

Vorlesung



Programmieren I und II

Unit 6

Objektorientierte Programmierung und Unified Modeling Language
(UML)

Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme

1

Disclaimer



Zur rechtlichen Lage an Hochschulen:

Dieses Handout und seine Inhalte sind durch den Autor selbst erstellt. Aus Gründen der Praktikabilität für Studierende lehnen sich die Inhalte stellenweise im Rahmen des Zitatrechts an Lehrwerken an.

Diese Lehrwerke sind explizit angegeben.

Abbildungen sind selber erstellt, als Zitate kenntlich gemacht oder unterliegen einer Lizenz, die nicht die explizite Nennung vorsieht. Sollten Abbildungen in Einzelfällen aus Gründen der Praktikabilität nicht explizit als Zitate kenntlichgemacht sein, so ergibt sich die Herkunft immer aus ihrem Kontext: „Zum Nachlesen ...“.

Creative Commons:

Und damit andere mit diesen Inhalten vernünftig arbeiten können, wird dieses Handout unter einer Creative Commons Attribution-ShareAlike Lizenz (CC BY-SA 4.0) bereitgestellt.



<https://creativecommons.org/licenses/by-sa/4.0>

Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme

2



Prof. Dr. rer. nat. Nane Kratzke

*Praktische Informatik und
betriebliche Informationssysteme*

- Raum: 17-0.10
- Tel.: 0451 300 5549
- Email: kratzke@fh-luebeck.de



@NaneKratzke

Updates der Handouts auch über Twitter #prog_inf und
#prog_itd

Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme

3

Units

Unit 1
Einleitung und
Grundbegriffe

Unit 2
Grundelemente
imperativer Programme

Unit 3
Selbstdefinierbare
Datentypen und
Collections

Unit 4
Einfache I/O
Programmierung

Unit 5
Rekursive
Programmierung und
rekursive
Datenstrukturen

Unit 6
Einführung in die
objektorientierte
Programmierung und
UML

Unit 7
Weitere Konzepte
objektorientierter
Programmiersprachen
(Selbststudium)

Unit 8
Testen
(objektorientierter)
Programme

Unit 9
Generische Datentypen

Unit 10
Objektorientierter
Entwurf und
objektorientierte
Designprinzipien

Unit 11
Graphical User
Interfaces

Unit 12
Multithread
Programmierung

Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme

4

Abgedeckte Ziele dieser UNIT



University of Applied Sciences

Kennen existierender Programmierparadigmen und Laufzeitmodelle

Sicheres Anwenden grundlegender programmiersprachlicher Konzepte (Datentypen, Variable, Operatoren, Ausdrücke, Kontrollstrukturen)

Fähigkeit zur problemorientierten Definition und Nutzung von Routinen und Referenzen (insbesondere Liste, Stack, Mapping)

Verstehen des Unterschieds zwischen Werte- und Referenzsemantik

Kennen und Anwenden des Prinzips der rekursiven Programmierung und rekursiver Datenstrukturen

Kennen des Algorithmusbegriffs, Implementieren einfacher Algorithmen

Kennen objektorientierter Konzepte Datenkapselung, Polymorphie und Vererbung

Sicheres Anwenden programmiersprachlicher Konzepte der Objektorientierung (Klassen und Objekte, Schnittstellen und Generics, Streams, GUI und MVC)

Kennen von UML Klassendiagrammen, sicheres Übersetzen von UML Klassendiagrammen in Java (und von Java in UML)

Kennen der Grenzen des Testens von Software und erste Erfahrungen im Testen (objektorientierter) Software

Sammeln erster Erfahrungen in der Anwendung objektorientierter Entwurfsprinzipien

Sammeln von Erfahrungen mit weiteren Programmiermodellen und -paradigmen, insbesondere Multithread Programmierung sowie funktionale Programmierung

Am Beispiel der Sprache JAVA

Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme

5

Themen dieser Unit



University of Applied Sciences

Warum eigentlich OO?

- Beherrschung von Komplexität
- Kapselung
- Polymorphie
- Abstraktion

Objekte

- haben ein Verhalten
- haben einen (gekapselten) Zustand
- können kommunizieren
- sind unterschiedlich (aber ähnlich, bzw. polymorph)

Modellieren

- Objekte schützen
- Objekte verknüpfen
- Objekte abstrahieren

Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme

6

Zum Nachlesen ...



University of Applied Sciences



Kapitel 1

Einleitung

Kapitel 2

Die Basis der Objektorientierung

Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme

7

Objektorientierung als Mittel zur Beherrschung von Komplexität



University of Applied Sciences

Komplexität

- steigt in der Regel bei einem SW-System mit zunehmender Größe
- senkt häufig die Qualität von SW

Objektorientierung

- Komplexität beherrschbar machen
- Steigerung der Qualität von SW

„Die Techniken der objektorientierten SW-Entwicklung unterstützen [...] dabei, Software einfacher erweiterbar, besser testbar und besser wartbar zu machen.“

[LR09, S. 27]

Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme

8

Grundelemente der Objektorientierung



- **Objektorientierung** kann als ein Werkzeugkasten verstanden werden, um die Zielsetzungen der Entwicklung von Software anzugehen.
- **Basiswerkzeuge** sind:

