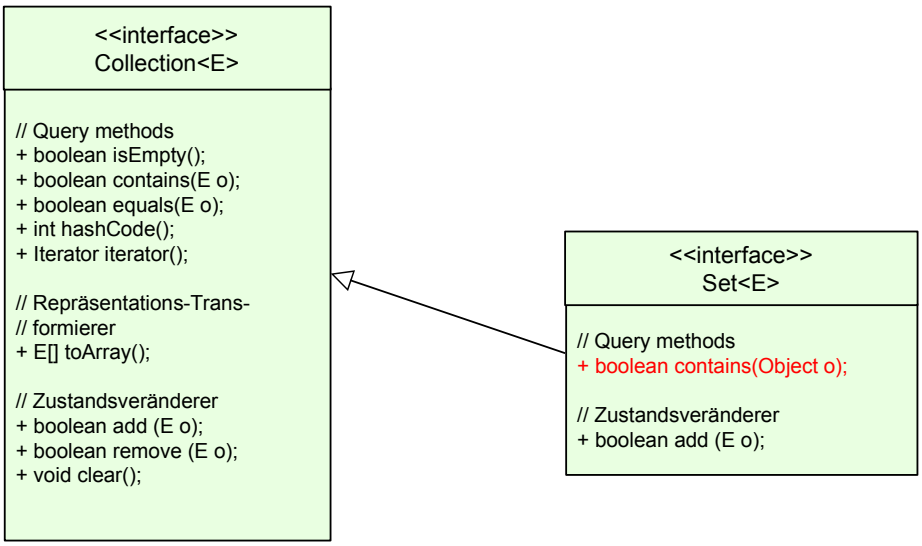
21.5 Ungeordnete Collections mit Set

Schnittstellen und Implementierungen im Collection-Framework bilden generische Behälterklassen



Im Folgenden betrachten wir die unterschiedlichen Implementierungen von Set.

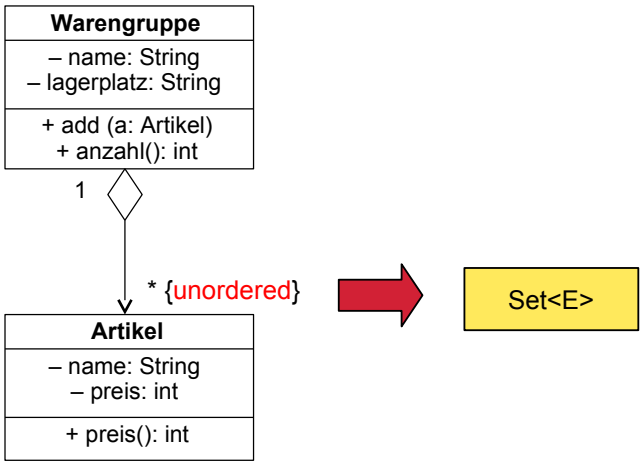
Ungeordnete Mengen: java.util.Set<E> (Auszug)



Bei Mengen dürfen Elemente nur einmal eingetragen sein; daher ruft add() zunächst contains() auf und fügt erst dann ein, wenn das Element noch nicht enthalten ist.

