Objektorientierte Softwareentwicklung

Inhalte:

1. Voraussetzungen
2. OO-Softwareentwicklung- Einstieg
3. Objekte ohne Methoden
4. Objekte mit Methoden
5. Methoden mit Parameter
6. Methoden und Rückgabewerte
7. Datenkapselung und „getter/setter“
8. Konstruktoren
9. Objekte als Parameter
10. Objekte als Rückgabewert
11. Stringverarbeitung
12. Beziehungen zwischen Objekten I
13. Beziehungen zwischen Objekten II
14. Übersicht Beziehungstypen
15. Vererbung
16. Konstruktoren und Vererbung
17. Vererbung und Listen
18. Sequenzdiagramme mit UML
19. Anwendungsfalldiagramme
20. Datenbankanbindung
21. Softwareentwicklung und „Wasserfallmodell“

Version 3.8.5 info

Voraussetzungen

- Vorkenntnisse in einer strukturierten Programmiersprache bzgl.:
  - E/A mit Bildschirm und Tastatur
  - Variablen und Datentypen (int, double, boolean, Texte)
  - Stringverarbeitung (Vergleichen, Konkatenieren)
  - Verzweigungsstrukturen (IF – ELSE),
  - Schleifen (while, for)
  - Arrays
- Darstellung von strukturierten Programmabläufen:
  - PAP
  - Struktogramm

Voraussetzungen

- Die genannten Inhalte werden nur noch implizit thematisiert.
- Falls Vorkenntnisse nicht vorhanden sind: **Selbst aneignen!** (siehe auch Moodle).

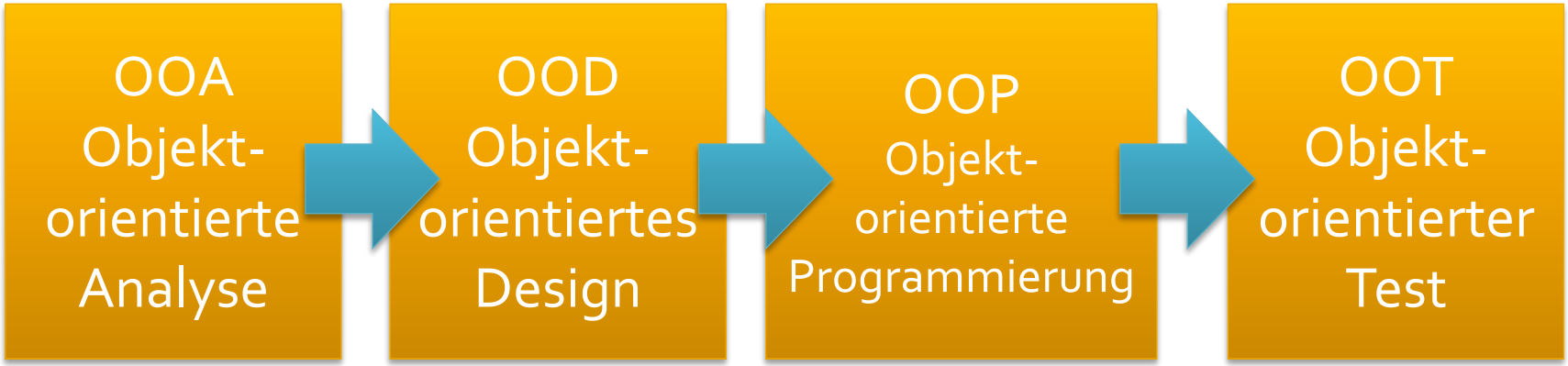
OOSE – Was ist das?

Übersicht | nächste Folie >

OOSE Einstieg

- Objektorientierte Softwareentwicklung heißt:
  - Systematischer Programmentwurf in mehreren Schritten (OOA,OOD,OOP,OOT)
  - Die „Welt“ wird mit Objekten/Klassen beschrieben, welche bestimmte Eigenschaften und Fähigkeiten haben. Die Objekte/Klassen stehen dabei miteinander in Beziehung und interagieren.

Schritte der OOSE



OOA

OOA
Objekt-
orientierte
Analyse

- Analyse des „Problems“
- WAS soll das Programm tun?
- grundsätzliche Strukturen werden identifiziert
- Vorstrukturierung von Klassen-/Objektstrukturen
- Fragen der technischen Implementierung VÖLLIG vernachlässigt.
- Ergebnis der OOA : Use-Case-Diagramme, Aktivitätsdiagramme oder (sehr grob gehaltene) Klassendiagramme.

OOD

OOD
Objekt-
orientiertes
Design

- WIE soll das Programm es tun?
- Ergebnisse der vorigen Schritte werden konkretisiert.
- Wie werden einzelne Prozesse und Interaktionen realisiert?
- Wie wird die Datenhaltung realisiert?
- Wie spielen die einzelnen Schichten (z.B. MVC) zusammen?
- Aspekte der zu verwendenden Programmiersprache werden relevant (Datentypen; ist Überladung, Mehrfachvererbung möglich?).
- Als Ergebnis steht ein möglichst konkretes Klassendiagramm.
- Anmerkung: Die Grenzen zwischen OOA und OOD sind fließend.

OOP

OOP
Objekt-orientierte Programmierung

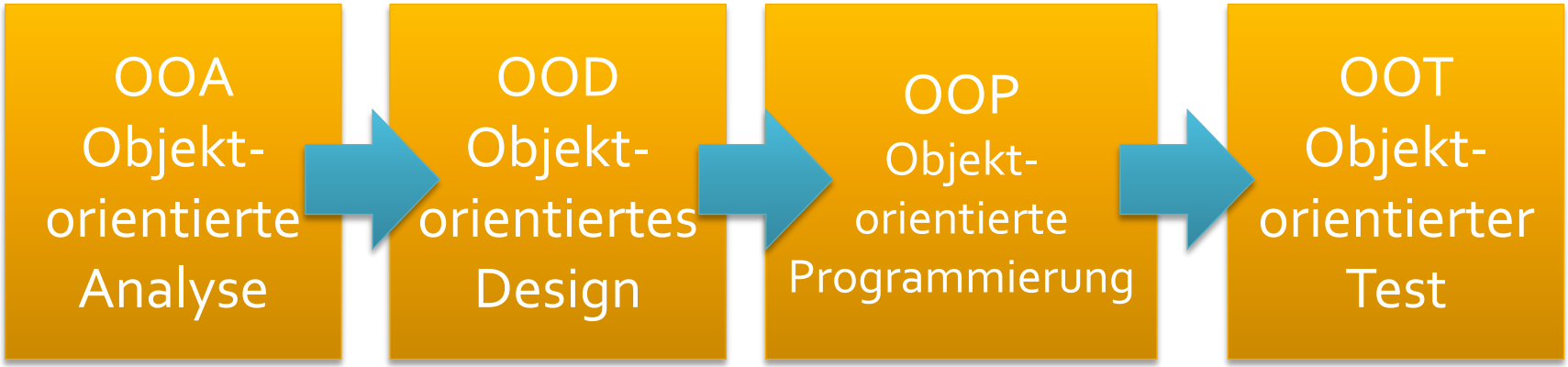
- Umsetzung, d.h. Implementierung der Ergebnisse aus OOA/OOD in Programmcode einer konkreten objektorientierten Programmiersprache.

OOT

OOT
Objekt-
orientierter
Test

- Macht das Programm das, was es soll?
- Definieren und Durchführen von Testfällen.

Schritte der OOSE



Problem Analysieren

Lösungskonzept planen

Lösung auf Basis der Planung umsetzen

Lösung testen

Objektorientierte Softwareentwicklung

Objekte ohne Methoden

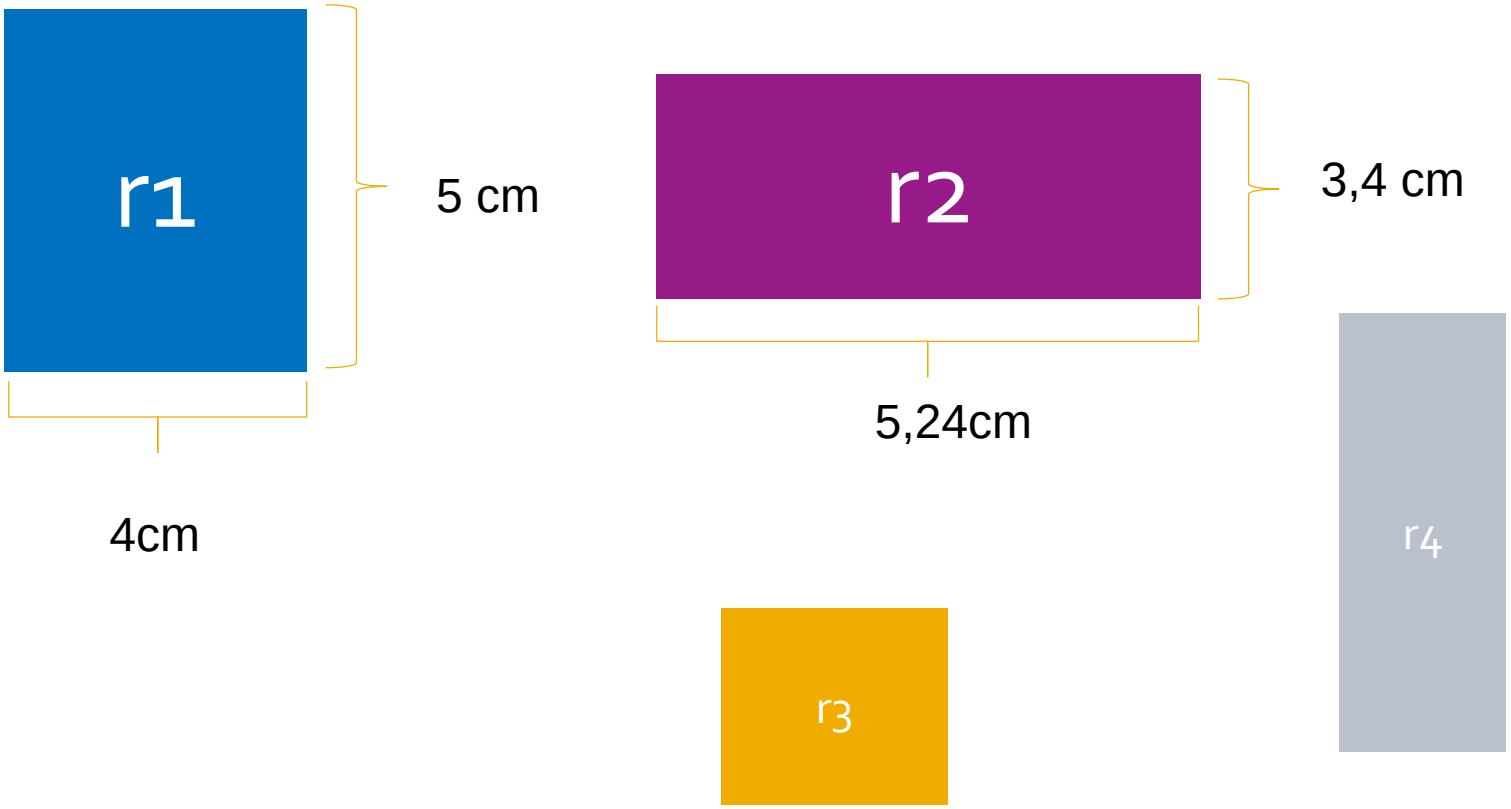
Übersicht | nächste Folie >

Problemstellung

- Problembeschreibung: Sie sind Mitglied einer Entwicklungsabteilung, welche ein Vektorzeichenprogramm entwickeln soll.
- Sie bekommen den Auftrag die Komponente zur Darstellung von Rechtecken zu entwickelt.
- Die Software soll objektorientiert sein.

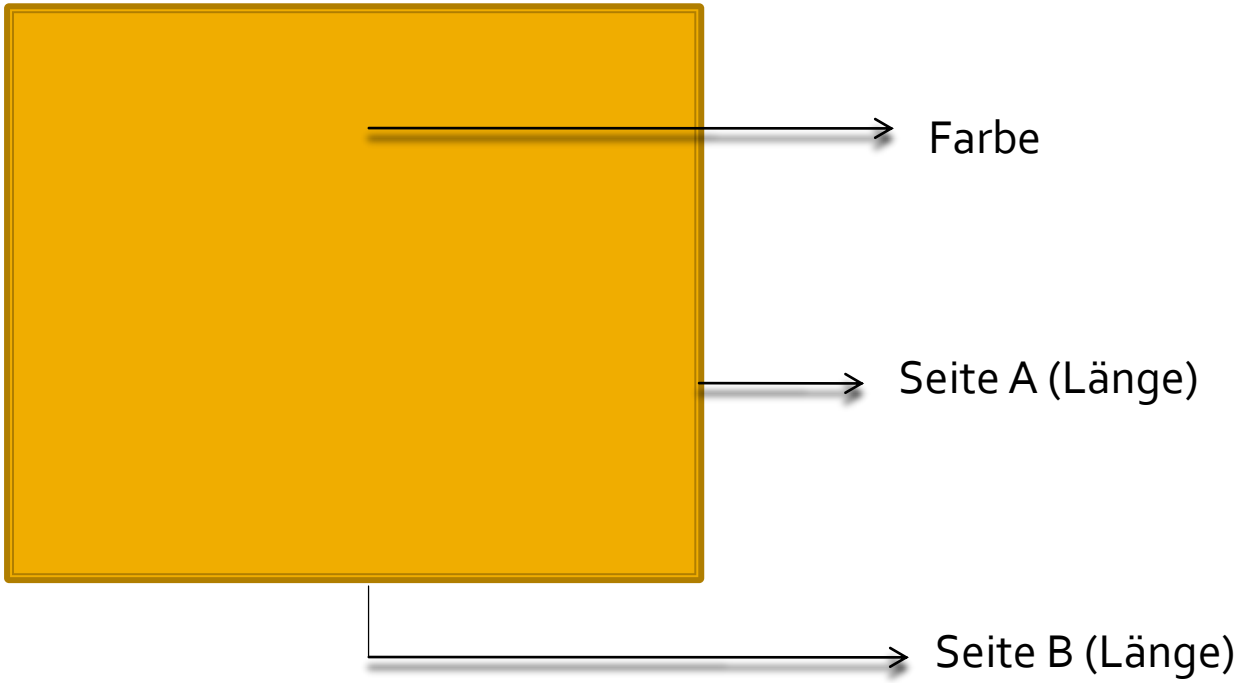
OOA - Problemanalyse

- Wie sehen Beispiel-Rechtecke aus?



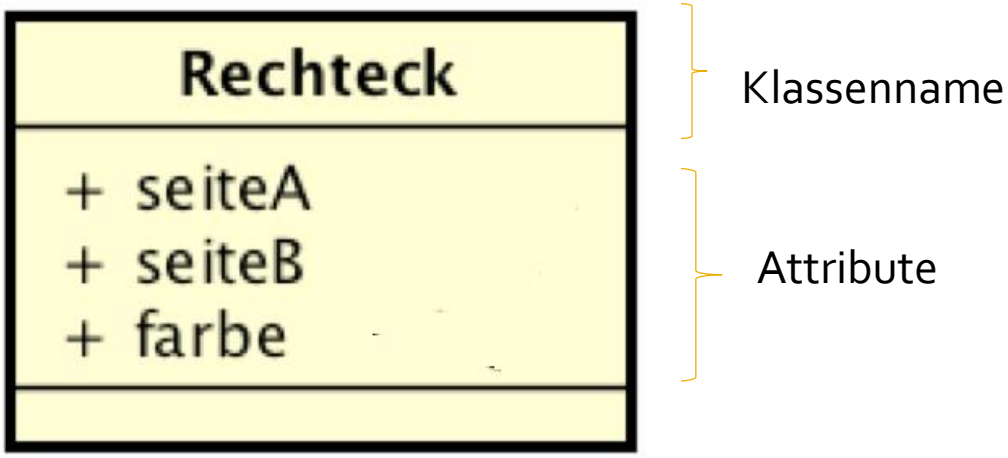
OOA: allgemeine Eigenschaften

- Welche allgemeinen Eigenschaften hat ein Rechteck?



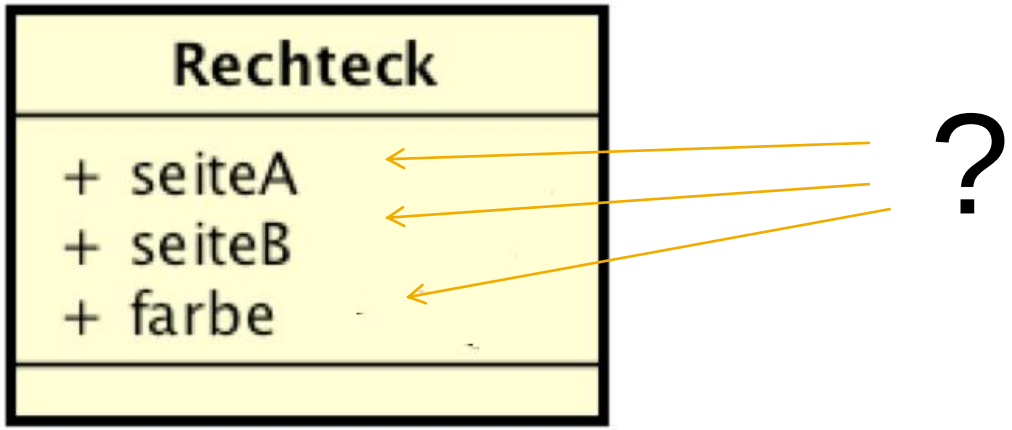
OOA: vereinfachtes UML-Klassendiagramm

Die allgemeinen Eigenschaften werden in einer so genannten **Klasse** festgehalten. Klassen werden in UML wie folgt dargestellt:



OOD: Klassendiagramm vervollständigen

- Überlegen: Was soll in den Attributen gespeichert werden?
- Sinnvolle Datentypen wählen (abhängig von Programmiersprache).



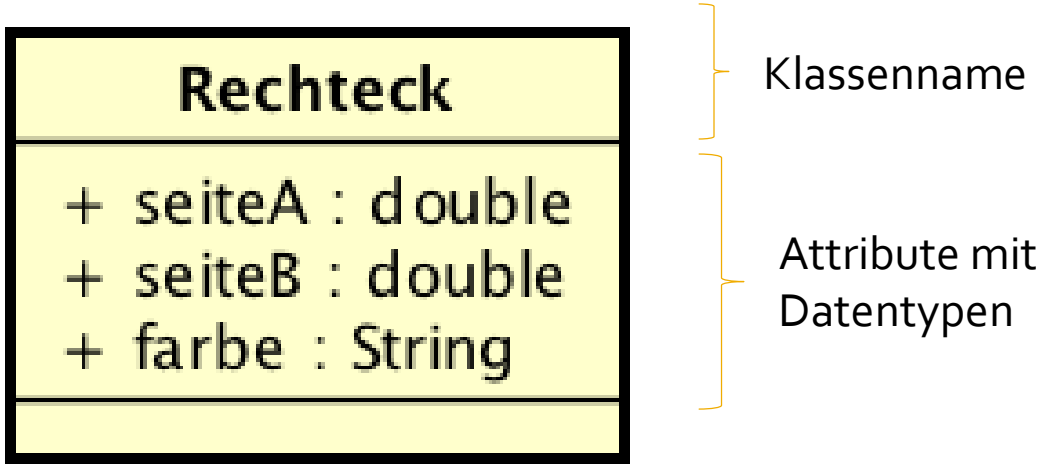
OOD: Klassendiagramm vervollständigen

(Einige) Datentypen in JAVA:

Datentyp	Art	Wertebereich/mögliche Werte	Größe	Beispiel
byte	Ganzzahl	-128 bis +127	8 Bit	byte x = 105;
short	Ganzzahl	-32.768 bis +32.767	16 Bit	short y = -64;
int	Ganzzahl	(ca.) -2 Milliarden bis +2 Milliarden	32 Bit	Int z = 45637;
long	Ganzzahl	(ca.) -10E18 bis +10E18	64 Bit	long l = 123343333;
float	Kommazahl	-3.4E+38 bis +3.4E+38	32 Bit	floaf f = 3.232;
double	Kommazahl	-1.7E+308 bis 1.7E+308	64 Bit	double d = -344.2322;
boolean	Wahrheitswert	true; false	1 Bit	boolean b = false;
char	Zeichen	alle Zeichen	8 Bit	char c = 'g';
String	Zeichenkette	Texte	unterschiedlich	String txt = "hallo";

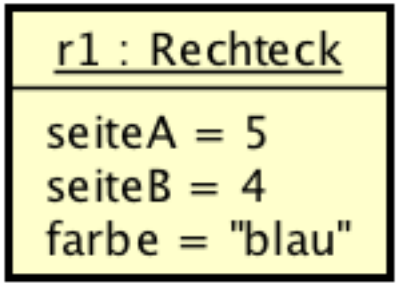
OOD: UML-Klassendiagramm

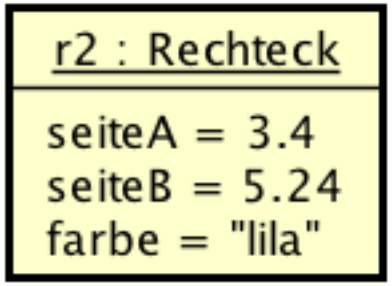
Datentypen angeben:



OOD: Beipiel-Objekte als UML-Objektdiagramm darstellen

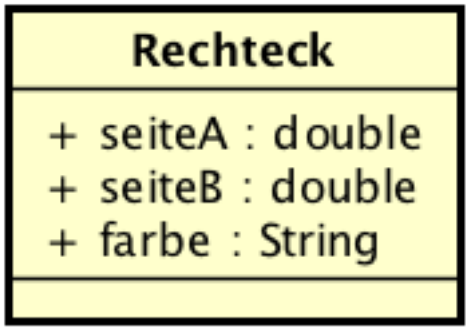
- Das UML-Objektdiagramm für die konkreten Rechtecke r1 und r2 lautet:





OOP: Umsetzung der Klasse in Java

Klassen in UML-Darstellung werden wie folgt in JAVA implementiert:



Datei Rechteck.java:

```
public class Rechteck {

    public double seiteA;
    public double seiteB;
    public String farbe;

}
```

OOP: Objekte erzeugen

Die Objekte werden **innerhalb einer anderen Datei (hier Testrechteck.java)** erzeugt. Diese Datei und die Datei Rechteck.java müssen im selben Verzeichnis liegen!

```
class Testrechteck{

    public static void main(String args[]){

        Rechteck r1 = new Rechteck();

        r1.seiteA = 5;
        r1.seiteB = 4;
        r1.farbe = "blau";

        Rechteck r2 = new Rechteck();

        r2.seiteA = 3.4;
        r2.seiteB = 5.24;
        r2.farbe = "lila";

    }

}
```

r1 : Rechteck
seiteA = 5 seiteB = 4 farbe = "blau"

r2 : Rechteck
seiteA = 3.4 seiteB = 5.24 farbe = "lila"

Hier startet das Programm...

Objekt erzeugen

Objekt mit Werten füllen.

OOT: Testen

- Überlegen von Testfällen, um zu überprüfen, ob das Programm das macht, was es soll.
- Für das Beispiel: Ausgabe der Objekteattribute nach Objekterzeugung und Wertezuweisung auf dem Bildschirm.

OOT

```
public class TestRechteck {
        public static void main(String[] args) {
                Rechteck r1 = new Rechteck();
                r1.seiteA = 5;
                r1.seiteB = 4;
                r1.farbe = "blau";

                Rechteck r2 = new Rechteck();
                r2.seiteA = 3.4;
                r2.seiteB = 5.24;
                r2.farbe = "lila";

                // Ausgabe der Objekte
                System.out.println("Rechteck r1:");
                System.out.println("SeiteA:" + r1.seiteA);
                System.out.println("SeiteB:" + r1.seiteB);
                System.out.println("Farbe:" + r1.farbe);

                // Ausgabe der Objekte
                System.out.println("\nRechteck r2:");
                System.out.println("SeiteA:" + r2.seiteA);
                System.out.println("SeiteB:" + r2.seiteB);
                System.out.println("Farbe:" + r2.farbe);

        }
}
```

Klasse dokumentieren

- Damit andere Benutzer (oder man selbst nach einiger Zeit) eine Klasse möglichst einfach benutzen kann, sollte diese möglichst gut dokumentiert werden.
- Dazu wird die Klasse allgemein (Zweck der Klasse, Autor, Version) und jedes (public) Attribut (Was wird darin gespeichert?) dokumentiert.
- In Java muss dazu eine Form eingehalten werden. Dies hat den Vorteil, dass die Dokumentation später als HTML-Seite generiert werden kann.

Klasse dokumentieren

```
/**
 * Diese Klasse modelliert einen allgemeinen Quader
 *
 * @author Christian Frye
 * @version 1.0
 */
public class Rechteck {

        /**
         * Laenge der Seite a des Rechtecks.
         */
        public double seiteA;

        /**
         * Laenge der Seite b des Rechtecks.
         */
        public double seiteB;

        /**
         * Die Farbe des Rechtecks.
         * Darstellt als Text (z.B. rot, gelb usw.).
         */
        public String farbe;
```

Objektorientierte Softwareentwicklung

Objekte mit einfachen Methoden

Übersicht | nächste Folie >

Problemstellung

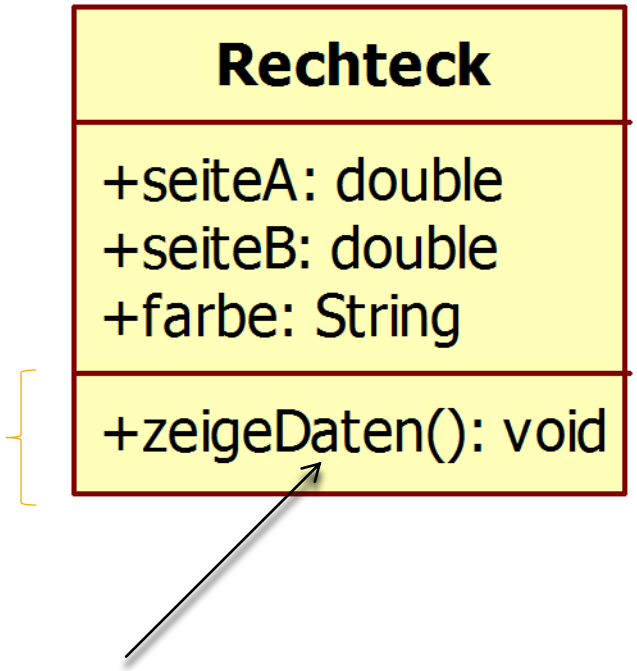
- Bislang ist das Objekt rein **passiv**. Es können über die Attribute nur Werte zugewiesen bzw. wieder ausgelesen werden.
- Um den Objekt eigene Funktionalität bzw. „**Fähigkeiten**“ zu geben müssen Methoden (Operationen, Funktionen) zugefügt werden.

Beispiel

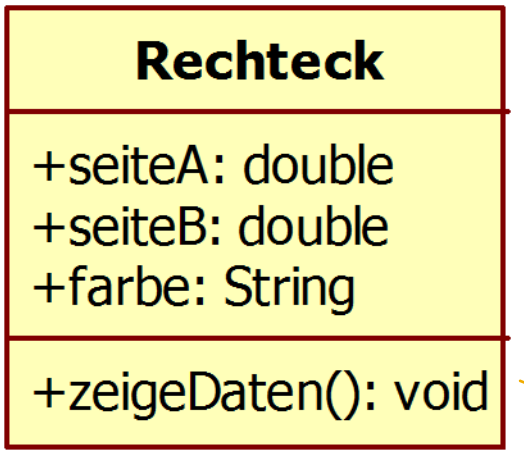
- Das Objekt soll die Fähigkeit bekommen, die Werte seiner Attribute **selbst** auszugeben.
- Diese „Fähigkeit“ soll **„zeigeDaten“** heißen.

OOA/OOD: UML-Klassendiagramm

Fähigkeiten werden als Methoden realisiert. Diese erscheinen im Klassendiagramm im unteren Teil aufgelistet. Methoden werden **nicht** im Objektdiagramm dargestellt.



OOP: Umsetzung in JAVA




```
class Rechteck{
        double seiteA;
        double seiteB;
        String farbe;

        void zeigeDaten(){

        }

}
```

OOP: Programmcode

Datei Rechteck.java:

```
class Rechteck{
        double seiteA;
        double seiteB;
        String farbe;

        void zeigeDaten(){
                System.out.println("Seite a: "+this.seiteA);
                System.out.println("Seite b: "+this.seiteB);
                System.out.println("Farbe: "+this.farbe);

        }
}
```

Zugriff die eigenen Objektattribute.

OOT:

- Zum Testen wird ein Beispielobjekt erzeugt und die Methode „zeigeDaten“ dieses Objektes aufgerufen.
- Die Objektattribute sollten dann vollständig auf dem Bildschirm erscheinen.