Kapselung

Poly-morphie

Abstraktion

Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme

9

Vorläufer der objektorientierten Programmierung



- **Prozedurale Programmierung**
- Ausgangspunkt Inhalt eines Computerspeichers
 - Daten
 - Instruktionen

Strukturierung von Instruktionen

- Verzweigungen
- Zyklen
- Routinen mit Aufruf- und Rückgabeparametern

Strukturierung von Daten

- Datentypen
- Zeiger, Records, Arrays, Listen, Bäume, Mengen

Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme

10

Prozedurale Programmierung



Typische (prozedurale) Programmiersprachen

- C
- Pascal
- Fortran
- COBOL

Objektorientierte Erweiterungen

- Kapselung von Daten
- Polymorphie
- Vererbung
- Bspw: geboten durch
 - C++, C#
 - JAVA
 - Python
 - PHP

Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme

11

Verantwortlichkeit des Entwicklers bei prozeduralen Programmiersprachen



Das dies manchmal nicht funktioniert, lassen manche C Programme vermuten.

- ProgrammiererIn hat volle Kontrolle welche Routinen, welche Daten aufrufen.

Kontrolle

- ProgrammiererIn hat auch die Verantwortung, dass die richtigen Routinen die richtigen Daten nutzen.

Verantwortung

Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme

12

Grundelemente der Objektorientierung



University of Applied Sciences

- **Objektorientierung** kann als ein Werkzeugkasten verstanden werden, um die Zielsetzungen der Entwicklung von Software anzugehen.
- **Basiswerkzeuge** sind:

Kapselung

Poly-morphie

Abstraktion

Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme

13

Kapselung von Daten



University of Applied Sciences

Daten gehören einem Objekt



Kein direkter Zugriff auf Daten



Datenzugriff grundsätzlich nur über Methoden eines Objekts

Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme

14

Hintergrund der Datenkapselung



University of Applied Sciences

- Objekt sorgt für Konsistenz seiner Daten
- dient dem Zwecke:
 - Konsistenz der Daten einfacher sicherzustellen
 - Reduktion des Aufwands von Änderungen
 - Änderungen lassen sich auf Einzelobjekte (bzw. deren Klassen) beschränken

Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme

15

Prinzip der Kapselung



University of Applied Sciences

Daten

- Satz von Variablen
- Für jedes Objekt neu angelegt (**Instanzvariablen**)
- Instanzvariablen repräsentieren den **Zustand** eines Objekts
- Zustand eines Objekts kann sich während Lebensdauer ändern
- Zugriff kann eingeschränkt werden

Methoden

- Auf Daten operierende Routinen
- **Methoden** nur einmal vorhanden
- Methoden **operieren aber auf Instanzvariablen**
- Methoden definieren das Verhalten eines Objekts
- Zugriff auf Methoden kann eingeschränkt werden

Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme

16

Daten- und Methodensichtbarkeiten **public, protected und private**



```
class An_Object {  
    public Object forall;  
  
    protected Object forchildren;  
  
    private Object my_eyes_only;  
  
    public Object public_method() {};  
  
    protected Object protected_method() {};  
  
    private Object private_method() {};  
}
```

Details folgen ...

Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme

17

Daten- und Methodensichtbarkeiten **public, protected und private**



Daten- und Methodensichtbarkeiten können dazu genutzt werden

- Daten zu verbergen (zu kapseln)
- Datenzugriffe einzuschränken
- Datenzugriffe nur über definierte Schnittstellen zuzulassen.

- Code zu verbergen (zu kapseln)
- Codeaufrufe einzuschränken
- Codebereiche festzulegen, die für zukünftige Anpassungen gesperrt sind.
- Codebereiche festzulegen, in denen zukünftige Anpassungen stattzufinden haben.

Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme

18

Objekte werden geschützt



Lord Protector lässt nicht mehr alles zu ...

Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme

19

Grundelemente der Objektorientierung



- **Objektorientierung** kann als ein Werkzeugkasten verstanden werden, um die Zielsetzungen der Entwicklung von Software anzugehen.
- **Basiswerkzeuge** sind:

Kapselung

Poly-morphie

Abstraktion

Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme

20

Prinzip der Polymorphie

Polymorphie bedeutet im Wortsinne „Vielgestaltigkeit“

Bsp.: Fassung und Leuchtmittel

Standardisierte Fassungen arbeiten sowohl mit

Klassischen Glühbirnen

Energie-sparlampen

LED-Lampen



Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme

21

Prinzip der Polymorphie

- Einheitliche Schnittstellen
- unterschiedliche Ausprägungen von Funktionalitäten
- dient dem Zwecke:

Bereiche im Code für „Plugins“

Wiederverwendbarkeit von „Meta“funktionalitäten

Wesentlich flexiblere Software

Steigerung der Wartbarkeit und Änderbarkeit

Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme

22

Polymorphie ist so etwas wie die Steckdose der OO-Programmierung



Schließe an was Du willst, Hauptsache es passt in die Steckdose.

(implementiert eine Schnittstelle, bzw. Aufrufsignatur)

Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme

23

Grundelemente der Objektorientierung



- **Objektorientierung** kann als ein Werkzeugkasten verstanden werden, um die Zielsetzungen der Entwicklung von Software anzugehen.
- **Basiswerkzeuge** sind:

Kapselung

Poly-
morphie

Abstraktion

Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme

24

**Objekte sind unterschiedlich
(aber ähnlich)**



University of Applied Sciences



Vieles kann also **wiederverwendet** werden.