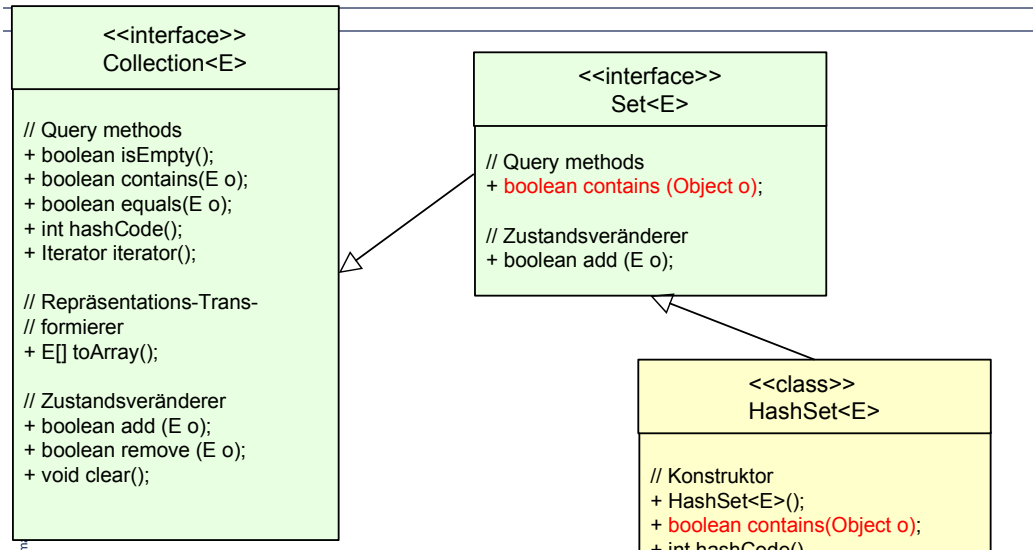
Anwendungsbeispiel für Set<E>

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



Hier sieht man den Einsatz von “tagged values” in UML: man kann die genaue Art von Collection angeben, die benötigt ist.

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

21.5.2 Re-Definition der Gleichheit von Elmenten in Set

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

- Die Operation **==** prüft auf *Referenz- bzw. Pointer-Gleichheit*, d.h. physische Identität der Objekte in der Halde
  - 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
    - String s.equals(String r) prüft auf *strukturelle Gleichheit*, d.h. auf eine gleiche Folge von Zeichen
  - für selbstdefinierte Klassen genau überlegen, was man möchte:
    - Standardbedeutung Referenzgleichheit
    - oder strukturelle Gleichheit
  - 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:

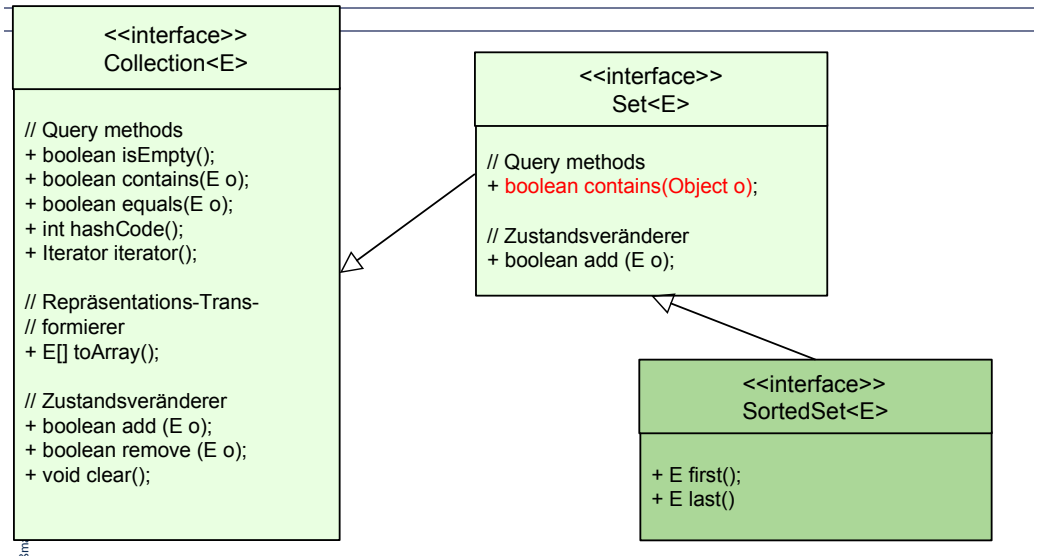
```
Warengruppe Moebel
Schrank(300) Tisch(200) Stuhl(100)
```

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>`
- Modifikation 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.TreeSet<E>.compareTo(TreeSet<E>.java, Compiled Code)

corrected

- 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)