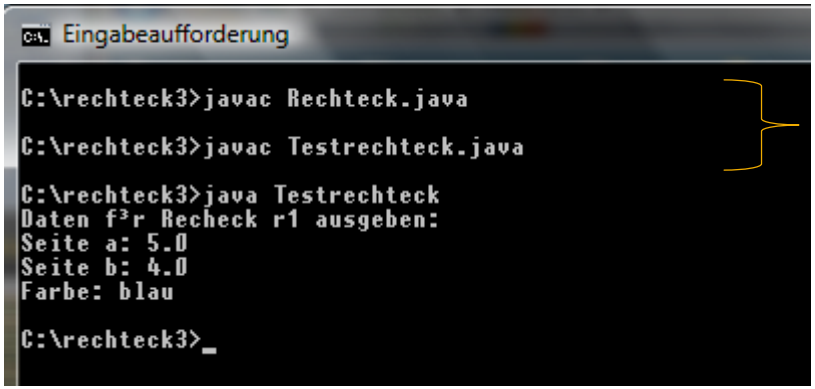
OOT: Objekt erzeugen und Test

Datei Testrecheck.java

```
class Testrechteck{

        public static void main(String args[]){

                Rechteck r1 = new Rechteck();

                r1.seiteA = 5;
                r1.seiteB = 4;
                r1.farbe = "blau";

                System.out.println("Daten für Recheck r1 ausgeben:");

                r1.zeigeDaten();


        }

}
```

Interaktion im Programm

Programmcode ausführen



Programmcode übersetzen

Datei Testrechteck.java:

```
class Testrechteck{

        public static void main(String args[]){

                Rechteck r1 = new Rechteck();

                r1.seiteA = 5;
                r1.seiteB = 4;
                r1.farbe = "blau";

                System.out.println("Daten für Recheck r1 ausgeben:");

                r1.zeigeDaten();


        }
}
```

Objekt erzeugen

Objekt mit Werten belegen

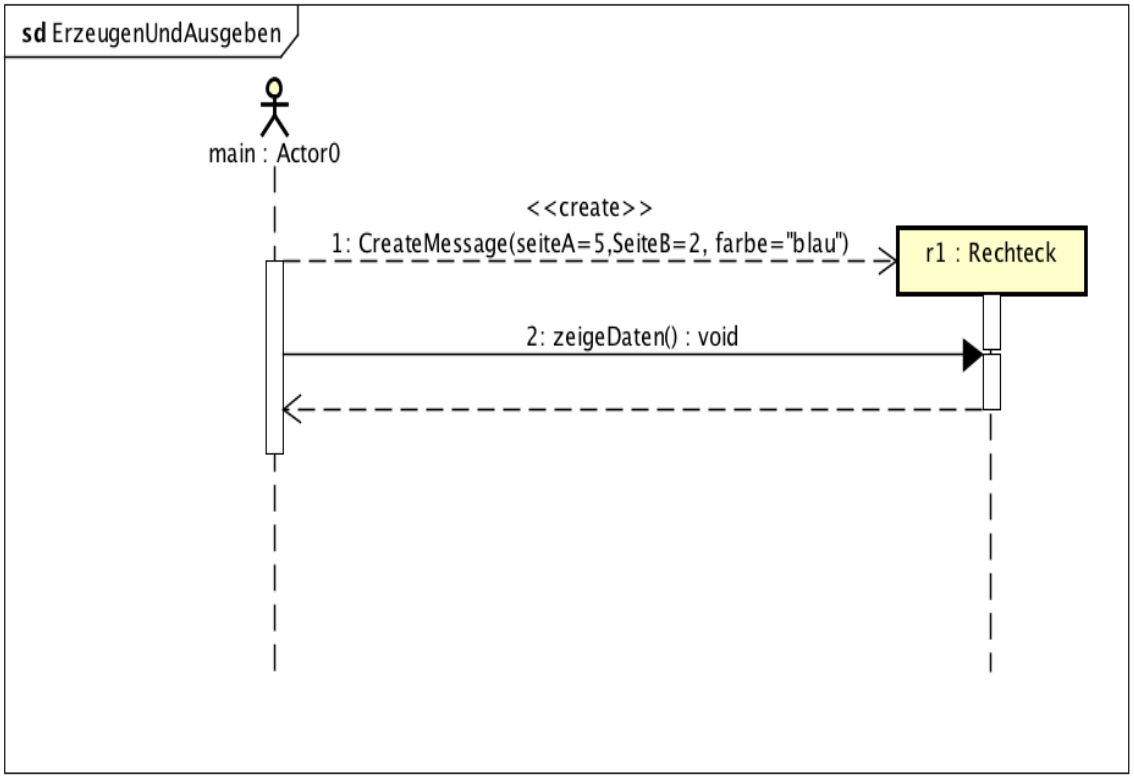
Methode aufrufen

Datei Rechteck.java:

```
class Rechteck{
        double seiteA;
        double seiteB;
        String farbe;

        void zeigeDaten(){
                System.out.println("Seite a: "+this.seiteA);
                System.out.println("Seite b: "+this.seiteB);
                System.out.println("Farbe: "+this.farbe);
        }
}
```

Sequenzdiagramm

```
class Testrechteck{

	public static void main(String args[]){

		Rechteck r1 = new Rechteck();

		r1.seiteA = 5;
		r1.seiteB = 4;
		r1.farbe = "blau";

		System.out.println("Daten für Recheck r1 ausgeben:");

		r1.zeigeDaten();


	}

}
```

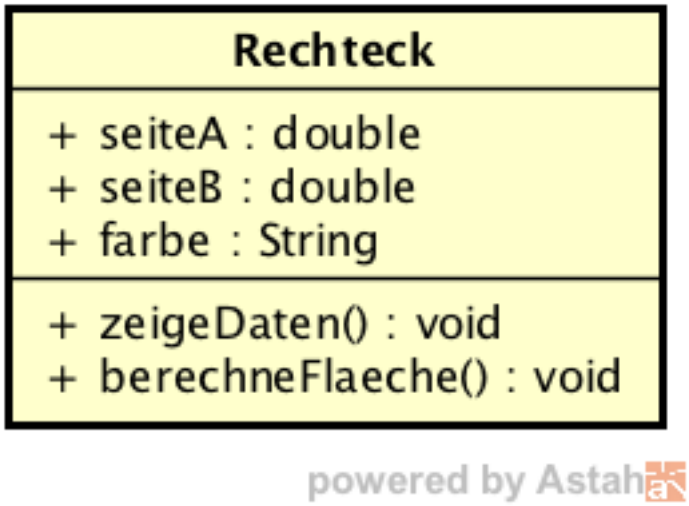


Durch den Aufruf von Methoden kommunizieren die Methoden miteinander /
tauschen Botschaften aus.

Beispiel 2

- Das Rechteck soll die Fähigkeit bekommen seine Fläche zu berechnen und das Ergebnis auf dem Bildschirm anzuzeigen.

OOA/OOD



OOP

```
public class Rechteck {
	public double seiteA;
	public double seiteB;

	public void berechneFlaeche(){
		double flaeche = this.seiteA*this.seiteB;
		System.out.println("Die Flaeche ist:"+flaeche);

	}

	// Methode zeigeDaten nicht dargestellt

}
```

OOT

```
public class testRechteck {
	public static void main(String[] args) {
		// Rechteck erzeugen
		Rechteck r1 = new Rechteck();

		r1.seiteA = 2.4;
		r1.seiteB = 4.6;


		// Flaeche ausrechnen und ausgeben
		r1.berechneFlaeche();


	}

}
```

Methoden dokumentieren

- Jede Methode sollte dokumentiert werden, so dass deutlich wird, was die Methode macht und wie man diese aufruft (dazu später mehr).

Methoden dokumentieren

```
public class Rechteck {
	public double seiteA;
	public double seiteB;

	/**
	 * berechnet die Flaeche des Rechtecks
	 * und gibt das Ergebnis auf dem Bildschirm aus.
	 */
	public void berechneFlaeche(){
		double flaeche = this.seiteA*this.seiteB;
		System.out.println("Die Flaeche ist:"+flaeche);

	}


}
```

Objektorientierte Softwareentwicklung

Objekte und Methoden mit Parametern

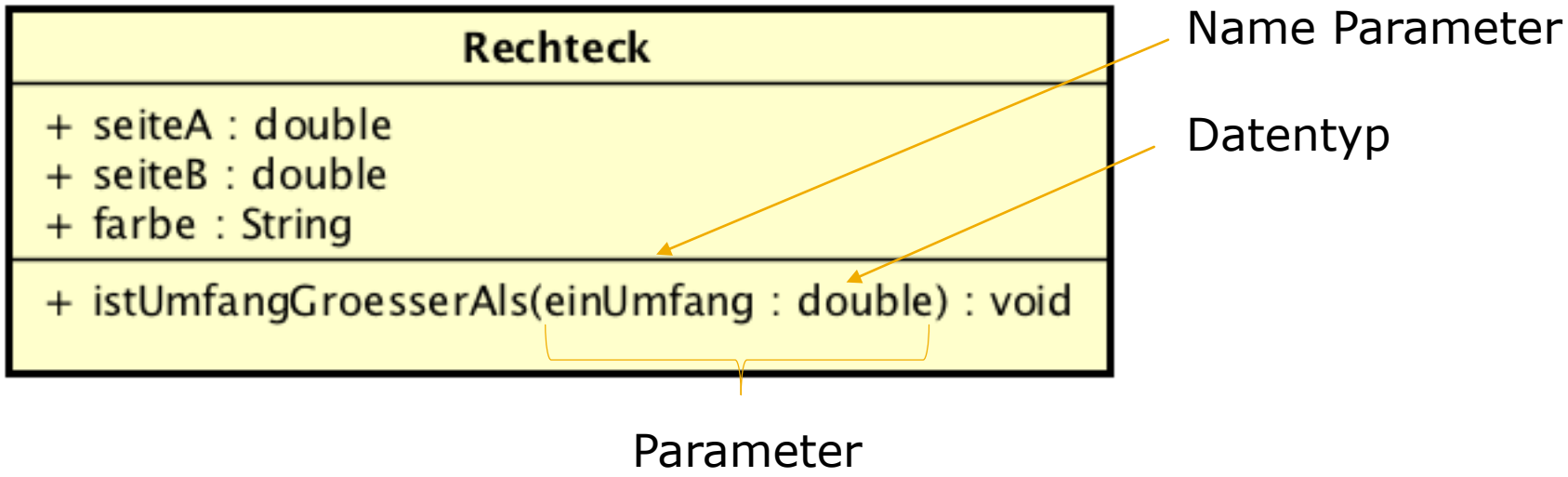
Übersicht | nächste Folie >

Problemstellung

- Das "Rechteck"-Objekt soll um eine Methode „istUmfangGroesserAls" erweitert werden:
  - Dabei wird der Methode wird ein Wert für einen Umfang übergeben, mit dem der Umfang des Rechtecks verglichen werden soll.
  - Die Methode ermittelt, ob der Umfang des Rechtecks größer ist als der übergebene Wert.
  - Das Ergebnis wird auf dem Bildschirm ausgegeben.
- Problem: Wie kann einer Methode ein Wert (hier ein Umfang) übergeben werden?

OOA/OOD

Werte werden in Form von Parametern übergeben
Der Parameter hat einen (sinnvollen) Namen und einen Datentyp:



OOP

```
public class Rechteck {
     public double seiteA;
     public double seiteB;

     public void istUmfangGroeßerAls(double einUmfang){
       // Umfang ermitteln
       double umfangRechteck = 2*this.seiteA+2*this.seiteB;

       if (umfangRechteck>einUmfang){
         System.out.println("Der Umfang ist groesser");
       }else{
         System.out.println("Der Umfang ist nicht groesser");
       }

     }

}
```

OOT

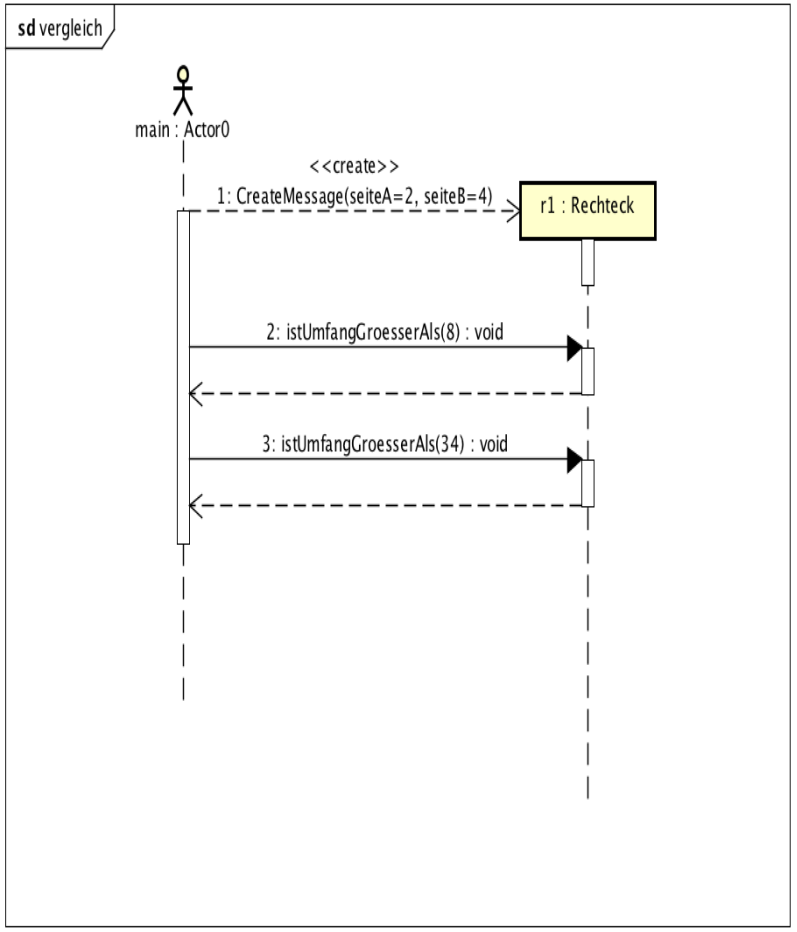
```
public class testRechteck {
	public static void main(String[] args) {

		// Rechteck erzeugen
		Rechteck r1 = new Rechteck();
		r1.seiteA=2;
		r1.seiteB=4;

		// Test grosser
		r1.istUmfangGroeßerAls(8);
		// Test kleiner
		r1.istUmfangGroeßerAls(34);

	}
}
```

```
<terminated> testRechteck [Java Application] /Library/Java/JavaVirtualMachi
Der Umfang des Rechtecks ist groesser
Der Umfang des Rechtecks ist nicht groesser
```


| nächste Folie >

Ablauf als Sequenzdiagramm

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

          // Rechteck erzeugen
          Rechteck r1 = new Rechteck();
          r1.seiteA=2;
          r1.seiteB=4;

          // Test grosser

          r1.istUmfangGroeßerAls(8);

          // Test kleiner
          r1.istUmfangGroeßerAls(34);

    }
}
```



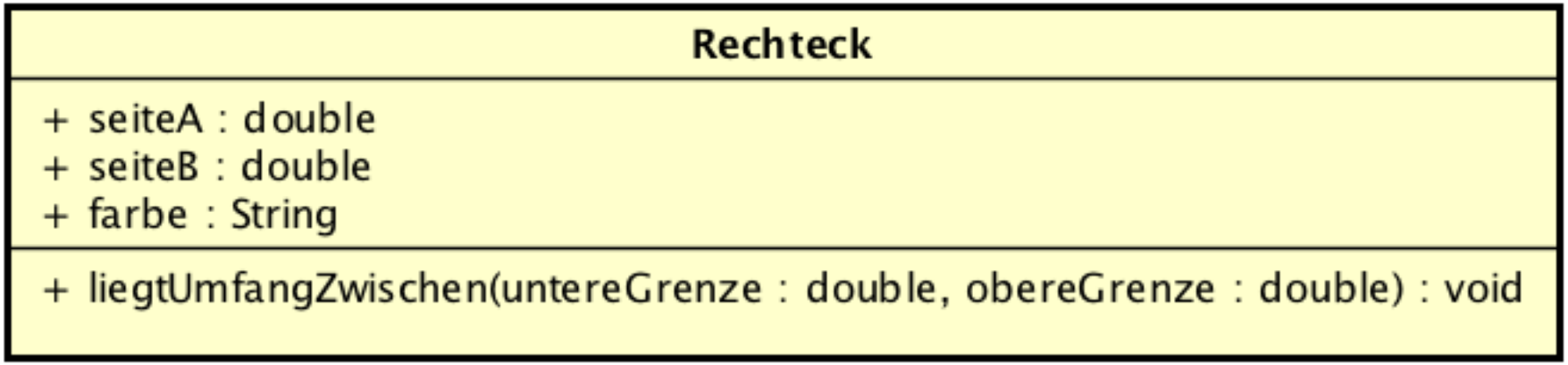
Dokumentieren

```
public class Rechteck {
     public double seiteA;
     public double seiteB;

     /**
     * Die Methode üeberprueft, ob der Umfang des Rechecks
     * groesser als ein uebergebener Wert ist.
     * Das Ergebnis wird auf dem Bildschirm ausgegeben.
     * @param einUmfang Der Wert, der mit dem Umfang verglichen
     *         werden soll
     */

     public void istUmfangGroeßerAls(double einUmfang){
       // der Uebersicht halber weggelassen
     }

}
```

Beispiel 2

- Das "Rechteck"-Objekt soll um eine Methode „liegtUmfangZwischen" erweitert werden:
  - Der Methode werden zwei Werte.
  - Die Methode ermittelt, ob der tatsächliche Umfang zwischen diesen beiden Werten liegt.
  - Das Ergebnis wird auf dem Bildschirm ausgegeben.
- Problem: Wie kann einer Methode mehrere Wert übergeben werden?

OOA/OOD



Mehrere Parameter werden mit Komma abgetrennt.
Für jeden Parameter muss ein Datentyp angegeben werden.
Es können mehrere Methoden mit gleichen Namen bereitgestellt werden, sofern sich die Parameter (Anzahl, Datentypen) unterscheiden.
Dies nennt man Überladen von Methoden.
Es wird dann die passende Methode aufgerufen.

OOP

```
public class Rechteck {
 public double seiteA;
 public double seiteB;

 public void liegtUmfangZwischen(double untereGrenze, double obereGrenze){
     // Umfang ermitteln
     double umfangRechteck = 2 * this.seiteA + 2 * this.seiteB;

     if (umfangRechteck>untereGrenze && umfangRechteck<obereGrenze) {
       System.out.println("Der Umfang liegt dazwischen.");
     } else {
       System.out.println("Der Umfang liegt nicht dazwischen.");
     }
   }
}
```

OOT

```
public class testRechteck {
	public static void main(String[] args) {
		// Rechteck erzeugen
		Rechteck r1 = new Rechteck();
		r1.seiteA=2;
		r1.seiteB=4;

		// Test liegt dazwischen
		r1.liegtUmfangZwischen(8, 16);
		// Test liegt nicht dazwischen
		r1.liegtUmfangZwischen(2, 10);
	}
}
```

```
<terminated> testRechteck (1) [Java Application] /Library/Java/JavaV
Der Umfang des Rechtecks liegt dazwischen.
Der Umfang des Rechtecks liegt nicht dazwischen.
```

Objektorientierte Softwareentwicklung

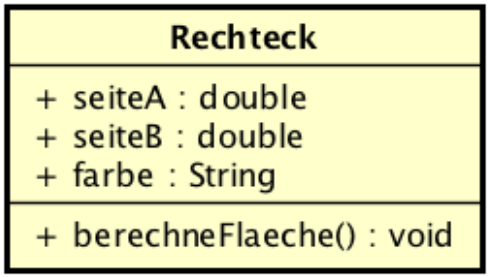
Objekte und Methoden mit Rückgabewert

Übersicht | nächste Folie >

Beispiel

- Ein Programmteil (z.B. main) möchte überprüfen, ob die Fläche eines Rechtecks größer 10 ist.
- Die Klasse Rechteck hat dazu eine „berechneFlaeche"-Methode zur Flächenberechnung und Ausgabe des Ergebnisses.

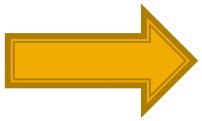
OOA – bisheriger Ansatz




```
public class Rechteck {
        public double seiteA;
        public double seiteB;

        public void berechneFlaeche(){
          double flaeche = this.getSeiteA()*this.getSeiteB();
          System.out.println(flaeche);

        }
}
```

OOP/OOT – erster Ansatz

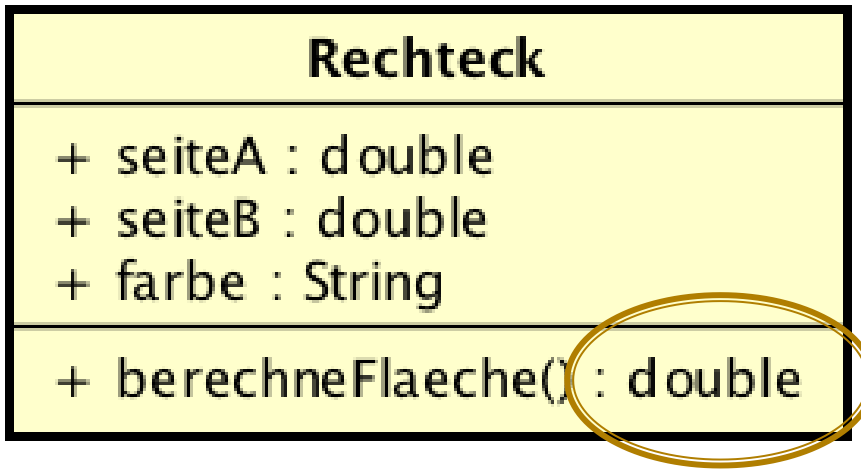
```
public class testRechteck {
          public static void main(String[] args) {
                    // Rechteck erzeugen
                    Rechteck r1 = new Rechteck();
                    r1.seiteA=10;
                    r1.seiteB=4;

                    // Fleche erfragen:
                    r1.berechneFlaeche();

                    // Wie kommt man an die Flaeche dran?

                    // Ueberpruefen, on Flaeche groesser 10
                    if (????>10){
                       System.out.println("Flaeche ist groesser 10");
                    }else{
                       System.out.println("Flaeche ist NICHT groesser 10");
                    }

                    // mit Werten a und b kann nun weiter gearbeitet werden

          }

}
```



Die Methode berechneFlaeche muss eine Antwort geben (der Wert der Fläche).

OOA/OOD – zweiter Ansatz

Die Methode berechneFlaeche bekommt ein Rücvkgabewert:



Datentyp der Rückgabe
(Die Fläche ist eine Kommazahl)

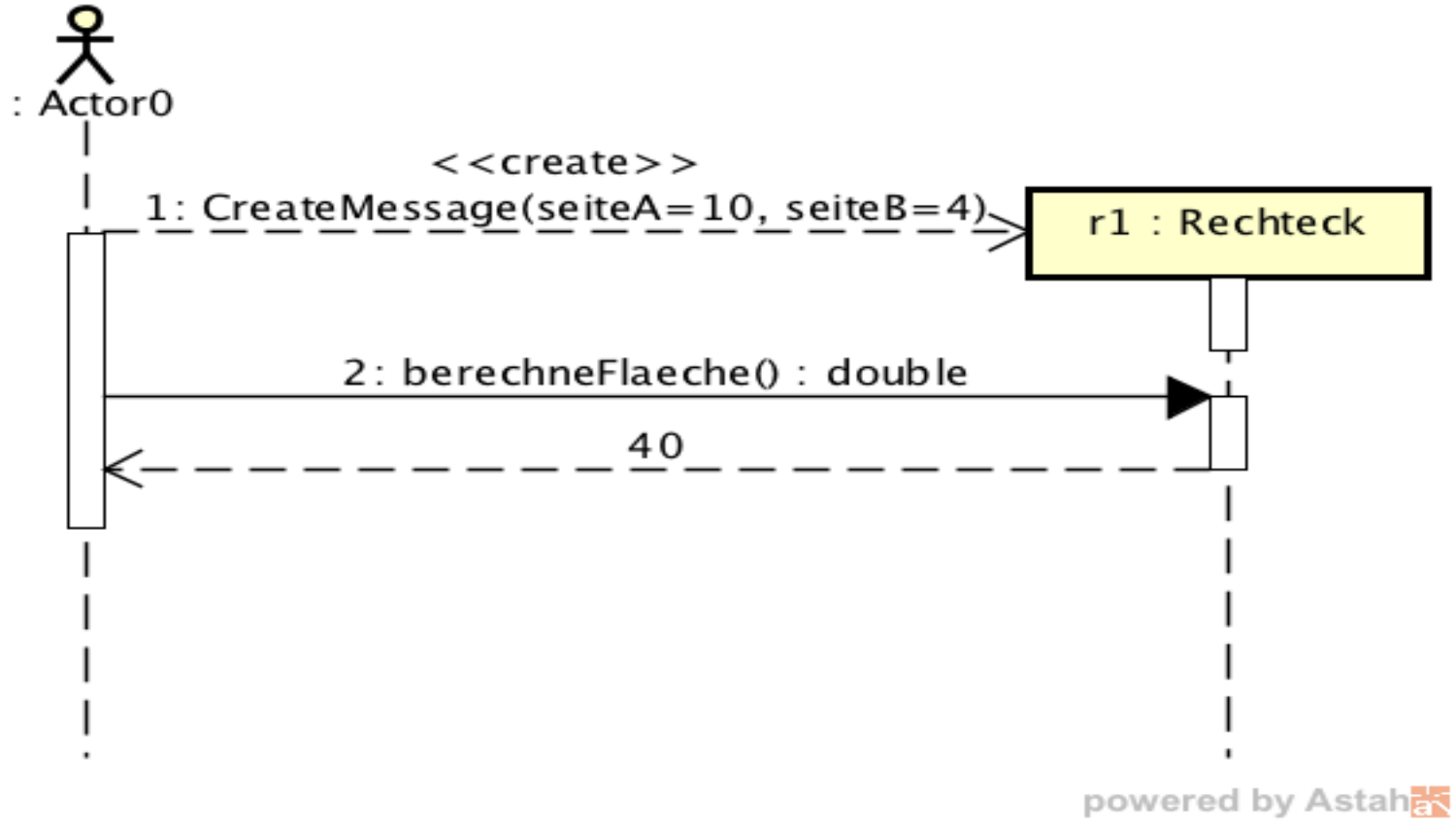
OOP

```
public class Rechteck {
	public double seiteA;
	public double seiteB;

	public double berechneFlaeche(){
		double flaeche=this.seiteA * this.seiteB;
		return flaeche;

	}
}
```

OOT

```
public class testRechteck {
        public static void main(String[] args) {
                // Rechteck erzeugen
                Rechteck r1 = new Rechteck();
                r1.seiteA=10;
                r1.seiteB=4;

                // Flaeche erfragen und Ergebis aufgangen
                double fl = r1.berechneFlaeche();

                // Ueberpruefen, on Flaeche groesser 10
                if (fl>10){
                        System.out.println("Flaeche ist groesser 10");
                }else{
                        System.out.println("Flaeche ist NICHT groesser 10");
                }

        }

}
```

Sequenzdiagramm



Dokumentieren

```
public class Rechteck {
	public double seiteA;
	public double seiteB;

	/**
	 * Methode zum Berechnen der Flaeche
	 * des Rechtecks
	 * @return Die Flaeche des Rechtecks (Kommazahl)
	 */
	public double berechneFlaeche(){
		double flaeche=this.seiteA * this.seiteB;
		return flaeche;

	}
}
```

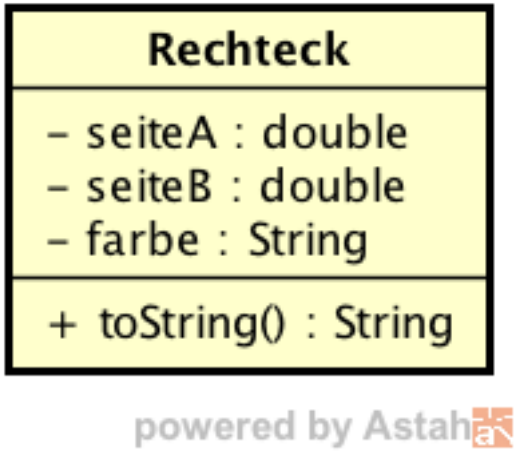
Weiteres Beispiel

- Die Klasse Rechteck soll um eine Methode „toString“ erweitert werden, welche die Werte aller Attribute als ein String zurückliefert. Sie einzelnen Werte sind durch ein Semikolon voneinaner getrennt.
- Beispiel:

r2 : Rechteck
seiteA = 3.4 seiteB = 5.24 farbe = "lila"

Liefert den String „3.4;5.24;lila“ zurück.

OOA


Rechteck
- seiteA : double
- seiteB : double
- farbe : String
+ toString() : String
powered by Astah

OOP

```
public class Rechteck {
	public double seiteA;
	public double seiteB;
	public String Farbe;


	public String toString(){
	    // lokale Variable fuer Rueckgabe
	    String result="";

	    // Rueckgabe zusammenbauen:
	    result = this.seiteA+";"+this.seiteB+";"+this.Farbe;
	   // Ueberlegung: was passiert mit den double-Werten?


	       return result;
	}
}
```

OOT

```
public class testRechteck {
	public static void main(String[] args) {
		// Rechteck erzeugen
		Rechteck r1 = new Rechteck();

		r1.seiteA = 2.4;
		r1.seiteB = 4.6;
		r1.Farbe = "lila";


		String RechteckAlsString = r1.toString();
		System.out.println(RechteckAlsString);


	}

}
```

Objektorientierte Softwareentwicklung

Datenkapselung, set-/Get-Methoden

Übersicht | nächste Folie >

Beispiel

- Bislang werden die Objektattribute durch direkten Zugriff von außen gesetzt.
- Problem:

```
class Testrechteck{

        public static void main(string args[]){
                Rechteck r1 = new Rechteck();
                r1.seiteA = 5;
                r1.seiteB = -2;
                r1.farbe = "hans";
        }
}
```

Problem

- Nachteile:
  - Das Objekt hat nicht die Möglichkeit, auf fehlerhafte Wertzuweisungen zu reagieren (negative Seitenlänge, nicht sinnvolle Farbe).
  - Jeder kann jederzeit die Objektattribute ändern. Dies führt besonders bei komplexen Programmen zu Fehlern, wenn z.B. Objekte den internen Zustand anderer Objekte in unerwarteter Weise lesen oder ändern.
- Lösung: mögliche Fehler schon im Vorhinein ausschließen durch „Datenkapselung“ bzw. Wahrung des „Geheimnisprinzips“.

Geheimnisprinzip

- Geheimnisprinzip:
  - Für das Verwenden der Klasse, z.B. durch den Programmierer, muss man so wenig wie möglich wissen.
  - Das Innenleben der Klasse ist weitgehend geheim und von Außen nicht sichtbar.

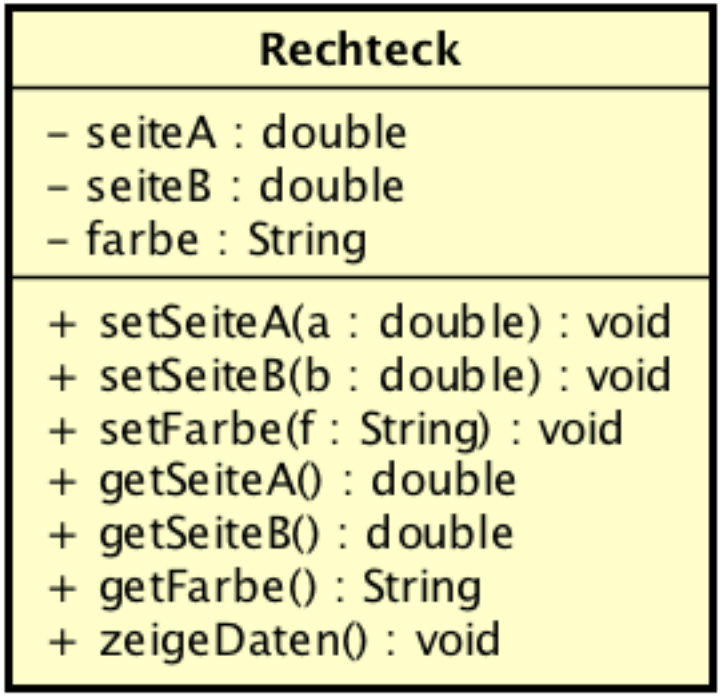
Datenkapselung

- Datenkapselung:
  - Der direkte Zugriff auf Attribute wird unterbunden.
  - Der Zugriff auf Attribute erfolgt indirekt über spezielle Methoden, welche ggf. fehlerhafte Eingaben herausfiltern.
  - Attribute und Methoden, welche rein intern für die Klasse benutzt werden, werden komplett versteckt.
  - Klassen können den internen Zustand anderer Klassen nicht in unerwarteter Weise lesen oder ändern.

Beispiel

- Die Klasse Rechteck wird wie folgt abgeändert :
  - Der direkte Zugriff auf die Attribute wird unterbunden.
  - Zum Setzen sollen der Attribute sollen „set-Methoden“ bereitgestellt werden.
  - Die „set-Methoden“ überprüfen auch, ob die Werte sinnvoll sind.
  - Zum Auslesen der Attribute werden „get-Methoden“ bereitgestellt.

OOA/OOD



Der direkte Zugriff auf die Attribute
wird verboten (- d.h. private).
Die Attribute können nur indirekt über **set-Methoden** gesetzt werde. Diese überprüfen auch ggf., ob die Werte sinnvoll sind.
Um Attribute auszulesen werden **get-Methoden** bereitgestellt.

OOP