Klassen werden uns ermöglichen zu abstrahieren und wiederzuverwenden
bzw. Polymorphie (Vielgestaltigkeit) in unseren Entwurf einzubetten.

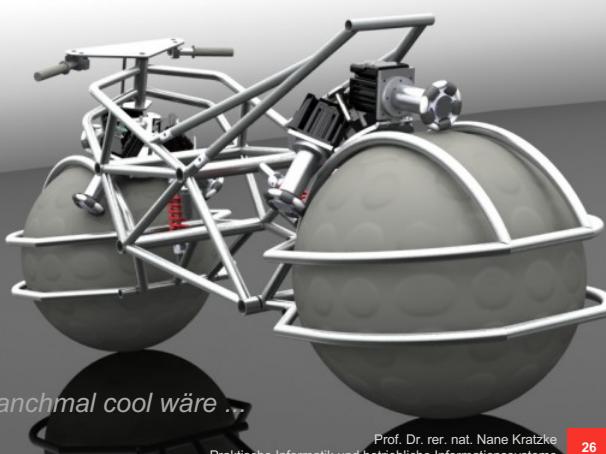
Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme

25

D.h. wir müssen das Rad nicht neu erfinden!



University of Applied Sciences



Auch wenn es vielleicht manchmal cool wäre ...

Quelle: Pixabay

Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme

26

Zusammenfassung



- Objektorientierung ist ein Art Werkzeugkasten, um die Entwicklung und Wiederverwendung von Software zu optimieren (steigende Komplexität größerer SW-Systeme zu beherrschen)
- Einleitung in die Kernkonzepte der Objektorientierung
- **Einheit von**
 - Daten (Zustand eines Objekts) und
 - Code (Verhalten eines Objekts)
- **Kapselung**
- **Polymorphie**
- **Abstraktion**



Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme

27

Themen dieser Unit



Warum eigentlich OO?

- Beherrschung von Komplexität
- Kapselung
- Polymorphie
- Abstraktion

Objekte

- haben ein Verhalten
- haben einen (gekapselten) Zustand
- können kommunizieren
- sind unterschiedlich (aber ähnlich, bzw. polymorph)

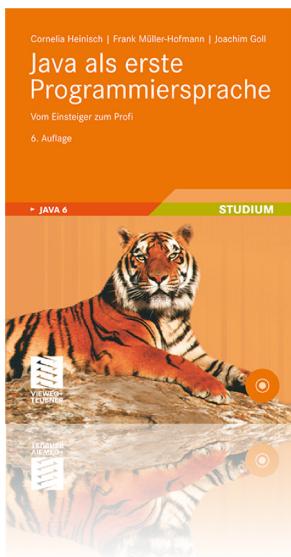
Modellieren

- Objekte schützen
- Objekte verknüpfen
- Objekte abstrahieren

Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme

28

Zum Nachlesen ...



University of Applied Sciences

Kapitel 2

Objektorientierte Konzepte

- 2.1 Modellierung mit Klassen und Objekten
- 2.2 Das Konzept der Kapselung
- 2.3 Abstraktion und Brechung der Komplexität

Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme

29

Noch mehr zum Nachlesen ...



University of Applied Sciences

Kapitel 4

UML Grundlagen

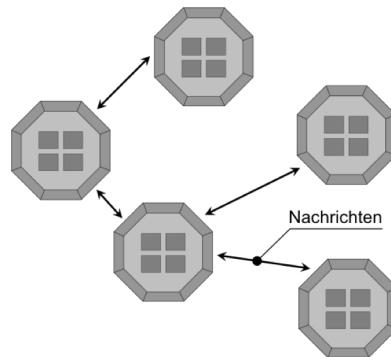
- 4.3.1 Klasse
- 4.4.1 Generalisierung, Spezialisierung
- 4.4.2 – 4.4.5 Assoziation (gerichtet, attribuiert, qualifiziert)
- 4.4.7 – 4.4.8 Aggregation und Komposition

Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme

30

Modellierung mit Klassen und Objekten

- Entscheidend für den objektorientierten Ansatz, ist nicht das objektorientierte Programmieren,
- sondern das Denken in Objekten
- Bei der objektorientierten Modellierung denkt man lange Zeit hauptsächlich im Problembereich



Klassen und UML

- Eine Klasse
 - trägt einen **Klassennamen**
 - enthält **Datenfelder** (Attribute)
 - und **Methoden**, die auf diese Klasse zugreifen.

