

# Programmieren 3

## Kapitel 5: Anwendungsprogrammierung

- ▶ Software-Kategorien
- ▶ A-Komponenten
- ▶ Datentypen
- ▶ Enumerationen
- ▶ Entitätstypen
- ▶ enge vs. lose Kopplung
- ▶ Sichten und Abhängigkeiten





# Der rote Faden: Wie implementiere ich große Softwaresysteme?



Klassen und  
Schnittstellen

Wie kann ich von (techn.) Details abstrahieren?  
Wie kann ich die Implementierungen austauschen?  
Wie bringt man Implementierung und Schnittstelle zusammen?

Komponenten  
und  
Schnittstellen

Was sind Komponenten?  
Wie kommunizieren Komponenten?  
Wie strukturiert man Komponenten?

Anwendungs-  
programmierung

Wie findet man Komponente?  
Wie implementiert man Datenmodell/Anwendungsfälle?  
Was ist enge bzw. lose Kopplung?

# Software Kategorien – SW-Blutgruppen

## Software ist ...

0	bestimmt durch nichts (container, strings) <i>ideal wieder verwendbar, steht überall zur Verfügung</i>
A	bestimmt durch die Anwendung (Kunde, Bestellung, Lieferung) <i>das ist das eigentliche System</i>
T	bestimmt durch mindestens eine technische API (z.B. Datei-System) <i>ohne T geht es nicht</i>
AT	bestimmt durch die Anwendung und mindestens eine technische API <i>sollte vermieden oder zumindest sorgfältig getrennt werden</i>
R	Repräsentations-Software (Transformation zwischen A und T; einfache Variante von AT; idealer Kandidat für Generatoren)

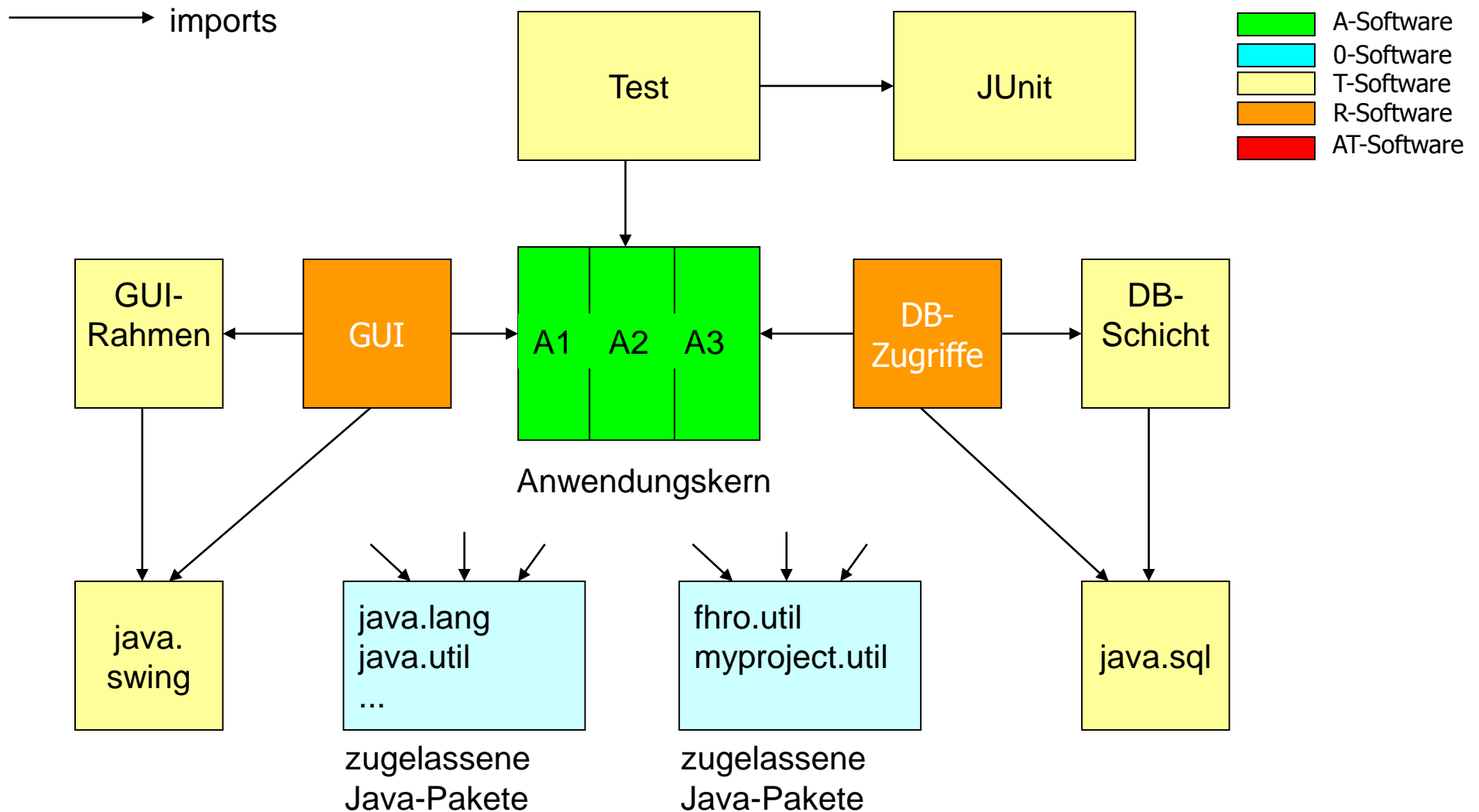
### Kombination von Kategorien

$$A + 0 = A$$

$$T + 0 = T$$

$$A + T = AT$$

# Paketstruktur eines Informationssystems



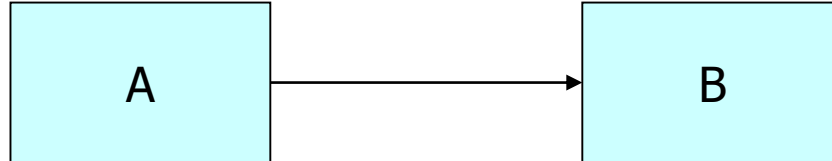


# Der Anwendungskern

- ▶ Anwendungskern: A-Kategorien eines Systems
- ▶ Anwendungskern macht wenig Annahmen zu seinen Aufrufern
  - ▶ prüft alle Eingaben
  - ▶ offen für verschiedene Ausnahmebehandlungen
  - ▶ offen für verschiedene Transaktionsstrategien
- ▶ Anwendungskern bestimmt durch
  - ▶ Datenmodell (fachlich), bestehend aus
    - ▶ Entitätstypen
    - ▶ Zugehörigen Datentypen
  - ▶ Anwendungsfälle

## ▶ Anwendungskomponenten / A-Komponenten

- ▶ Jede A-Komponente verwaltet eine oder mehrere Entitätstypen; jeder Entitätstyp gehört zu genau einer Komponente (**Datenhoheit**).
- ▶ A-Komponente implementiert die zu den Daten zugehörigen Anwendungsfälle und exportiert diese über Schnittstellen
- ▶ Komponenten werden nur über Schnittstellen angesprochen (z.B. Java Interfaces)
- ▶ Braucht-Beziehung (A requires B) = enge Koppelung



Um A zu übersetzen, braucht man das Interface von B.  
Um A zu testen, braucht man eine (Dummy)-Implementierung von B.

- ▶ Das Komponentendiagramm zeigt die Braucht-Beziehungen VOLLSTÄNDIG.



# Wie mache ich eine Komponente?

- ▶ **Schritt 1:** Informationen zu Anforderungen
- ▶ **Schritt 2:** Spezifikation des Datenmodells (aus Pflichtenheft / Anforderungsspezifikation)
- ▶ **Schritt 3:** Komponentenschnitt durch Schnitt durch das Datenmodell (Berücksichtigung von Abhängigkeiten)
- ▶ **Schritt 4:** Definition der Schnittstellen und Entitätstypen
- ▶ **Schritt 5:** Überlegungen zur Package Struktur
- ▶ **Schritt 6:** Konfiguration zur Applikation



# Beispiel: Verwaltung von Bundesjugendspielen



## Schritt 1: Anforderungen

- ▶ Unterstützung bei der Planung von Bundesjugendspielen
  - ▶ Verwaltung von Sportstätten
  - ▶ Bildung von Zeitplänen für die verschiedenen Sportstätten, Einteilung von Teilnehmern und Betreuern
  - ▶ Informationsbasis erstellen (Schüler, Betreuer erfassen)
  
- ▶ Unterstützung nach der Durchführung
  - ▶ Erfassung der erbrachten Leistungen
  - ▶ Erstellung der Urkunden mit Hilfe fester Berechnungsformeln und des Punktespiegels





# Schritt 1: Berechnungsformeln



## Mädchen

Ausgabe 2003

Disziplin/Distanz		a	c
50 m	h+e	3.64800	0.00660
75 m	h+e	3.99800	0.00660
100 m	h+e	4.00620	0.00656
800 m		2.02320	0.00647
2 000 m		1.80000	0.00540
3 000 m		1.75000	0.00500
Hochsprung		0.88070	0.00068
Weitsprung		1.09350	0.00208
Kugelstoß		1.27900	0.00398
Schleuderball		1.08500	0.00921
200-g-Ballwurf		1.41490	0.01039
80-g-Schlagballwurf		2.02320	0.00874

## Jungen

Disziplin/Distanz		a	c
50 m	h+e	3.79000	0.00690
75 m	h+e	4.10000	0.00664
100 m	h+e	4.34100	0.00676
1 000 m		2.15800	0.00600
2 000 m		1.78400	0.00600
3 000 m		1.70000	0.00580
Hochsprung		0.84100	0.00080
Weitsprung		1.15028	0.00219
Kugelstoß		1.42500	0.00370
Schleuderball		1.59500	0.009125
200-g-Ballwurf		1.93600	0.01240
80-g-Schlagballwurf		2.80000	0.01100

Lauf (h): 
$$p = \frac{(D: (M + \text{Zuschlag})) - a}{c}$$

Die über die Formel errechneten Werte sind in jedem Falle auf ganze Zahlen abzurunden.