```
public class Rechteck {
        private double seiteA;
        private double seiteB;
        private String farbe;

        public void setSeiteA(double a){
                if (a>0){
                        this.seiteA=a;
                }else{
                        System.out.println("kein sinnvoller Wert");
                }
        }

        public void setSeiteB(double b){
                if (b>0){
                        this.seiteB=b;
                }else{
                        System.out.println("kein sinnvoller Wert");
                }
        }
```

OOP

```
public void setFarbe(String f){

        // Anmerkung: der Einfachheit halber nur Test
        // auf rot oder blau
        if (f.equals("rot") || f.equals("blau")){
                this.farbe=f;
        }else{
                System.out.println("kein sinnvoller Wert");
        }
}
```

OOP

```
public double getSeiteA(){
        return this.seiteA;
}

public double getSeiteB(){
        return this.seiteA;
}

public String getFarbe(){
        return this.farbe;
}

public void zeigeDaten(){
        System.out.println("Seite A:"+this.getSeiteA());
        System.out.println("Seite B:"+this.getSeiteB());
        System.out.println("Farbe:"+this.getFarbe());
}
// Ende Klasse Rechteck
```

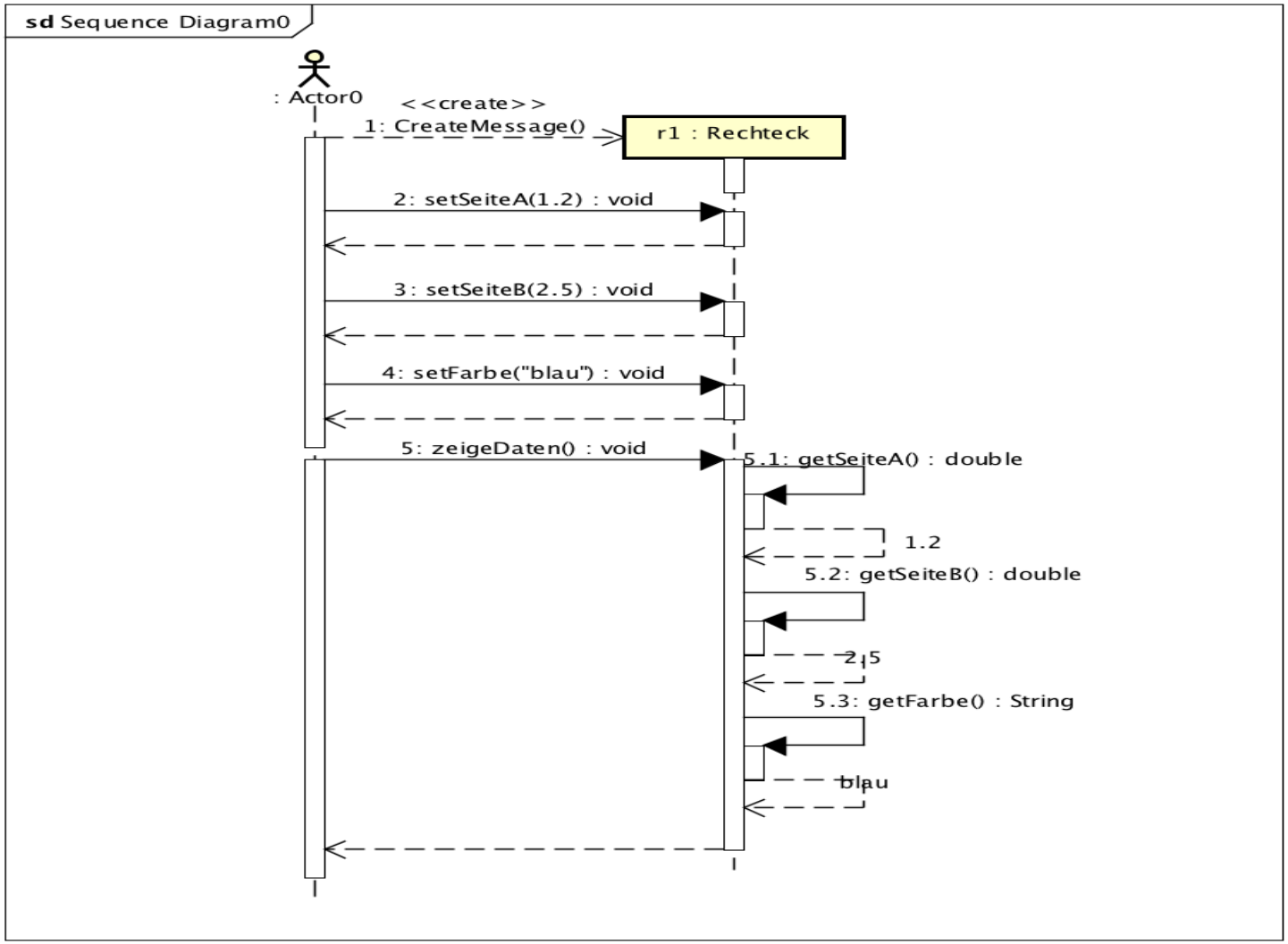
OOT: Testobjekte erzeugen

```
public class testRechteck1 {
	public static void main(String[] args) {
		Rechteck r1 = new Rechteck();

		// nicht mehr moeglich, da Attribute private:
		// r1.seiteA = 5;

		// Rechteck erzeugen und
		// mit set-Methoden Attribute setzen:
		r1.setSeiteA(1.2);
		r1.setSeiteB(2.5);
		r1.setFarbe("blau");

		// Objektattribute zur Kontrolle ausgeben:
		r1.zeigeDaten();
	}
}
```

OOT

- Ist das Programm vollständig getestet?
- Welche weiteren Testfälle sollten durchgeführt werden?

Sequenzdiagramm



Objektorientierte Softwareentwicklung

Konstruktoren

Übersicht | nächste Folie >

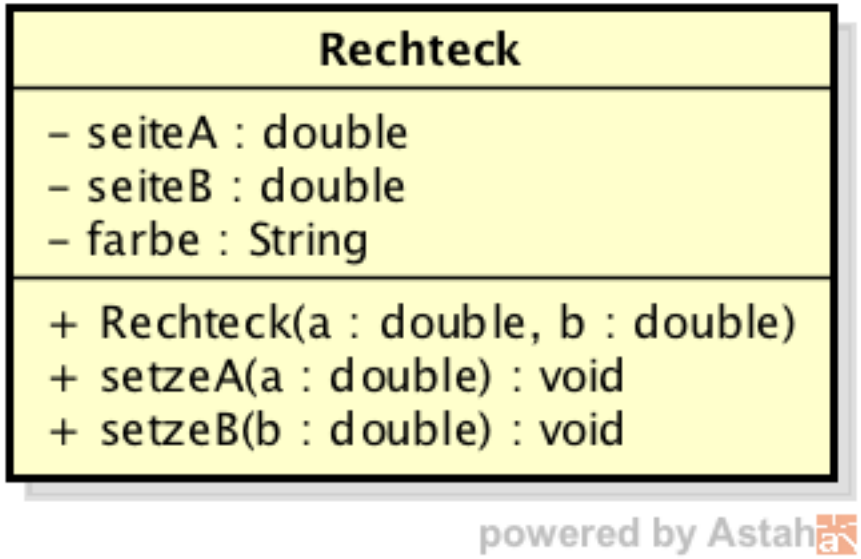
Konstruktoren

- Die Aufgabe des Konstruktors ist es bei der Objekterzeugung bestimmte Vorbereitungen zu „erledigen“ (z.B. Variablen mit Startwerten zu belegen), ohne die ein Arbeiten mit dem Objekt nicht möglich ist.

Beispiel

- Es soll eine Klasse Rechteck entwickelt werden.
- Bereits **bei Objekterzeugung** sollen die Seitenlängen gesetzt werden.
- Es wird ein Konstruktor zum Setzen der Seitenlängen benötigt.

Planung



Konstruktoren werden als **spezielle Methoden** aufgeführt.

Der **Methodenname** ist gleich dem **Klassennamen.**

Implementierung und Test

Implementierung der Klasse Rechteck:

```
public class Rechteck {

	double seiteA;
	double seiteB;
	String farbe;

	Rechteck(double a, double b) {
		this.setzeSeiteA(a);
		this.setzeSeiteB(b);
	}

	public void setzeSeiteA(double a) {
		this.seiteA = a;
	}

	public void setzeSeiteB(double b) {
		this.seiteB = b;
	}

}
```

Objekterzeugung:

```
public class TestRechteck {
	public static void main(String[] args) {
		Rechteck r1 = new Rechteck(4, 5);
	}
}
```

Nicht mehr möglich ist jedoch:

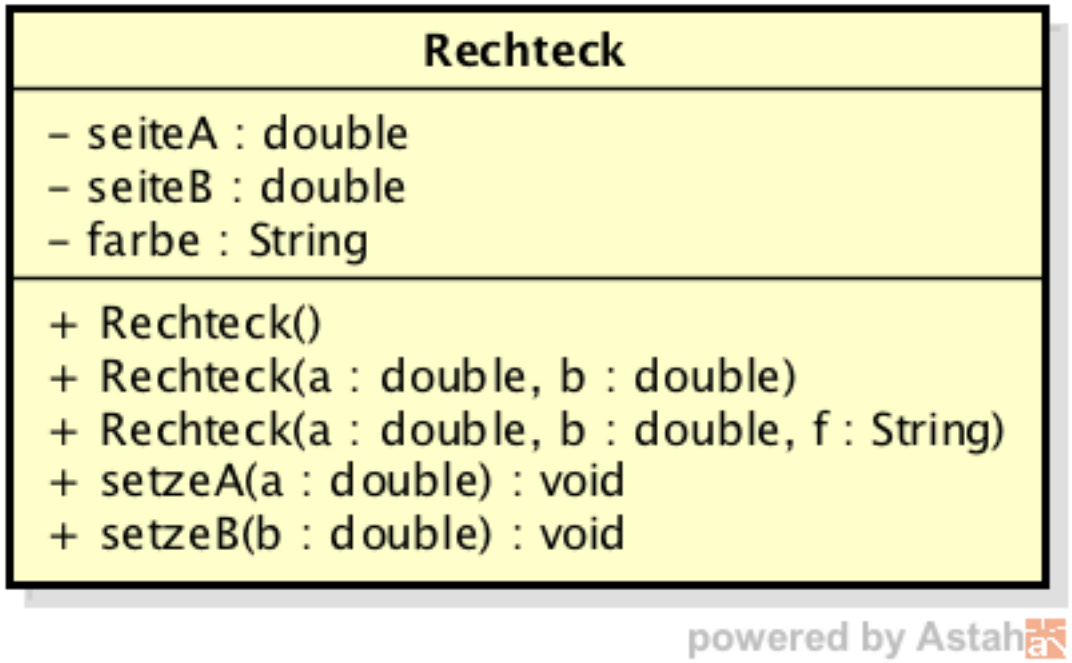
```
public class TestRechteck {
	public static void main(String[] args) {
		Rechteck r1 = new Rechteck();
	}
}
```

Mehrere Konstruktoren

Die Klasse Rechteck soll erweitert werden:

- Die Objekterzeugung ohne Werteübergabe bei Objekterzeugung soll ermöglich werden.
- Zusätzlich soll bei Objekterzeugung optional auch das Attribut „Farbe“ gesetzt werden können.

Planung



Es werden mehrere Konstruktoren eingeführt. Diese können wahlweise für die Objekterzeugung verwendet werden.
Die Konstruktoren werden hier **überladen**.

Implementierung

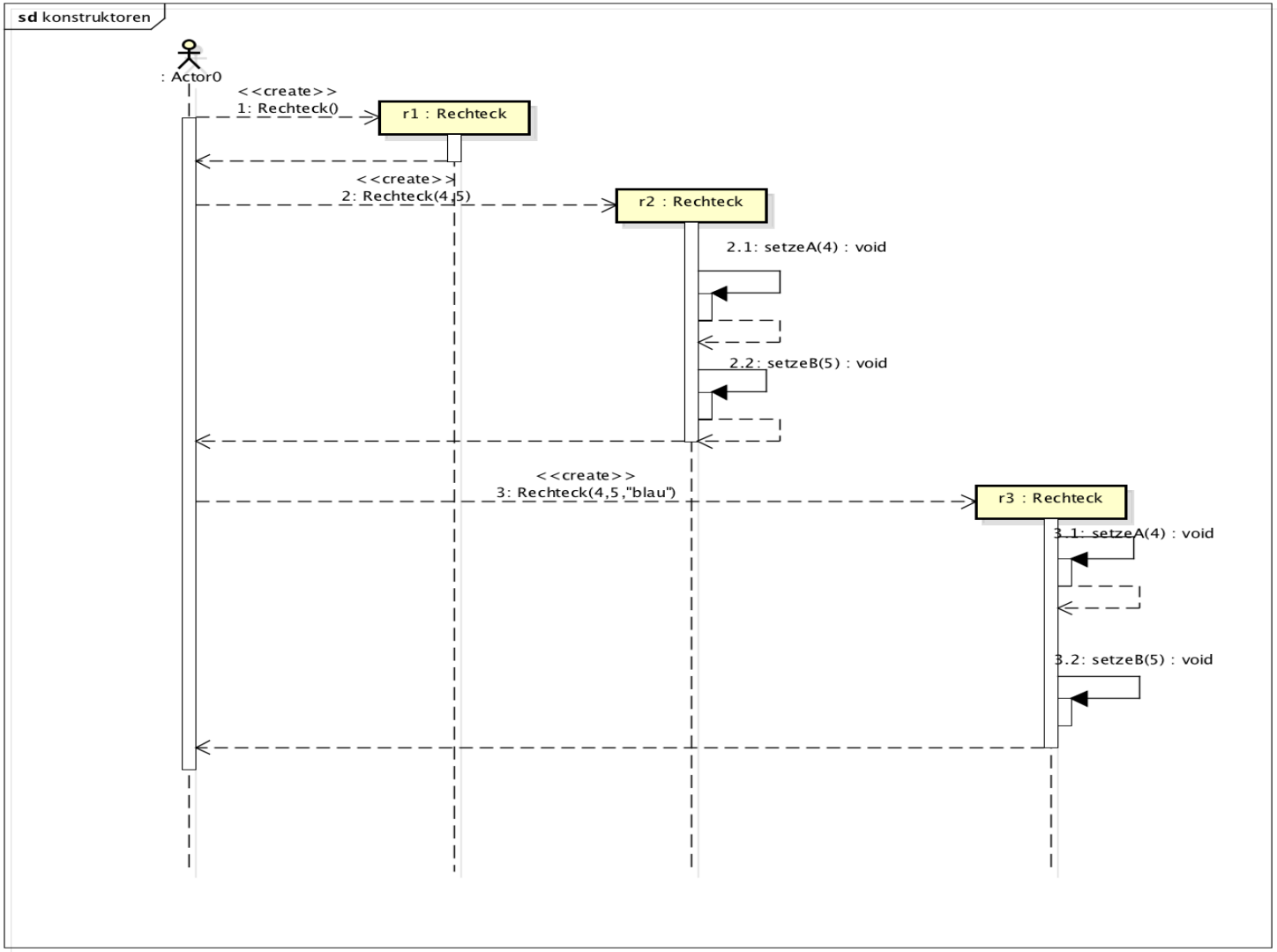
Implementierung der Klasse Rechteck:

```
public class Rechteck {

	double seiteA;
	double seiteB;
	String farbe;

	Rechteck(){}

	Rechteck(double a, double b) {
		this.setzeSeiteA(a);
		this.setzeSeiteB(b);
	}

	Rechteck(double a, double b, String farbe) {
		this.setzeSeiteA(a);
		this.setzeSeiteB(b);
		this.farbe=farbe;
	}

	public void setzeSeiteA(double a) {
		this.seiteA = a;
	}

	public void setzeSeiteB(double b) {
		this.seiteB = b;
	}

}
```

Möglichkeiten der Objekterzeugung:

```
public class TestRechteck {

	public static void main(String[] args) {

		Rechteck r1 = new Rechteck();

		Rechteck r2 = new Rechteck(4,5);

		Rechteck r3 = new Rechteck(4,5,"blau");

	}
}
```

Sequenzdiagramm



OOSE

Objekte als Parameter

Übersicht | nächste Folie >

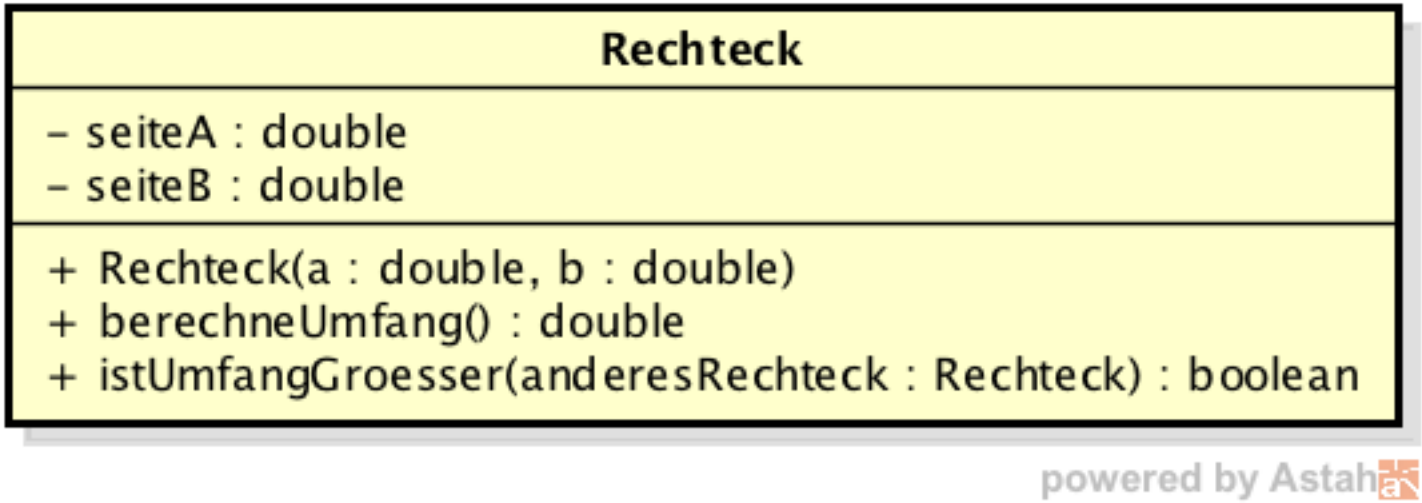
Aufgabenstellung

- Zwei Rechtecke sollen bzgl. des Umfangs verglichen werde.
- Dazu wird einem Rechteckobjekt ein anderes Rechteck-Objekt einer Methode istUmfangGrößer übergeben.
- Die Methode gibt „true“ zurück, wenn der Umfang des übergebenen Objektes größer ist, ansonsten „false“.

OOA

- Vorüberlegungen:
  - Es wird eine Methode zur Umfangsberechnung benötigt.
  - Neu: Der Methode istUmfangGrosser muss ein Objekt vom Typ Rechteck übergeben werden.

OOA/OOD



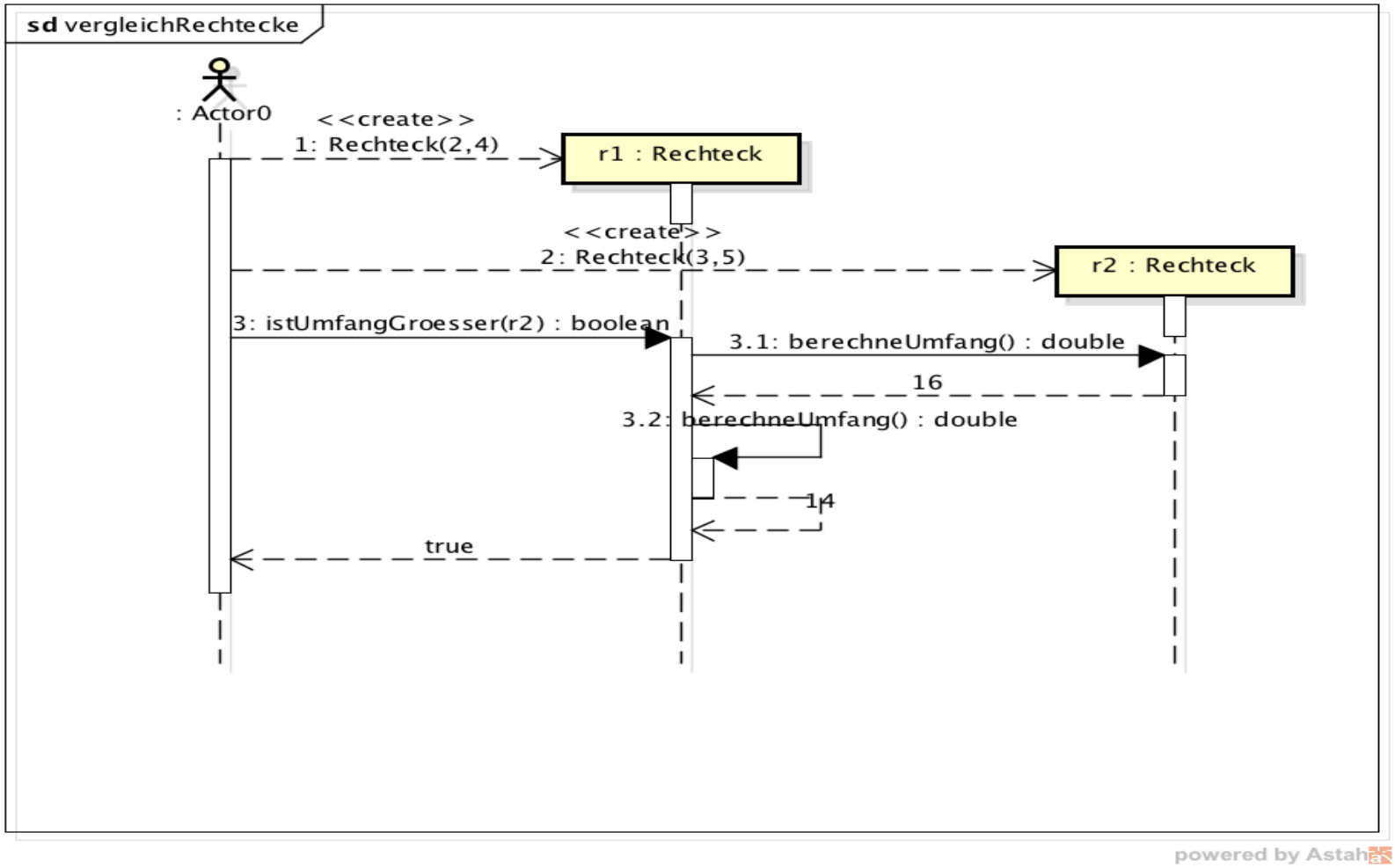
OOA/OOD

```
public class Rechteck {
        private double seiteA;
        private double seiteB;

        public Rechteck(double a, double b){
                this.seiteA=a;
                this.seiteB=b;
        }

        public double berechneUmfang(){
                return 2*this.seiteA+2*this.seiteB;
        }


        public boolean istUmfangGroesser(Rechteck anderesRechteck) {
                // Umfang ermitteln
                double umfangAnderesRechteck = anderesRechteck.berechneUmfang();
                double meinUmfang = this.berechneUmfang();

                if (umfangAnderesRechteck > meinUmfang) {
                        return true;
                } else {
                        return false;
                }
        }
}
```

OOT

```
public class testRechteck {
 public static void main(String[] args) {
       // Rechtecke erzeugen
       Rechteck r1 = new Rechteck(2,4);
       Rechteck r2 = new Rechteck(3,5);

       // Rechtecke vergleichen und Ergebnis auffangen
       boolean istGroesser = r1.istUmfangGroesser(r2);

       if (istGroesser){
         System.out.println("Umfang von R2 ist groesser");
       }else{
         System.out.println("Umfang von R2 ist nicht groesser");
       }
 }
}
```

Darstellung als Sezquenzdiagramm



Objektorientierte Softwareentwicklung

Objekte als Rückgabewert

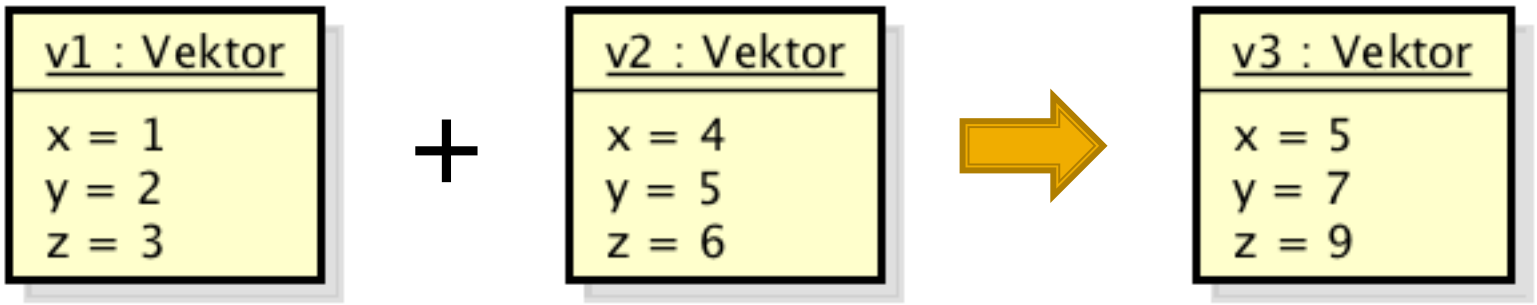
Übersicht | nächste Folie >

Problemstellung

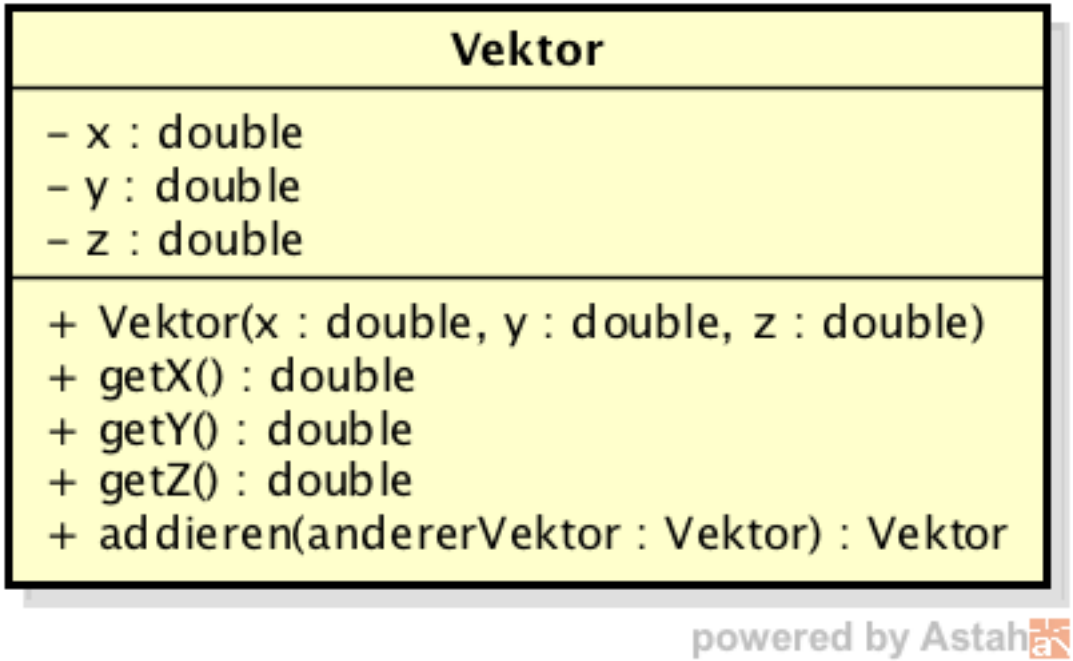
- Sie erhalten den Auftrag für eine Mathematiksoftware dreidimensionale „Vektoren“ objektorientiert zu programmieren.
- Ein Vektor soll über eine Methode „addieren“ verfügen. Der Methode wird ein anderer Vektor übergeben. Die Vektoren werden addiert und das Ergebnis als neuer Vektor zurückgegeben. Die beiden Ausgangsvektoren sollen unverändert bleiben.

Beispiel

$$\begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix} + \begin{pmatrix} 4 \\ 5 \\ 6 \end{pmatrix} = \begin{pmatrix} 5 \\ 7 \\ 9 \end{pmatrix}$$



OOA/OOD



OOP