Punkt
x : int y : int
zeichne() verschiebe() loesche()

Klassename Punkt
Datenfeld x vom Typ int
Datenfeld y vom Typ int

Methode zeichne()
Methode verschiebe()
Methode loesche()

Darstellung einer Klasse mittels UML

Exkurs: UML Unified Modelling Language

- Die Unified Modeling Language (UML) ist eine graphische Modellierungssprache zur
 - Spezifikation,
 - Konstruktion und
 - Dokumentation von (objektorientierter) Software
- UML hat sich insbesondere im OO-Umfeld als Quasistandard etabliert
- UML definiert graphische Notationen (Diagramme) für statische Strukturen und dynamischen Abläufen
- UML wird von der Object Management Group (OMG) entwickelt und ist zertifizierter ISO Standard (ISO/IEC 19501)



Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme

33

Exkurs UML: Diagrammarten

Strukturdiagramme

- **Klassendiagramme**
- Montagediagramm
- Komponentendiagramm
- Verteilungsdiagramm
- Objektdiagramm
- Profildiagramm

Verhaltensdiagramme

- **Aktivitätsdiagramm**
- Use Case Diagramm
- Interaktionsübersichtsdiagramm
- Kommunikationsdiagramm
- **Sequenzdiagramm**
- Zeitverlaufsdiagramm
- **Zustandsdiagramm**

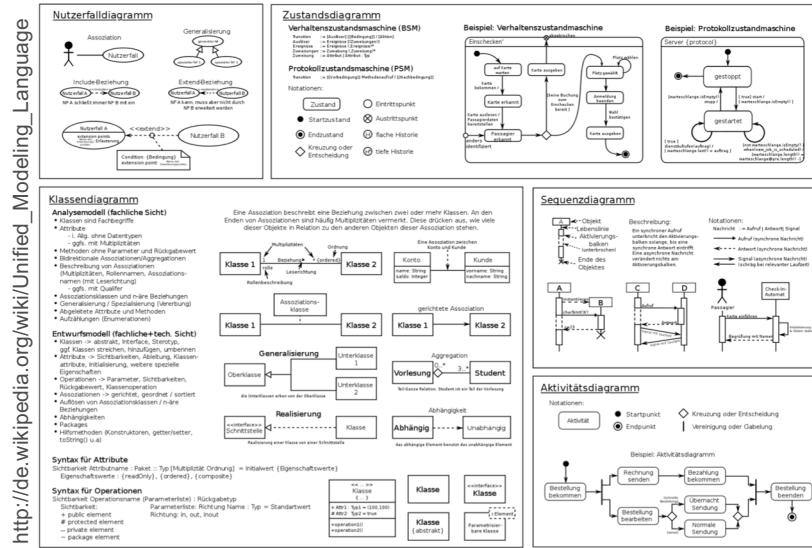
UML kennt die oben stehenden Diagrammarten. Die markierten Diagramme sind die gebräuchlichsten und werden im Rahmen der Vorlesung genutzt.

Die grafische UML-Notation wird Stück für Stück an den geeigneten Stellen im Verlaufe der Vorlesung eingeführt.

Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme

34

Exkurs UML: Diagramm Übersicht nur zur Information



Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme

35

Klassen und UML

Punkt
x : int y : int
zeichne() verschiebe() loesche()

Klassenname Punkt
Datenfeld x vom Typ int
Datenfeld y vom Typ int
Methode zeichne()
Methode verschiebe()
Methode loesche()

```
class Punkt {

    int x;
    int y;

    void zeichne() { ... }
    void verschiebe() { ... }
    void loesche() { ... }

}
```

UML

JAVA

Derselbe Sachverhalt – andere Notation

Im Rahmen dieser Vorlesung wird UML primär zur Darstellung struktureller oder ablauforientierter Sachverhalte genutzt und JAVA für programmiertechnische Implementierungen.

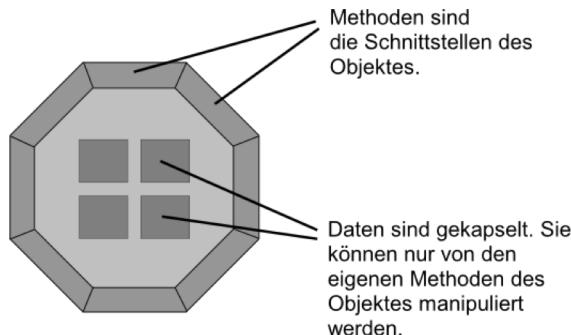
Beide Formen werden aber parallel genutzt. Lauffähig programmieren lässt sich übrigens nur in JAVA.

Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme

36

Klassen und Objekte

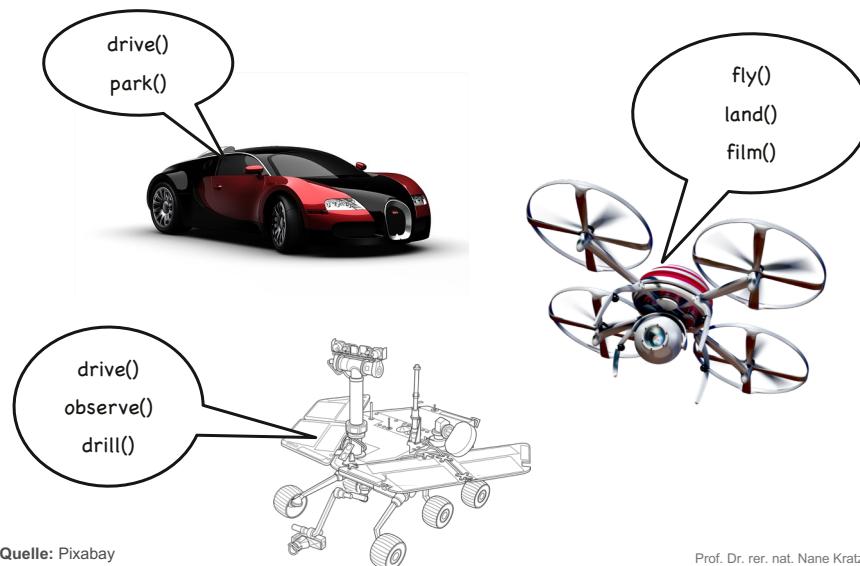
- Bei der Objektorientierung werden die
 - Daten eines Objektes und
 - Die Daten verändernden Methoden
 - als eine Einheit betrachtet – das Objekt.