Lauf (e): 
$$p = \frac{(D:M) - a}{c}$$

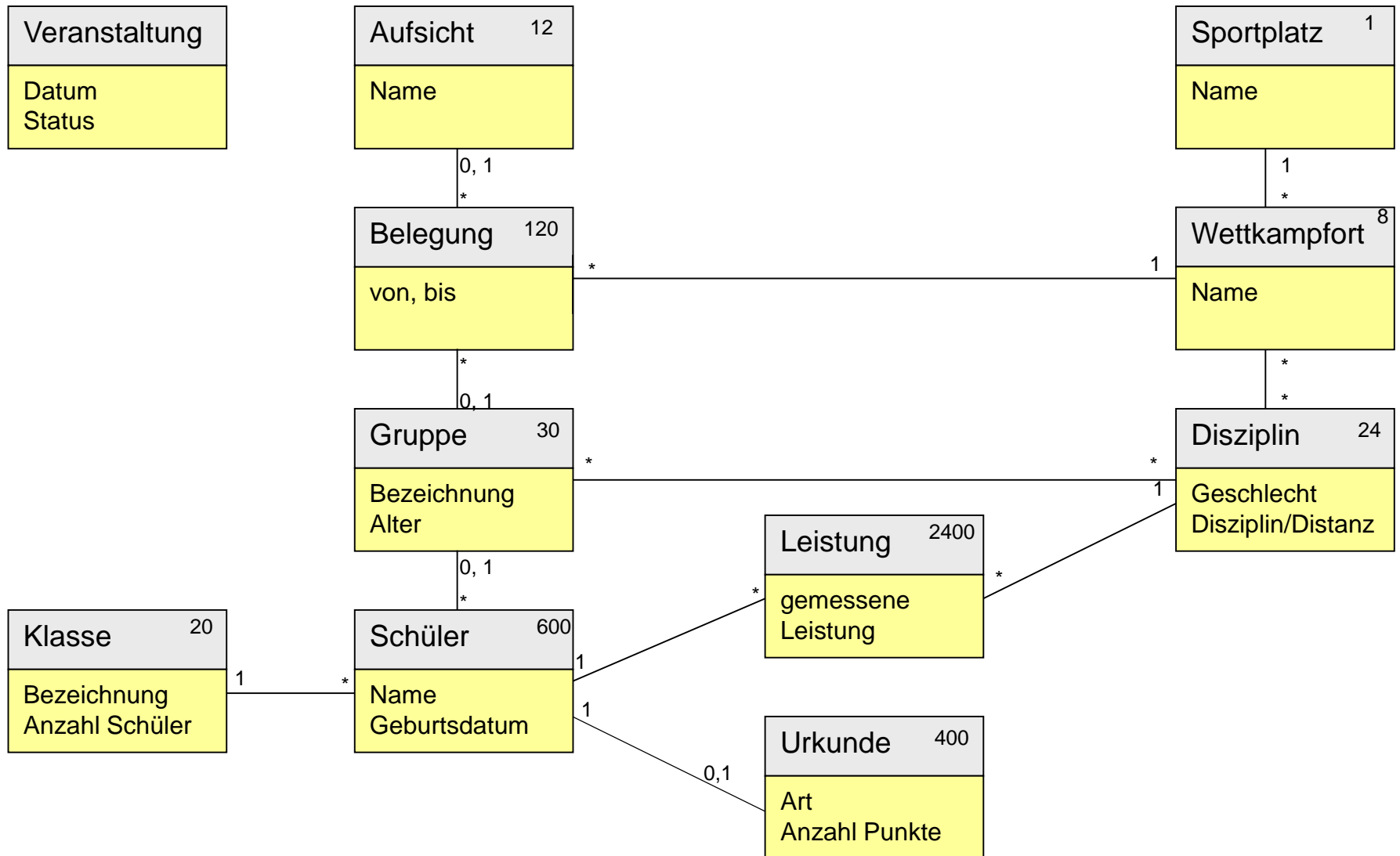
Sprung + Stoß/Wurf: 
$$p = \frac{\sqrt{M} - a}{c}$$

Erläuterung:

- h= Handzeitmessung
- e= Elektronische Zeitmessung
- p= Punkte
- D= Distanz; Laufstrecke in Metern
- Zuschlag= Strecken bis einschließlich 300 m = 0,24  
Strecken von 300 bis 400 m = 0,14  
über 400 m kein Zuschlag
- M= Messwert oder Zeiten in Sekunden,  
Leistung Höhen/Weiten in Metern umzurechnen!

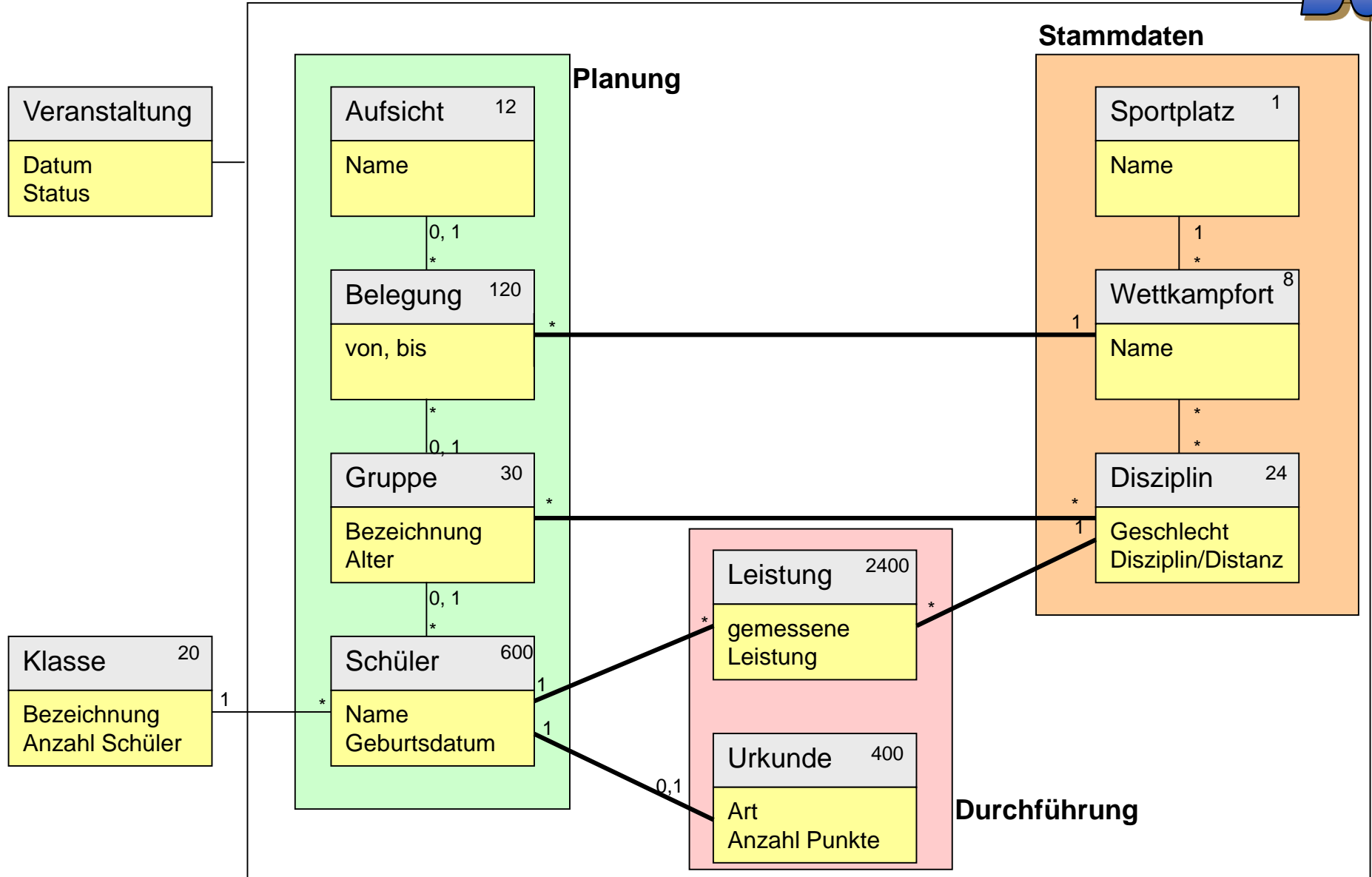


## Schritt 2: Anwendungskern BJS - Datenmodell

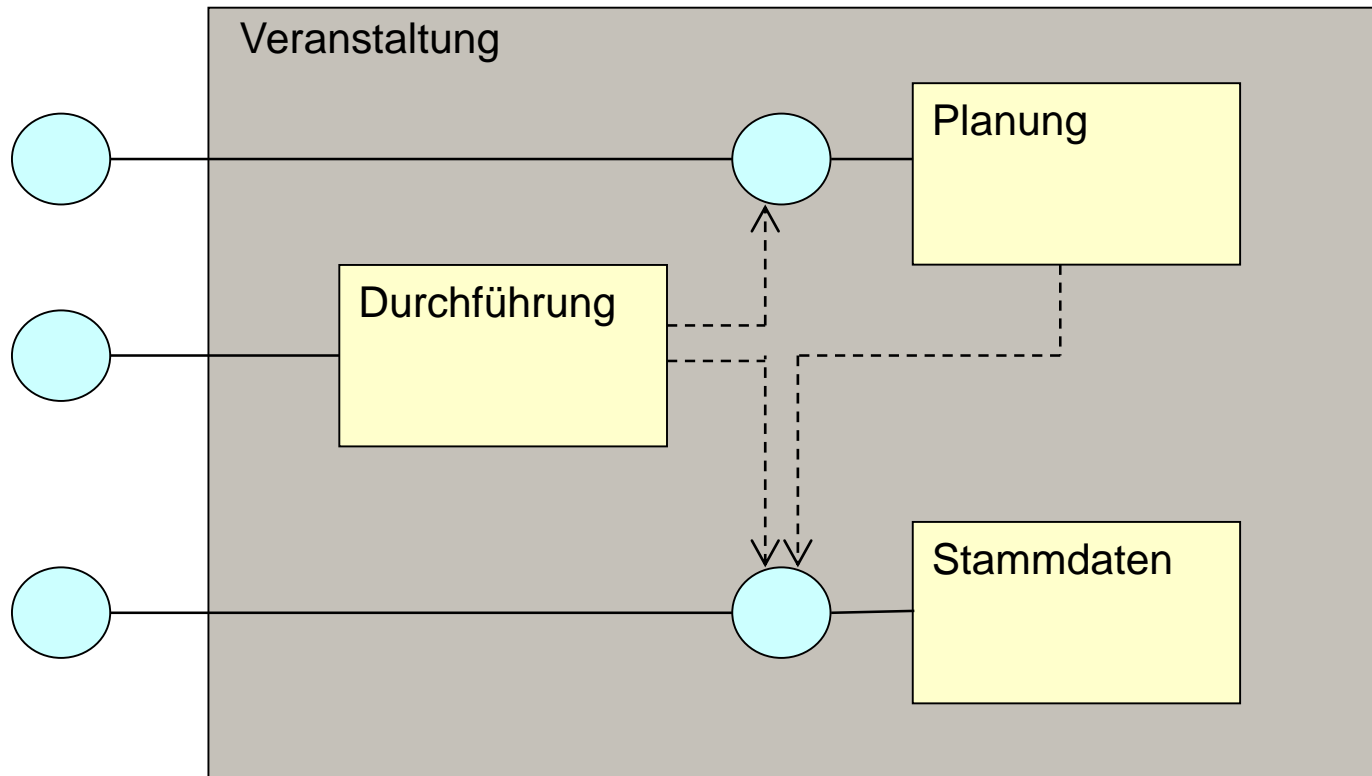




## Schritt 3: Anwendungskern BJS - Komponenten



## Schritt 3: BJS-Komponenten (Import und Export)



—— exportiert

----> importiert

## Schritt 4: Schnittstelle BJS-Stammdaten



Stammdaten

An der Schnittstelle sichtbare Klassen;  
oft Interfaces oder Maps (!)

```
class Sportplatz { .. }           // r/o  
class Wettkampfort { .. }        // r/o  
class Disziplin { .. }           // r/o
```

```
public interface BJSStammdaten {  
    Sportplatz getSportplatz();  
    List<Wettkampfort> getWettkampforte(Sportplatz sportplatz);  
    List<Disziplin> getDisziplinen(Wettkampfort wettkampfort);  
    List<Wettkampfort> getWettkampforte(Disziplin disziplin);  
}
```