```
public class Vektor {
        double x;
        double y;
        double z;
        public double getX() {
                return x;
        }
        public double getY() {
                return y;
        }

        public double getZ() {
                return z;
        }

        public Vektor(double x, double y, double z) {
                this.x = x;
                this.y = y;
                this.z = z;
        }
        // Fortsetzung nächste Seite
```

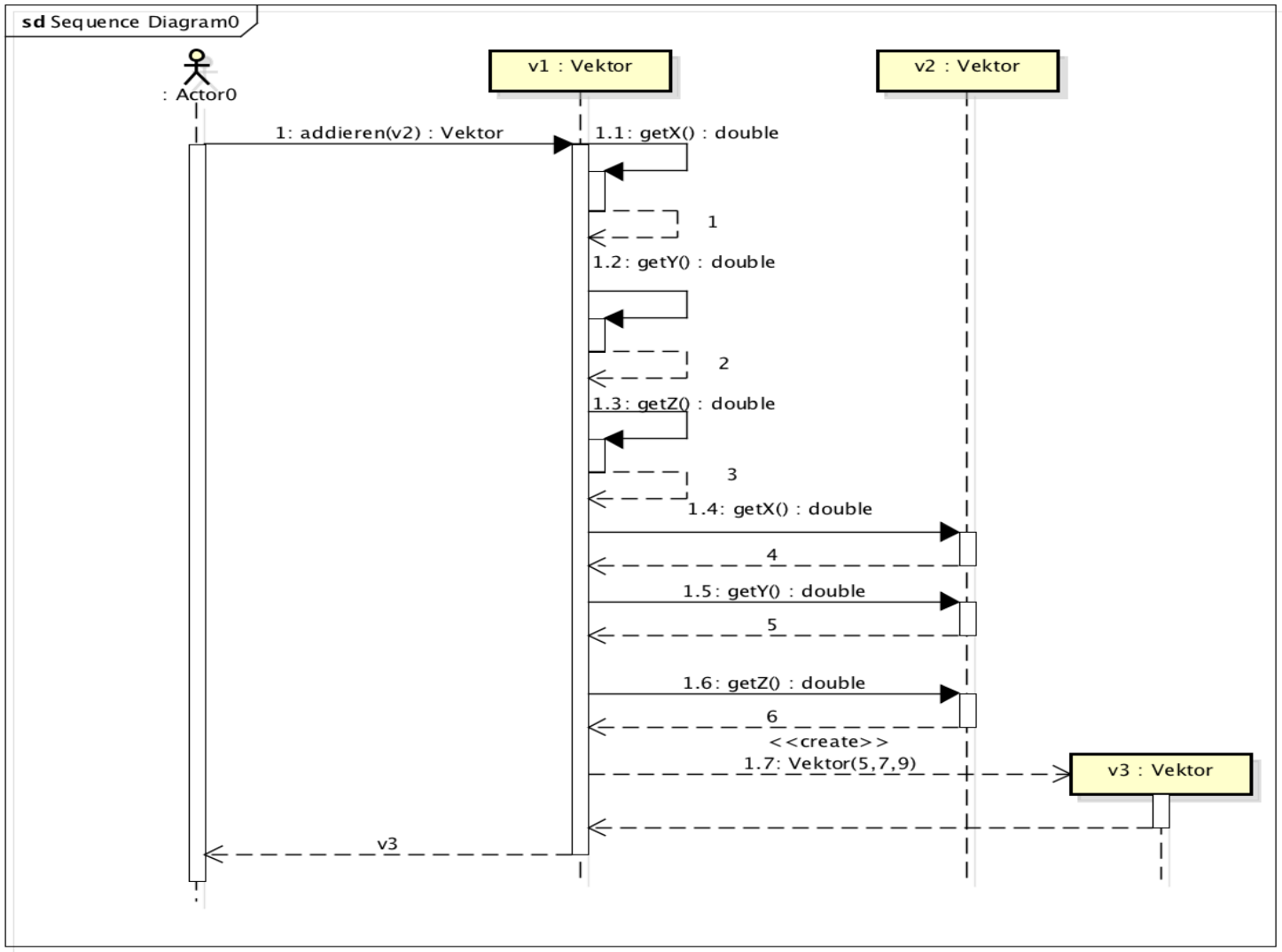
OOP

```
public Vektor addieren(Vektor andererVektor) {
            // Komponenten addieren und zwischenspeichern
            double x_sum = this.getX() + andererVektor.getX();
            double y_sum = this.getY() + andererVektor.getY();
            double z_sum = this.getZ() + andererVektor.getZ();
            // Ergebnis Vektor erzeugen
            Vektor erg = new Vektor(x_sum, y_sum, z_sum);
            // Ergebnisvektor zurueckgeben
            return erg;

      }

}
```

OOT

```
public class testVektor {

        public static void main(String[] args) {
                // zwei Vektoren erzeugen
                Vektor v1 = new Vektor(1,2,3);
                Vektor v2 = new Vektor(4, 5, 6);
                // addieren und Rueckgabe auffangen
                Vektor v3 = v1.addieren(v2);
                // Ergebnis ausgeben
                System.out.println("x:"+v3.getX());
                System.out.println("y:"+v3.getY());
                System.out.println("z:"+v3.getZ());

        }

}
```

Sequenzdiagramm



Objektorientierte Softwareentwicklung

Beziehungen zwischen Objekten I

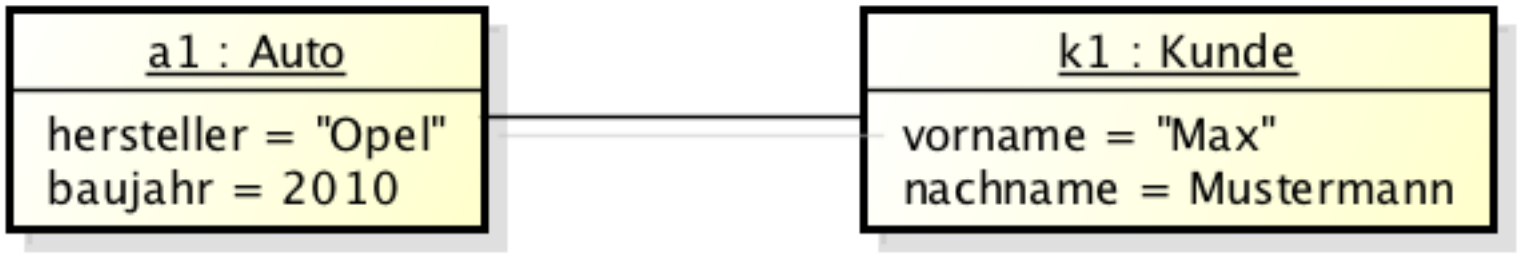
Übersicht | nächste Folie >

Problemstellung

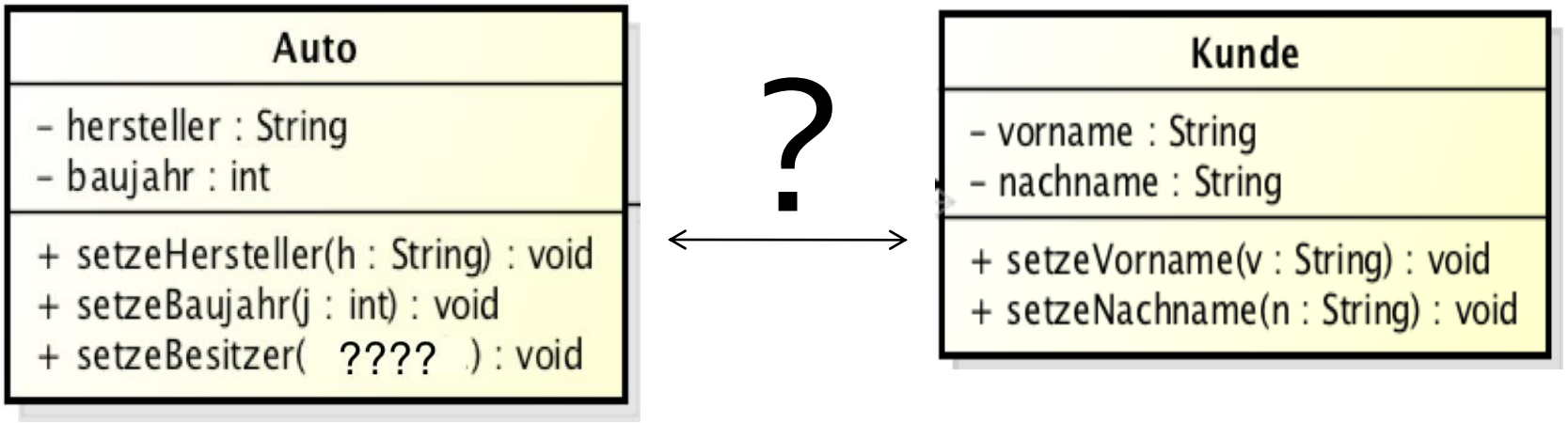
- Für ein Autohaus soll eine objektorientierte Software zur Verwaltung von Fahrzeugen und Kunden entwickelt werden.
- Jedem Auto kann dabei ein Kunde als Besitzer zugeordnet werden (unverkaufte Autos haben keinen offiziellen Besitzer).

OOA/OOD: Bespielobjekte

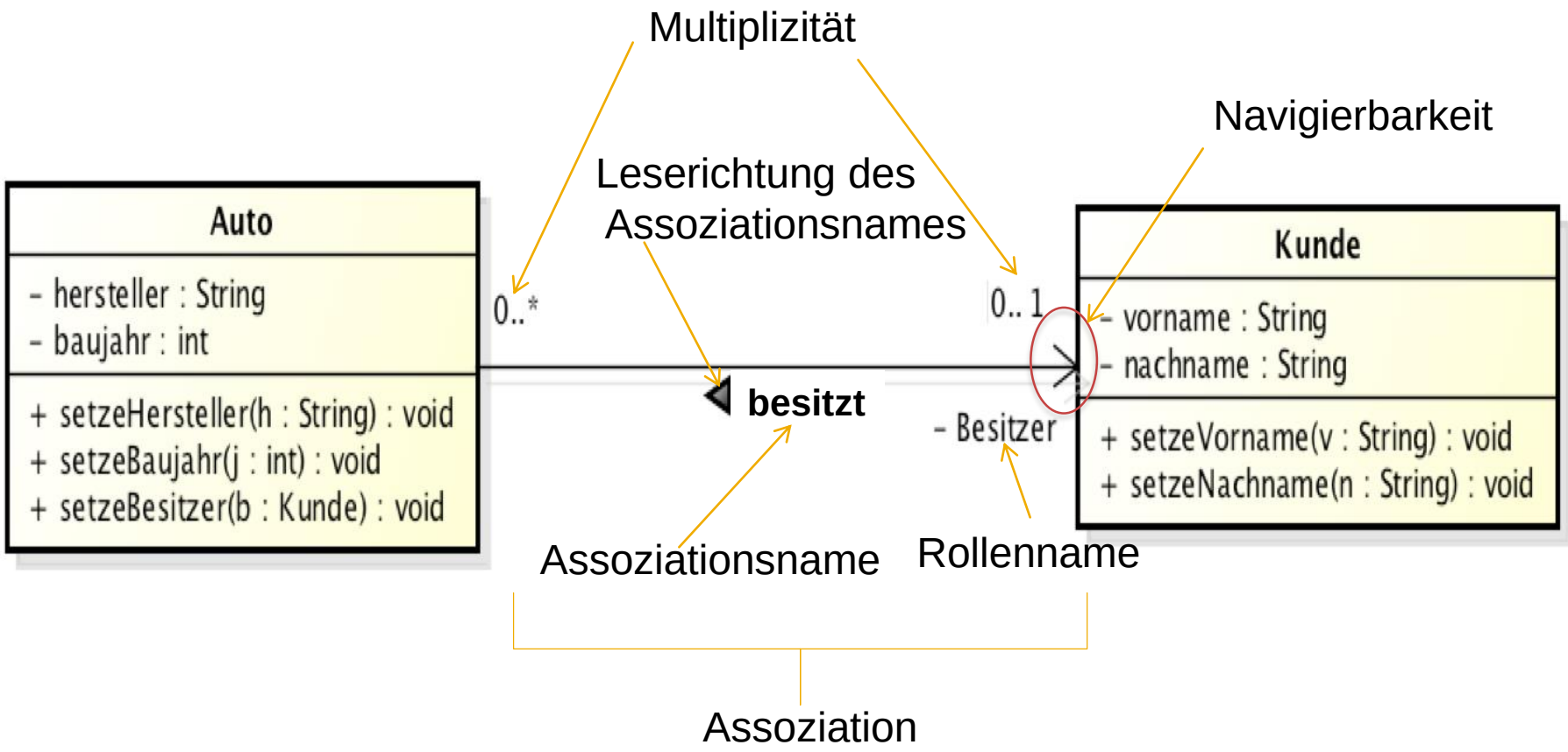
Beispiel: Der Kunde „Max Mustermann“ steht mit einem Opel Baujahr 2010 in Beziehung:



OOA/OOD: Klassen in UML



OOA/OOD: Assoziationen in UML



Anmerkung: Die Multiplizität sagt aus, mit wie vielen anderen Objekten ein Objekt in Beziehung stehen kann. Die Kardinalität sagt aus, mit wie vielen es wirklich in Beziehung steht (macht also nur in Objektdiagrammen Sinn).

OOP: Umsetzung in Java


```
public class Auto {

 private String hersteller;
 private int baujahr;

 private Kunde Besitzer;

 public void setzeHersteller(String h) {
          this.hersteller=h;
 }

 public void setzeBaujahr(int j) {
          this.baujahr=j;
 }

 public void setzeBesitzer(Kunde b) {
          this.Besitzer=b;
 }
}
```

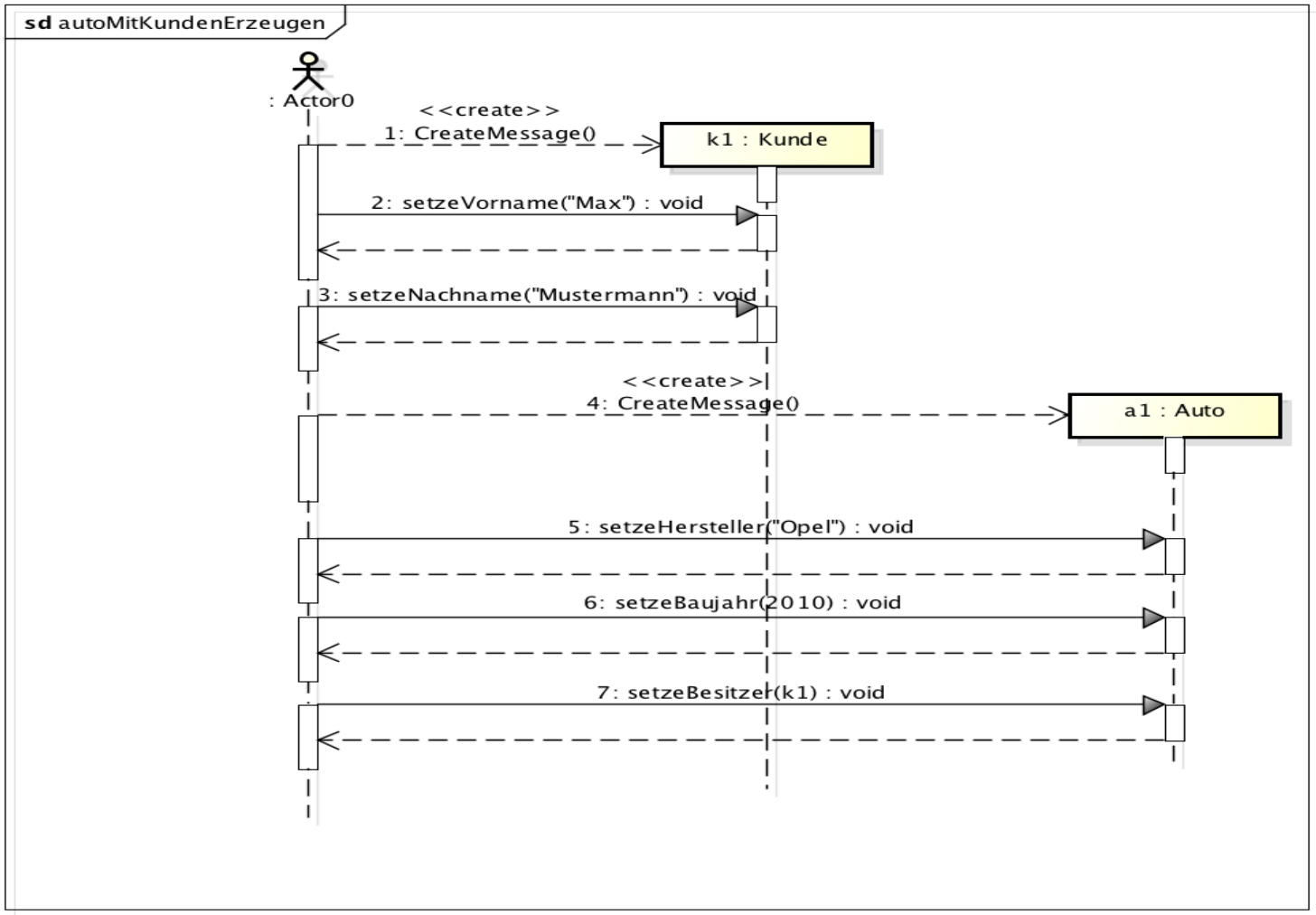
```
public class Kunde {

 private String vorname;

 private String nachname;

 public void setzeVorname(String v) {
          this.nachname=v;
 }

 public void setzeNachname(String n) {
          this.nachname=n;
 }

}
```

Anmerkung: welche Auswirkung hätte die Multiplizität 1 statt 0..1 für die Umsetzung?

OOP: Umsetzung in Java

```
public class test {
        public static void main(String[] args) {
                // Kunde erzeugen
                Kunde k1 = new Kunde();
                k1.setzeVorname("Max");
                k1.setzeNachname("Mustermann");

                // Auto erzeugen
                Auto a1 = new Auto();
                a1.setzeHersteller("Opel");
                a1.setzeBaujahr(2010);

                // Dem Auto einen Kunden zuordnen
                a1.setzeBesitzer(k1);

        }
}
```

Sequenzdiagramm



Objektorientierte Softwareentwicklung

Beziehungen zwischen Objekten II

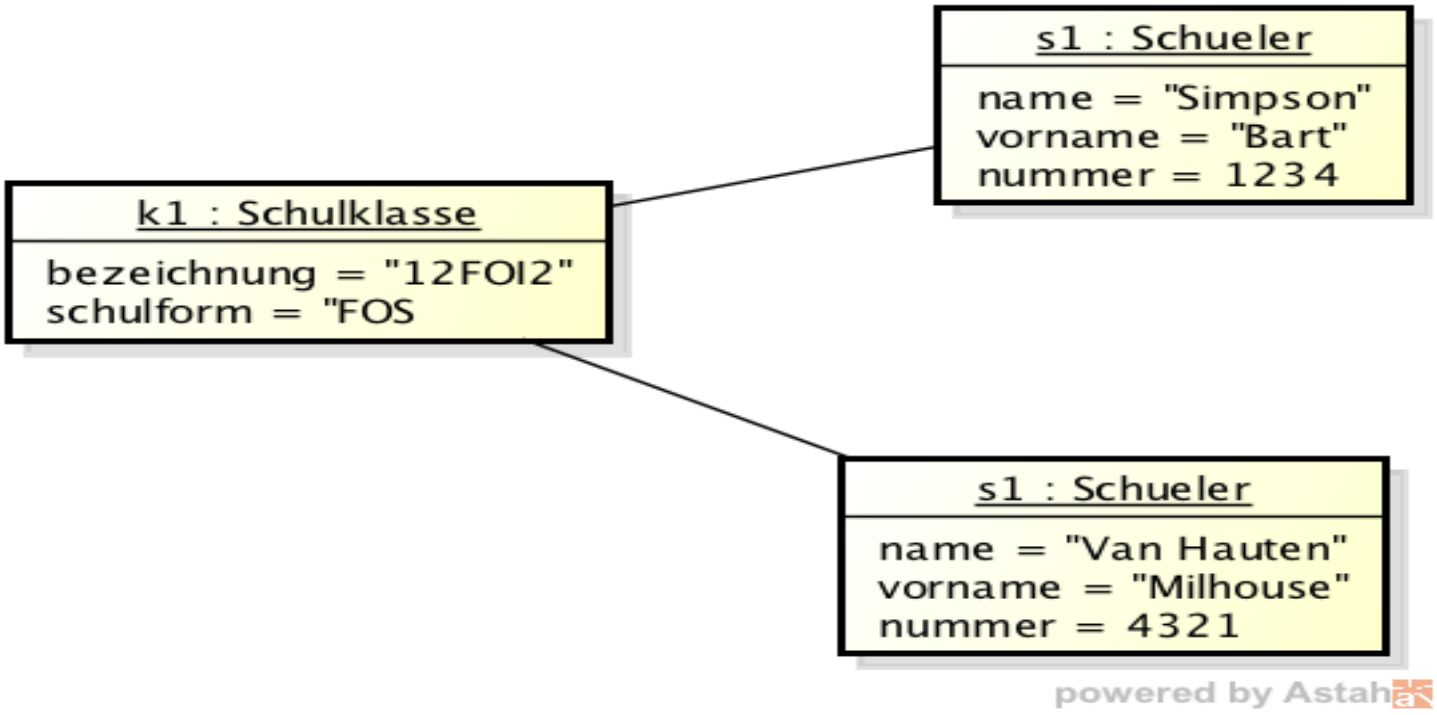
Übersicht | nächste Folie >

Problemstellung

- Es soll eine objektorientierte Software zur Verwaltung von Schülern und Schulklassen entwickelt werden.
- Sowohl Schüler als auch Schulklassen sollen in der Software als Objekte repräsentiert werden.
- Die Schulklasse soll sich ihre Schüler „merken“. Dazu soll diese über eine Methode zum Hinzufügen eines Schülers und eine Methode zum Anzeigen aller Schüler verfügen.

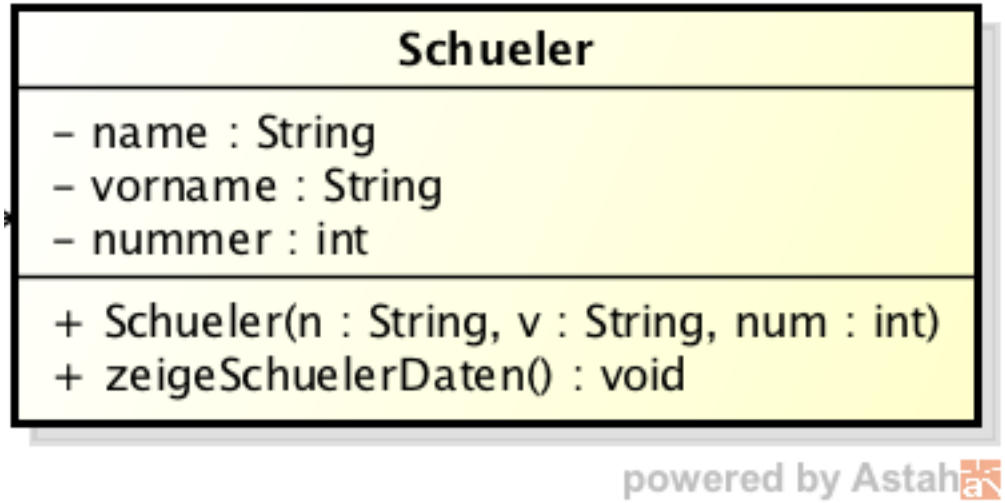
OOA/OOD

- Testbeispiel-Objektstruktur:



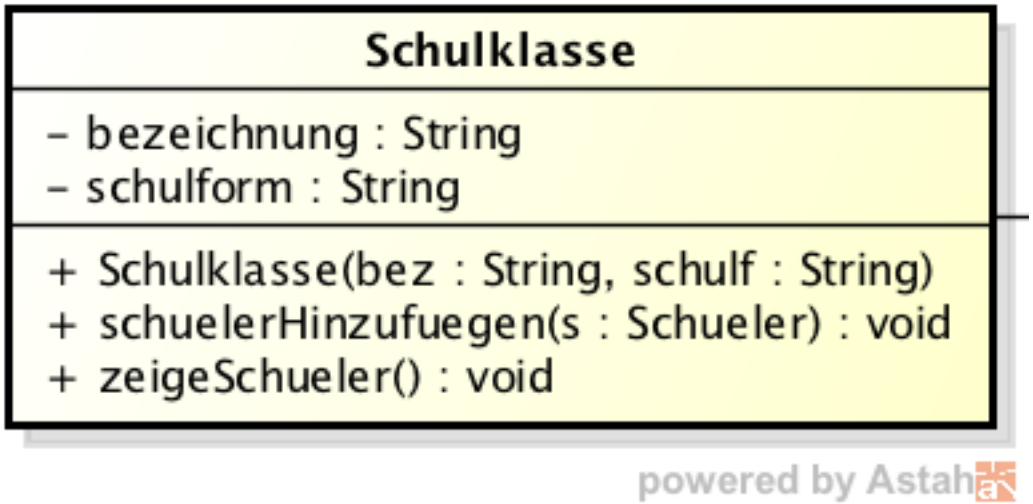
OOA/OOD: Planung

- Darstellung eines Schülers:



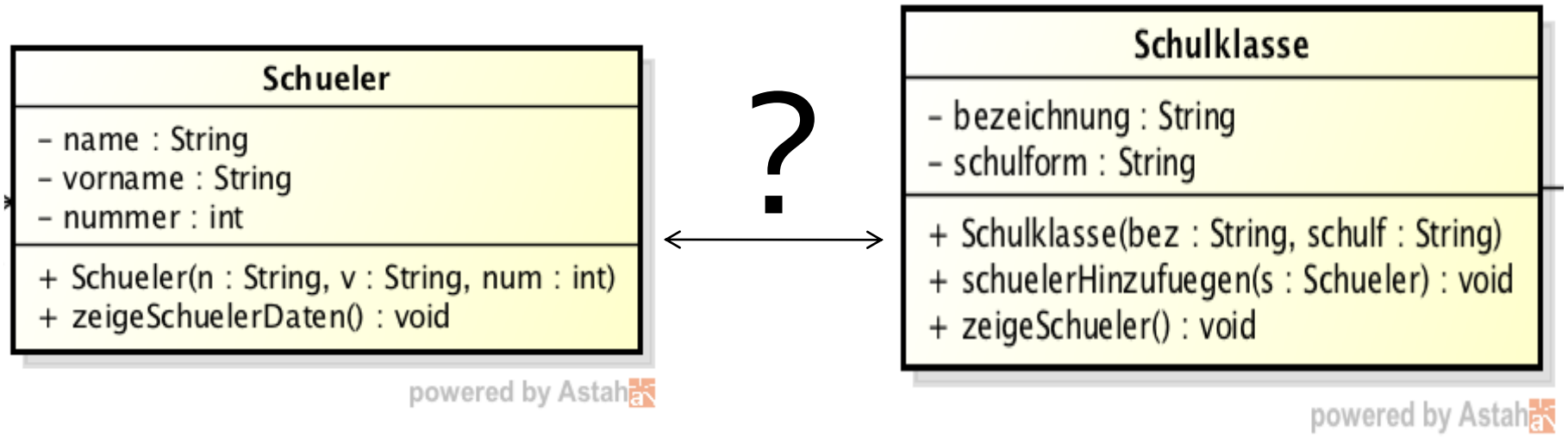
OOA/OOD

- Darstellung einer Schulklasse:

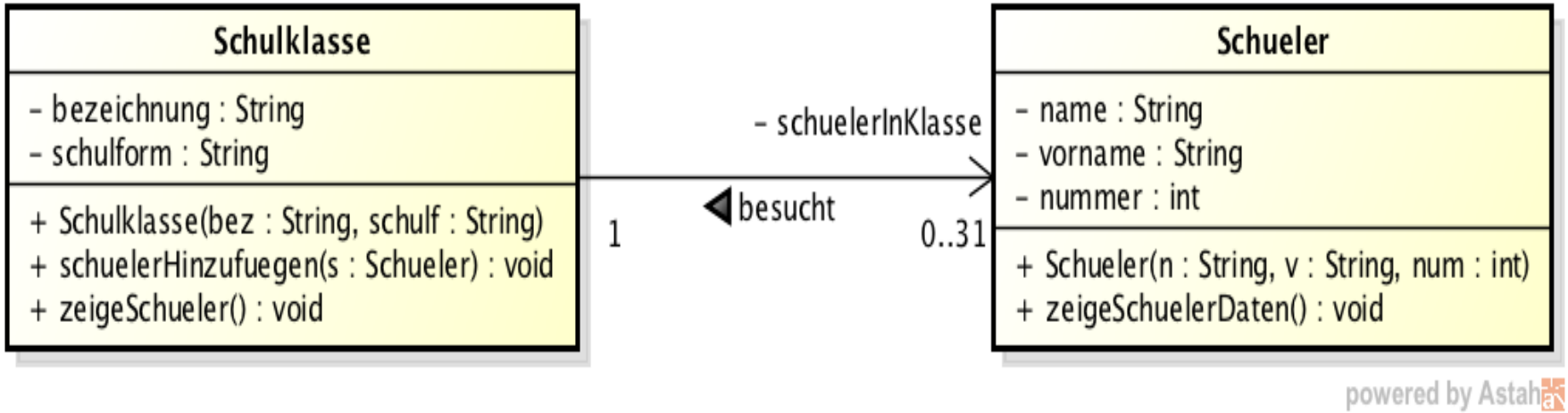


OOA/OOD

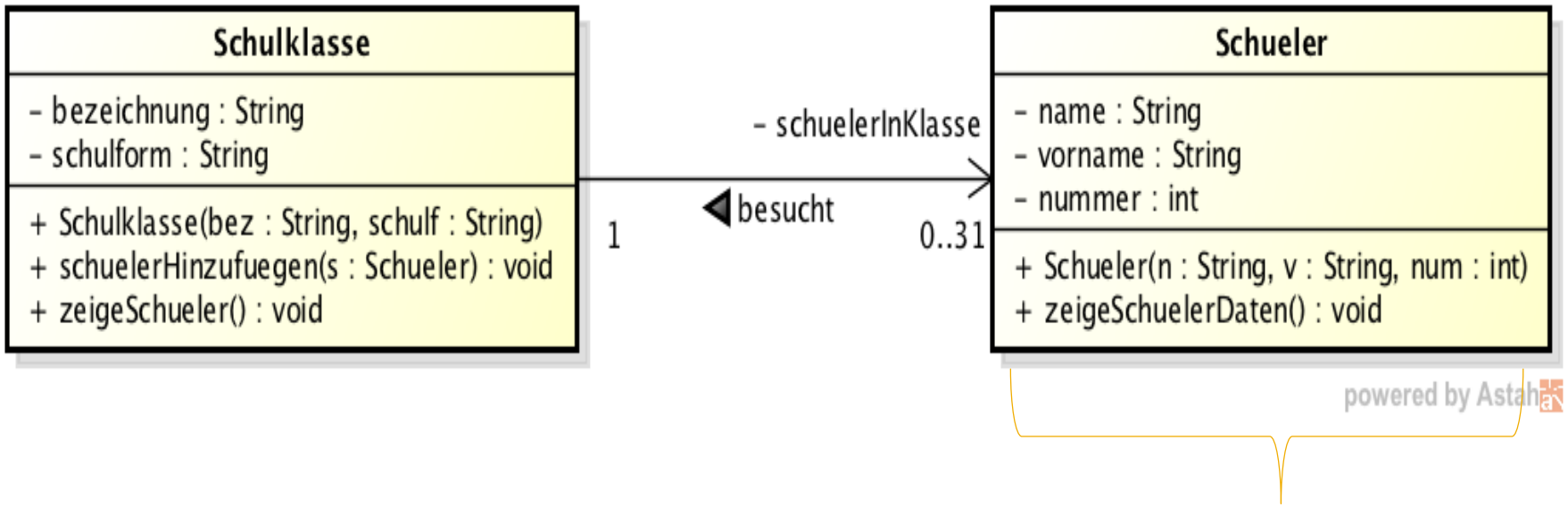
- Beziehung zwischen Schulklasse und Schüler



OOA/OOD



OOD: Überlegungen zu Schüler

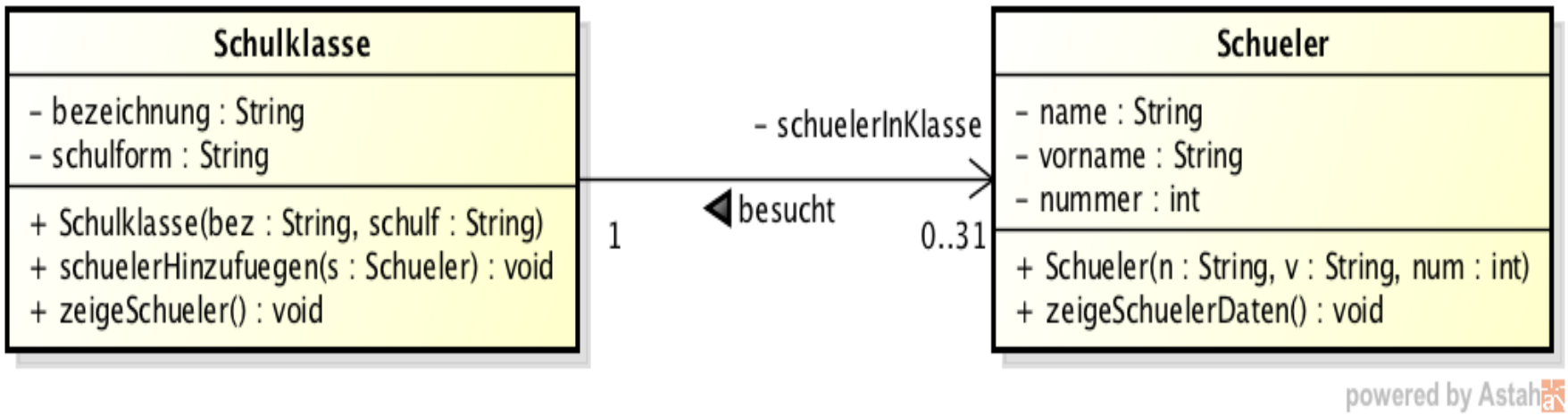


Für die Klasse „Schueler“ gibt es keine Besonderheiten.

Sie wird wie gehabt implementiert.

OOD: Überlegungen zu Schulklasse

Problem: Die Klasse „Schulklasse“ muss sich **viele** Schüler „merken“.

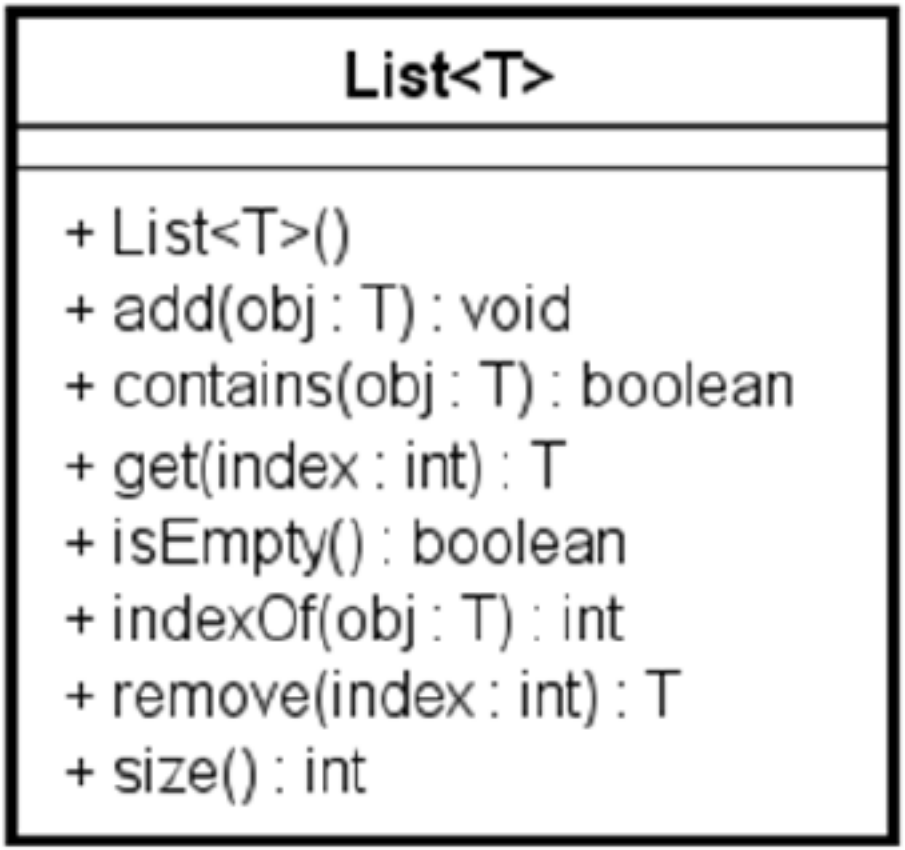


Lösung: Die Klasse „Schulklasse“ merkt sich in einem **Container** alle Schüler.

OOD: Umsetzung der Beziehung

- Die Klasse „Schulklasse“ braucht eine Struktur, in dem sie alle Schüler speichern kann.
- Klassen, die solche Fähigkeiten haben, werden “Container-Klassen“ genannt.
- In JAVA gibt es dazu ein Interface „List“, welches durch die Klasse „ArrayList“ implementiert wird.

Schnittstelle Container List/Arraylist



Anmerkung:
List ist nur ein Interface. Eine Implementierung ist z.B. ArrayList.

Die genaue Verwendung ist der JAVA-Dokumentation zu entnehmen.

OOP: Implementierung Schüler