Klassen und Objekte

- Methoden** erfüllen die Aufgaben:
 - Werte der Datenfelder **auszugeben**.
 - Datenfelder zu **verändern**.
 - Neue Ergebnisse mittels in Datenfeldern gespeicherter Werte zu **berechnen**.
- Datenfelder** definieren mögliche **Zustände** der Objekte (Datenstruktur),
- die **Methoden** bestimmen das **Verhalten** der Objekte.

Objekte haben ein Verhalten



Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme

39

Objekte haben ein Verhalten (I)

Bislang haben wir Objekte (Instanzen von Klassen) nur als strukturierte Datentypen ohne nennenswertes Verhalten kennen gelernt (bspw. Adresse). Objekte können jedoch auch ein Verhalten zeigen.

Dieses Verhalten wird durch die Methoden des Objekts (eigentlich der Klasse, dazu später mehr) definiert.

Wir definieren nun zwei Klassen, um freundliche und unfreundliche Personen erzeugen zu können (d.h. mit freundlichem und unfreundlichem Verhalten).

Objekte der Klasse `FriendlyPerson` zeigen ein anderes Verhalten als Objekte der Klasse `UnfriendlyPerson`.

Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme

40

Objekte haben ein Verhalten (II)

```
public class FriendlyPerson {  
    public String name;  
    public FriendlyPerson(String n) { this.name = n; }  
  
    public void sayHello() {  
        System.out.println("[" + this + "]: Hi, I am " + this);  
    }  
    public String toString() { return name; }  
}  
  
public class UnfriendlyPerson {  
    public String name;  
    public UnfriendlyPerson(String n) { this.name = n; }  
  
    public void sayHello() {  
        System.out.println("[" + this + "]: Go away. I am busy.");  
    }  
    public String toString() { return name; }  
}
```



University of Applied Sciences

Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme

41

Objekte haben ein Verhalten (III)



University of Applied Sciences

```
FriendlyPerson p1 = new FriendlyPerson("Max");  
UnfriendlyPerson p2 = new UnfriendlyPerson("Moritz");  
p1.sayHello();  
p2.sayHello();
```

Ergibt auf der Konsole:

```
[Max]: Hi, I am Max.  
[Moritz]: Go away. I am busy.
```

D.h. Max und Moritz zeigen ein anderes Verhalten
(aufgrund ihrer Programmierung).

!!! Methoden definieren das Verhalten von Objekten !!!

Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme

42

Objekte haben einen (inneren) Zustand



Quelle: Pixabay

Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme

43

Objekte haben einen Zustand (I)

```
public class Person {  
    public String name;  
    public Person(String n) { this.name = n; }  
  
    public void sayHello() {  
        System.out.println("[" + this + "]: Hi, I am " + this);  
    }  
    public String toString() { return name; }  
}
```

Datenfelder eines Objekts definieren die Zustände, die ein Objekt annehmen kann.

```
Person p1 = new Person("Max");  
Person p2 = new Person("Maya");  
p1.sayHello();  
p2.sayHello();
```

Hier besteht der Zustand einer Person nur aus einem Namen.

Ergibt auf der Konsole:

```
[Max]: Hi, I am Max.  
[Maya]: Hi, I am Maya.
```

Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme

44

Objekte haben einen Zustand (II)



University of Applied Sciences

- Ein Objekt hat einen Satz von Datenfeldern (und Methoden)
- Jedes Datenfeld hat Werte
- **Zustand eines Objekts == momentane Wertbelegung der Datenfelder des Objekts**
- **Beispiel Fahrstuhl**
 - Gewichtssensor im Fahrstuhl
 - **Mikroskopischer Zustand** des Fahrstuhls == aktueller Wert des Sensors
 - **Makroskopischer Zustand** des Fahrstuhls == Überladen oder nicht Überladen



Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme

45

Objekte haben einen Zustand (III)



University of Applied Sciences

```

public class SemiFriendlyPerson {
    public String name;
    public int helloCounter;

    public SemiFriendlyPerson(String n) { name = n; }

    public void sayHello() {
        helloCounter++;
        if (helloCounter < 5) {
            System.out.println(this + "Hi, I am " + name);
        } else {
            System.out.println(this + "Hi");
        }
    }

    public String toString() { return "[" + name + "]: " ; }
}
  
```

Hier haben wir einen Zustand bestehend aus zwei Datenfeldern. **sayHello()** ändert nun zudem den **Zustand** des Objekts und sein **Verhalten** ist **abhängig** vom **Zustand**.

Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme

46

Objekte haben einen Zustand (IV)



University of Applied Sciences

```
SemiFriendlyPerson p3 = new SemiFriendlyPerson("Willi");
p3.sayHello();
p3.sayHello();
p3.sayHello();
p3.sayHello();
p3.sayHello();
```

Ergibt auf der Konsole:

```
[Willi]: Hi, I am Willi
[Willi]: Hi
```

Das Verhalten von Willi ändert sich nach dem fünften Methodenaufruf von sayHello() aufgrund seines Zustands (*vielleicht ist er müde die ganze Zeit zu grüßen*).

Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme

47

Objekte haben einen Zustand (V)



University of Applied Sciences

```
public class SemiFriendlyPerson {
    public String name;
    public int helloCounter;

    public SemiFriendlyPerson(String n) { name = n; }

    public boolean tiredToGreet() { return helloCounter >= 5; }

    public void sayHello() {
        helloCounter++;
        if (tiredToGreet()) { System.out.println(this + "Hi"); }
        else { System.out.println(this + "Hi, I am " + name); }
    }

    public String toString() { return "[" + name + "]: "; }
}
```

„Zustandsgruppen“ (Makrozustand) die das Verhalten eines Objekts beeinflussen werden häufig (aber nicht immer, Klausur !!!) als boolesche Methoden definiert.
Gleichzeitig machen sie den Code so häufig lesbarer („natürlich sprachlicher“).

Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme

48

Miniübung:



Es kann aber natürlich auch komplexere Zustände geben (aus mehr als einem Datenfeld). Methoden können den Zustand eines Objekts verändern.

```
class Auto {  
    private double fuel = 0.0;  
    private double kmstand = 0.0;  
  
    public Auto() {  
        this.fuel = 5.0;  
    }  
  
    public void tanke(double l) {  
        this.fuel += l;  
    }  
  
    public void fahre(double km) {  
        this.kmstand += km;  
        this.fuel -= 7.0 * km / 100;  
    }  
}
```

Geben Sie den Mikrozustand des erzeugten Objekts nach den entsprechenden Methodenaufrufen an.

```
Auto car = new Auto();
```

```
car.tanke(50.0);
```

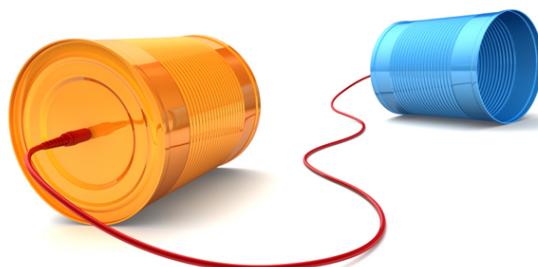
```
car.fahre(50.0);
```

```
car.fahre(200.0);  
car.tanke(10.0);
```

Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme

49

Objekte können kommunizieren



Quelle: Pixabay

Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme

50

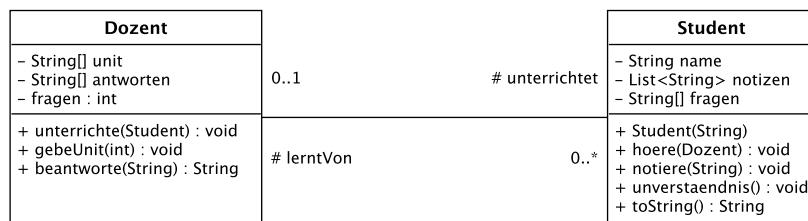
Objekte können kommunizieren



University of Applied Sciences

Damit Objekte miteinander kommunizieren (d.h. sich gegenseitig ihre Methoden aufrufen) können, müssen sie einander kennen.

Auf Ebene von UML kann man solch eine Kenntnisbeziehung als **Assoziation** modellieren. UML Assoziationen lassen sich programmiertechnisch als Referenzen auf (Listen von) Objekte(n) abbilden.



Hier einmal das Beispiel, dass ein Dozent mehrere Studenten unterrichtet und ein Student von maximal einem Dozenten unterrichtet wird (zu einem Zeitpunkt). Studenten notieren dabei Inhalte und können Fragen stellen (bei Unverständnis). Dozenten geben Units und beantworten Fragen.

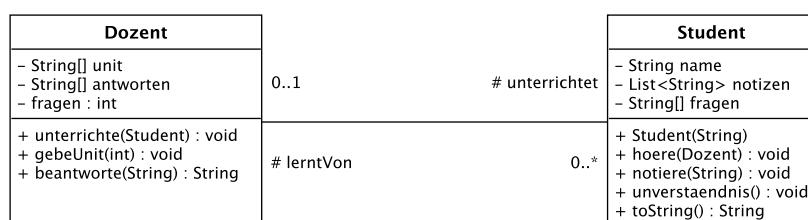
Prof. Dr. rer. nat. Nane Kratzke
 Praktische Informatik und betriebliche Informationssysteme

51

Objekte kennen sich (mittels Referenzen)



University of Applied Sciences



```
public class Student {
    protected Dozent lerntVon;
}
```

Jeder Student kennt also seinen Dozenten (lerntVon).

```
public class Dozent {
    protected List<Student> unterrichtet = new LinkedList<Student>();
}
```

Jeder Dozent kennt seine Studenten (unterrichtet).