## Schritt 4: Komponente BJS-Planung – Anwendungsfälle (Use Cases)



```
public interface Planung {
    Gruppe makeGruppe(String nummer,
                       Geschlecht geschlecht,
                       Alter alter);
    boolean addSchueler(Gruppe gruppe, ISchueler schueler);
    boolean removeSchueler(Gruppe gruppe, ISchueler schueler);
    Gruppe getGruppe(String schuelerId);

    List findAufsichten(Map filter);
    Klasse findKlassen(Map filter);
    List findSchueler(Map filter);
    List findGruppen(Map filter);

    List getBelegteSlots(Aufsicht aufsicht);
    List getBelegteSlots(Gruppe gruppe);
    List getBelegteSlots(Wettkampfort wettkampfort);
    List getFreieSlots(Aufsicht aufsicht);
    List getFreieSlots(Gruppe gruppe);
    List getFreieSlots(Wettkampfort wettkampfort);
    boolean belegeSlot(Aufsicht aufsicht, Wettkampfort wettkampfort,
                      Gruppe gruppe);
}
```

## Schritt 5: Paketstruktur Anwendungskern



### ► Beispiel: Bundes Jugend Spiele (BJS)

Konfiguration

BjsAppl

Interfaces +  
Transportstrukturen

interfaces

et

uc

dt

stammdaten

interfaces

et

uc

dt

leistung

interfaces

et

uc

dt

planung

...

interfaces

et

uc

dt

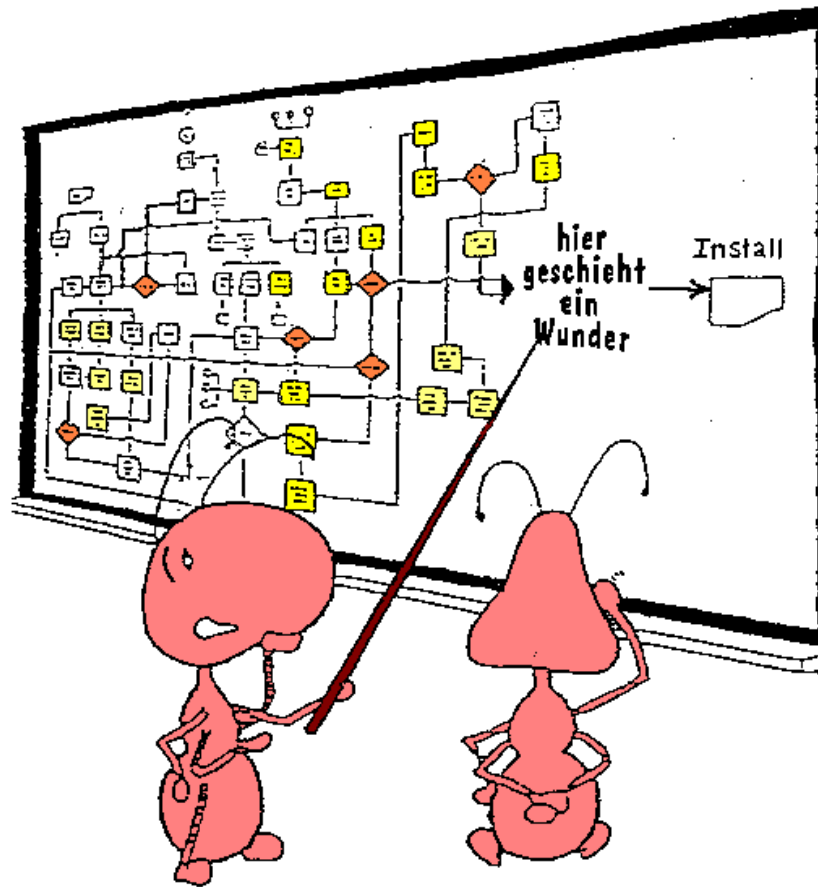
Entitätstypen  
(Datenmodell)

Use Cases  
(Anwendungsfälle)

Datentypen



Jetzt müssen wir die vielen Kästchen „nur“ noch implementieren



**Sehr gute Arbeit!**  
**Aber sollten wir hier vielleicht nicht**  
**noch ein wenig detaillierter werden...?**



# Implementierung des Anwendungskerns

interfaces		
et	uc	dt

- ▶ Für jeden Datentyp: eine Datentypklasse
- ▶ Für jeden Entitätstyp: eine Entitätsklasse
  - ▶ get- und set-Methoden nur dort, wo es Sinn macht: "einzahlen", "abheben" statt "setSaldo".
  - ▶ Konstruktoren: Standardkonstruktor fast immer unsinnig (was soll ein leeres Konto?).  
Oft: ein Hauptkonstruktor mit einer langen Parameterliste.
  - ▶ Nummernvergabe im Konstruktor mit Hilfe eines Nummernservers / ID-Generators
  - ▶ Verwaltung vieler Attribute: Konsistenz von Konstruktor(en), clone, equals, hashCode und Attributdefinitionen!!
- ▶ Für jede Komponente mindestens eine Klasse mit Anwendungsfällen



# Intelligente (fachliche) Datentypen

## ▶ Was ist das?

- ▶ Datentypen wie z.B. Versichertenart, Flughafencode oder Currency
- ▶ Datentypen die Prüfungen, Plausibilitäten, Formatierungen ohne Fehler und frei von Redundanz programmieren.
- ▶ Sie verstecken die Komplexität technischer Datentypen, wie z.B. die der Datumsdarstellung in Java (Date, Calendar, ...)
- ▶ Intelligente Datentypen erleichtern das Programmieren von großen Projekten
- ▶ Sie können häufig wieder verwendet werden
- ▶ Intelligente Datentypen müssen korrekt und performant implementiert werden (in Java z.B. equals, hashCode)
- ▶ Sie unterstützen den Programmierer indem sie fachliche Komplexität kapseln (z.B. Prüfungen, Transformationen)



# Position in Komponenten-Hierarchie

- ▶ Einfache Datentypen
- ▶ **Intelligente Datentypen**
- ▶ Entitätstypen
- ▶ Komplexe Klassen
- ▶ Muster (Pattens)
- ▶ Komponenten
- ▶ Subsysteme
- ▶ Systeme (Anwendungen)
- ▶ Enterprise Applications



## Gruppen von Datentypen

- ▶ **Klassiker:** String, Datum, Geld, Intervall
- ▶ **Minigrammatiken:** Dateinamen, URLs, Prüfziffertypen, Flugfrequenz
- ▶ **Enumerationen:** Anrede, Wochentag, Bonität, Rating, Versicherungstarif
- ▶ **Tabellentypen:** Flughafencode+Land, Versicherungstarif+Höchstalter
- ▶ **Zusammengesetzte Typen:** Adresse, Bankverbindung

**Unsinn: STRING10, STRING20, ..**



## Beispiel: Versichertenart

### Der Software-GAU

```
if v_art in ( 101, 102, 103, 104, 105, 106, 107, 108, 109, 110, 111,
             114, 115, 116, 117, 118, 119, 120, 121, 122, 123, 124,
             401, 402, 403, 404, 406, 407, 408, 601, 602, 603)
then  -- Pflichtversicherung
      -- tu was vernünftiges
end if;
```

### So sollte es sein

```
if versart.ist_pflichtversichert(v_art)
then
      -- tu was vernünftiges
end if;
```



# Datentypen: Sinn und Benutzung

```
class Kunde {  
    private Kundenummer nummer;  
    private Adresse    lieferadresse;  
    private Adresse    rechnungsadresse;  
    private Bonitaet    bonitaet;  
    private Geld        umsatzLfdJahr;  
    ..  
  
    private String    hinweisfeld  
}
```

**Prüfungen, Formatierungen und Transformationen werden nach Möglichkeit im Datentyp abgehandelt. So sind die Anwendungsklassen vom Kleinkram befreit. Jede Art von String-Analyse passiert im Datentyp.**



# Wie baut man Datentypen?

- ▶ **änderbar** oder nicht?      Evtl. zwei Varianten (wie String/StringBuffer)
- ▶ **interne/externe Darstellung**      sinnvoll wählen
- ▶ **Konstruktoren:**      Initialisierung immer kritisch  
    `DateTime() { ... ??? ... }`  
    `DateTime( String s, int format ) { ... ok ... } // wirft Ausnahme!`
- ▶ **statische Prüfmethode**      `static boolean check( String s, int format ) { ... }`
- ▶ **nicht-statische Prüfmethode**      `boolean check( String s, int format ) { ... }`
- ▶ **Transformationsmethode**      `String toString( int format )`
- ▶ **fachliche Abfrage**      `boolean istPflichtversichert() { ... harte Abfrage ... }`
- ▶ **Konstanten**      `Farbe.ROT`
- ▶ **sonstige Utilities**      `int getTimeDiffInDays( DateTime date ){ .. }`  
    `int getTimeDiffInSeconds( DateTime date ){ .. }`
- ▶ **Java Objektprotokoll (*equals*, *hashCode*, *clone*, *compareTo*)**      sorgfältig machen!



## Datentypen MailAdress

```
public class MailAddress {                                // immutable
    private String address;
    private static Pattern pattern =
        Pattern.compile("\\w+(\\.\\w+)*@\\w+(\\.\\w+)*");

    public MailAddress(String address)
        throws IllegalArgumentException {
        Matcher m = pattern.matcher(address);
        if (m.matches())
            this.address = address;
        else
            throw new IllegalArgumentException ("invalid mail address");
    }

    public String toString() {
        return address;
    }

    // equals, hashCode, clone
}
```





## Mit der *equals()* Funktion kann ermittelt werden ob Objekte gleich sind

**Tücke:** Es ist ein Unterschied, ob zwei Objekte gleich oder identisch sind. Wechselt man beides, so können logische Fehler entstehen.

```
String s1 = „Prg3“;
String s2 = „Prg3“;
if (s1 == s2) {    // true
    // identical objects
}
if (s1.equals(s2)) {    // true
    // same objects
}
```



```
String s3 = new String(“Prg3”);

if (s1 == s3) {    // false
    // not identical objects
}

if (s1.equals(s3)) {    // true
    // same objects
}
```

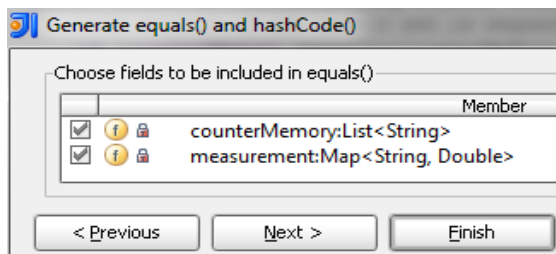
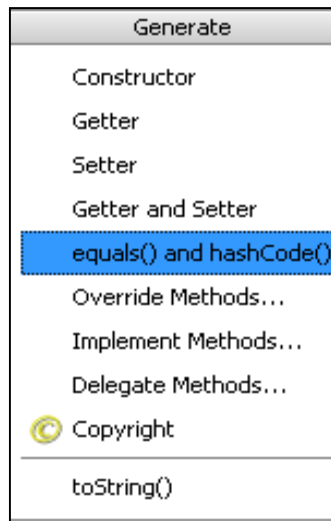
**Gleichheit:** Inhalte zweier Objekte sind logisch gleich. Die Anwendung definiert "Gleichheit":  
Besonders bei Daten- und Entitätstypen.