```
public class Schueler {
	private String name;
	private String vorname;
	private int nummer;

	public Schueler(String n, String v, int num) {
		this.name = n;
		this.vorname = v;
		this.nummer = num;
	}

	public void zeigeSchuelerdaten() {
		System.out.println("Name: " + this.name);
		System.out.println("vorname:" + this.vorname);
		System.out.println("Nummer: " + this.nummer);
	}
}
```

OOP: Implementierung Schulklasse

Container für Schüler

Container erzeugen

Schüler in Array einfügen

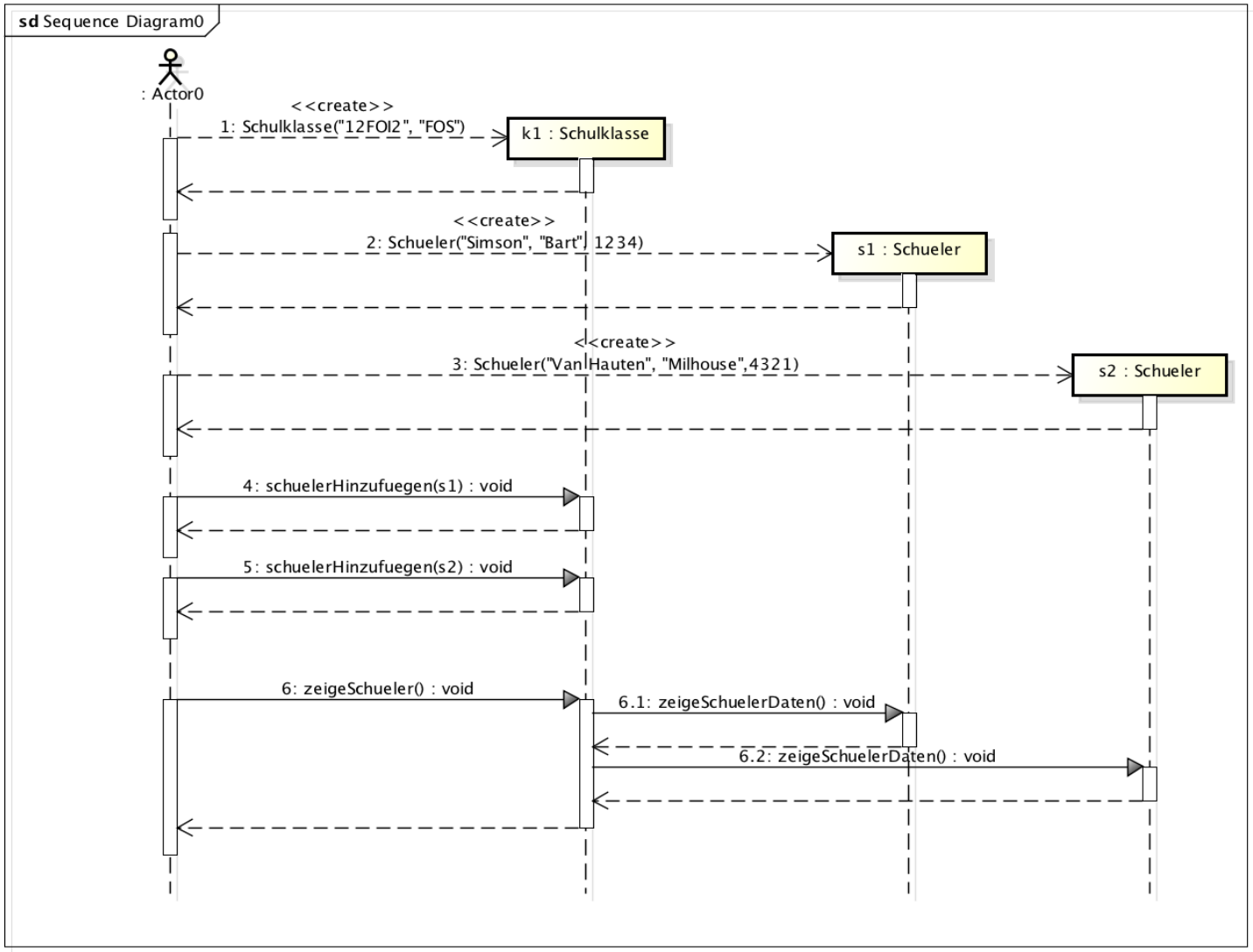
Alle Schüler im Array der Reihe nach ausgeben.

```
import java.util.*;

public class Schulklasse {
        private String bezeichnung;
        private String schulform;
        private List<Schueler> schuelerInKlasse;

        public Schulklasse(String bez, String schulf) {
                this.bezeichnung = bez;
                this.schulform = schulf;
                schuelerInKlasse=new ArrayList<Schueler>();
        }
        public void schuelerHinzufuegen(Schueler s) {
                if (schuelerInKlasse.size()<31){
                        schuelerInKlasse.add(s);
                } // hier ggf. Fehlermeldung , wenn Klasse
        }
        public void zeigeSchueler() {
                System.out.println("Schueler in Klasse"+bezeichnung+":");
                for (int i = 0; i < schuelerInKlasse.size(); i++) {
                        Schueler s = schuelerInKlasse.get(i);
                        s.zeigeSchuelerdaten();
                        System.out.println("--------------");
                }
        }
}
```

Objekterzeugung und Test

```
public class Testklasse {

        public static void main(String[] args) {

                // Klasse erzeugen
                Schulklasse k1 = new Schulklasse("12FOI2","FOS");

                // Schueler erzeugen
                Schueler s1 = new Schueler("Simpson", "Bart", 1234);
                Schueler s2 = new Schueler("Van Hauten", "Milhouse", 4321);

                // Schueler der Klasse zuweisen
                k1.schuelerHinzufuegen(s1);
                k1.schuelerHinzufuegen(s2);

                // Klasse inkl. Schueler ausgeben
                k1.zeigeSchueler();

        }

}
```

Sequenzdiagramm



Anmerkung:
Die Kommunikation mit dem Container-Objekt ist nicht dargestellt.

Objektorientierte Softwareentwicklung

Übersicht Beziehungstypen

Übersicht | nächste Folie >

Assoziation

Assoziation:

Klasse A und B stehen „locker“ in Beziehung. Es ist keine Aussage darüber getroffen „wer wen kennt“.

Assoziation mit Navigierbarkeit (in eine Richtung):

Klasse A und B stehen in Beziehung. Dabei kennt / benutzt Klasse A Klasse B.
Umsetzung: Referenz auf Klasse B in Klasse A.

Assoziation

Klasse A und Klasse B stehen in Beziehung.
Klasse A kennt / benutzt Klasse B und umgekehrt (Navigierbarkeit geht in beide Richtungen).
Umsetzung: Die Klassen referenzieren sich gegenseitig.

Klasse A und Klasse B stehen in Beziehung.
Klasse A kennt / benutzt Klasse B. Von Klasse B ist Klasse A aber nicht erreichbar (d.h. keine Referenz in diese Richtung).

Aggregation und Komposition

Aggregation:

Komposition:

Aggregationen bzw. Kompositionen sind spezielle Assoziationen, die **"Teile/Ganzes"-Beziehung** beschreiben.

Das "Teil“ des "Ganzen“ kann bei der Aggregation auch einzeln existieren, bei der Komposition nur, wenn auch das "Ganze" existiert. Das "Ganze" befindet sich auf der Seite der Raute.

Bei der Implementierung werden Aggregationen und Kompositionen häufig nicht anders als normale Assoziationen behandelt.

Beispiele

Beispiel für eine Aggregation:

Die Räder sind Teil von einem Auto und können auch ohne das Auto existieren.

Beispiel für eine Komposition:

Ein Gehirn ist ein Teil von einem Menschen.
Ohne den Menschen kann das Gehirn aber nicht existieren.

Objektorientierte Softwareentwicklung

Vererbung

Übersicht | nächste Folie >

Problemstellung

- Für einen Betrieb ist eine Mitarbeiterverwaltung zu entwickeln.
- In dem Betrieb gibt es zwei Arten Mitarbeiter: Angestellte und Arbeiter
- Angestellte erhalten einen Basislohn und eine Provision, während Arbeiter einen Stundenlohn erhalten.

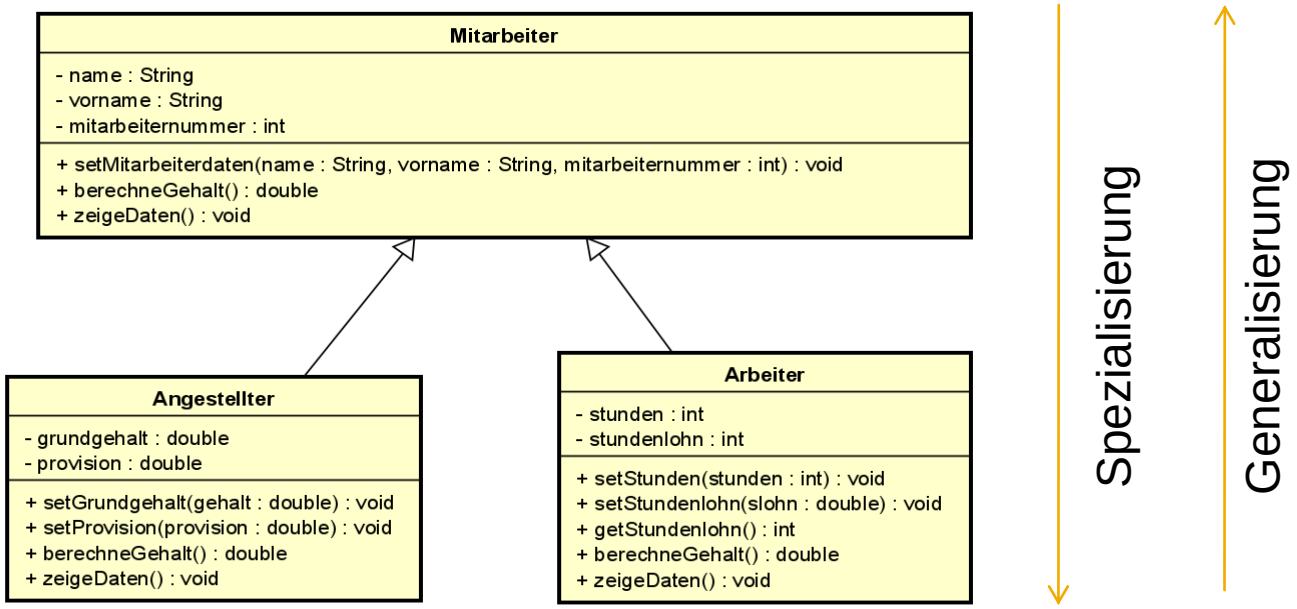
OOA

- Angestellte und Arbeiter haben als Mitarbeiter gemeinsame Eigenschaften (Name, Vorname,..) und Methoden.
- Zusätzlich haben die Mitarbeiter spezielle Eigenschaften und Methoden, die sich voneinander unterscheiden je nachdem, ob die Angestellte oder Arbeiter sind.
- Idee: Gemeinsamkeiten zusammenfassen und nach Unterschieden spezialisieren.

OOA/OOP: UML-Klassendiagramm

Gemeinsamkeiten

Unterschiede



Alle Angestellte und Arbeiter haben / „erben“ alle Eigenschaften und Methoden von „Mitarbeitern“ und erweitern diese entsprechend. Erben können jedoch nicht direkt auf geerbte private-Attribute zugreifen. Der Zugriff erfolgt über die set/get-Methoden. „berechneGehalt“ und „zeigeDaten“ tauchen erneut in den Erben auf, da ihr Verhalten angepasst („überschrieben“) werden muss.

OOP

```
public class Mitarbeiter {
	private String name;
	private String vorname;
	private int mitarbeiternummer;

	public void setMitarbeiterdaten(String name, String vorname, int mitarbeiternummer) {
		this.name = name;
		this.vorname = vorname;
		this.mitarbeiternummer = mitarbeiternummer;

	}

	public double berechneGehalt() {

		// an dieser Stelle kann kein Gehalt berechnet werden
		return 0;

	}

	public void zeigeDaten() {
		System.out.println("vorname:" + this.vorname);
		System.out.println("Nachname:" + this.name);
		System.out.println("Mitarbeiternummer:" + this.mitarbeiternummer);
	}
}
```

OOP

```
public class Angestellter extends Mitarbeiter {
        private double grundgehalt;
        private double provision;

        public void setGrundgehalt(double gehalt) {
                this.grundgehalt = gehalt;
        }

        void setProvision(double provision) {
                this.provision = provision;
        }

        // Methode überschreiben
        @Override public double berechneGehalt() {
                return this.grundgehalt + this.provision;
        }

        // Methode überschreiben
        @Override public void zeigeDaten() {
                // Name, Vorname und Mitarbeiternummer ausgeben
                super.zeigeDaten();
                // spezielle Daten ausgeben
                System.out.println("Ich bin ein Angestellter!");
                System.out.println("Grundgehalt: " + this.grundgehalt);
                System.out.println("Provision: " + this.provision);
        }
}
```

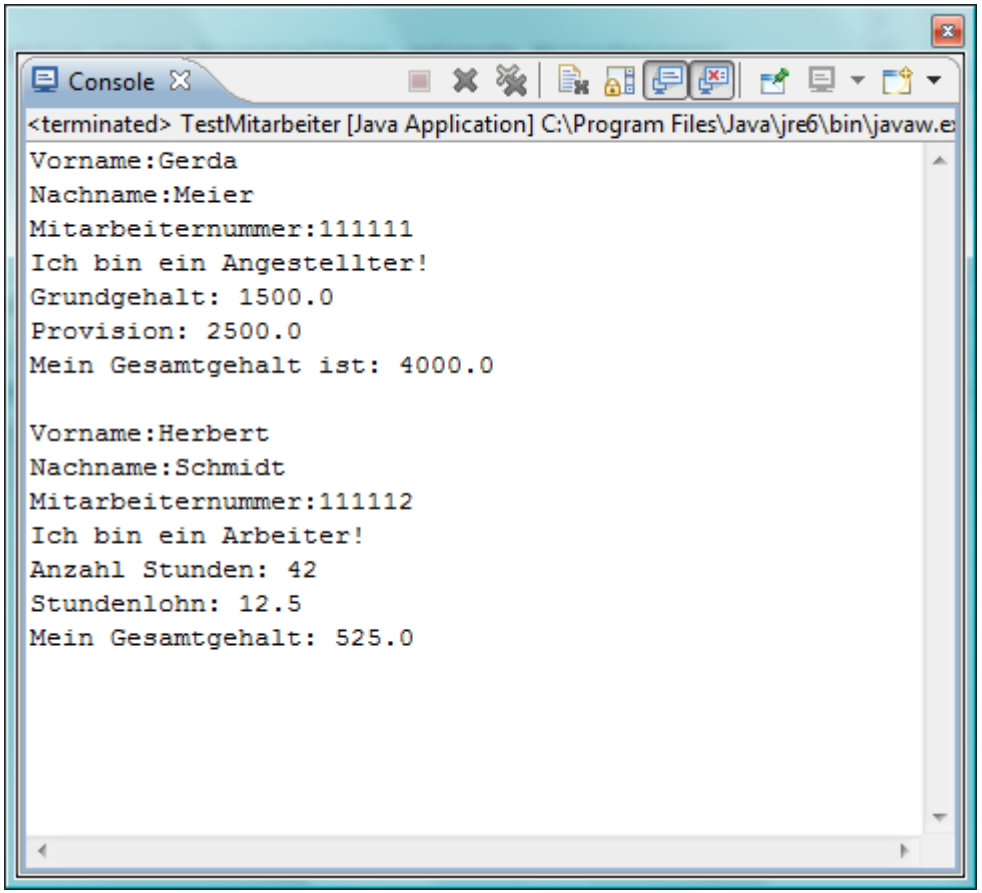
OOP

```
public class Arbeiter extends Mitarbeiter {
        private int stunden;
        private double stundenlohn;

        public void setStunden(int stunden) {
                this.stunden = stunden;
        }

        void setStundenlohn(double slohn) {
                this.stundenlohn = slohn;
        }

        // Methode überschreiben
        @Override public double berechneGehalt() {
                return this.stundenlohn * this.stunden;
        }

        // Methode überschreiben
        @Override public void zeigeDaten() {
                // Name, vorname und Mitarbeiternummer ausgeben
                super.zeigeDaten();
                // spezielle Daten ausgeben
                System.out.println("Ich bin ein Arbeiter!");
                System.out.println("Anzahl Stunden: " + this.stunden);
                System.out.println("Stundenlohn: " + this.stundenlohn);
        }
}
```

OOT

```
public class TestMitarbeiter {
          public static void mainn(String[] args) {
                    Angestellter an1 = new Angestellter();
                    an1.setMitarbeiterdaten("Meier", "Gerda", 111111);
                    an1.setGrundgehalt(1500);
                    an1.setProvision(2500);
                    an1.zeigeDaten();

                    System.out.println("Mein Gesamtgehalt ist:" + an1.berechneGehalt());
                    System.out.println("");

                    Arbeiter ar1 = new Arbeiter();

                    ar1.setMitarbeiterdaten("Schmidt", "Herbert", 111112);
                    ar1.setStunden(42);
                    ar1.setStundenlohn(12.50);
                    ar1.zeigeDaten();

                    System.out.println("Mein Gesamtgehalt: " + ar1.berechneGehalt());
                    System.out.println("");

          }
}
```

OOT


Console
<terminated> TestMitarbeiter [Java Application] C:\Program Files\Java\jre6\bin\javaw.e
Vorname:Gerda
Nachname:Meier
Mitarbeiternummer:111111
Ich bin ein Angestellter!
Grundgehalt: 1500.0
Provision: 2500.0
Mein Gesamtgehalt ist: 4000.0
Vorname:Herbert
Nachname:Schmidt
Mitarbeiternummer:111112
Ich bin ein Arbeiter!
Anzahl Stunden: 42
Stundenlohn: 12.5
Mein Gesamtgehalt: 525.0

Zusammenfassung / Begriffe

- Die Klasse, von der geerbt wird, nennt man **Oberklasse**. Die erbende heißt **Unterklasse**.
- Die Methoden „zeigeDaten“ und „berechneGehalt“ werden von der Unterklasse **überschrieben (Override)**.
- Methoden, die überschrieben werden sollen, nennt man auch „virtuelle Methoden“.
- Durch das Überschreiben **verhalten** sich die Methoden von Mitarbeitern **unterschiedlich.**

Zusammenfassung / Begriffe

- Die Vererbung hat zur Folge, dass **ein Objekt** sich bzgl. seinen Eigenschaften und seinem Verhalten (Methoden) **unterschiedlich verhalten kann**.
- Z. B. kann ein Arbeiter-Objekt sowohl die **Gestalt** als Arbeiter als auch als Mitarbeiter annehmen und sich entsprechend verhalten.
- Die Möglichkeit des unterschiedlichen Verhaltens bzw. „Gestaltannehmens“ wird **Polymorphie** (Vielgestaltigkeit) genannt.

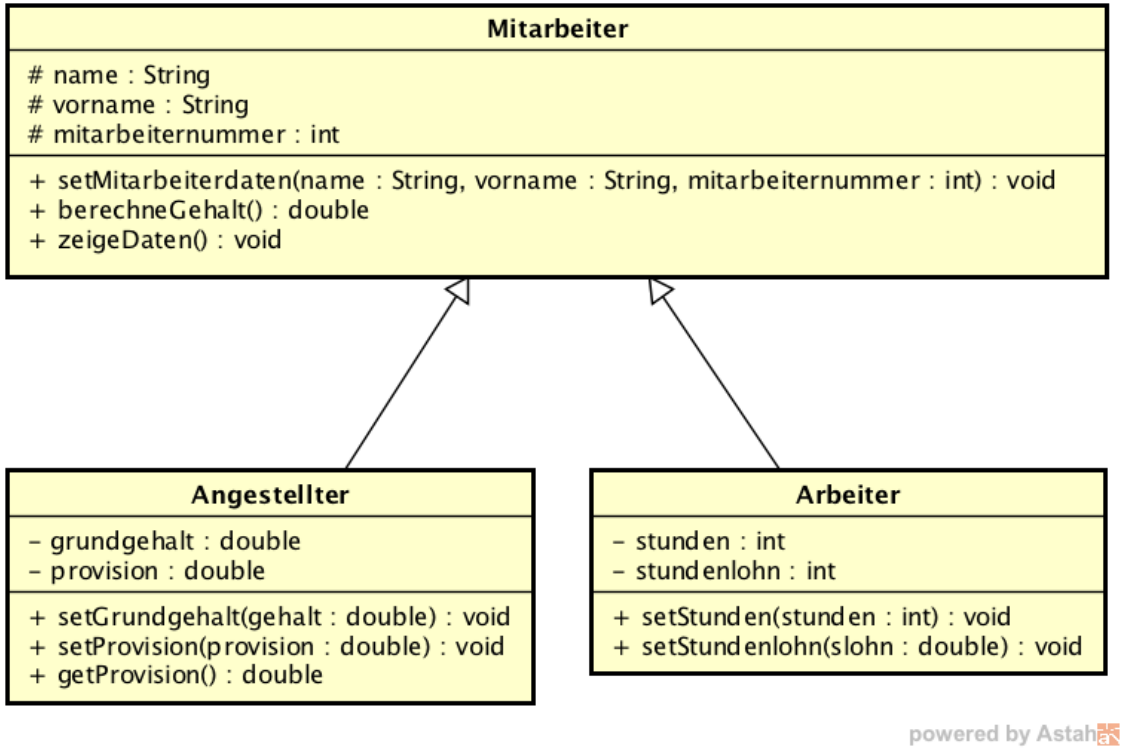
Zusammenfassung / Begriffe

- Oft wird erst zur Laufzeit eines Programmes entschieden, welches Verhalten ein Objekt konkret hat; dies nennt man „late-Binding“.
- Durch die Polymorphie-Eigenschaft kann man z.B. Objekte unterschiedlichen-Typs (aber mit gleichen „Vorfahren“ in einer Erbhierarchie) in einer Liste verwalten.

Beispiel: Polymorohie

```
// Angestellten erzeugen
Angestellter an1 = new Angestellter();
an1.setMitarbeiterdaten("Meier", "Gerda", 111111);
an1.setGrundgehalt(1500);
an1.setProvision(2500);


// Arbeiter umwandeln als Mitarbeiter
Mitarbeiter m1 = (Mitarbeiter) an1;
// Der Angestellte hat seine Gestalt geaendert und
// und tritt jetzt als Mitarbeiter „auf“
double = m1.getProvision(); // nicht mehr moeglich
// zurück wandeln:
Angestellter ang = (Angestellter) m1;
double = ang.getProvision(); // geht!