Prof. Dr. rer. nat. Nane Kratzke
 Praktische Informatik und betriebliche Informationssysteme

52

Ein kommunizierender Dozent



University of Applied Sciences

```

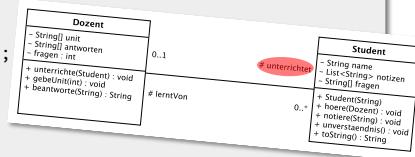
public class Dozent {
    private String[] unit = {
        "Ein Objekt hat ein Verhalten.", "Ein Objekt hat einen Zustand.",
        "Ein Objekt kann kommunizieren.", "Ein Objekt ist vielgestaltig." };
    private String[] antworten = { "Eine sehr gute Frage.",
        "Bitte arbeiten Sie dies zu Hause durch.", "Dazu kommen wir noch." };
    private int fragen;
    protected List<Student> unterrichtet = new LinkedList<Student>();

    public void unterrichte(Student s) { unterrichtet.add(s); s.hoere(this); }

    public String beantwortete(String s) {
        return antworten[fragen++ % antworten.length];
    }

    public void gebeUnit(int n) {
        for (Student s : unterrichtet) { s.notiere(unit[n]); }
    }
}

```



Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme

53

Ein kommunizierender Student

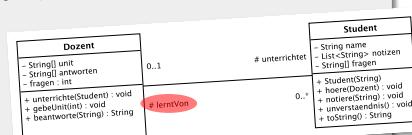


```

public class Student {
    private String name;
    private String[] fragen = { "Gibt es dazu mal ein Beispiel?", "Das war mir viel zu schnell!", "Fehlt da nicht ein Semikolon?" };
    private List<String> notizen = new LinkedList<String>();
    protected Dozent lerntVon;

    public Student(String n) { this.name = n; }
    public void hoere(Dozent d) { this.learntVon = d; }
    public void unverstaendnis() {
        if (this.learntVon == null) return;
        Random r = new Random();
        String frage = this.fragen[r.nextInt(this.fragen.length)];
        String antwort = this.learntVon.beantwortete(frage);
        System.out.println(this.name + ": " + frage + " Dozent: " + antwort);
    }
    public void notiere(String s) { this.notizen.add("- " + s); }
    public String toString() {
        String ret = "Notizen von: " + name + "\n";
        for (String notiz : notizen) ret += notiz + "\n";
        return ret;
    }
}

```



Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme

54

Eine exemplarische Kommunikation Dozent -> Student

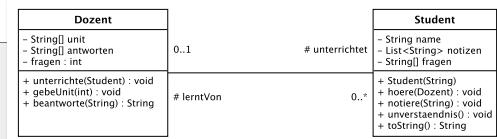


```
Dozent d = new Dozent();
Student[] students = {
    new Student("Max"),
    new Student("Maren"),
    new Student("Tessa")
};

for (Student s : students) d.unterrichte(s);

d.gebeUnit(0);
d.gebeUnit(2);
d.gebeUnit(1);

for (Student s : students) {
    System.out.println(s);
}
```



Notizen von: Max

- Ein Objekt hat ein Verhalten.
- Ein Objekt kann kommunizieren.
- Ein Objekt hat einen Zustand.

Notizen von: Maren

- Ein Objekt hat ein Verhalten.
- Ein Objekt kann kommunizieren.
- Ein Objekt hat einen Zustand.

Notizen von: Tessa

- Ein Objekt hat ein Verhalten.
- Ein Objekt kann kommunizieren.
- Ein Objekt hat einen Zustand.

Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme

55

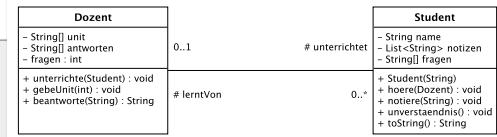
Eine exemplarische Kommunikation Student -> Dozent



```
Dozent d = new Dozent();
Student[] students = {
    new Student("Max"),
    new Student("Maren"),
    new Student("Tessa")
};

for (Student s : students) d.unterrichte(s);

for (Student s : students) {
    s.unverstaendnis();
}
```



Nur so am Rande:

Die Kommunikation bleibt in unserem Beispiel dieselbe, wenn wir die Stoffvermittlung sein lassen ;)

Max: Gibt es dazu mal ein Beispiel? Dozent: Eine sehr gute Frage.
 Maren: Das war mir viel zu schnell! Dozent: Bitte arbeiten Sie dies zu Hause durch.
 Tessa: Fehlt da nicht ein Semikolon? Dozent: Dazu kommen wir noch.

Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme

56

Objekte sind unterschiedlich (aber ähnlich)



University of Applied Sciences



also polymorph (vielgestaltig)

Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme

57

Objekte sind vielgestaltig



University of Applied Sciences

Ein berechtigter Einwand an unserem Beispiel wäre, dass nicht alle Studierende gleich sind.

Es gibt bspw. unterschiedliche Strategien Notizen anzufertigen.

- Der SkriptStudent notiert sich gar nichts und vertraut aufs Skript.
- Der EifrigStudent notiert alles und sicherheitshalber doppelt und mit Ausrufezeichen.
- Der LazyStudent notiert sich Teile (so zu etwa 50%).
- Der TiltedStudent schafft es nicht mehr als zwei Units zu notieren.
- Der EmotionaleStudent notiert mehr seine Empfindungen, weniger den Inhalt.