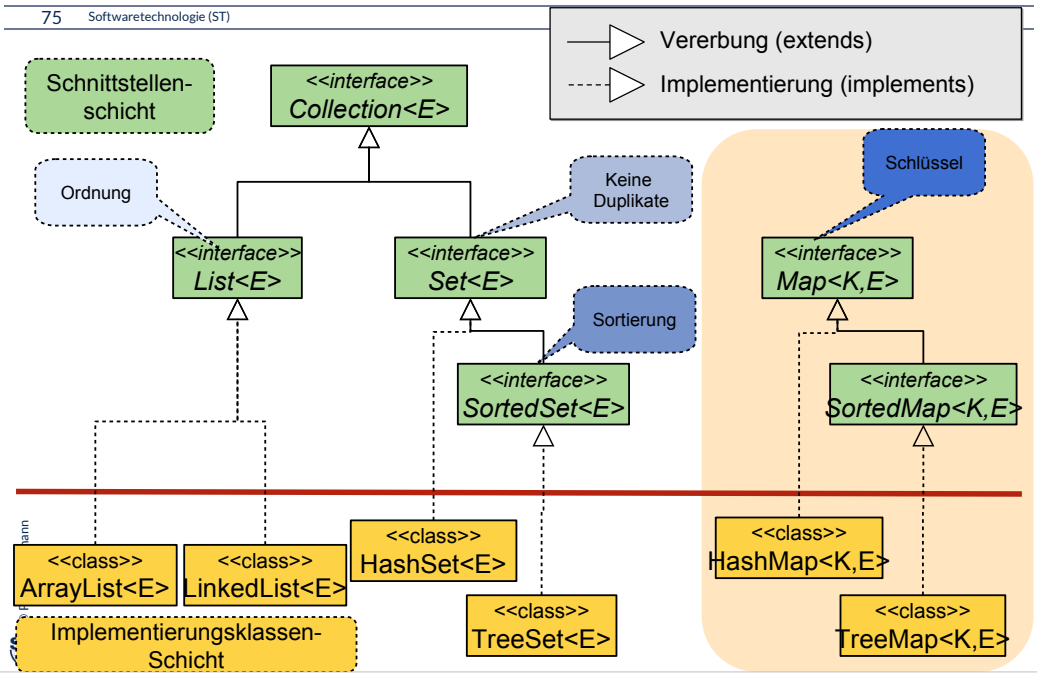
Typ	Einfügen	Enthalten	Iteration
HashSet	36,14	106,5	39,39
TreeSet	150,6	177,4	40,04

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

21.6 Kataloge mit Map

- Ein **Katalog (Wörterbuch, dictionary, map)** ist eine Abbildung eines Schlüssel-Ausgangsbereiches in einen Wertebereich.
- Achtung: Im Collection-Hierarchie der Map wird “Element” als “Value” bezeichnet

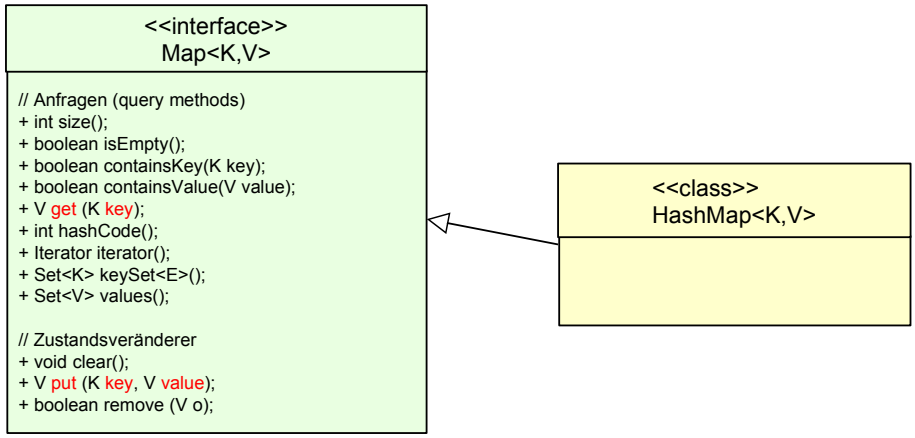
Schnittstellen und Implementierungen im Collection-Framework bilden generische Behälterklassen



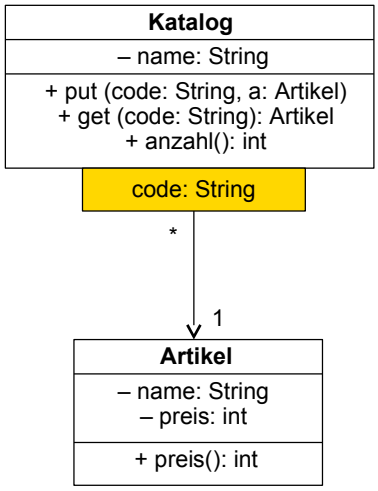
Im Folgenden schauen wir uns Maps (Kataloge, Dictionary) an.