// auf dasselbe Objekt im Speicher wird mit drei
// Unterschiedlichen Bezeichnern zugegriffen (an1, m1, ang)
// je nach „Gestalt“ m1 bzw. an1, ang verhält
// es sich aber Unterschiedlich
```

Ergänzung 1: protected statt private

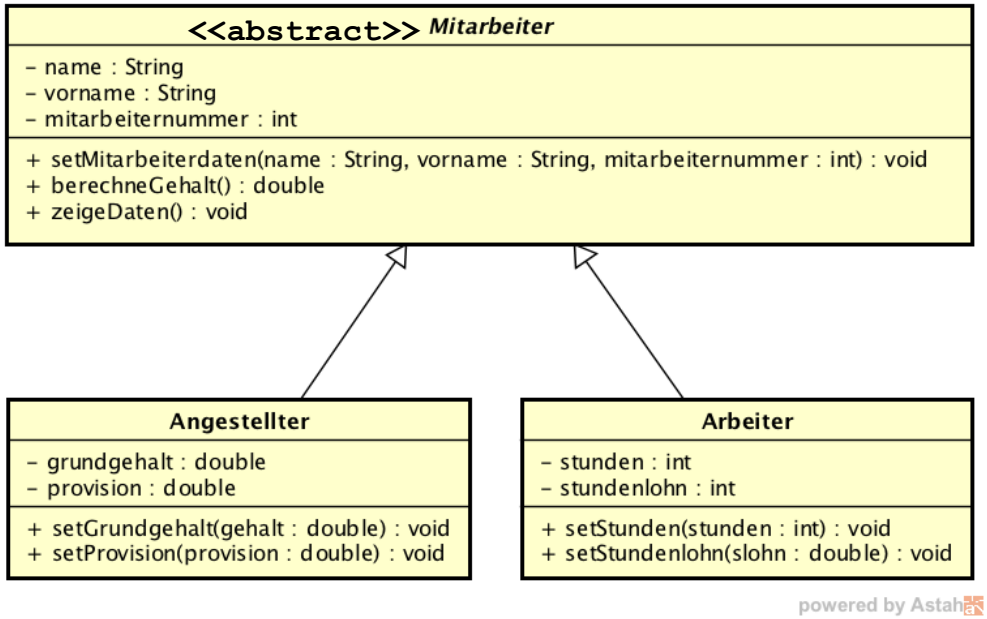


Möchte man, dass Erben doch direkt auf die Attribute zugreifen können, will man diese aber nicht generell nach außen freigegeben, kann man die Attribute „protected“ setzen (Kennzeichnung # in UML).

Ergänzung 2: Abstrakte Klassen

- Ein Objekt vom Typ „Mitarbeiter“ zu erzeugen, ist nicht sinnvoll (Gehalt kann nicht berechnet werden).
- Wenn man dies verhindert will, kann man eine Klasse als „abstract“ kennzeichnen.

Ergänzung 2: Abstrakte Klassen




```
public abstract class Mitarbeiter(){
  ...
}
```

main:

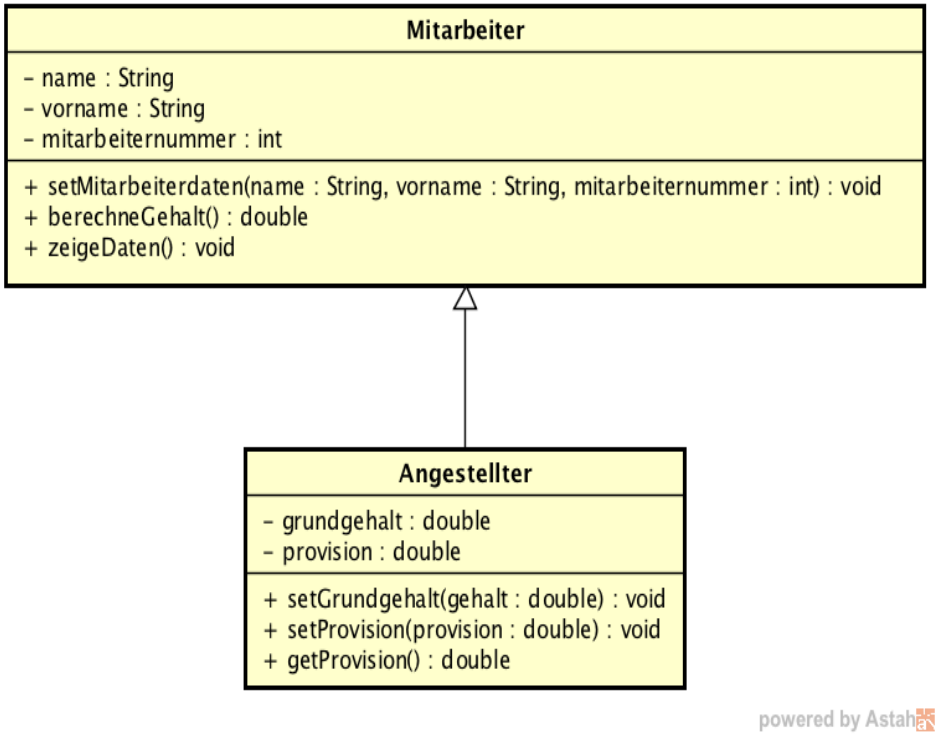
```
// nicht mehr möglich
Mitarbeiter m1 = new Mitarbeiter();
// weiterhin möglich
Angesteller a1 = new Angestellter();
```

Objektorientierte Softwareentwicklung

Vererbung und Konstruktoren

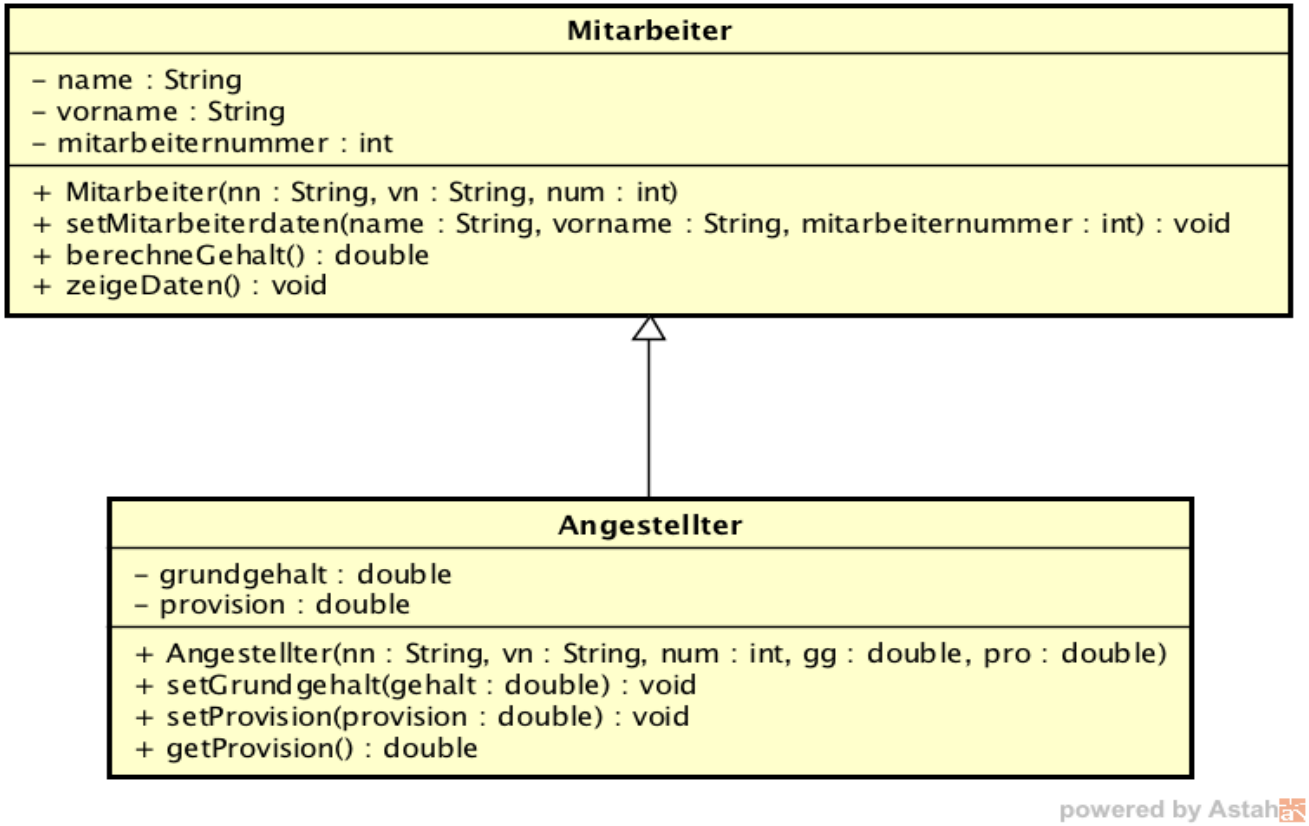
Übersicht | nächste Folie >

Konstruktoren in erbenden Klasssen



- Das Beispiel „Mitarbeiter und Angestellter“ soll um Konstruktoren zum Initialisieren aller Objektattribute erweitert werden.

OOA/OOD



OOP

```
public class Mitarbeiter {
        private String name;
        private String vorname;
        private int mitarbeiternummer;

        Mitarbeiter(String nn, String vn, int num) {
                this.setzeMitarbeiterdaten(nn, vn, num);
        }


        public void setzenMitarbeiterdaten(String name, String vorname, int mitarbeiternummer)
                this.name = name;
                this.vorname = vorname;
                this.mitarbeiternummer = mitarbeiternummer;

        }

        // Anmerkung: berechnegehalt und zeigeDaten fehlen hier
}
```

OOP: Konstruktor der Elternklasse mitverwenden

```
public class Angestellter extends Mitarbeiter {
        private double grundgehalt;
        private double provision;

        // Konstruktoren werden nicht mitvererbt
        Angestellter(String nn, String vn, int num, double grundgeh, double provision){
                // auf den Konstruktor der „Vaterklasse“ kann zugegriffen werden:
                super(nn,vn,num);
                // zusätzliche Attribute setzen
                this.setzeGrundgehalt(grundgeh);
                this.setzeProvision(provision);

        }

        public void setzeGrundgehalt(double gehalt) {
                this.grundgehalt = gehalt;
        }

        void setzeProvision(double provision) {
                this.provision = provision;
        }

        // Anmerkung: berechnegehalt und zeigeDaten fehlen hier
}
```

OOT: Testobjekt erzeugen

```
public class TestMitarbeiter {

          public static void main(String[] args) {
                    Angestellter an1 =
                              new Angestellter("Mustermann","Max", 1111, 5000.25, 1232.34);
                    an1.zeigeDaten();
          }

}
```

Häufiger Fehler

```
public class Angestellter extends Mitarbeiter {
        private double grundgehalt;
        private double provision;

        Angestellter(double grundgeh, double provision){
                this.setzeGrundgehalt(grundgeh);
                this.setzeProvision(provision);

        }

        public void setzeGrundgehalt(double gehalt) {
                this.grundgehalt = gehalt;
        }

        void setzeProvision(double provision) {
                this.provision = provision;
        }

        // Anmerkung: berechnegehalt und zeigeDaten fehlen hier
}
```

Fehlermeldung:
Implicit super constructror Mitarbeiter() is undefined. Must explicitly invoke another constructor

Objektorientierte Softwareentwicklung

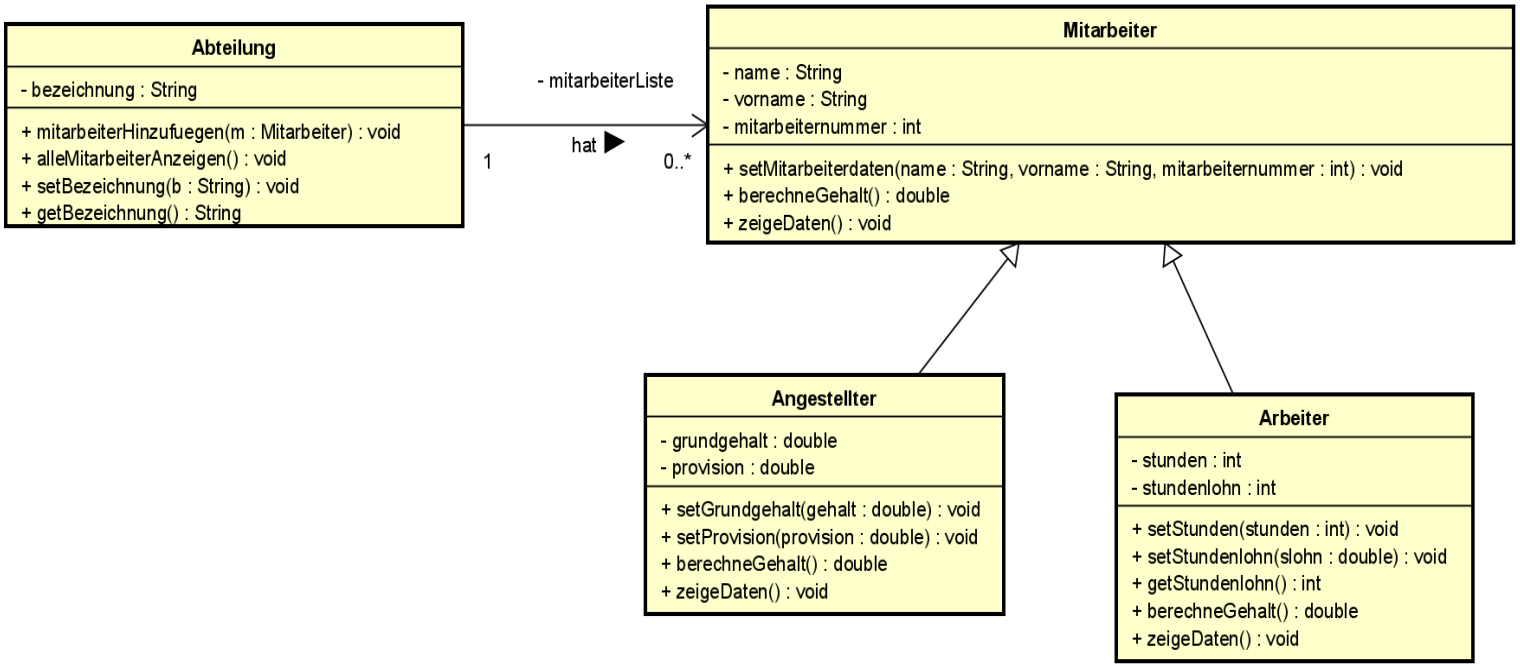
Vererbung und Listen

Übersicht | nächste Folie >

In gemeinsamer Liste Verwalten

- Die Mitarbeiter (Angestellte, Arbeiter) sollen nun einer Abteilung zugeordnet werden.
- Dabei kennt die Abteilung ihre Mitarbeiter.
- Beide Arten von Mitarbeitern (Angestellte, Arbeiter). Sollen in **derselben** Liste verwaltet werden.

UML OOA/OOD



OOP

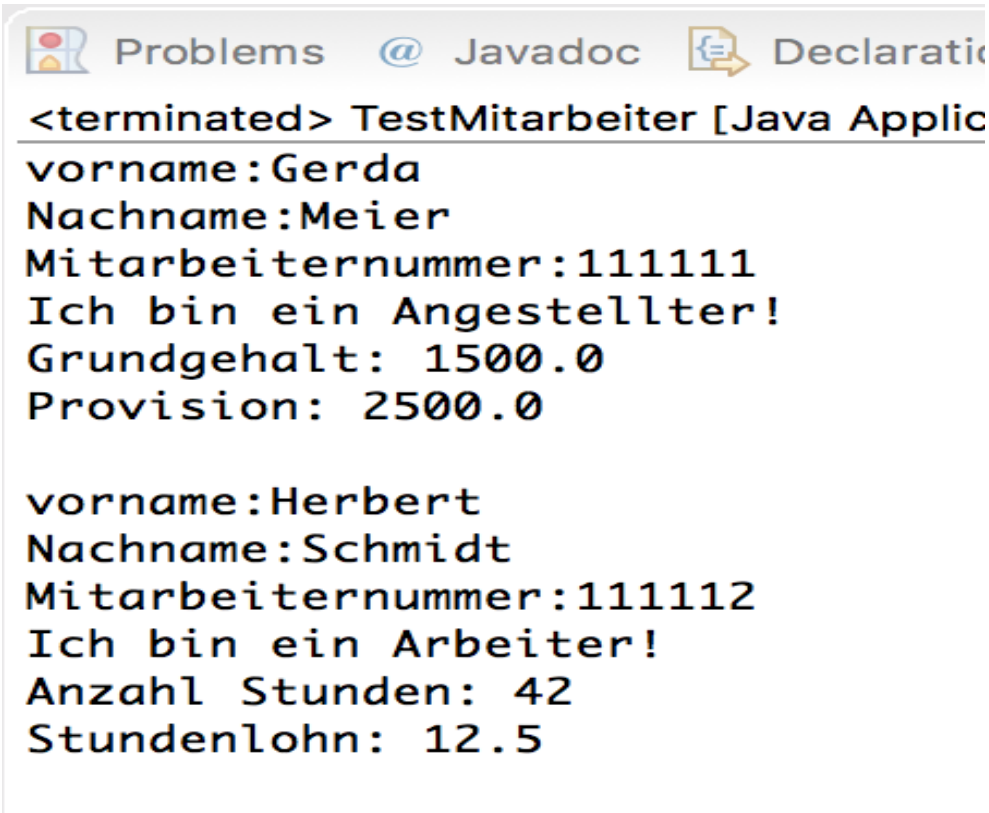
```
import java.util.*;

public class Abteilung {
          private String bezeichnung;

          // ArrayListe vom Typ Mitarbeiter erzeugen. So können sowohl
          // Angestellte als auch Arbeiter in die Liste eingetragen werden
          List<Mitarbeiter> mitarbeiterliste = new
                              ArrayList<Mitarbeiter>();

          // get-/set-Methoden hier nicht dargestellt

          public void mitArbeiterhinzufuegen(Mitarbeiter m) {
                    // in Liste einfuegen
                    // dabei ist egal, welcher konkrete Typ vorliegt
                    this.mitarbeiterliste.add(m);
          }

          public void alleMitarbeiterAnzeigen() {
                    for (Mitarbeiter m : mitarbeiterliste) {
                              // Datenanzeigen:
                              m.zeigeDaten();
                              // je nach konkretem Typ werden
                              // die Attribute angezeigt,
                              // da die zeigeDaten()-Methode entsprechend
                              // ueberschrieben wurde
                              System.out.println(“”); // Zeilenumbruch
}
```

OOT

```
// Programmgeruest nicht dargestellt

Angestellter an1 = new Angestellter();
an1.setzeMitarbeiterdaten("Meier", "Gerda", 111111);
an1.setzeGrundgehalt(1500);
an1.setzeProvision(2500);

Arbeiter ar1 = new Arbeiter();
ar1.setzeMitarbeiterdaten("Schmidt", "Herbert", 111112);
ar1.setzeStunden(42);
ar1.setzeStundenlohn(12.50);

Abteilung a1 = new Abteilung();

// Mitarbeiter der Abteilung hinzufuegen
a1.mitArbeiterhinzufuegen(an1); // geht
a1.mitArbeiterhinzufuegen(ar1); // geht auch
// alle Daten Anzeigen lassen:
a1.alleMitarbeiterAnzeigen();
// Daten werden angezeigt
// unterschiedlich, je nachdem welcher konkrete Typ vorliegt
```

OOT

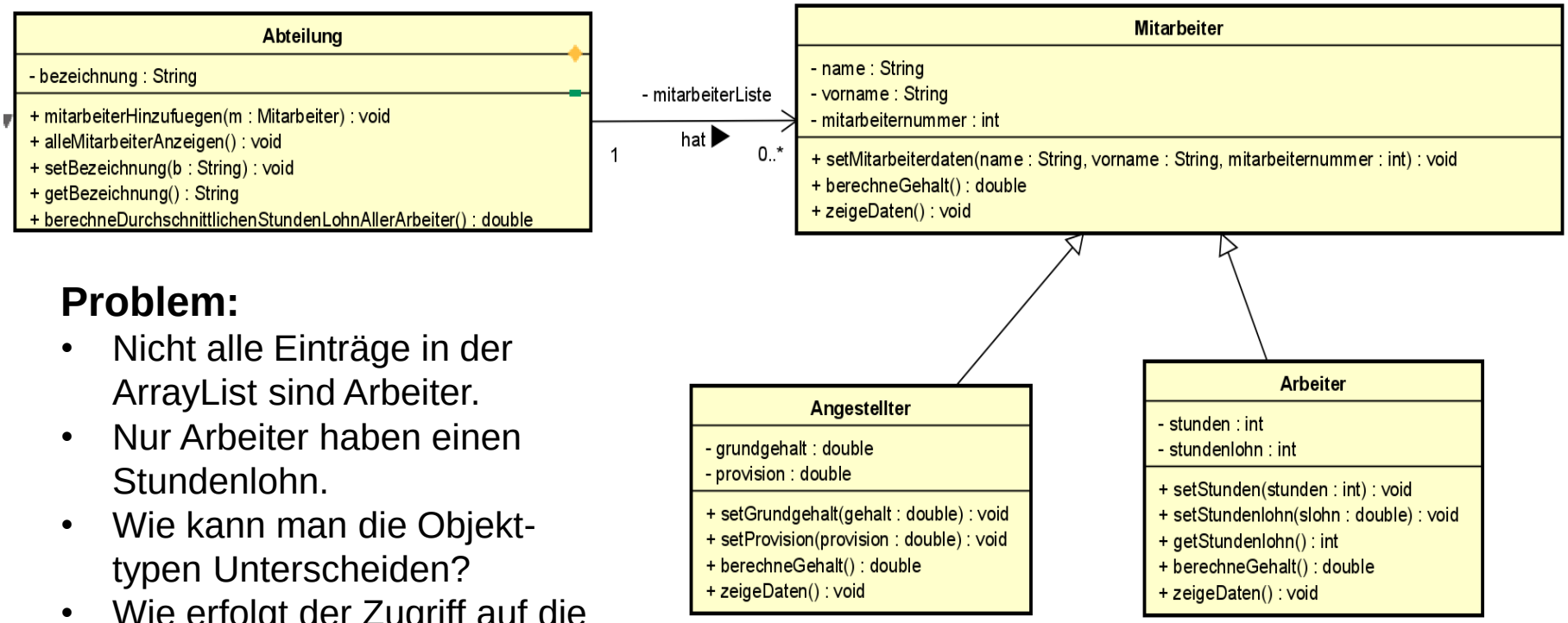
Die Ausgabe ist angepasst: je nach konkretem Typ (Angesteller/Arbeiter)



Problemstellung: Einträge unterscheiden

- Es soll eine Methode bereitgestellt in der Klasse Abteilung bereitgestellt werden, welche die Stundenlöhne aller Arbeiter in der Abteilung aufsummiert und das Ergebnis zurückliefert.

OOA/OOD



Problem:

- Nicht alle Einträge in der ArrayList sind Arbeiter.
- Nur Arbeiter haben einen Stundenlohn.
- Wie kann man die Objekt-typen Unterscheiden?
- Wie erfolgt der Zugriff auf die Methode „getStundenlohn()“?

OOP