**Identität:** Zwei Objekte zeigen auf den gleichen Bereich im Hauptspeicher. Es ist ein und dasselbe Objekt. Das Laufzeitsystem definiert „Identität“. Die Standardimplementierung von *equals()* prüft die Identität.



## Wie überschreibt man *equals()*?

- ▶ Man muss sich entscheiden, welche Attribute zu vergleichen sind.
- ▶ Den Rest erledigt die Entwicklungsumgebung.



```
@Override
public boolean equals(Object o) {
    if (this == o) return true;
    if (o == null || getClass() != o.getClass())
        return false;

    ModStatusPerformanceCounter that
        = (ModStatusPerformanceCounter) o;

    if (counterMemory != null ?
        !counterMemory.equals(that.counterMemory) :
        that.counterMemory != null) return false;
    if (measurement != null ?
        !measurement.equals(that.measurement) :
        that.measurement != null) return false;

    return true;
}
```



## ***hashCode()* muss immer konsistent zu *equals()* sein**

- ▶ *hashCode()* wird von allen Hash-Containern benötigt.
- ▶ Die Standardimplementierung bestimmt den hashCode nur aus der ObjectId

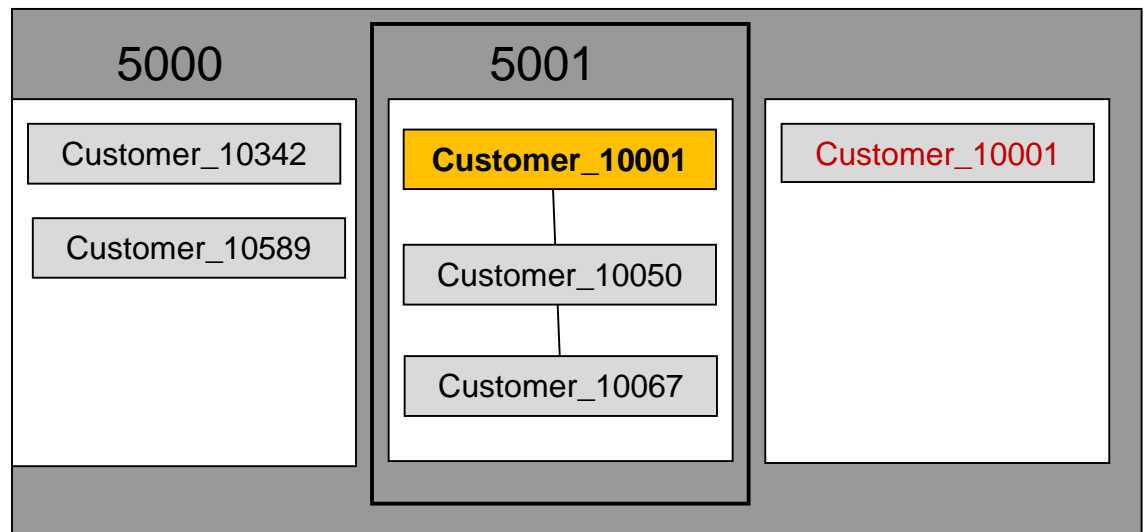
```
Map map = new HashMap();  
Customer customer = new Customer("10001");  
map.put(customer, new Double(500.0));      // hashCode 5001  
map.get(customer); // finds the customer  
map.get(new Customer("10001")); // hashCode != 5001 → returns null
```

### **Problem:**

Zwei *gleiche* Kunden  
mit *unterschiedlichem*  
hashCode.

### **Genauer:**

Die Klasse Customer hat  
`equals` überschrieben, die  
`hashCode`-Methode nicht.





## Warum überschreibt man *toString()*?

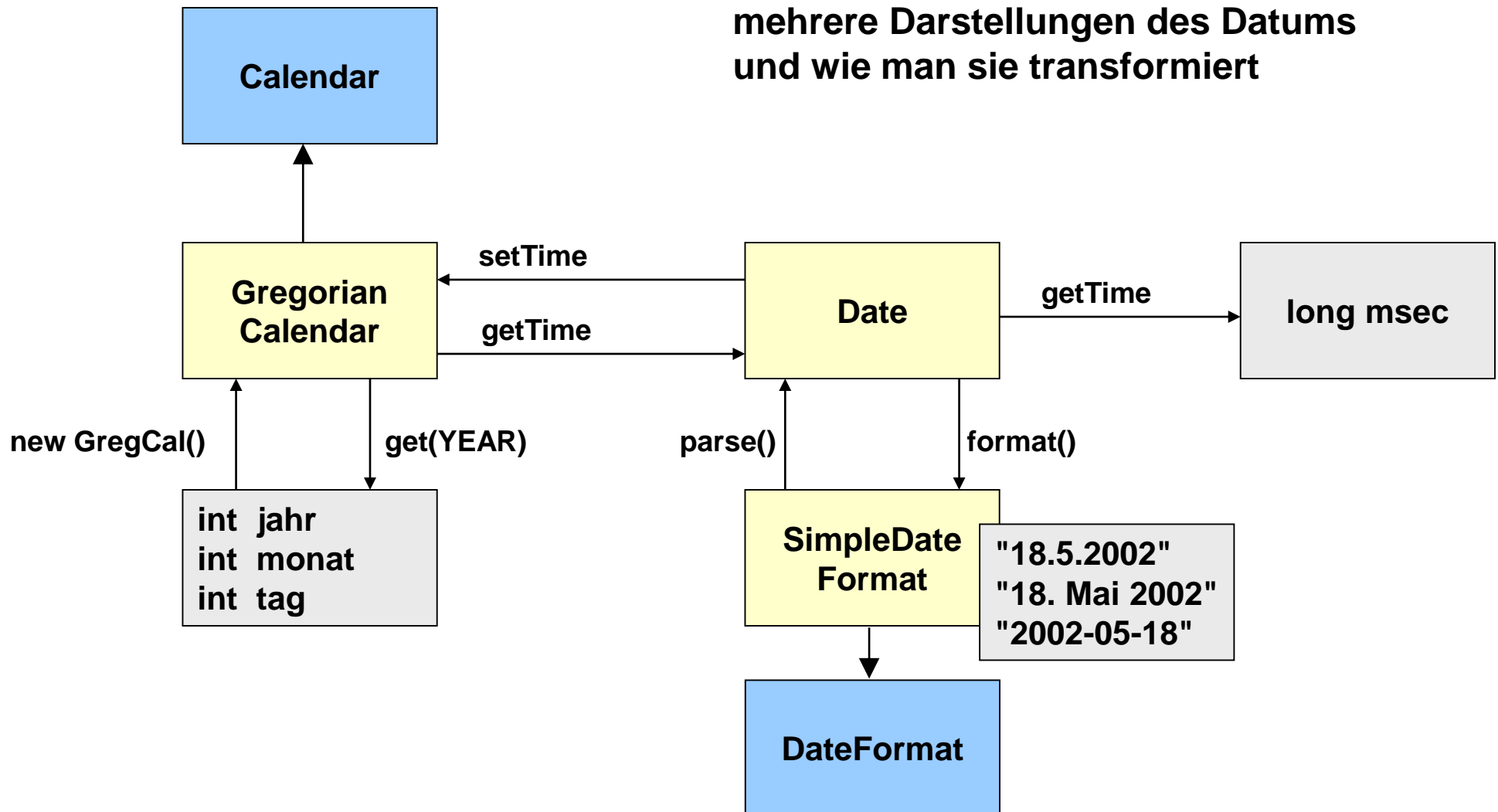
- ▶ *toString()* liefert eine Zeichenkette, die ein Objekt in lesbarer Form beschreibt
- ▶ Diese Methode sollte man bei Daten- und Entitätstypen überschreiben, denn:
  - ▶ Die Standardimplementierung liefert `getClass().getName() + '@' + Integer.toHexString(hashCode())`, also eine weniger gut lesbare Form.
  - ▶ Die meisten Debugger lassen sich so einstellen, dass sie Objekte im *toString*-Format darstellen.
  - ▶ Das Ergebnis lässt sich beim Logging / Tracing verwenden, um den Zustand zu protokollieren. Achtung: Dabei aber natürlich auf das richtige Log-Level achten.



# Das Datum – eine nicht ganz triviale Angelegenheit

- ▶ Zeitzonen: Was ist 14:30 Uhr? GMT/UTC/CET
  - ▶ Konkretes Problem: Unterschiedliche Zeitzonen Client <> Server
- ▶ Eine Minute hat nicht immer 60 Sekunden (Schaltsekunden)
- ▶ Ein Tag hat nicht immer 24 Stunden (Sommer- bzw. Winterzeit)
- ▶ Eine Woche hat nicht immer 7 Tage (z.B. erste Woche im Jahr)
- ▶ Ein Jahr hat nicht immer 365 Tage (Schaltjahre)
- ▶ Schaltjahrberechnungen (Modulo 4 reicht nicht!)
- ▶ Der Wochenanfang ist manchmal mit Sonntag und manchmal mit Montag definiert
- ▶ Es gibt unterschiedliche Behandlungen von zweistelligen Jahreszahlen
- ▶ Die Angabe 01/02/03 kann in Deutschland 2003-02-01, in USA 2003-01-02 und in asiatischen Staaten 2001-02-03 bedeuten.
- ▶ Und viele andere Kleinigkeiten...

## Transformation: Calendar, Date, SimpleDateFormat





## Wissenswertes rund um das Datum

- ▶ Java rechnet intern in Millisekunden seit 1.1.1970 (nach Universal Time Coordonné)
- ▶ **SimpleDateFormat** unterstützt variable Formatierungen
- ▶ Bei nicht elementarer Zeitbehandlung bietet sich eine eigene Implementierung einer Klasse „Datum“ an.
- ▶ Der **GregorianCalendar** bietet sich als Basis für eigene Zeit Klassen an.

```
SimpleDateFormat formatDe =  
    new SimpleDateFormat("dd.MM.yyyy");  
SimpleDateFormat formatUs =  
    new SimpleDateFormat("yyyy.dd.MM");  
  
...  
Calendar today = Calendar.getInstance();  
cal = new GregorianCalendar(  
    today.get(Calendar.YEAR),  
    today.get(Calendar.MONTH),  
    today.get(Calendar.DAY_OF_MONTH)  
);  
  
...  
public String toString(String representation)  
{  
    if(represantation.equals("Us"){  
        return formatUs.format(cal.getTime());  
    }  
    if(represantation.equals("De"){  
        return formatDe.format(cal.getTime());  
    }  
    }...
```

```
String s = "31. März 2008";  
Date date = formatDe.parse(s);
```



## Eine eigene einfache Datum Klasse

```
public class SimpleDate {

    private SimpleDateFormat format = new SimpleDateFormat("dd.MM.yyyy");
    private Calendar date;           // das Datum als Calendar

    public SimpleDate(int year, int month, int day){
        date = new GregorianCalendar(year, month-1, day);
    }
    public SimpleDate(){              // date wird auf die Systemzeit gesetzt
        Calendar today = Calendar.getInstance();
        date = new GregorianCalendar(
            today.get(Calendar.YEAR),
            today.get(Calendar.MONTH),
            today.get(Calendar.DAY_OF_MONTH) );
    }
    public void changeCurrentDate(int year, int month, int day){
        date.clear();
        date.set(year, month-1, day);
    }
    public String toString(){
        return format.format(date.getTime());
    }
}
```





# JodaTime

- ▶ Ist als neuer Java Standard für die Version 8 geplant
- ▶ Zeit und Datum Behandlung
- ▶ Unterstützt verschiedene Kalender Systeme
- ▶ JodaTime ist benutzerfreundlicher als die aktuelle JDK Lösung
- ▶ Open Source und leicht erweiterbar
- ▶ DateTime ist immutable!