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
- Achtung: wir nennen nun im Folgenden “E – Element” “V – Value”, wie in der JCF



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

Nachbarmengen mit Schlüsseln heißen in UML *qualifizierte Assoziationen.*

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 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:

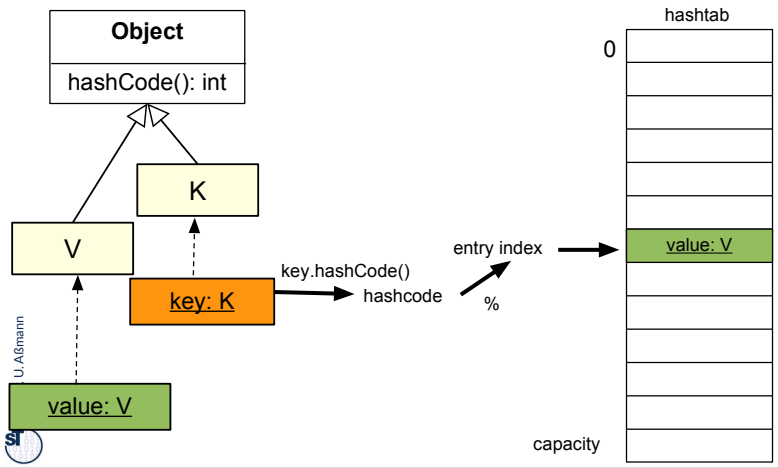
```
Katalog Katalog1
 M03 -> Schrank(300)
 M02 -> Stuhl(100)
 M01 -> Tisch(200)

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

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

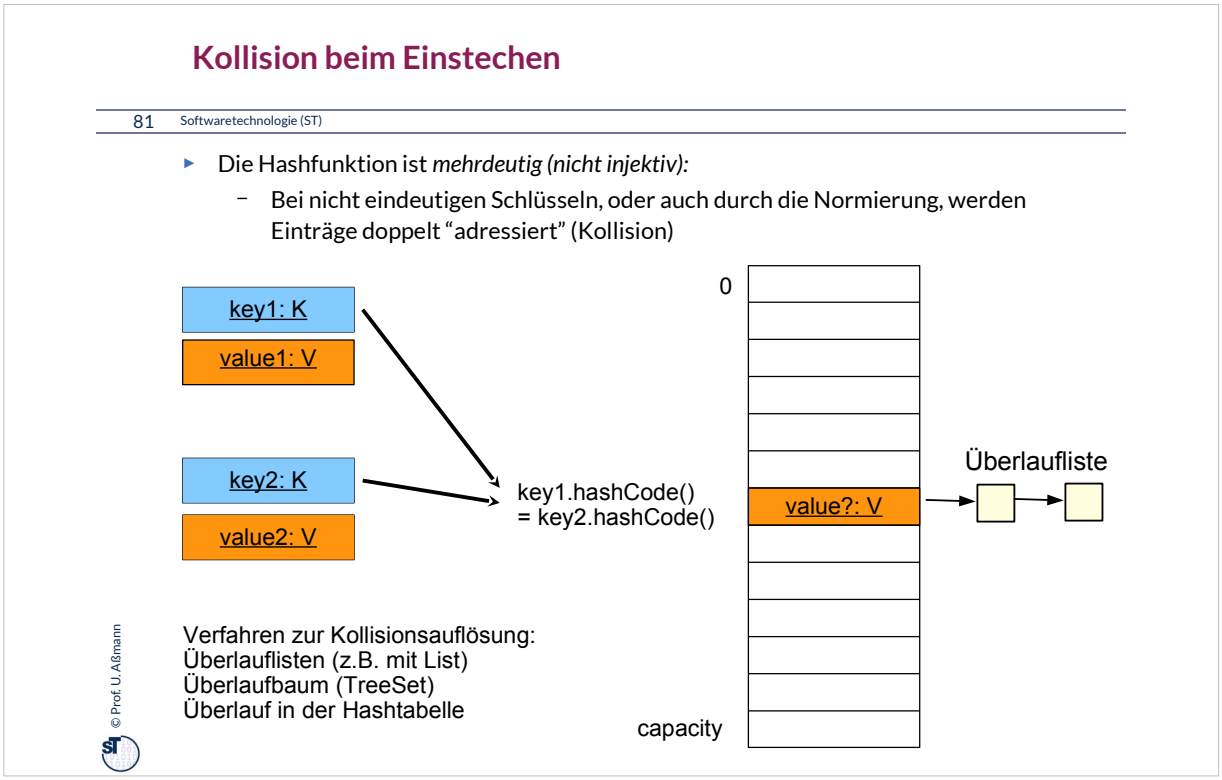
Ordnung auf den Schlüsseln: SortedMap (Implementierung z.B.TreeMap).

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



Hashen ist eine der wichtigsten Operationen der Softwaretechnik, denn der Zugriff von Schlüssel auf Wert kann in nahezu konstanter Zeit realisiert werden, wenn hashCode und Hashtabelle auf einander gut abgestimmt sind.



Möglichkeiten zur Kollisionsauflösung:

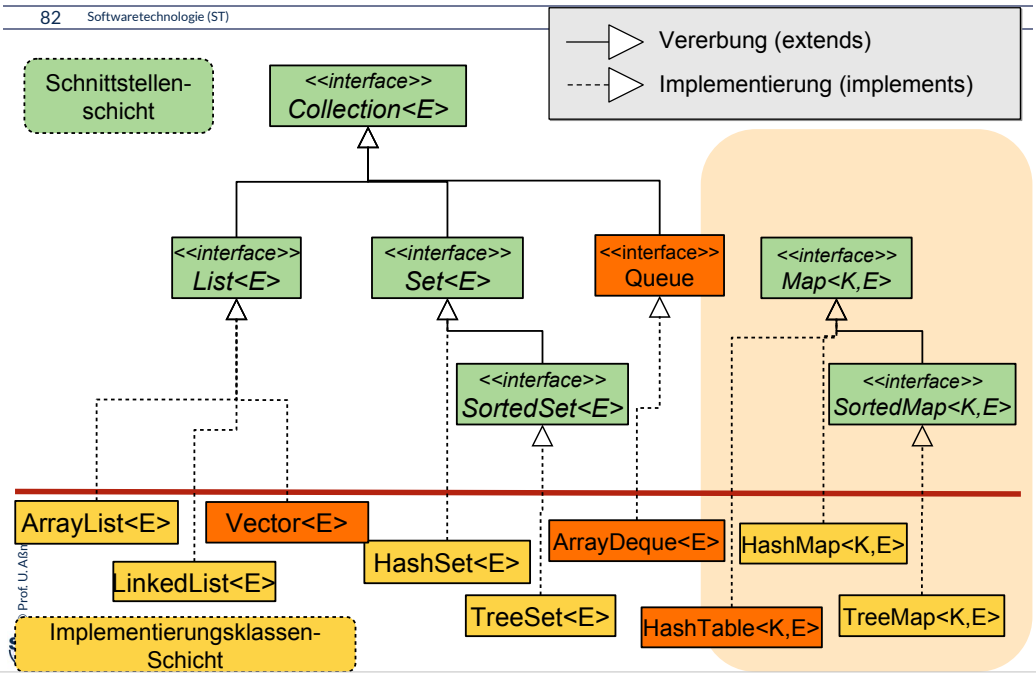
- Verwendung des nächsten Felds der Tabelle, iterativ