```
public class Abteilung {
          private String bezeichnung;
          List<Mitarbeiter> mitarbeiterliste = new ArrayList<Mitarbeiter>();
// ...
public double berechneDurchschnittlichenStundenLohnAllerArbeiter(){
          double dstundenlohn = 0;
          for (Mitarbeiter m : mitarbeiterliste) {
                    // m.getStundenlohn hier nicht moeglich,
                    // da Mitarbeiter diese Methode nicht haben
                    // alle Objekte treten im Moment nur als Mitarbeiter auf
                    // testen, welcher Typ vorliegt
                    if (m instanceof Arbeiter){
                              // Objekt ist ein Arbeiter,
                              // m.getStundenlohn aber immer noch nicht moeglich
                              // Umwandeln:
                              Arbeiter a = (Arbeiter)m;
                              // Das Objekt tritt nun als Arbeiter auf (Polymorphismus)
                              // Die Gestalt wird hier zur Laufzeit geändert (lateBinding)
                              dstundenlohn+=a.getStundenlohn();
                    }
          }
          return dstundenlohn;
}
}
```

Objektorientierte Softwareentwicklung

Sequenzdiagramme in UML

Übersicht | nächste Folie >

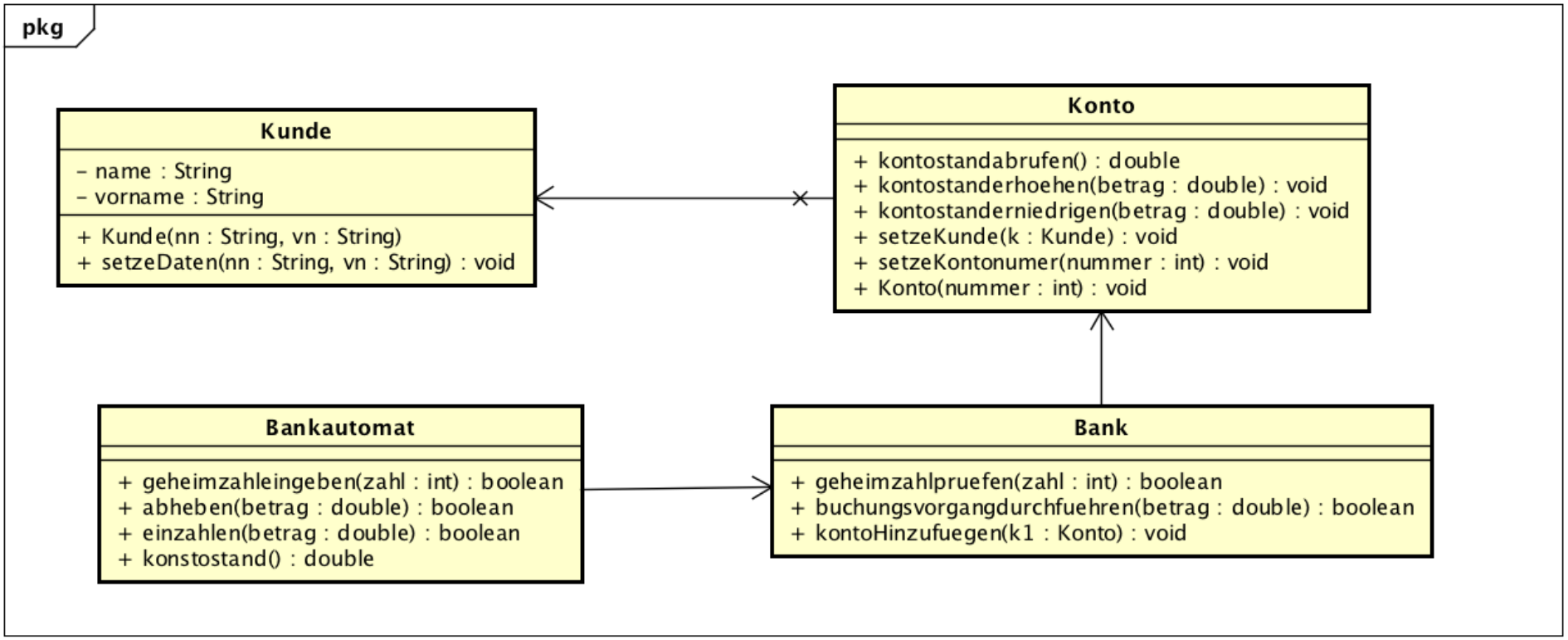
Wozu?

- UML Sequenzdiagramme dienen dazu Interaktionen zwischen Objekten in verschiedenen Szenarien darzustellen.
- Dabei interagieren Objekte über Methodenaufrufe miteinander.

Beispiel

- Es eine objektorientiere Software für eine Bank entwickelt werden.
- Im System sind Kunden, Konten, Bankautomaten und verschiedene Banken abzubilden.
- Für verschiedene Szenarien (z.B. Geld abheben), sollen die notwendigen Objektinteraktionen analysiert und dargestellt werden.

Klassendiagramm (unvollständig)

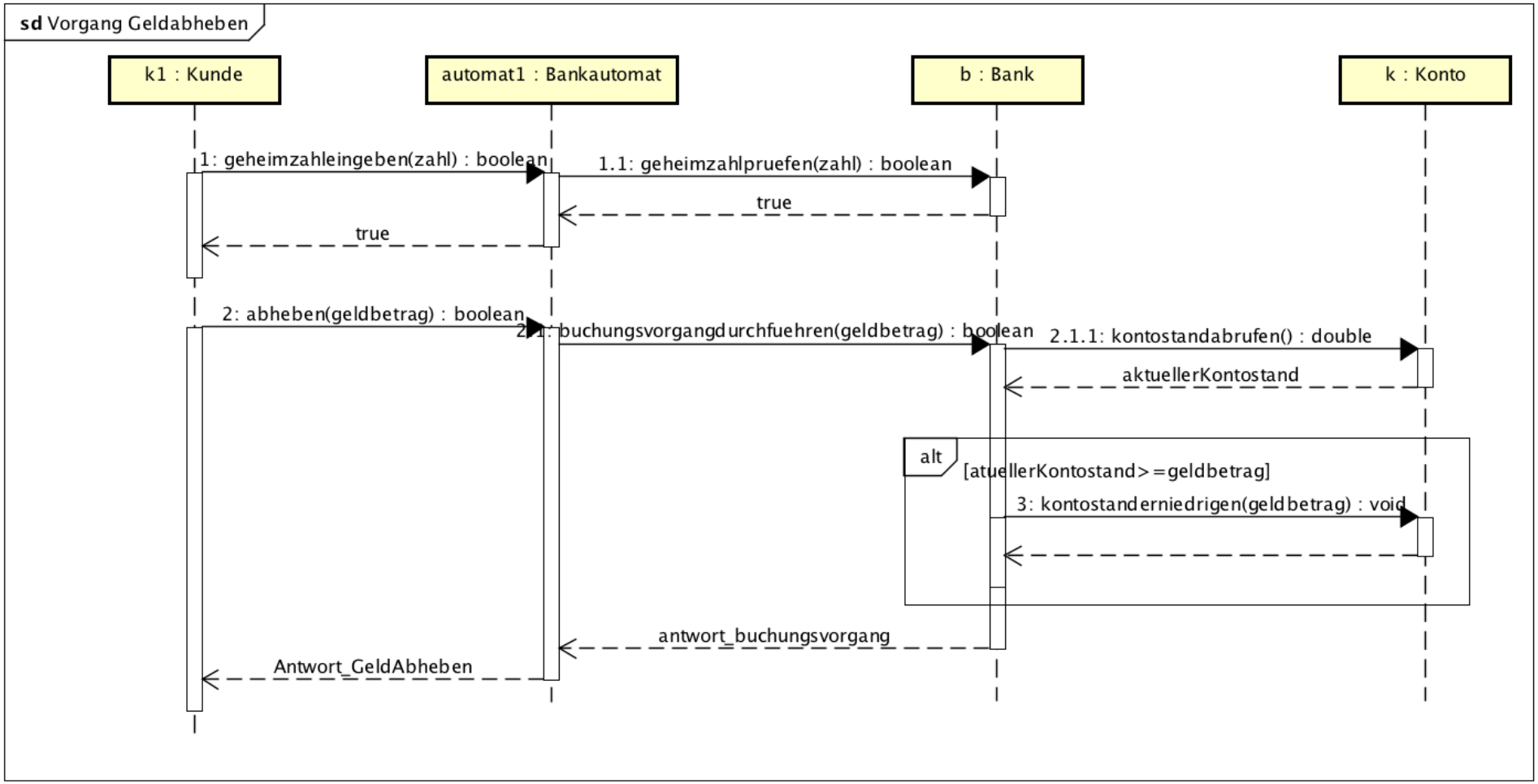


Anmerkung: Warum darf ein Kundenobjekt ein Kontoobjekt nicht direkt benutzen?

Senario: Geld abheben

- Dargestellt werden soll das Szenario, dass ein Kunde von einem Bankautomat Geld abhebt.
- Vorüberlegungen:
  - Ein Kunde kommuniziert mit einem Bankautomaten (PIN eingeben, Geldbetrag eingeben)
  - Der Bankautomat kommuniziert mit der Bank (Überprüfen der PIN)
  - Die Bank kommuniziert mit einem Konto (genug Geld vorhanden? Wenn ja Geldbetrag abbuchen.)

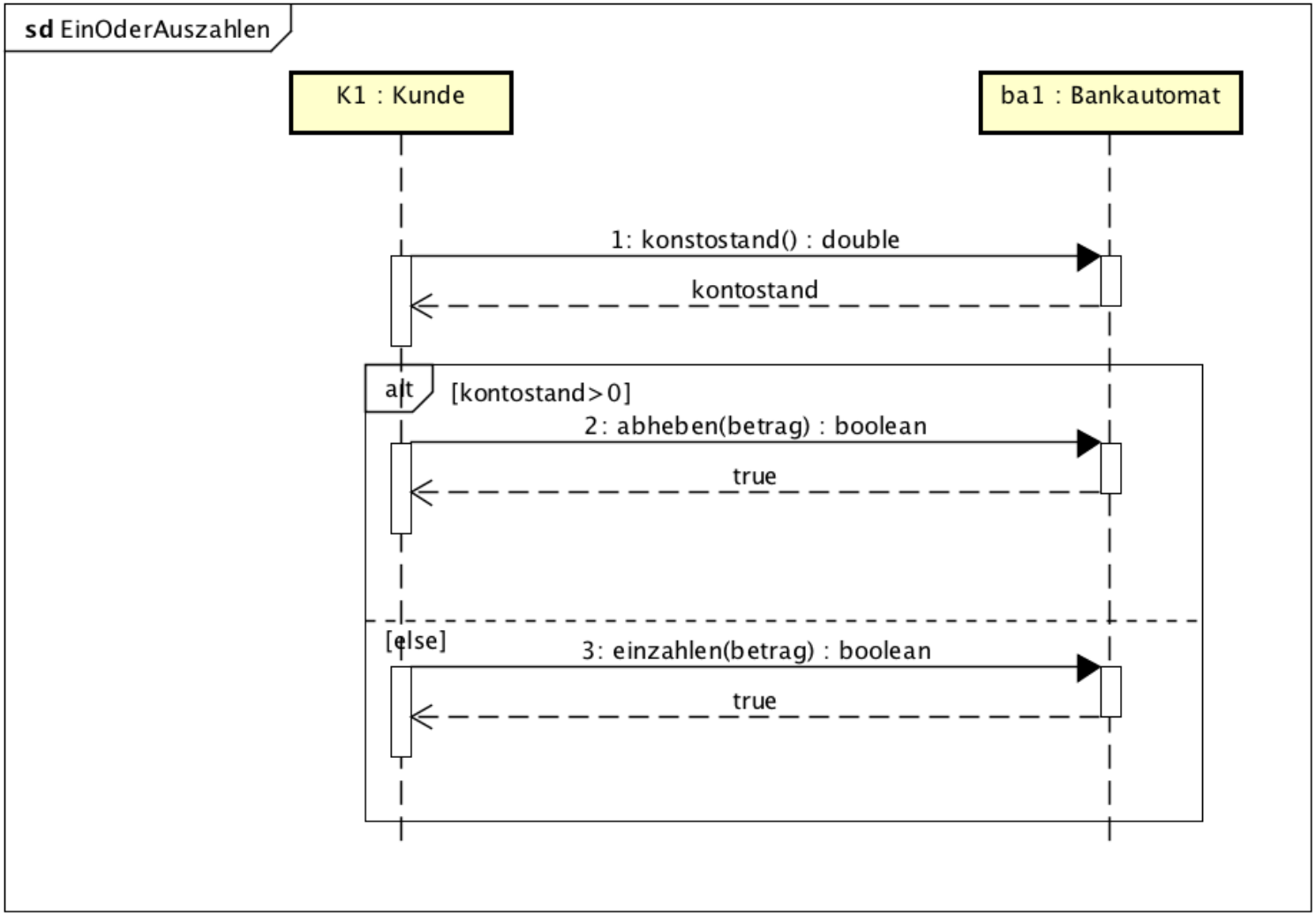
Sequenzdiagramm Geldabheben



Bedingte Aufrufe

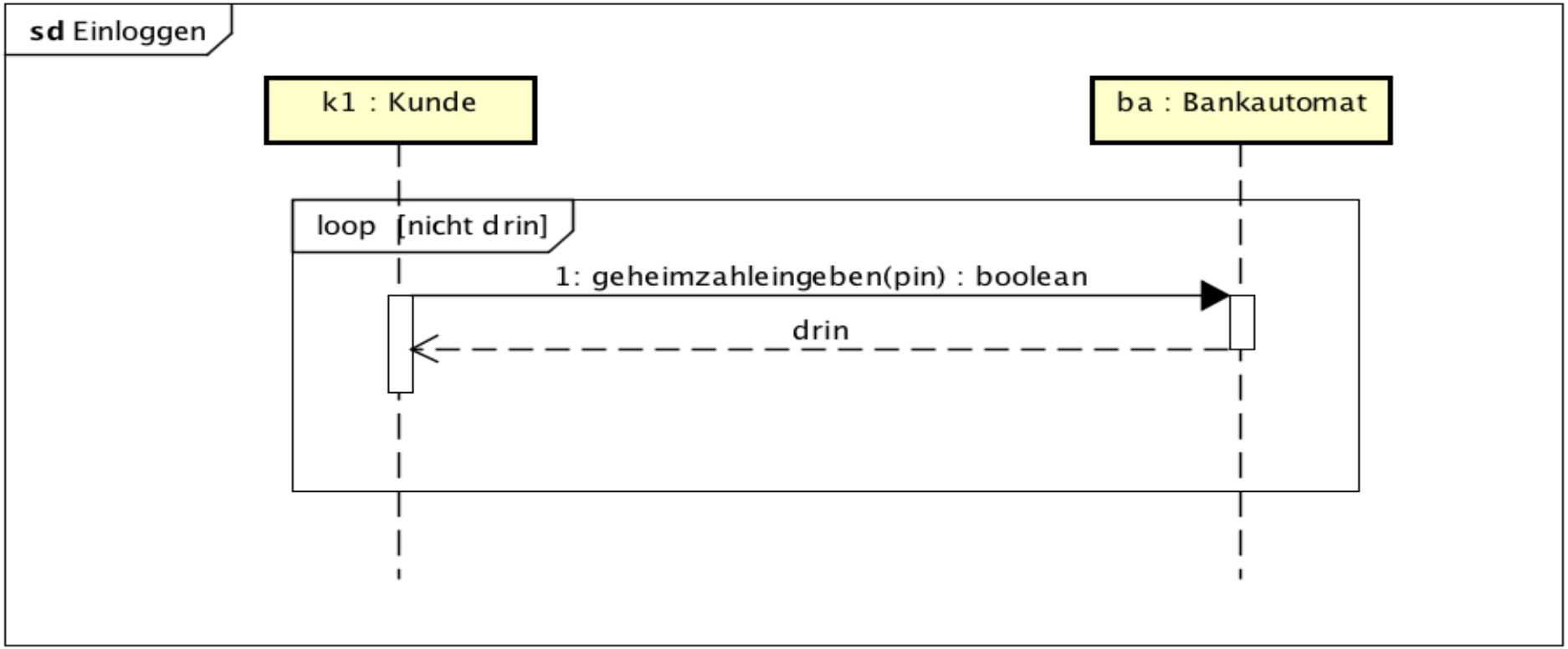
- Bespiel: Ein Kunde ruft seinen Kontostand ab.
- Wenn Geld auf dem Konto vorhanden ist, hebt er einen Betrag ab.
- Wenn kein Geld mehr vorhanden ist, zahlt er stattdessen einen Betrag ein.
- Dabei sollen nur die Interaktionen zwischen Kunde und Bankautomat dargestellt werden.

Bedingte Aufrufe



Wiederholungen

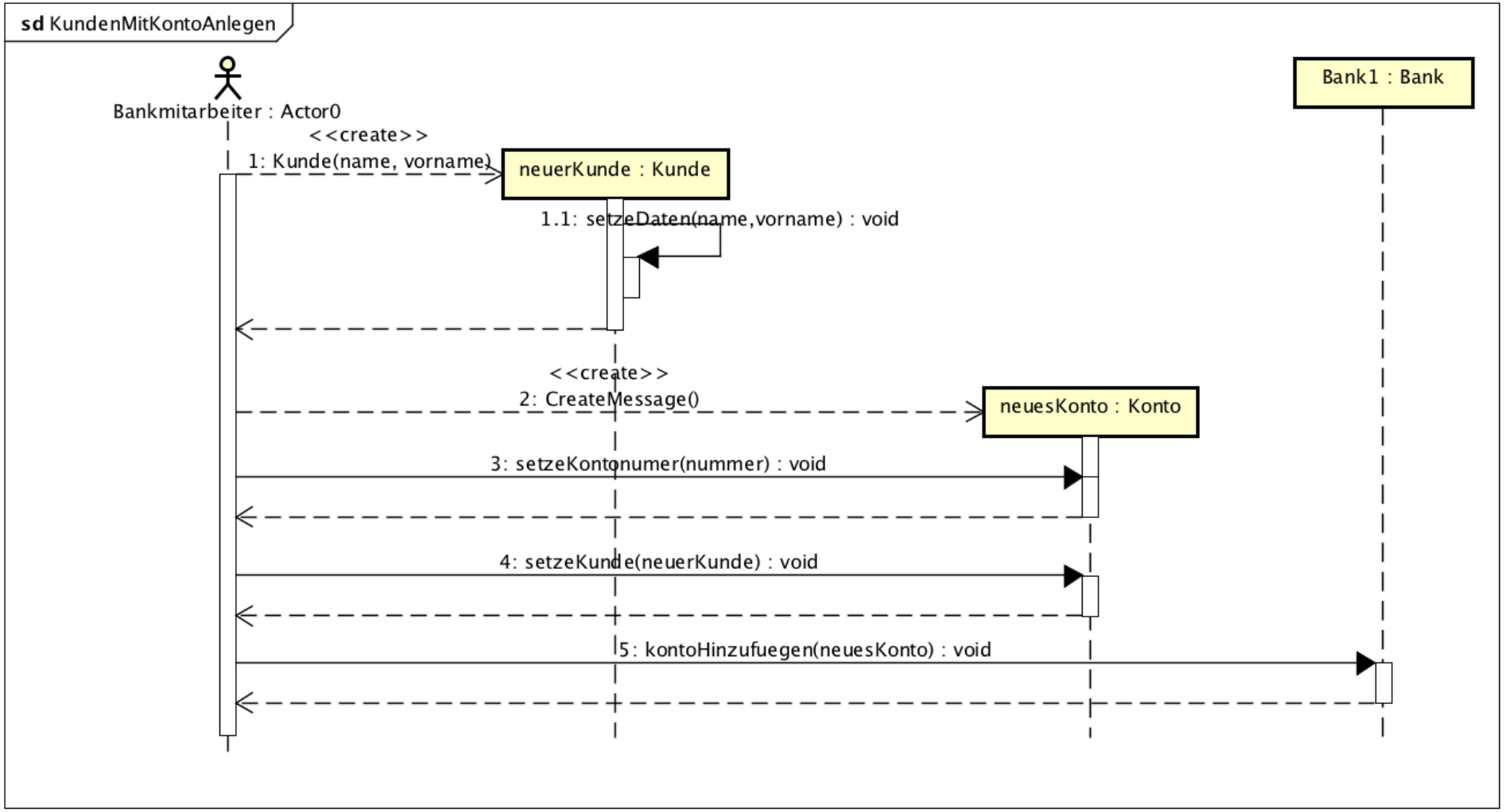
Bestimmte Interaktionen können wiederholt werden.
Beispiel: Ein Kunde gibt so lange seinen PIN ein, bis er richtig ist.



Objekte erzeugen und zuweisen

- Ein neuer Kunde soll ein Konto in der Bank bekommen.
- Vorüberlegungen:
  - Ein Bankangestellter erstellt einen neuen Kunden und ein neues Konto.
  - Der Kunde wird dem Konto zugeordnet.
  - Das Konto in der Bank registriert.

Objekte erzeugen und hinzufügen



UML Vertiefung

Anwendungsfalldiagramm

Übersicht | nächste Folie >

Wozu?

- Das Anwendungsfalldiagramm (Use-Case-Diagram) dient zur Planung eines Softwaresystems (in der OOA-Phase), indem mögliche **Akteure** (Menschen, Benutzer eines Systems), **Anwendungsfälle** (Szenarien, wie Akteure das System nutzen) und die **Beziehungen** dazwischen identifiziert und abgebildet werden.

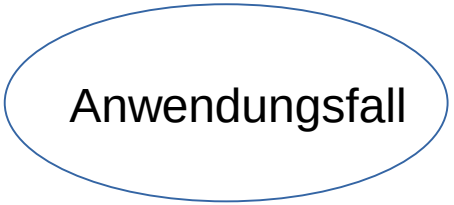
Beispiel Bankfiliale

- Es soll eine objektorientierte Software für eine Bankfiliale geplant werden.
- Dazu sind zunächst mögliche **Anwendungsfälle und Akteure** zu überlegen.

Identifizierung Anwendungsfälle/Akteure

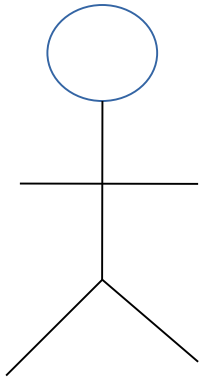
Anwendungsfälle:

- Geld vom Konto abheben
- Geld vom Konto einzahlen
- Konto eröffnen
- Kredit beantragen
- Kreditwürdigkeit überprüfen
- Beratungsgespräche führen



Akteure:

- Berater
- Kunde



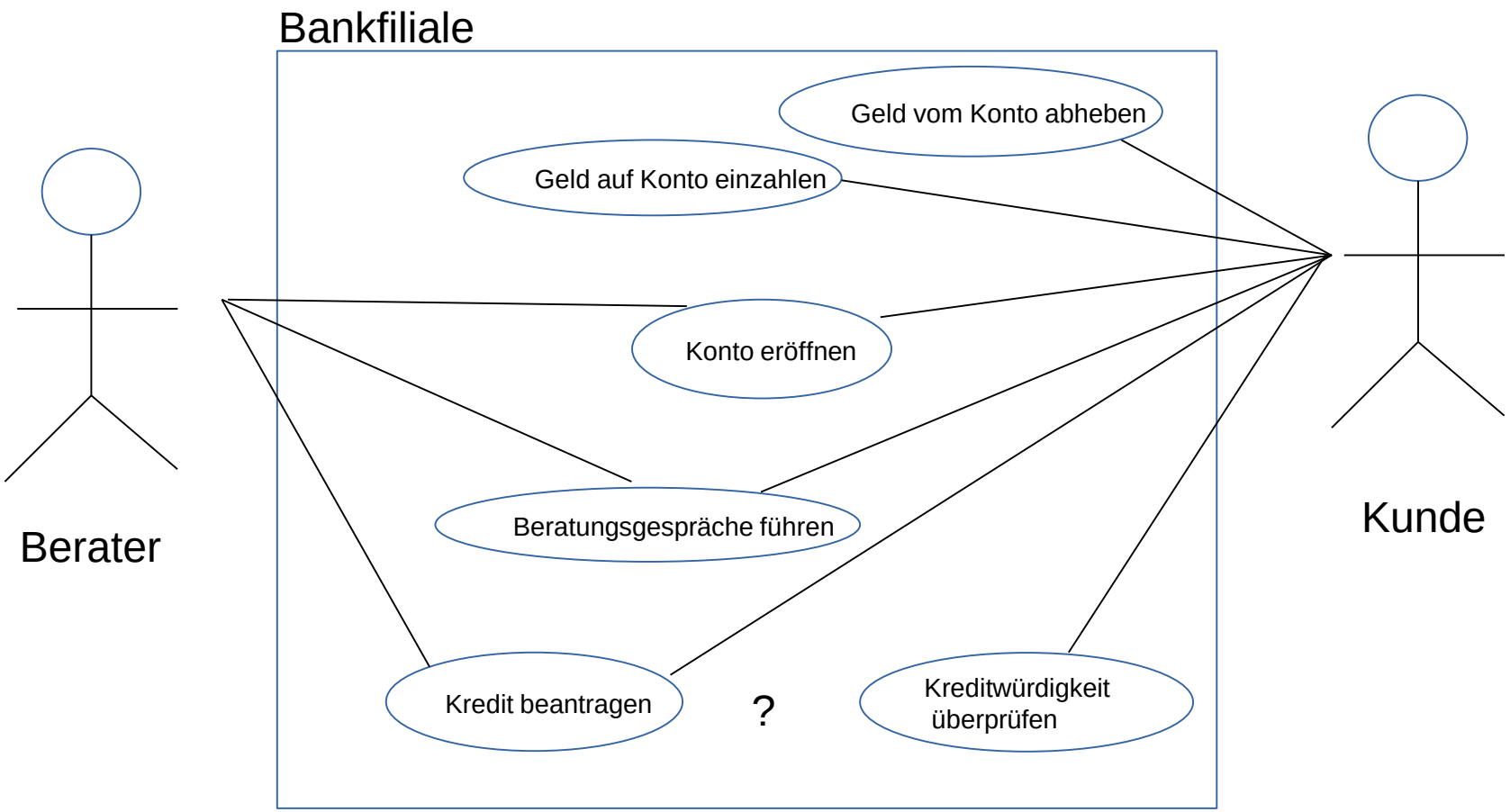
Akteur

Beziehungen

- Als nächstes sollen Beziehungen überlegt werden und zwar zwischen Akteuren und Anwendungsfälle.
- Leitfrage: “Welche Akteure sind an welchen Anwendungsfällen beteiligt?“

Beziehungen identifiziern

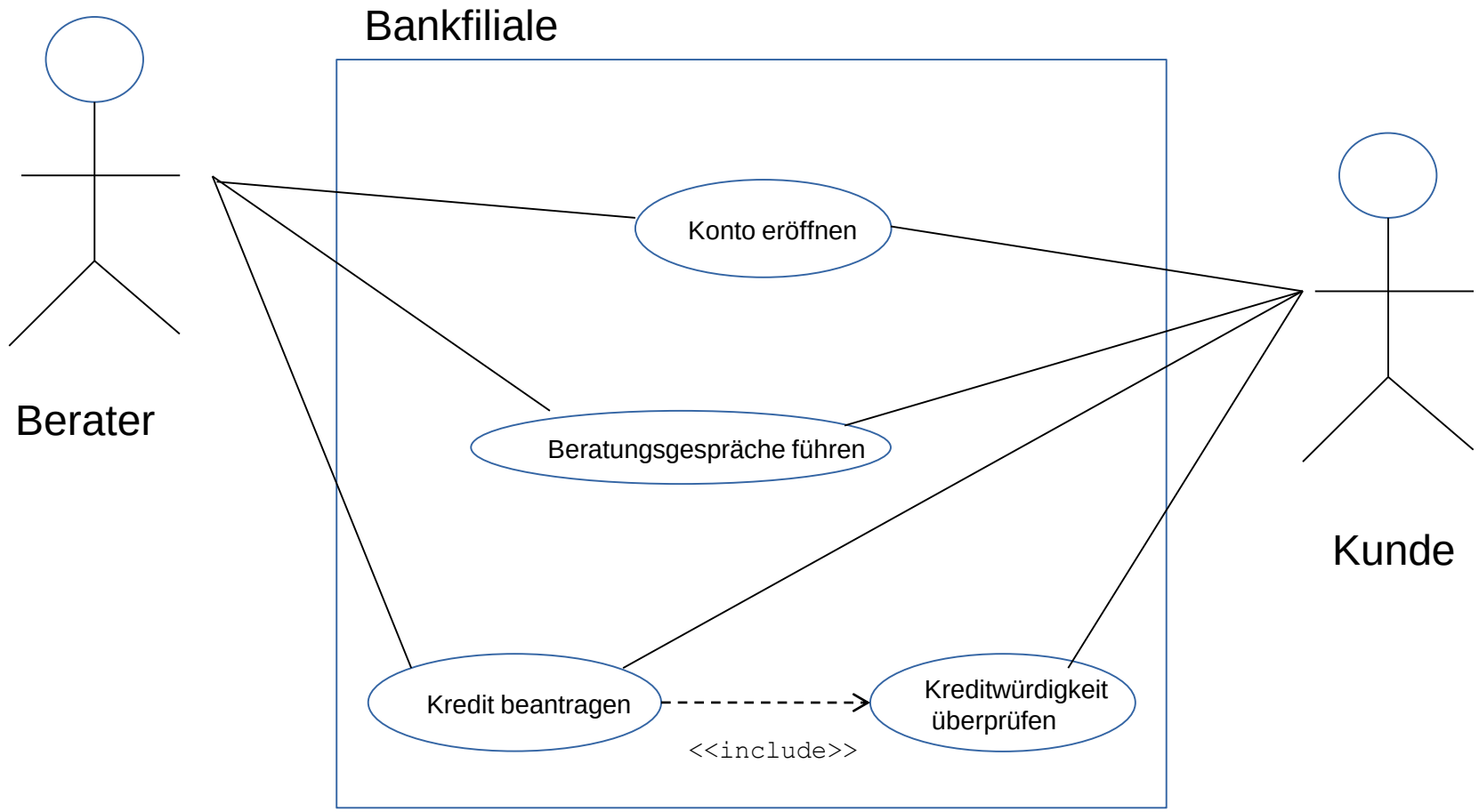
Welcher Akteur steht mit welchen Anwendungsfällen in Beziehung?



Beziehungen zwischen Anwendungsfällen 1

- Zwischen den Anwendungsfällen können auch Beziehungen bestehen:
  - Es kann sein, dass ein Anwendungsfall einen anderen Anwendungsfall **zwingend** mit einschließt.
  - Beispiel: Wenn ein Kredit beantragt wird, muss zwingend die Kreditwürdigkeit des Kunden überprüft werden.

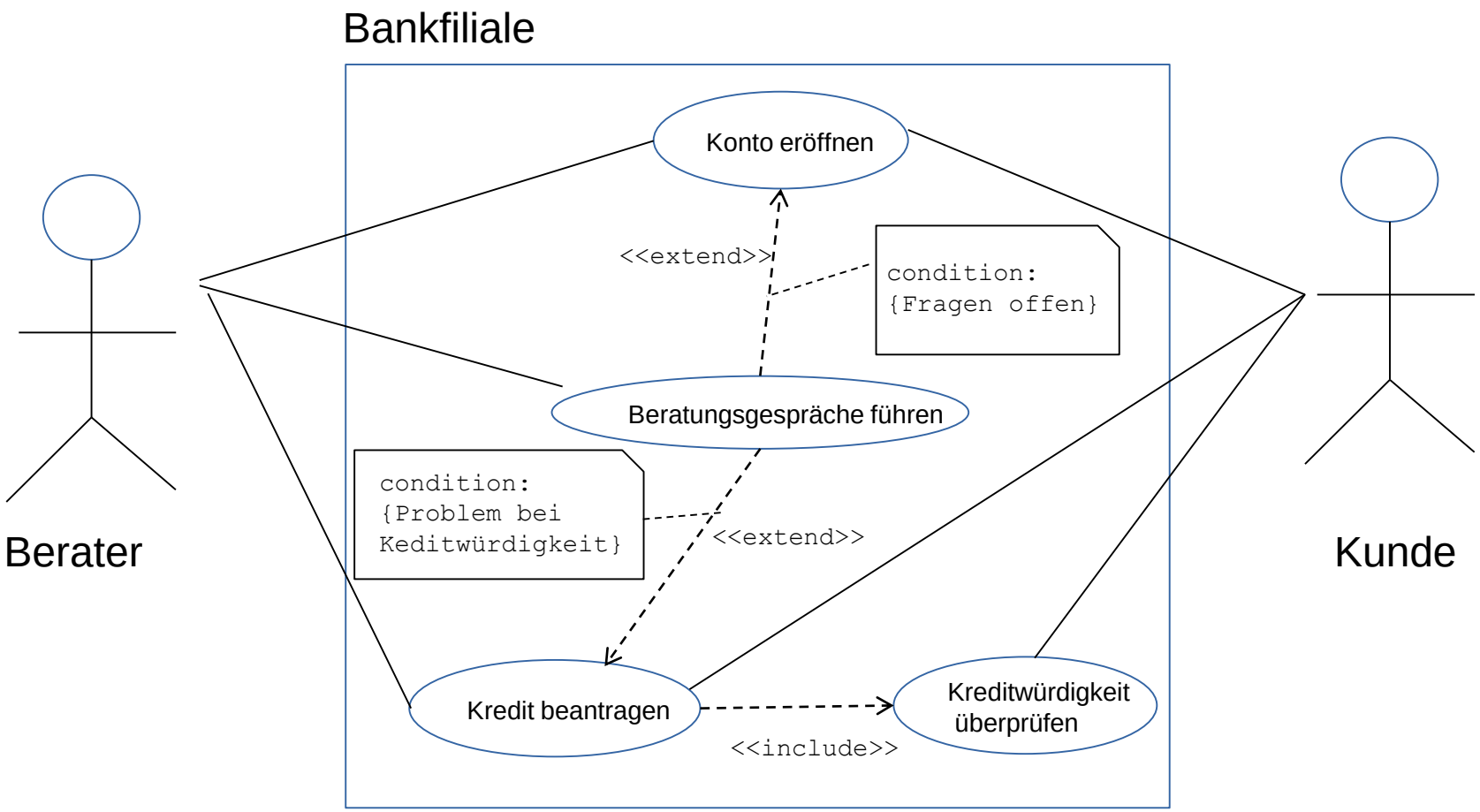
Beziehungen zwischen Anwendungsfällen (include)



Beziehungen zwischen Anwendungsfällen 2

- Nicht immer umfasst ein Anwendungsfall zwingend einen Anderen. Es kann sein, dass ein Anwendungsfall einen Anderen nur unter einer bestimmten Bedingung erweitert:
  - Wenn ein Konto eröffnet wird **kann** es sein, dass auch ein Beratungsgespräch notwendig wird, **wenn** z.B. noch Unklarheiten vorhanden sind.
  - Ebenso kann ein Beratungsgespräch für die Kredit vergabe notwendig sein, wenn z.B. die Kreditwürdigkeit Probleme bereitet.
  - Die Anwendungsfälle „Konto eröffnen" und „Kredit beantragen" machen also nur unter einer bestimmten Bedingung ein Gespräch notwendig.

Extend Beziehung

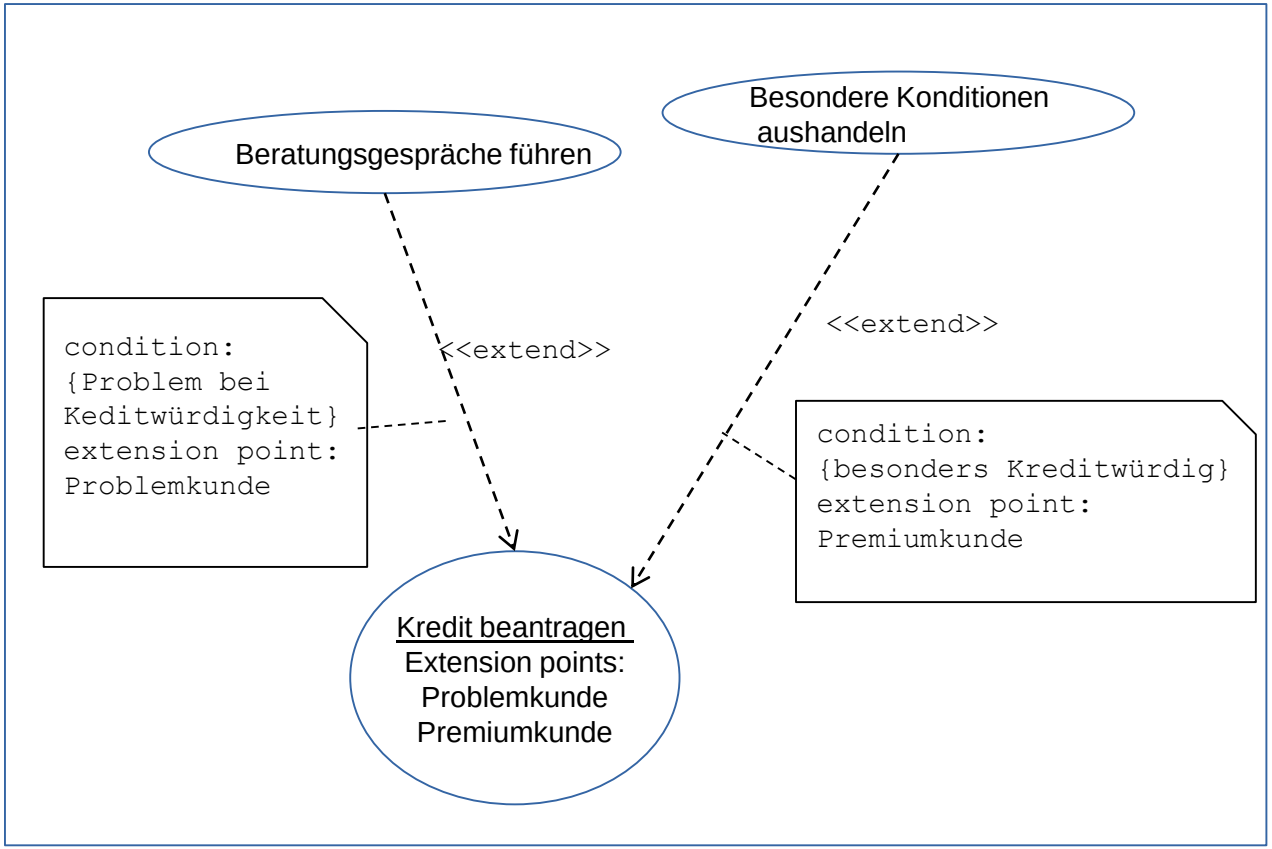


Extension points

- Ein Anwendungsfall kann durch mehrere Anwendungsfälle erweitert werden. Dann werden „Extension Points“ definiert und der erweitere Anwendungsfall erhält eine Liste mit “Extension points“.

Extension points

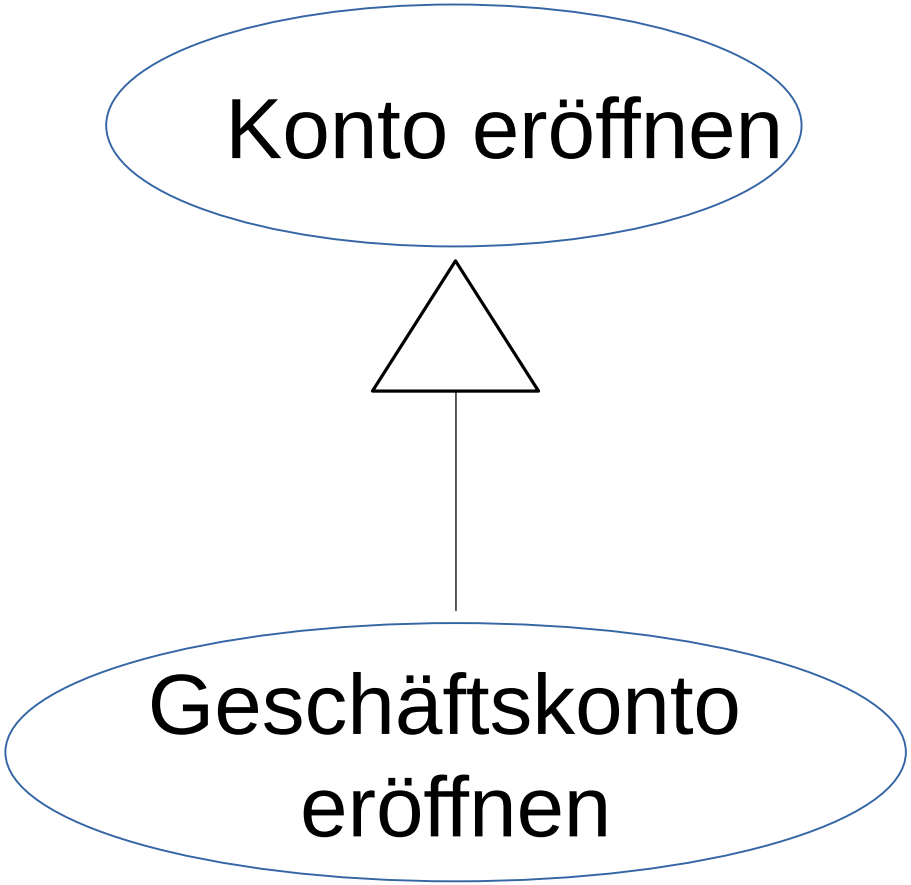
Bankfiliale



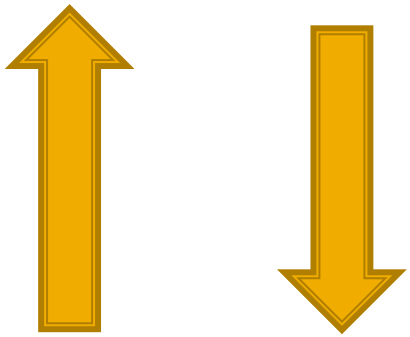
Beziehungen zwischen Anwendungsfällen 3

- Oft ist es sinnvoll Anwendungsfälle weiter zu „verfeinern“ (spezialisieren).
- So könnte es einen, dass es einen Anwendungsfall „Geschäftskonto eröffnen“ gibt, der alles aus dem Anwendungsfall „Konto eröffnen“ umfasst und zusätzlich weitere Aktionen erfordert.

Generalisierung/Spezialisierung



Generalisierung

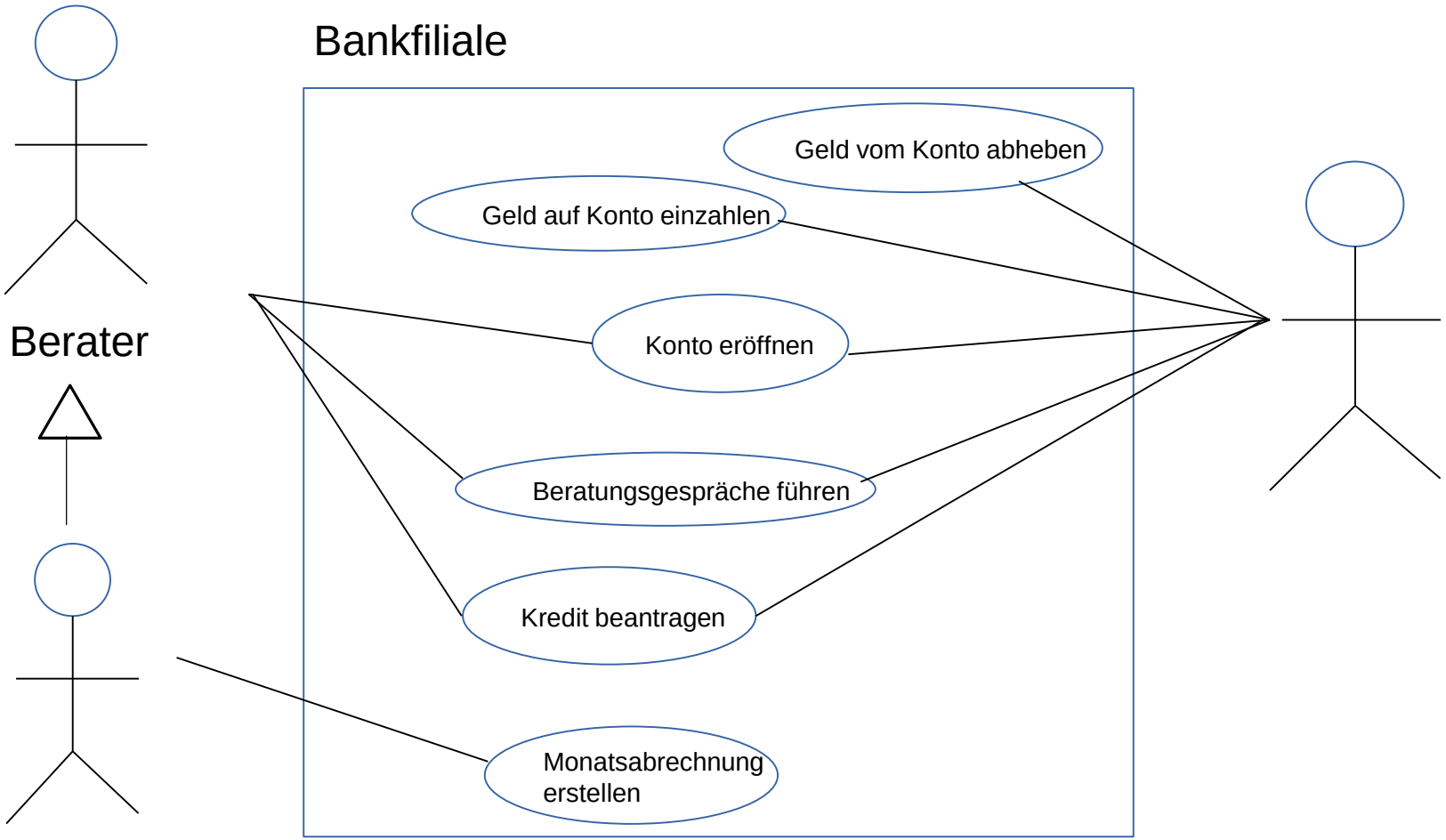


Spezialisierung

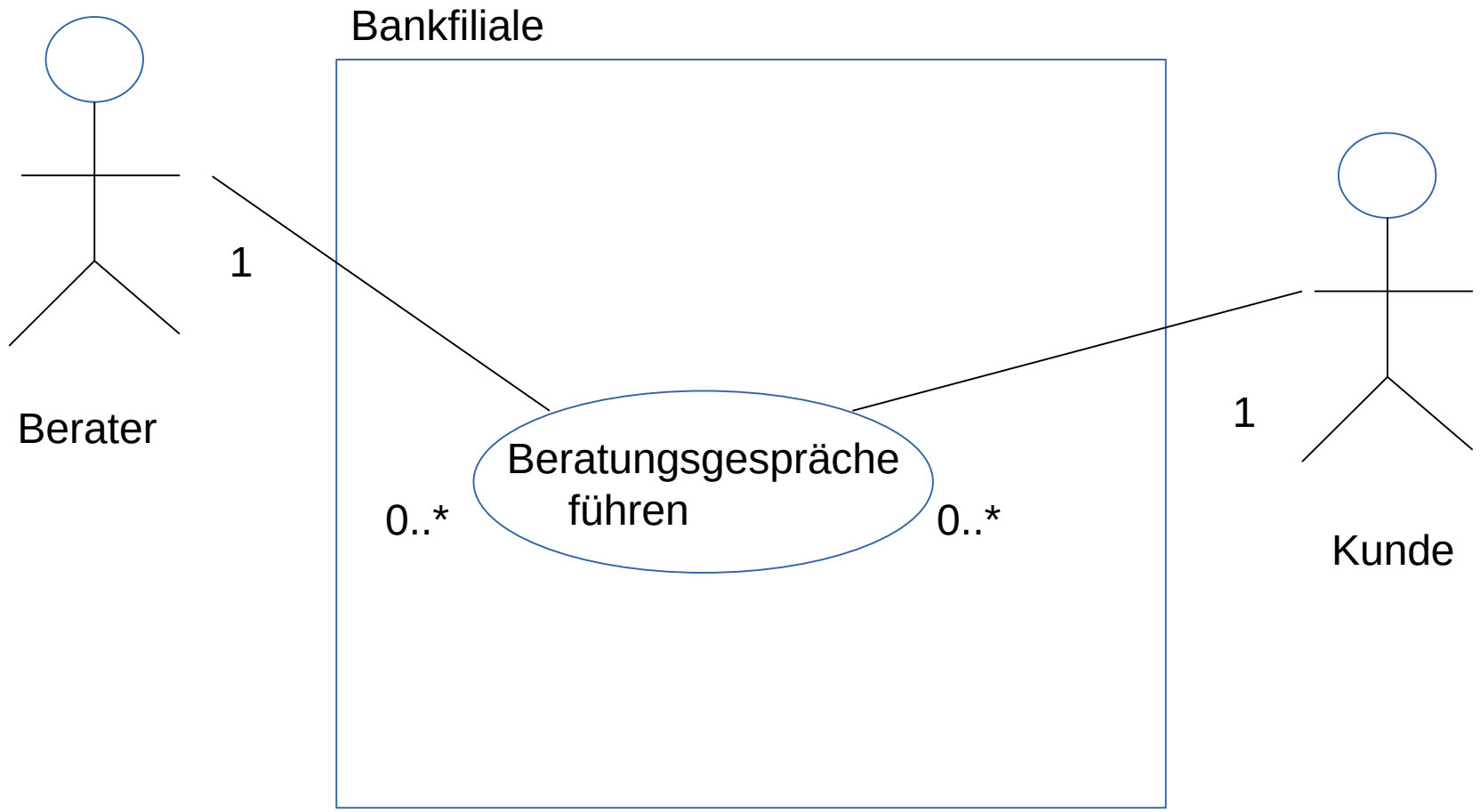
Beziehungen zwischen Akteuren

- Zusätzlich sollen Filialleiter als Akteur berücksichtigt werden.
- Diese dürfen alles, was ein Berater darf.
- Zusätzlich darf er eine Monatsabrechnung für die Filiale erstellen.

Akteure spezialisieren / generalisieren



Multiplizitäten



Zusammenfassung

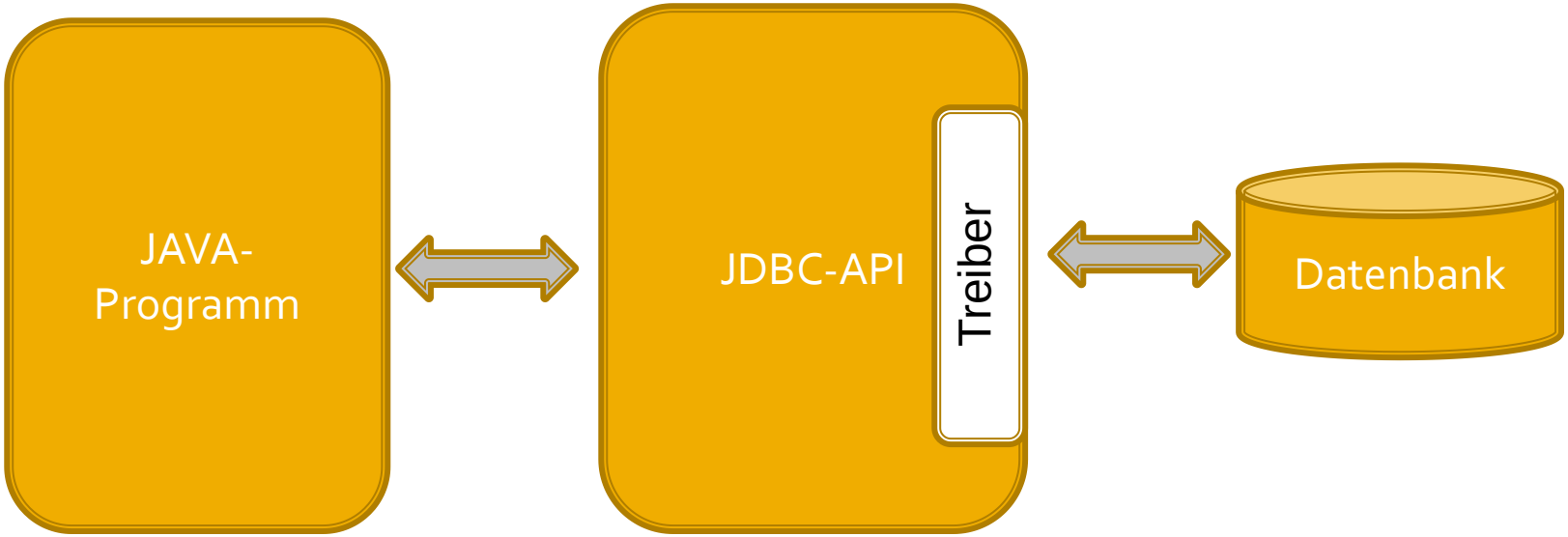
- Mit Hilfe eines Anwendungsfalldiagramms werden Anforderungen an das System analysiert und dargestellt (OOA).
- Basierend auf diesem Diagramm können nun weitere OOA/OOD-Planungsschritte folgen (Klassendiagramme, Sequenzdiagramme).

Objektorientierte Softwareentwicklung

Datenbankanbindung

Übersicht | nächste Folie >

Datenbankanbindung (Schema)



Installation des MySQL-Treibers:

1. Datei mysql-connector-java-5.1.6-bin.jar in das Verzeichnis \lib\ext der Java-Laufzeitumgebung kopieren.
2. Unter Eclipse: Datei mysql-connector-java-5.1.6-bin.jar importieren.

Ablauf (Prinzip)

- Kommunikation eines JAVA Programms mit einer MySQL-Datenbank:
  - Verbindung aufbauen
  - MySQL-Befehle generieren
  - MySQL-Befehl(e) abschicken
  - MySQL-Befehle ggf. auswerten
  - Verbindung schließen

Beispiel

- Es soll auf eine MySQL-Datenbank mit Namen „verwaltung“ zugegriffen werden.
- In die Tabelle „mitarbeiter“ sollen Daten eingefügt werden.
- Aus der Tabelle „mitarbeiter“ sollen alle Datensätze ausgelesen und auf dem Bildschirm angezeigt werden.

Verbindung auf- und abbauen