## Einige verfügbare Methoden:

```
// aktuelles Datum  
date = new DateTime();
```

```
date.getDayOfMonth();  
date.getDayOfYear();  
date.getDayOfWeek();  
date.getMonthOfYear();  
date.getYear();  
date.getMillis();  
date.getWeekyear();
```

```
String pattern = "YYYY-MM-dd";  
date.toString(pattern);
```

## Wichtige Punkte im Umgang mit Datentypen

- ▶ Datentypen werden häufig exportiert und sind daher für alle Komponenten sichtbar
  - ➔ Datentypen müssen technikfrei sein
- ▶ Vorsicht vor vielen nahezu identischen Datentypen
- ▶ Datentypen eher immutable als mutable
- ▶ Datentypen zum Kapseln von Komplexität nutzen
- ▶ Datentypen sollen unterstützen und vereinfachen, nicht behindern!



# Enumerationen Motivation

## Beispiel: Anrede

<b>INTERN</b>	0	1	2	3
<b>KURZ</b>	„?“	„Hr“	„Fr“	„Fa“
<b>LANG</b>	„?“	„Herr“	„Frau“	„Firma“
<b>ALTSYSTEM</b>	„00“	„01“	„02“	„04“

- ▶ Problem: unterschiedliche Darstellung von Aufzählungen
- ▶ Externe Darstellung: z.B. „Hr“, „Herr“, „01“
- ▶ Interne Darstellung: (Repräsentation, Wert), z.B. (KURZ, 1)

## Seit Java 5.0: Aufzählung als Sprachmittel

Anrede.java

```
public enum Anrede {  
    HERR,  
    FRAU,  
    FIRMA  
}
```

```
Anrede myAnrede = Anrede.HERR;
```

- ▶ **enum** ist ein neues Java-Schlüsselwort
  - ▶ Alte Programme, bei denen **enum** als Variablenname auftaucht können nicht mehr kompiliert werden



## enum-Methoden (Java 5.0) (1)

- ▶ `values()` liefert alle Enumerationswerte. `Iterable` wird nicht implementiert

```
for (Anrede a : Anrede.values())  
    System.out.println(a);    // ..., HERR, FRAU, FIRMA
```

- ▶ `ordinal()` gibt den Ordinalwert (Reihenfolge bei der Deklaration) zurück

```
assert ( Anrede.HERR.ordinal() == 0 );
```

- ▶ Eine Enumeration kann aus einem String erstellt werden, der dem Namen des Enumerationswertes entspricht

```
Anrede myAnredeFrau = Anrede.valueOf(„FRAU“);  
Anrede myAnredeHerr = Enum.valueOf(Anrede.class, „HERR“);
```



## enum-Methoden (Java 5.0) (2)

- ▶ Die Methoden `equals()`, `hashCode()`, `compareTo()` und `toString()` sind bereits vorhanden

- ▶ `compareTo()` vergleicht Enumerationswerte in Deklarationsreihenfolge

```
Anrede.HERR.compareTo>Anrede.FRAU) == -1;
```

- ▶ `toString()` gibt den Namen des Enumerationswerts zurück

```
assert (Anrede.FIRMA.toString().equals(„FIRMA“));
```

- ▶ `equals()` und `hashCode()` benutzen die Objektidentität. Da Enums Singletons sind, werden die Object-Kontrakte erfüllt.


```
assert ( myAnrede.equals>Anrede.HERR) );
```



## Aufzählungstypen (enum) (Java 5.0)

Eigenschaften der neuen Enumeration:

- ▶ Verhält sie wie eine Klasse (Unterklasse von `java.lang.Enum`)
- ▶ Kann eigene Methoden aufnehmen
- ▶ Kann Schnittstellen implementieren
- ▶ Kann **nicht** erben
- ▶ Kann auch innerhalb einer Klasse definiert werden, was dann einer statischen inneren Klasse gleich kommt
- ▶ Kann statisch importiert werden
- ▶ Ordinalwert kann **nicht** angegeben werden:



```
public enum Anrede {  
    HERR = 2, FRAU = 1, FIRMA = 3  
}
```

# Enumerationen in Java bis Version 1.4

- ▶ public static final int

```
public class AnredeEnum {  
    public static final int HERR = 1;  
    public static final int FRAU = 2;  
    public static final int FIRMA = 3;  
}
```

- ▶ Typesafe Enum Pattern (siehe [Bloch 2001])

```
public class AnredeEnum {  
    public static final AnredeEnum HERR = new AnredeEnum("HERR");  
    public static final AnredeEnum FRAU = new AnredeEnum("FRAU");  
    public static final AnredeEnum FIRMA = new AnredeEnum("FIRMA");  
  
    private final String value;  
  
    private AnredeEnum(String value){  
        this.value = value;  
    }  
  
    //equals(), hashCode(), compareTo(), toString() ...  
}
```

```
AnredeEnum a = AnredeEnum.FRAU;
```





## Java 5 enum mit Attributen

```
public enum Anrede {  
    HERR("Herr"),  
    FRAU("Frau"),  
    FIRMA("Firma");           //muss an erster Stelle stehen  
  
    private String text;  
  
    private Anrede(String text) {  
        this.text = text;  
    }  
  
    public String getText() {  
        return text;  
    }  
  
}
```

- ▶ Wichtig: Reihenfolge einhalten!
  - ▶ Zuerst die eigentliche Aufzählung, dann der Rest.



# Entitätstypen vs. Datentypen

## ▶ Entitätstypen

- ▶ Bilden Fachlichkeit des Systems ab
- ▶ Kann man zählen und verändern
- ▶ Haben eine Identität (techn.: Schlüssel)
- ▶ Entitätsobjekte haben Lebenszyklus und können über Assoziationen und Aggregationen verbunden sein
- ▶ Beispiele: Kunde, Konto, Bestellung

## ▶ Datentypen

- ▶ Für sich allein uninteressant
- ▶ Kann man weder zählen noch verändern
- ▶ Haben keine Identität (engl.: value objects) also auch keinen Schlüssel
- ▶ Datentypen enthalten nie Verweise auf Entitätstypen
- ▶ Beispiele: Datum, Adresse, ISBN



## Beispiel: Entität Schueler

```
public class SchuelerImpl implements ISchueler {
```

```
    private String    id;  
    private String    name;  
    private String    vorname;  
    private Datum     geburtsdatum;  
    private Adresse    wohnadresse;  
    ...
```

Fachliche /  
Intelligente  
Datentypen

```
    public SchuelerImpl(String vorname, String nachname  
                        Datum geburtsdatum) { .. }
```

Fachlich  
sinnvoller  
Konstruktor

```
    public String getName() { .. }  
    public Adresse getWohnort() { .. }
```

```
    ..  
    public void setWohnort(Adresse wohnadresse) { .. }
```

```
    //equals(), hashCode(), ...
```

```
}
```

Fachlich sinnv.  
get/set Methoden



## Entitätstypen: Beispiel Kunde

Fachliche /  
Intelligente  
Datentypen

```
public class Kunde implements IKunde{  
    private Kundennummer nummer;  
    private Adresse lieferadresse;  
    private Adresse rechnungsadresse;  
    private Bonitaet bonitaet;  
    private Geld umsatzLfdJahr;  
    ...  
    private String hinweisfeld;
```

Fachlich  
sinnvoller  
Konstruktor

```
    public Kunde(Kundennummer nummer,  
                Adresse lieferadresse, ..) { .. }
```

```
    public Kundennummer getKundennummer() { .. }  
    public Adresse getLieferadresse() { .. }
```

```
    ..  
    public void setLieferadresse(Adresse lieferadresse) { .. }
```

```
}
```

Fachlich sinnv.  
Get/set Methoden  
Achtung: Keine Navigation zulassen

## Entitätstypen: Implementierungshinweise

- ▶ Die Attribute sind in der Regel private Felder der Klasse. "protected" ist oft Unsinn. Aber auch: Lazy loading auf Feldebene möglich!
- ▶ get- und set-Methoden nur dort, wo es Sinn macht: "einzahlen", "abheben" statt "setSaldo".
- ▶ Zentrale Unterscheidung zwischen
  - a) richtigen Objekten, die einen Satz der Datenbank repräsentieren (DB-Repräsentanten)
  - b) Hilfsobjekten, die man aus technischen Gründen braucht (z.B. Before-Images)
- ▶ Konstruktoren: Standardkonstruktor fast immer unsinnig (was soll ein leeres Konto?). Oft: ein Hauptkonstruktor mit einer langen Parameterliste.
- ▶ Nummernvergabe im Konstruktor mit Hilfe eines Nummernservers / ID generators
- ▶ clone, copy
  - a) genau auseinander halten
  - b) macht in der Regel nur für technische Zwecke Sinn (z.B. Before-Image sichern), denn: Was passiert mit dem Schlüssel?
- ▶ Verwaltung vieler Attribute: Konsistenz von Konstruktor(en), clone und Attributdefinitionen!!



# Beziehungen zwischen Komponenten

## ▶ Enge Kopplung

- ▶ Komponenten kennen sich gut und machen viele Annahmen übereinander
- ▶ Entitätsklassen derselben Komponente sehen sich direkt
- ▶ Objektorientierte Schnittstelle, auf der Entitätsreferenzen übergeben werden
- ▶ Realisierung von Assoziationen über Objektreferenzen

## ▶ Lose Kopplung

- ▶ Komponenten kennen sich nicht im Detail und machen wenige Annahmen übereinander
- ▶ Dienstorientierte Schnittstelle, auf der nur einfache Datentypen oder Transportobjekte wie Records erlaubt sind
- ▶ Assoziationen per Übergabe des Fremdschlüssels

## ▶ Entscheidung über Kopplungsart muss bewusst getroffen werden:

- ▶ Bilde Gruppen eng gekoppelter Komponenten
- ▶ Komponenten verschiedener Gruppen können bei Bedarf lose gekoppelt werden



## Beispiel: Kopplung zwischen Schüler und Klasse

### ▶ Enge Kopplung:

```
interface Klasse {  
    void addSchueler(ISchueler schueler)  
}
```

```
interface ISchueler {  
    ...  
    String getId();  
}
```

### ▶ Lose Kopplung:

```
interface Klasse {  
    void addSchueler(SchuelerData schueler);  
    void addSchueler(String schuelerID);  
    void addSchueler(Map schueler);  
}
```

```
class SchuelerData {  
    public String name;  
    public String vorname;  
    public Datum geburtsdatum;  
}
```

über Transportobjekt

über Fremdschlüssel

über 0-Schnittstelle



# Transportobjekte

```
public class KundeData {  
  
    public String id;  
    public String vorname;  
    public String nachname;  
    public Adresse adresse;  
  
    // weitere Attribute, aber keine Beziehungen  
}
```

## Idee:

- ▶ Daten (auch) als Transportobjekte formulieren (übertragen)
- ▶ In Interfaces nur Transportobjekte
- ▶ get / set Methoden in reinen Transportobjekten überflüssig
- ▶ Für Remote Interfaces:
  - ▶ marshalling und unmarshalling erforderlich
  - ▶ von Hand oder modellgesteuert
  - ▶ ggf. Zeitstempel / Versionszähler





## 0-Schnittstelle: Keine Koppelung; Map als generisches Transportobjekt

```
public interface Pflege {  
  
    Iterator find(Query query, Map values); // Liste von Ids  
  
    String makeObject(Map values)           // liefert nur Id  
        throws ApplicationException;  
  
    Map getValues(String id);  
  
    void setValues(String id, Map values)  
        throws ApplicationException;  
  
}
```

- ▶ Syntaktische Koppelung ist nicht mehr da, da Komponenten nur noch über allg. Datenstrukturen kommunizieren.
- ▶ Aber: Aufrufer und Anbieter müssen sich darüber einig sein, was in der Map steht!  
D.h. eine SEMANTISCHE Abhängigkeit besteht nach wie vor und sie ist für einen Compiler unsichtbar.