- dito, mit steigender Schrittweite bei Iteration
- Verwendung einer „Überlaufliste“

Generell ist es wünschenswert, möglichst wenig Kollisionen zu haben. Dies kann z.B. durch eine geschickt gewählte, weit streuende Hashfunktion und ein günstiges Verhältnis Tabellengröße:Datenmenge erreicht werden.

Beispiel mit Hashtabelle für Strings und Länge als Hashfunktion.

In Java werden in den HashMap-Implementierungen spezielle Auslastungsfaktoren (load factors) berücksichtigt. Standardmäßig wird eine HashMap ab einer Auslastung von 75% automatisch neu organisiert.

Weitere Schnittstellen und Implementierungen im JCF



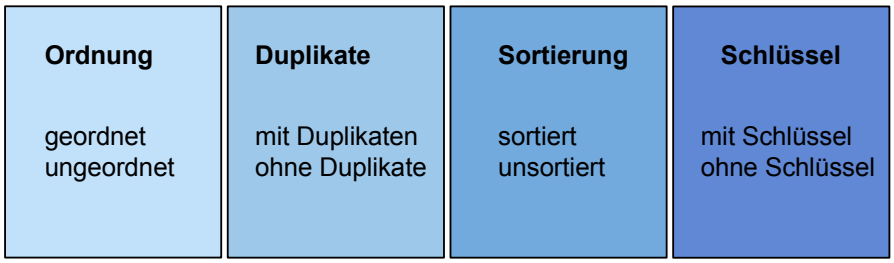
21.7 Optimierte Auswahl von Implementierungen von Datenstrukturen

- zur Implementierung verschiedener Arten von UML Assoziationen

Ermittlung der benötigten Assoziationsarten und Facetten der Implementierungsklassen

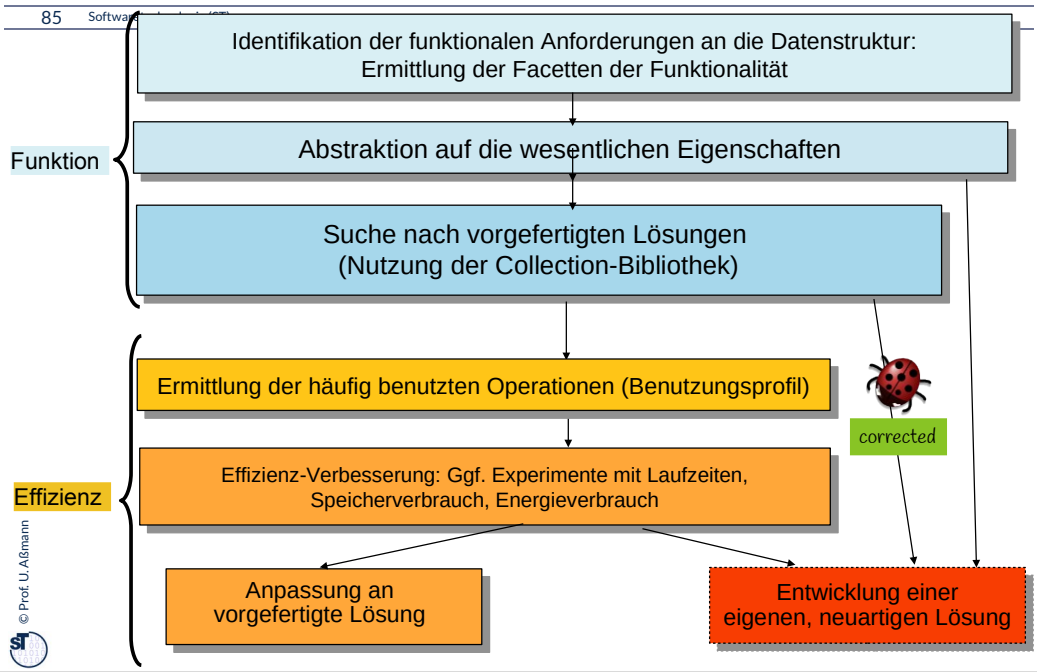
- An das Ende einer UML-Assoziation können folgende *Bedingungen* notiert werden:
  - Ordnung: {ordered} {unordered}
  - Eindeutigkeit: {unique} {non-unique}
  - Kollektionsart: {set} {bag} {sequence}