```
import java.sql.*; // Bibliothek für SQL benutzen

public class DBZugriff1 {

	public static void main(String[] args) throws Exception {

		//Datenbank-Treiber laden
		Class.forName("com.mysql.jdbc.Driver");

		// Strings mit Zugangsdaten anlegen:

		//                                        Host      Datenbankname
		String connectionURL = "jdbc:mysql://localhost/verwaltung";
		String user = "root"; // Benutzername
		String password = ""; // Passwort


		//Verbindung zur Datenbank aufbauen
		Connection conn = DriverManager.getConnection(connectionURL, user, password);

		// Verbindung schließen
		conn.close();
	}
}
```

Datensätze einfügen

```
import java.sql.*; // Bibliothek für SQL benutzen

public class DBZugriff2 {

	public static void main(String[] args) throws Exception {

		Class.forName("com.mysql.jdbc.Driver");


		String connectionURL = "jdbc:mysql://localhost/verwaltung";
		String user = "root"; // Benutzername
		String password = ""; // Passwort


		Connection conn = DriverManager.getConnection(connectionURL, user, password);

		//Statement-Objekt erzeugen
		Statement stmt = conn.createStatement();

		// SQL-Befehl absenden
		stmt.execute("insert into mitarbeiter values (1111, 'Herbert', 'Meier')");

		// Alles wieder schließen
		stmt.close();
		conn.close();
	}
}
```

Datensätze auslesen

```
import java.sql.*;

public class DBZugriff3 {

	public static void main(String[] args) throws Exception {
		//Datenbank-Treiber laden
		Class.forName("com.mysql.jdbc.Driver");

		//                                        Host      Datenbank
		String connectionURL = "jdbc:mysql://localhost/verwaltung";
		String user = "root";
		String password = "";
		//String password = null;

		//Verbindung zur Datenbank aufbauen
		Connection conn = DriverManager.getConnection(connectionURL, user, password);

		//Statement-Objekt erzeugen
		Statement stmt = conn.createStatement();


		//Datensätze aus Tabelle "benutzer" auslesen
		ResultSet rs = stmt.executeQuery("select * from mitarbeiter");

		//Durch alle Datensätze navigieren
		while (rs.next()) {
			int id = rs.getInt("nr");
			String vorname = rs.getString("vorname");
			String nachname = rs.getString("nachname");
			System.out.println(id + ", " + vorname + ", " + nachname);
		}

		//Alles wieder schliessen
		rs.close();
		stmt.close();
		conn.close();
	}

}
```

PostgreSQL Datenbank Anbinden

- Funktioniert analog zur MySQL-Anbindung.
- Unterschiede:
- Es muss die postgresql Bibliothek eingebunden werden.
- Der Klassenname des Treibers muss geändert werden.
- Die URL für die Verbindung ist abzuändern.
- Der Rest (Befehle generieren und abschicken, Ergebnisse verarbeiten) ist identisch.

Verbindung aufbauen mit postgreSQL

```
public class DBZugriff1 {

        public static void main(String[] args) throws Exception {

                //Datenbank-Treiber laden
                Class.forName("org.postgresql.Driver");

                // Strings mit Zugangsdaten anlegen:

                //                                        Host    Datenbankname
                String connectionURL = "jdbc:postgresql://localhost/frye";
                String user = "christianfrye"; // Benutzername
                String password = "1234"; // Passwort


                //Verbindung zur Datenbank aufbauen
                Connection conn =
                          DriverManager.getConnection(connectionURL, user, password);

                // Verbindung schließen
                conn.close();
        }
}
```

Objektorientierte Softwareentwicklung

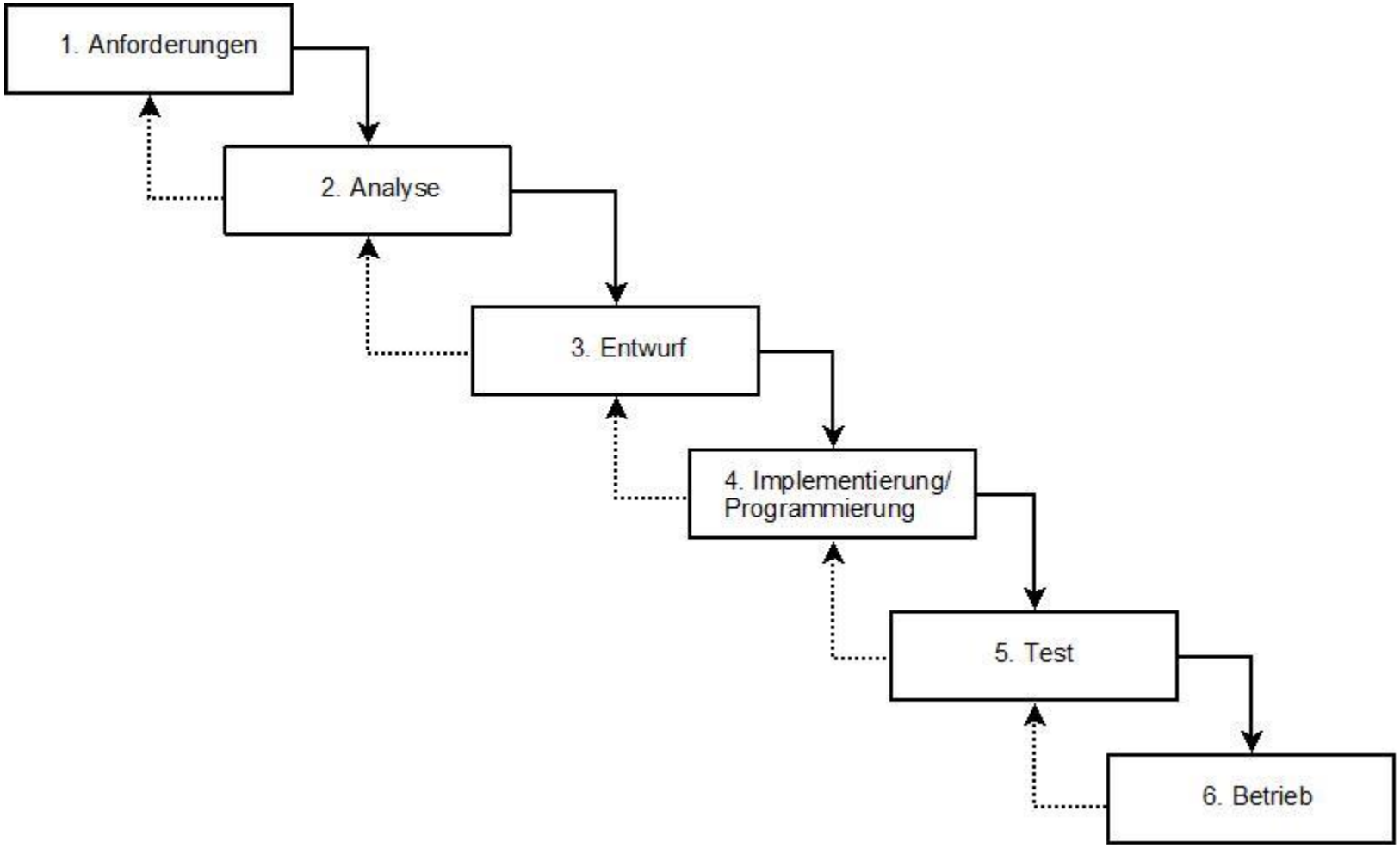
Systematische Softwareentwicklung nach dem Wasserfallmodell

Übersicht | nächste Folie >

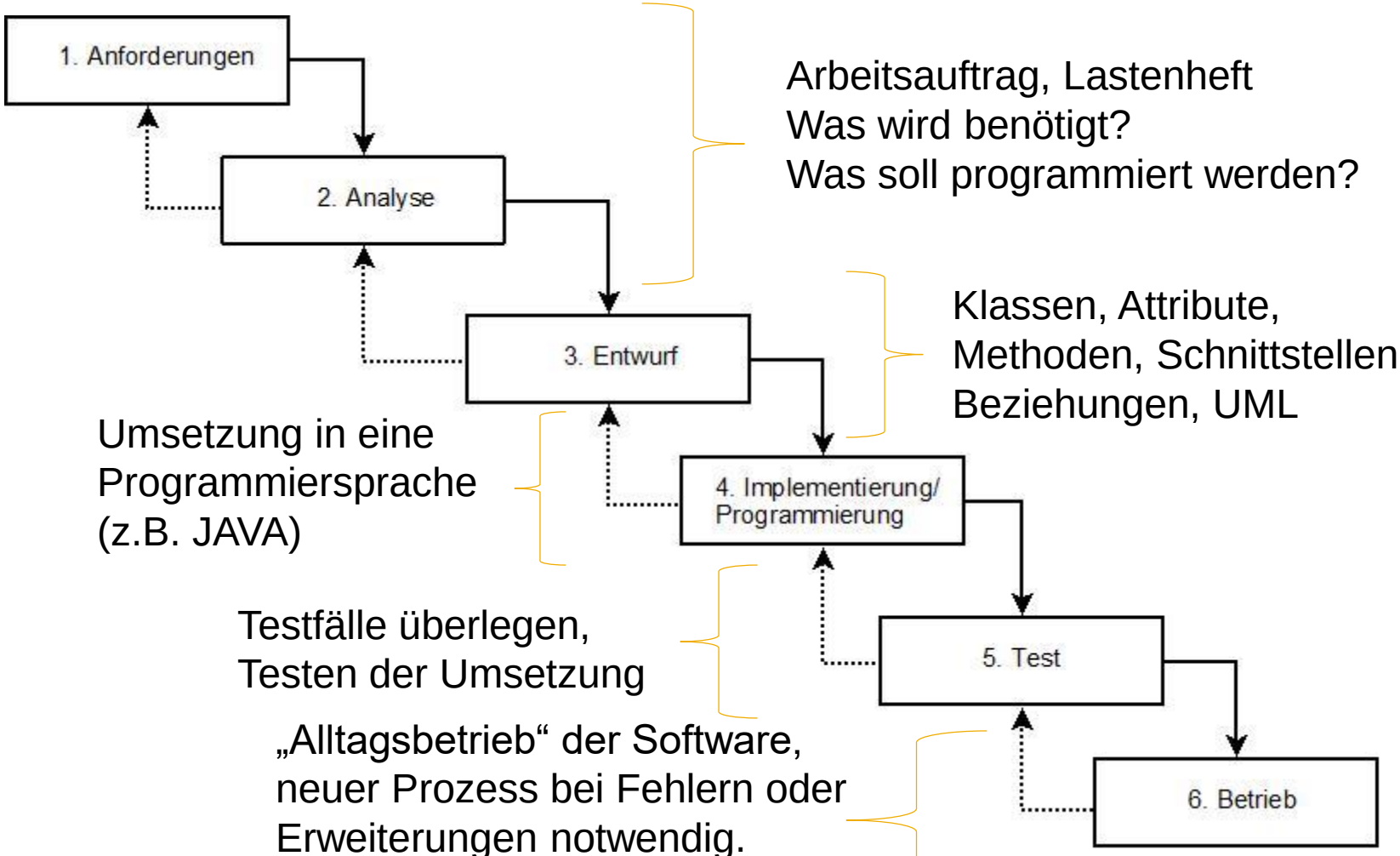
Das „Wasserfallmodell“

- schrittweise und systematische Vorgehensweise in der Softwareentwicklung
- Softwareentwicklungsprozess ist in Phasen organisiert
- jede Phase hat definierte Start- und Endpunkte mit eindeutig definierten Ergebnissen.
- die Ergebnisse einer Phasen sind bindende Vorgaben für die nächst tiefere Phase ein („Wasserfall“).
- „Aufwärtslaufen“ in den Phasen möglich, sofern in der aktuellen Phase etwas schieflaufen sollte, um den Fehler auf der nächsthöheren Stufe beheben zu können.

Das „Wasserfallmodell“



„Wasserfallmodell und OSE“



Strukturierte Programmierung

Stringverarbeitung

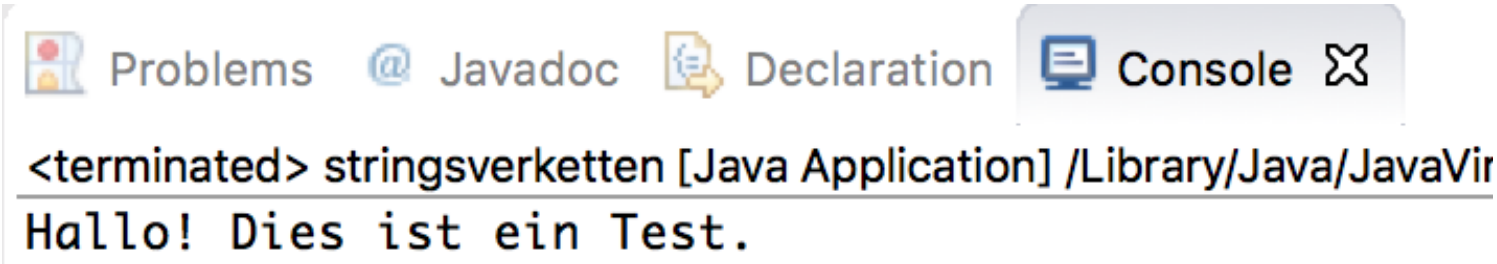
Übersicht | nächste Folie >

Strings verketten

- Mehrere Strings sollen zu einem String zusammengebaut werden.
- Dies geschieht mit dem „+“-Operator.
- Er wird im Prinzip wie die Addition bei Zahlen verwendet, nur dass die Strings verkettet werden.

Strings verketten

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

        String str1="Hallo";
        String str2="Dies";
        String str3="ist";
        String str4="ein";
        String str5="Test";

        String satz = str1+"! "+str2+" "+str3+" "+str4+" "+str5+".";

        System.out.println(satz);

}
}
```



Strings vergleichen

- Zwei Strings sollen miteinander verglichen werden, ob sie denselben Wert besitzen.

Ansatz 1

```
public class string_vergleich {

public static void main(String[] args) {

        String str1=new String("hallo");
        String str2=new String("hallo");


        if (str1 == str2){
                System.out.println("gleich");
        }else{
                System.out.println("nicht gleich");
        }

}
}
```

Auswertung

- Problem: Ansatz 1 erzeugt immer die Ausgabe “nicht gleich“.
- Strings sind keine elementaren Datentypen, sondern Objekte im Speicher.
- str1 und str2 haben zwar **denselben Inhalt** “hallo“, sind aber **verschiedene Objekte** im Speicher.
- str1 == str2 prüft, ob es **dieselben Objekte** sind (was nicht der Fall ist).
- Es wird eine Möglichkeit benötigt, die **Inhalte** zu vergleichen.

Ansatz 2

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

        String str1=new String("hallo");
        String str2=new String("hallo");


        if (str1.equals(str2)){
                System.out.println("gleich");
        } else{
                System.out.println("nicht gleich");
        }

}
}
```

Liefert das gewünschte Ergebnis.

Strings lexikographisch vergleichen

- Zwei Strings sollen miteinander verglichen werden, um festzustellen, welcher String im Alphabet vorher kommt.

Lexikographisch vergleichen

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

        String str1=new String("Berta");
        String str2=new String("Sommer");

        // funktioniert nicht:
        if (str1 < str2) {
                System.out.println(str1 +"steht vor "+str2);
        }

        // Vergleich, ob str1 vor str 2 steht
        if (str1.compareTo(str2)<0) {
                System.out.println(str1 +"steht vor "+str2);
        }

        // Vergleich, ob str1 hinter str 2 steht
        if (str1.compareTo(str2)>0) {
                System.out.println(str1+"steht hinter"+str2);
        }

}
```

Autor:
Christian Frye
Version:
3.8.5

Datum:
20.02.2020

Web:
http://moodle2.erasmus-kittler-schule.de
http://www.eks-darmstadt.de
Mail:
christian.frye@erasmus-kittler-schule.de

zurück zur Übersicht