## A-Schnittstelle: Enge Koppelung

```
public interface IKundenverwalter {  
  
    IKunde findById(String id);  
    Iterator findByPLZ(PLZ plz);    // Liste von Kunde  
  
    IKunde makeKunde(String nachname,  
                     String vorname;  
                     Adresse adresse;  
                     ...) throws ApplicationException;  
}  
  
public interface IKunde {  
  
    String getId();  
    String getVorname();  
    String getNachname();  
    Adresse getAdresse();  
    void setAdresse(Adresse adresse);  
    void addBestellung(IBestellung bestellung);  
    List<IBestellung> getBestellungen();  
}
```

## Zusammenfassung enge vs. lose Kopplung

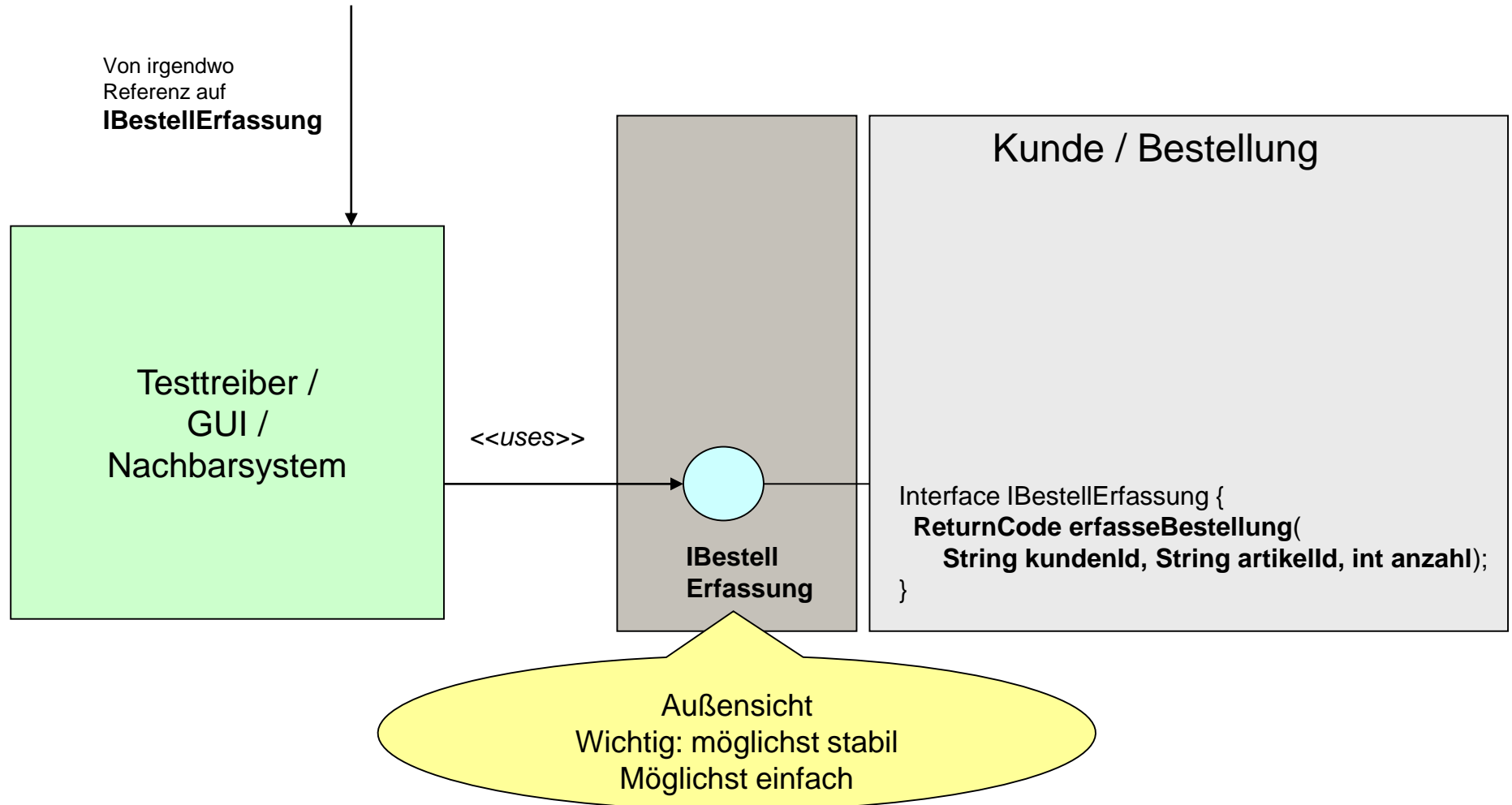
Enge Kopplung	Lose Kopplung
Komponenten kennen sich gut	Komponenten machen wenige Annahmen übereinander
Objektorientierte Schnittstelle	Dienstorientierte Schnittstelle
Objektreferenzen	Werte



# Komponente Bestellung/Kunde aus verschiedenen Perspektiven

## *Komponente aus der Sicht des Nutzers*

(Team, das GUI/Nachbarsystem/Nachbarkomponente baut)