- Beim Übergang zum Implementierungsmodell müssen diese Bedingungen auf Unterklassen von Collections abgebildet werden



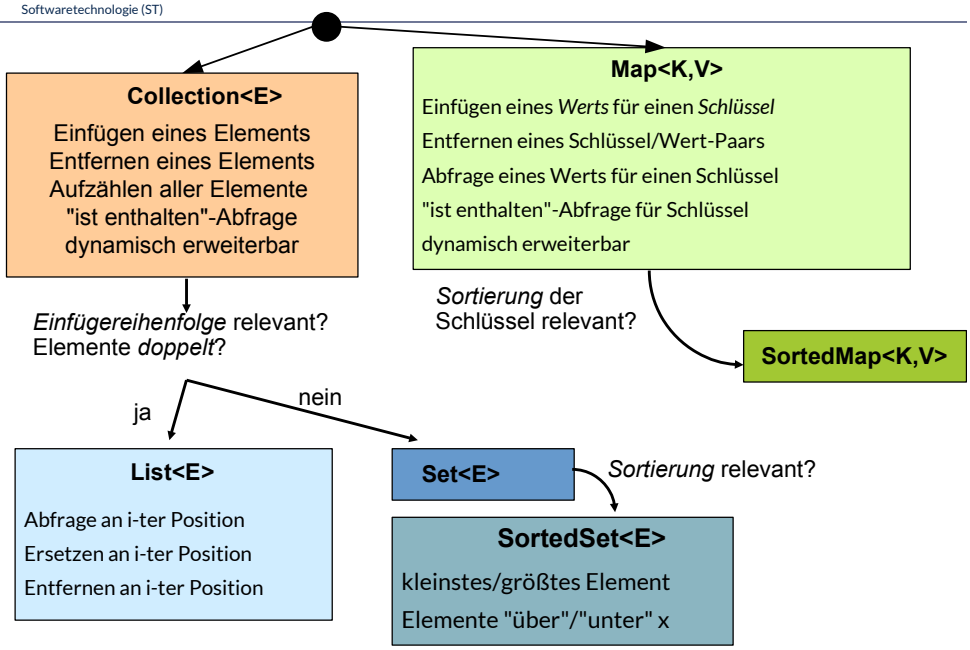
Die Facetten einer Collection können in UML durch “tagged values” notiert werden.

Vorgehensweise beim funktionalen und effizienzbasierten Datenstruktur-Entwurf (Ermittlung Benutzungsprofil)



Für die Entscheidung, welche Implementierung einer Datenstruktur ausgewählt wird, gibt es eine strukturierte Vorgehensweise (“Netz-Verfeinerung”).

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

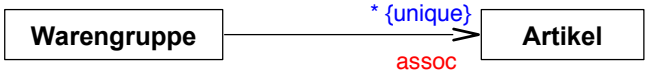


Beispiel: Realisierung von unidirektionalen Assoziationen

Datenstruktur im Warengruppe-Objekt für Artikel-Referenzen

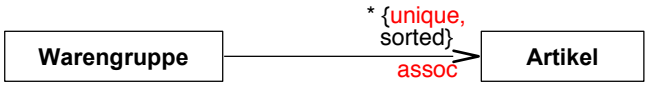
Anforderung	Realisierung
1) Assoziation anlegen	1) Einfügen (ohne Reihenfolge)
2) Assoziation entfernen	2) Entfernen (ohne Reihenfolge)
3) Durchlaufen aller bestehenden Assoziationen zu Artikel-Objekten	3) Aufzählen aller Elemente
4) Manchmal: Abfrage, ob Assoziation zu einem Artikel-Objekt besteht	4)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

Beispiel:
Realisierung von ungeordneten Assoziationen mit Set<E>