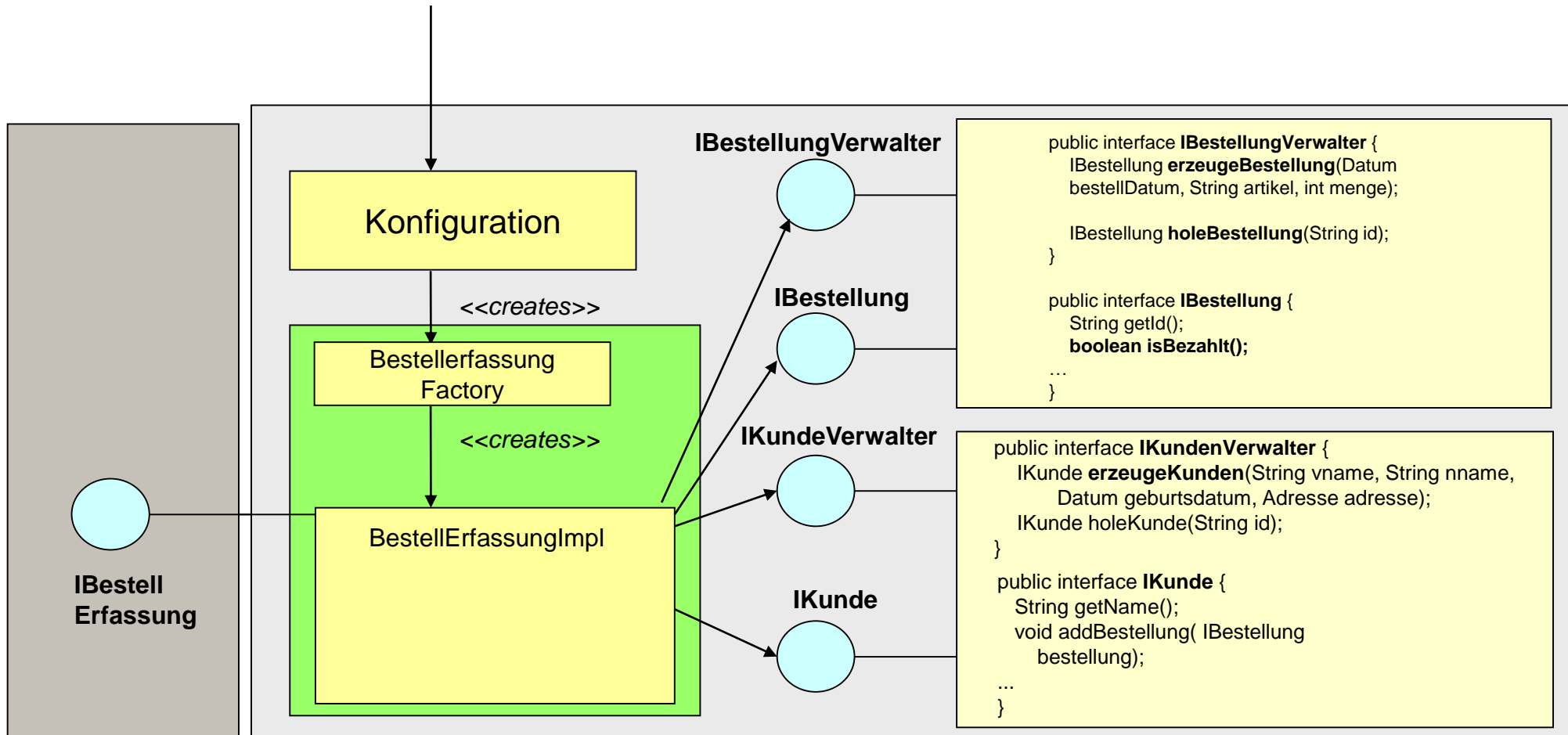




# Komponente Kunde/Bestellung aus verschiedenen Perspektiven

## Komponente aus der Sicht des Erstellers

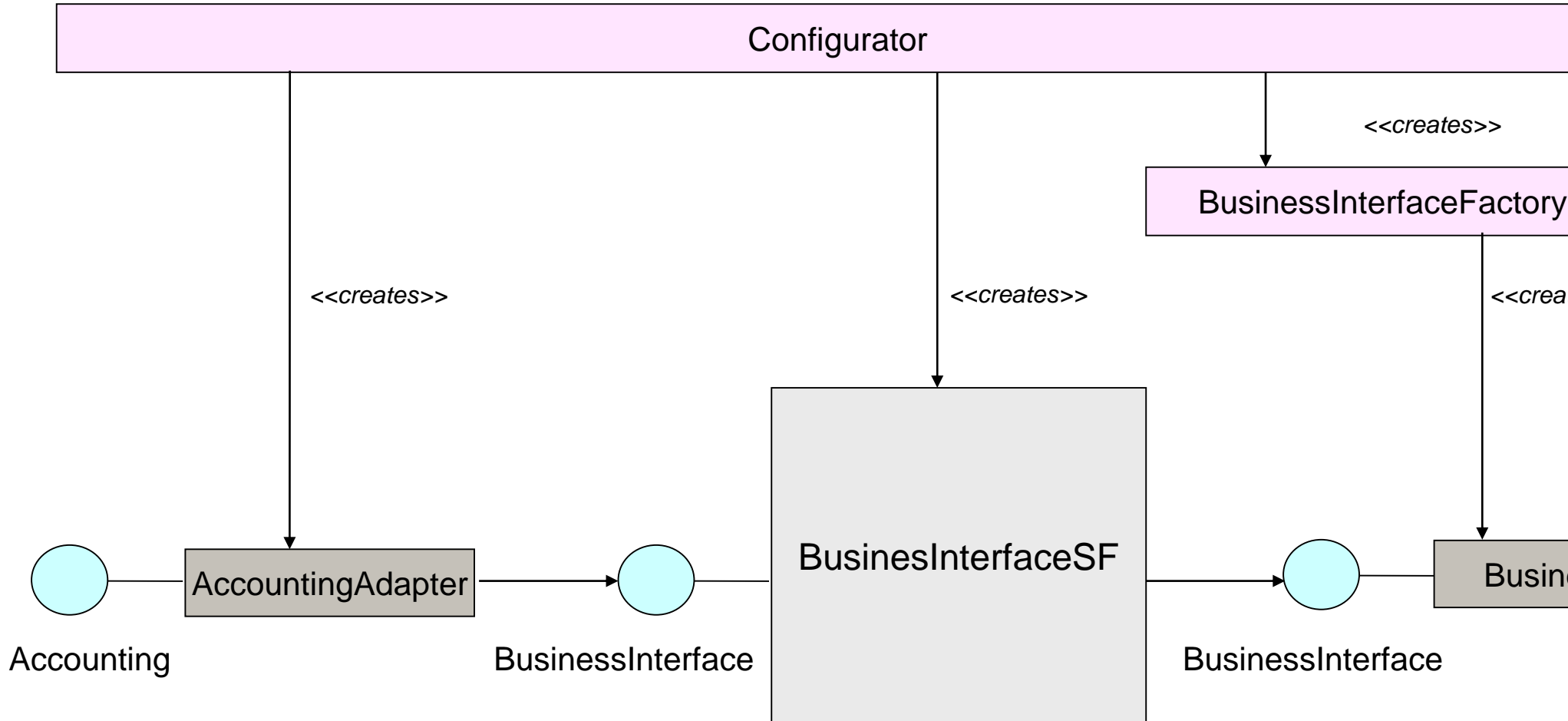
(Team, das die Komponente Kunde/Bestellung baut)





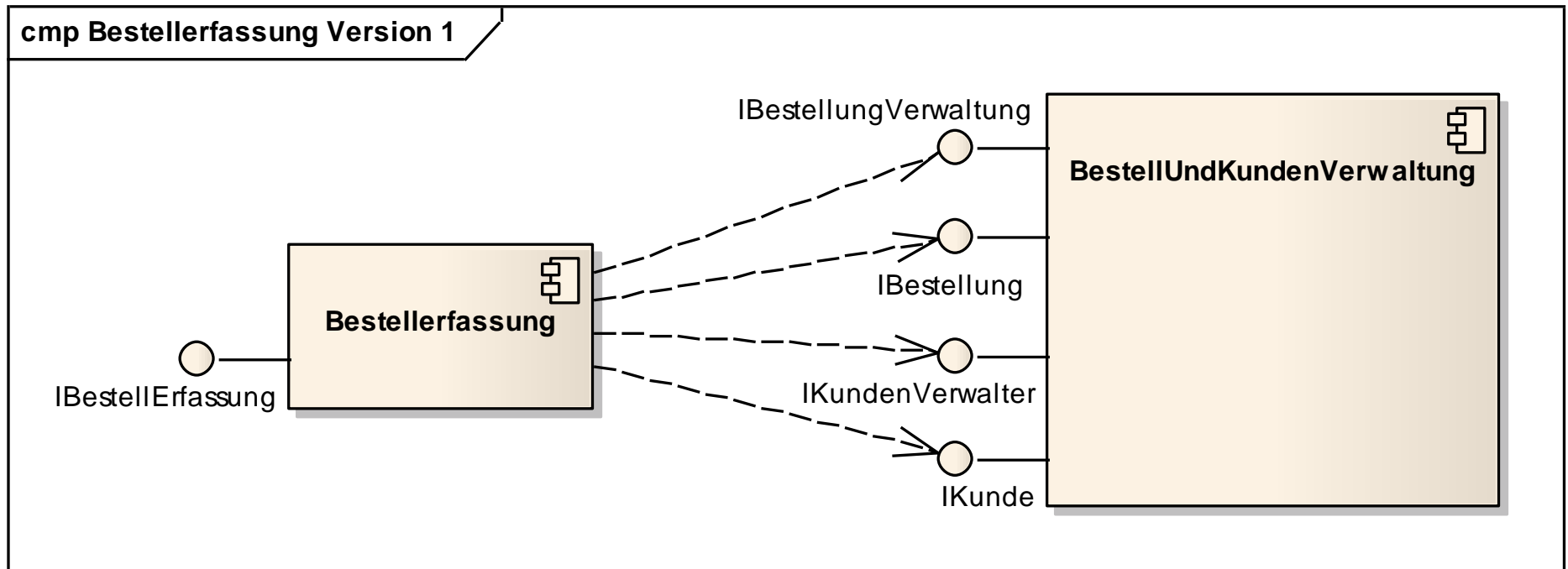
# Komponente Kunde/Bestellung aus verschiedenen Perspektiven

## Komp. aus der Sicht des Zusammenbauers



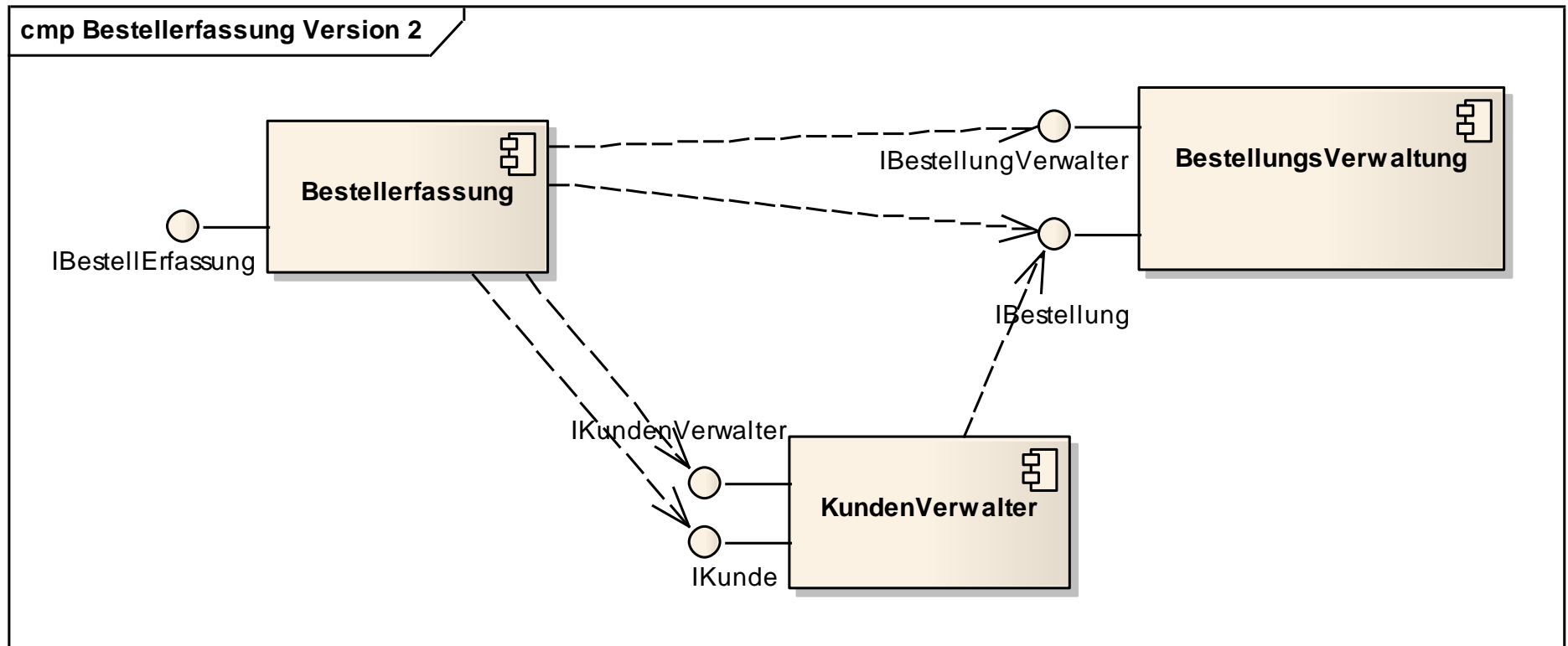
## Varianten des internen Komponenten-Design (1)

- ▶ Variante 1: Bestell- und Kundenverwaltung in einer Komponente realisiert



## Varianten des internen Komponenten-Design (2)

- ▶ Variante 2:
  - ▶ Bestell- und Kundenverwaltung in getrennten Komponenten realisiert
  - ▶ Kunde kennt seine Bestellungen (kunden-zentrische Sichtweise)