```
class Warengruppe {
  private Set<Artikel> assoc;
  ...
  public void addAssoc (Artikel ziel) {
    assoc.add(ziel);
  }

  public boolean testAssoc (Artikel ziel) {
    return assoc.contains(ziel);
  }

  public Warengruppe {
    assoc = new HashSet<Artikel>();
  }
}
```

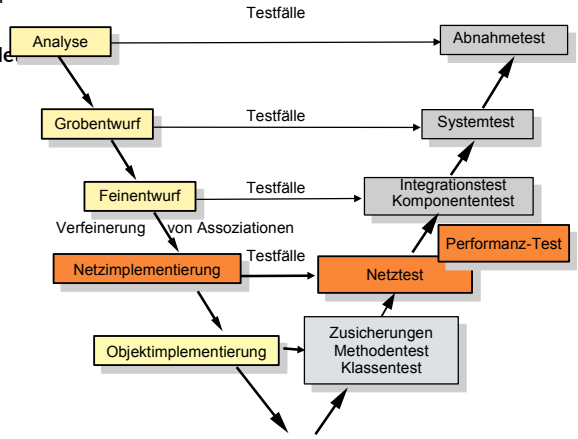
Beispiel:
Realisierung von sortierten Assoziationen mit Set<E>




```
class Warengruppe {
  private SortedSet<Artikel> assoc;
  ...
  public void addAssoc (Artikel ziel) {
    assoc.add(ziel);
  }