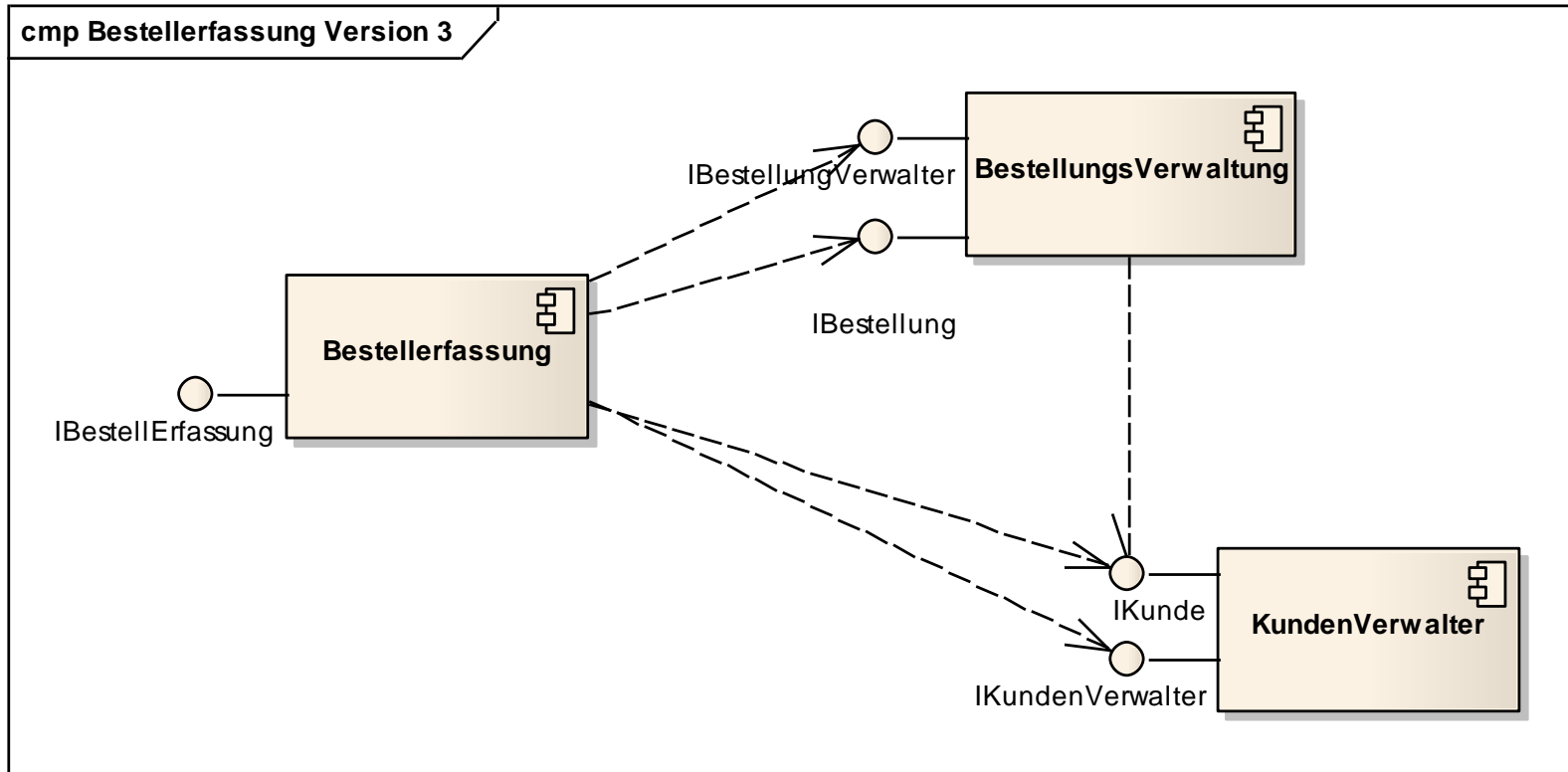


## Varianten des internen Komponenten-Design (3)



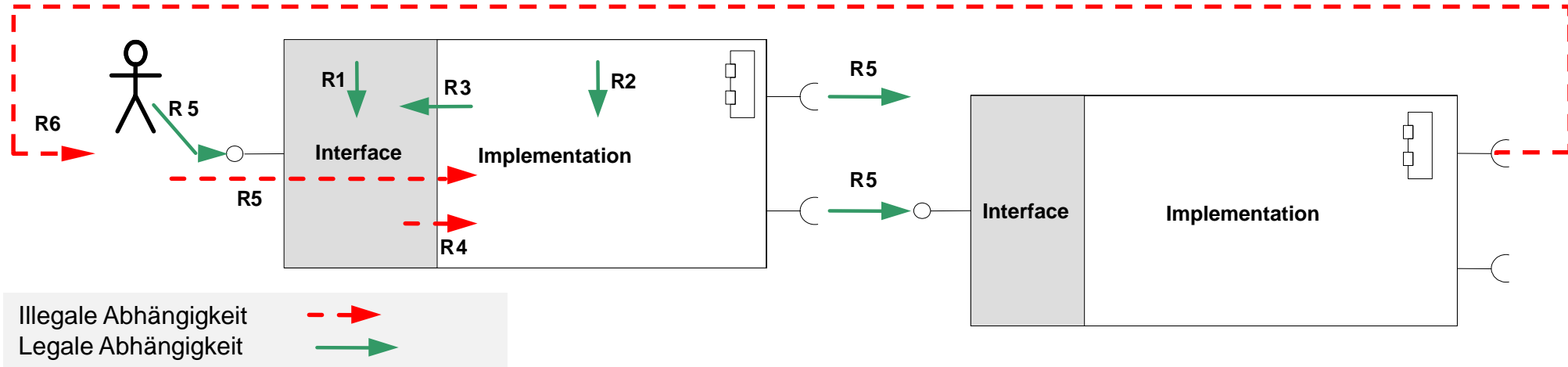
### Variante 3:

- ▶ Bestell- und Kundenverwaltung in getrennten Komponenten realisiert
- ▶ Bestellung kennt den Kunden (bestellungs-zentrische Sichtweise)





# Legale und illegale Abhängigkeiten in Komponenten



- ▶ **(R1)** Ein Artefakt der Schnittstelle einer Komponente C darf andere Artefakte einer Schnittstelle von C nutzen
- ▶ **(R2)** Ein Artefakt der Implementierung einer Komponente C darf andere Artefakte der Implementierung von C nutzen
- ▶ **(R3)** Ein Artefakt der Implementierung einer Komponente C darf alle Artefakte der Schnittstellen von C nutzen
- ▶ **(R4)** Artefakte der Schnittstellen dürfen keine Artefakte der Implementierung verwenden, um keine transitiven Abhängigkeiten zu einem Nutzer der Komponente zu erzeugen
- ▶ **(R5)** Ein externes Artefakt darf ausschließlich Beziehungen zu Artefakten der Schnittstellen einer Komponente besitzen.
- ▶ **(R6)** Der Beziehungsgraph von Komponenten ist zyklensfrei

## Law of Demeter: Don't talk to Strangers

- ▶ Minimierung der Kopplung auch bei Klassen wichtig
- ▶ Z.B. durch Minimieren von indirekten Routineaufrufe in andere Klassen

Viele Annahmen über Selection, Recorder und Location:

```
public void plotDate(Date d, Selection s) {  
    TimeZone tz = s.getRecorder().getLocation().getTimeZone();  
    plotDate(d, tz);  
}
```

Viel besser:

```
public void plotDate(Date d, TimeZone tz) { ... }  
  
plotDate(d, tz);
```

[Andrew Hunt, David Thomas: Der pragmatische Programmierer, Hanser Verlag (2003)]



## Law of Demeter: Talk only to friends

```
public class Demeter {  
    private MyClass myobject;  
    private int mymethod() { ... }  
  
    public void yourmethod(HisClass  
                           hisobject) {  
        YourClass yourobject = new YourClass();  
  
        int i = mymethod();  
  
        hisobject.invert();  
  
        myObject = new MyClass();  
  
        myObject.setActive();  
  
        yourobject.print();  
    }  
}
```

**Das darf man alles aufrufen:**

sich selbst

direkt übergebene Parameter

selbst gehaltene Objekte

selbst erzeugte Objekte



# Goldene Regeln: Anwendungsprogrammierung

- ▶ Denken in Kategorien und Softwareblutgruppen
- ▶ Denken in Komponenten und Schnittstellen.
- ▶ Erst die Außensicht, dann die Innensicht. Was mute ich meinem Aufrufer zu?
- ▶ Trennung von Admin- und Nutzungsschnittstelle von Komponenten.
- ▶ Komponenten haben Datenhoheit (Entitäten)
- ▶ Trennung von Anwendung und Technik; Abhängigkeit von Produkten kontrollieren.
- ▶ Abhängigkeiten bewusst eingehen, Abhängigkeiten minimieren.
- ▶ Verschiedene Varianten der Vermeidung von Abhängigkeiten prüfen



```
public class Arrays {

    List asList(Object[] a)

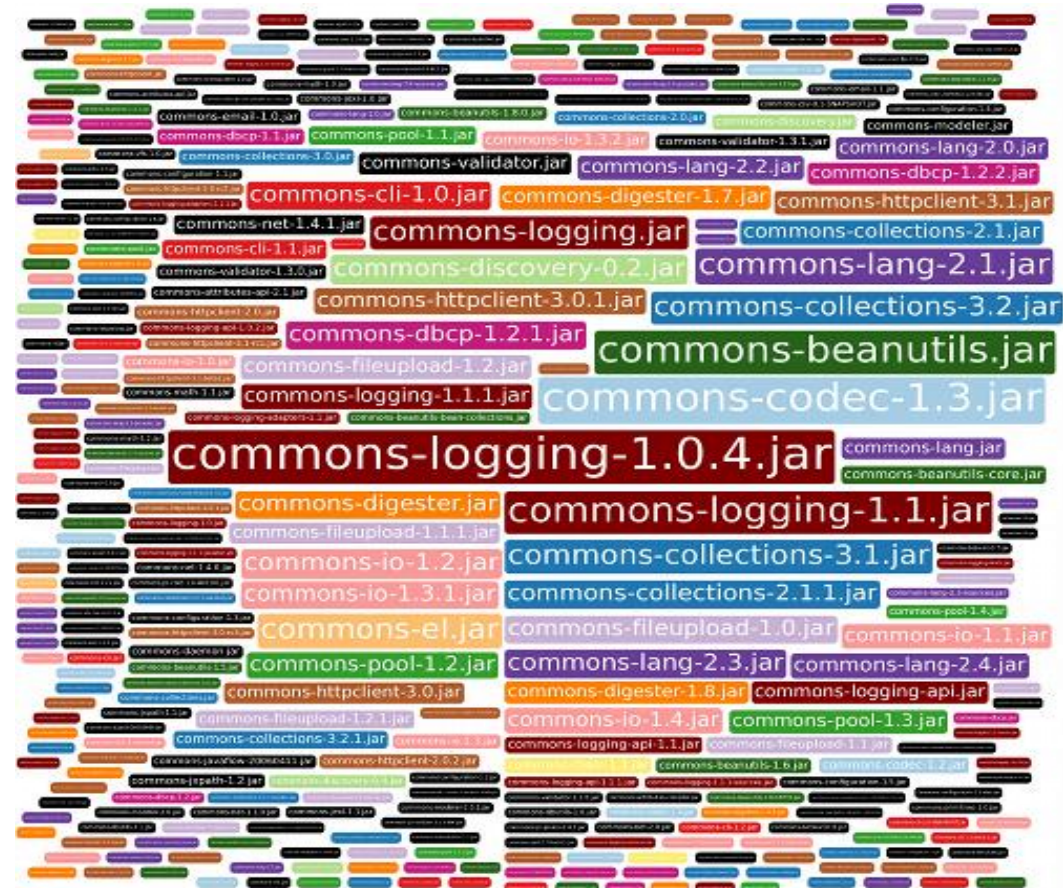
    int binarySearch(int[] a, int key)
    int binarySearch(Object[] a, Object key)
    int binarySearch(Object[] a, Object key,
                      Comparator c)

    boolean equals(int[] a, int[] a2)
    boolean equals(Object[] a, Object[] a2)

    void fill(Object[] a, int fromIndex,
              int toIndex,
              Object val)
    void fill(Object[] a, Object val)

    void sort(int[] a)
    void sort(Object[] a)
    void sort(Object[] a, Comparator c)

}
```



## Tag Cloud zur Nutzungshäufigkeit der Apache Commons Bibliotheken in den OSS Projekten auf Sourceforge und Google Code.



# Zusammenfassung Anwendungsprogrammierung

- ▶ Durch eine saubere Trennung von technischen Code und Anwendungskern erhöht sich die Wartbarkeit von Programmen
- ▶ Der Anwendungskern wird in Komponenten strukturiert
- ▶ Die Komponenten werden strukturiert in Datentypen, Entitätstypen und Anwendungsfälle
- ▶ Intelligente Datentypen erleichtern das Programmieren von großen Projekten indem sie fachliche Komplexität kapseln (z.B. Prüfungen, Transformationen)
- ▶ Sie können häufig wieder verwendet werden und müssen korrekt und performant implementiert werden (in Java z.B. equals, hashCode)
- ▶ Entitäten müssen korrekt implementiert werden (sinnvolle Konstruktoren, sinnvolle getter und setter, sprachspezifische Besonderheiten beachten, z.B. equals, hashCode)
- ▶ Eine saubere Paketstruktur erleichtert die Orientierung und Wartbarkeit von großen Programmsystemen
- ▶ Viele Aussagen über Komponenten, Schnittstellen und Anwendungsprogrammierung zeigen ihren wahren Nutzen erst bei großen Projekten