  public boolean testAssoc (Artikel ziel) {
    return assoc.contains(ziel);
  }

  public Warengruppe {
    assoc = new TreeSet<Artikel>();
  }
}
```

Ziel: 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 Ne entwerfen**
  - Dann die einzelne Klasse testen
  - **Dann das Netz testen**
  - Dann die Komponente
  - Dann das System
  - Dann der beta-Test
  - Dann der Geschwindigkeitstest
  - Zum Schluss der Akzeptanztest (Abnahmetest)



Was haben wir gelernt

- Wann wende ich welchen Entwicklungsprozess, SAD oder RAD, an?
  - Unterscheide statische vs. dynamische vs. keine Typisierung
  - Safe Application Development (SAD) ist nur mit statischen Typisierung möglich
  - Rapid Application Development (RAD) benötigt dynamische Typisierung
- Testen:
  - Test von Objektnetzen ist wichtig für die Qualität von Software
  - Performance-Test von Objektnetzen ist einfach mit Schnittstellen und verschiedenen Implementierungen (z.B. polymorphen Behälterklassen)
- Generische Collections besitzen den Element-Typ als Typ-Parameter
  - Element-Typ verfeinert Object
  - Weniger Casts, mehr Typsicherheit
- Das Java Collection Framework (JCF) bietet geordnete, ungeordnete Collections sowie Kataloge

The End

- Diese Folien bauen auf der Vorlesung Softwaretechnologie auf von © Prof. H. Hussmann, 2002. Used by permission.
- Warum ist das Lesen in einer ArrayList I.d.R. schneller als in der LinkedList?
- Warum ist das Löschen auf Index 0 in der ArrayList langsamer als in der LinkedList?
- Erklären Sie den Unterschied der 4 Facetten der Collections
- TreeSet verwendet eine baumartige Datenstruktur. Erklären Sie die Vorteile eines Baums für den Insert und das Suchen von Elementen.
- Warum sollte man sich in der Anforderungsanalyse mit aUML um die tagged values von Multiplizitäten kümmern?
- Wieso ist die Hörsaalübung wichtig?
- Welche Rolle spielen Prädikat-Schnittstellenklassen im JCF? Erklären Sie die Comparable-Schnittstelle.

Appendix A
Generische Command Objekte

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

Implementierungsmuster Command: Generische Methoden als Funktionale Objekte

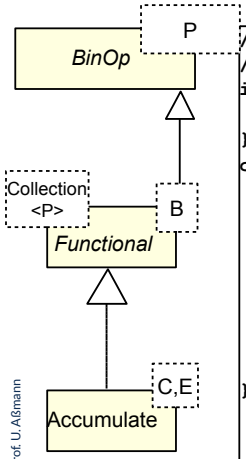
Ein **Funktionalobjekt (Kommandoobjekt)** ist ein Objekt, das eine Funktion darstellt (reifiziert).

- **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

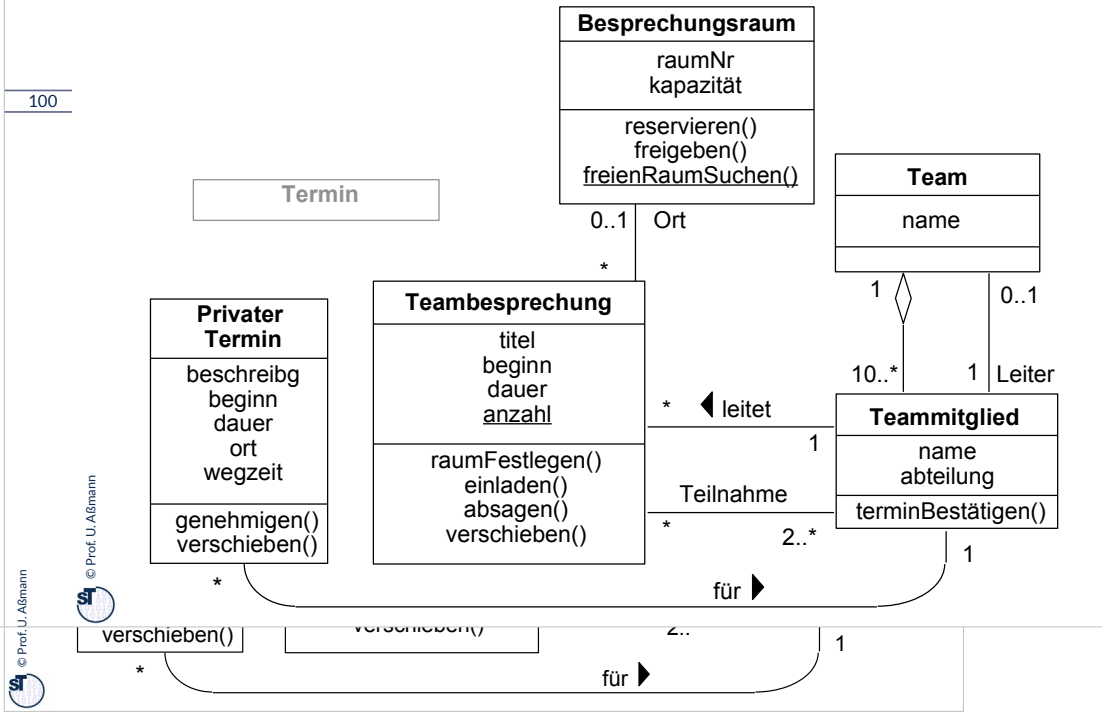
- Anwendung: Akkumulatoren und andere generische Listenoperationen




```
// 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;
  }
}
```

Appendix B
Bestimmung von konkreten Datentypen

Beispiel 2: Analysemodell der Terminverwaltung



Operationen werden in der Vorlesung interaktiv eingetragen.

Beispiel 2: Sortierte Liste von Räumen in der 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)

Online: TerminvTreeSet.java

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;
  } ...
}
```