Alle Strategien ändern nichts an der Tatsache, dass Objekte dieser Klassen Studenten bleiben. Der Dozent nimmt auf diese unterschiedlichen Strategien auch gar keine Rücksicht, sondern behandelt alle weiter als Student.

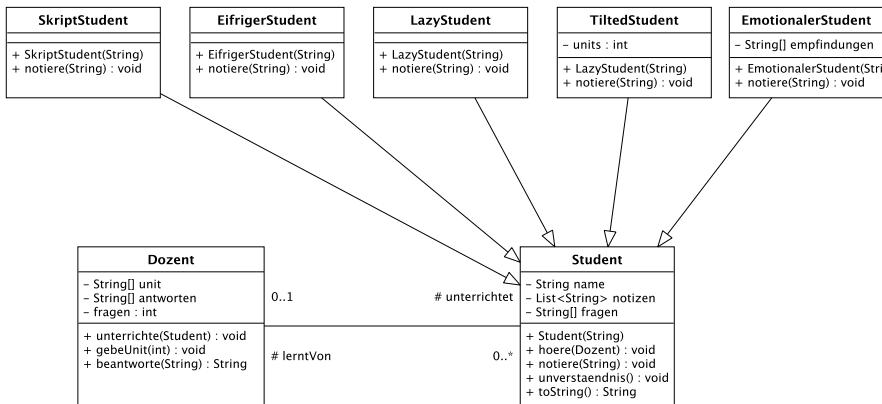
Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme

58

Vielgestaltige Studenten



University of Applied Sciences



Jetzt könnte man diese Strategien alle als eigene Klassen von Grund auf neu implementieren. Geschickter ist es jedoch ein bestehendes Konzept (**Student**) einfach zu erweitern und nur das geänderte Verhalten (**notiere**) neu zu implementieren.

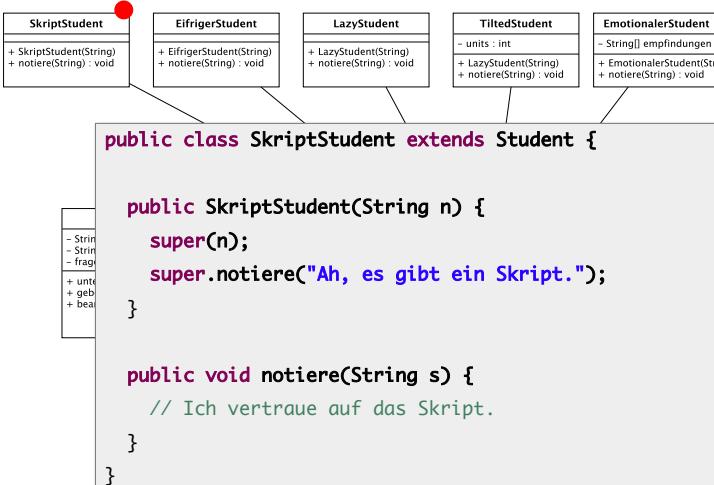
Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme

59

Vielgestaltige Studenten



University of Applied Sciences



Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme

60

Vielgestaltige Studenten

FACH
HOCHSCHULE
LÜBECK
University of Applied Sciences

```

classDiagram
    class SkriptStudent {
        + SkriptStudent(String)
        + notiere(String) : void
    }
    class EifrigerStudent {
        + EifrigerStudent(String)
        + notiere(String) : void
    }
    class LazyStudent {
        + LazyStudent(String)
        + notiere(String) : void
    }
    class TiltedStudent {
        - units : int
        + LazyStudent(String)
        + notiere(String) : void
    }
    class EmotionalerStudent {
        - String[] empfindungen
        + EmotionalerStudent(String)
        + notiere(String) : void
    }

    SkriptStudent <|-- EifrigerStudent
    SkriptStudent <|-- LazyStudent
    SkriptStudent <|-- TiltedStudent
    SkriptStudent <|-- EmotionalerStudent
  
```

```

public class EifrigerStudent extends Student {

    public EifrigerStudent(String n) {
        super(n);
    }

    public void notiere(String s) {
        super.notiere(s + " !!!");
        super.notiere("!!! " + s + " (Nacharbeiten !)");
    }
}
  
```

- String
- String
- frag
+ unte
+ geb
+ bea

Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme 61

Vielgestaltige Studenten

FACH
HOCHSCHULE
LÜBECK
University of Applied Sciences

```

classDiagram
    class SkriptStudent {
        + SkriptStudent(String)
        + notiere(String) : void
    }
    class EifrigerStudent {
        + EifrigerStudent(String)
        + notiere(String) : void
    }
    class LazyStudent {
        + LazyStudent(String)
        + notiere(String) : void
    }
    class TiltedStudent {
        - units : int
        + LazyStudent(String)
        + notiere(String) : void
    }
    class EmotionalerStudent {
        - String[] empfindungen
        + EmotionalerStudent(String)
        + notiere(String) : void
    }

    SkriptStudent <|-- EifrigerStudent
    SkriptStudent <|-- LazyStudent
    SkriptStudent <|-- TiltedStudent
    SkriptStudent <|-- EmotionalerStudent
  
```

```

public class LazyStudent extends Student {

    public LazyStudent(String n) {
        super(n);
        super.notiere("Jamaica, man!");
    }

    public void notiere(String s) {
        super.notiere(s.substring(0, s.length() / 2));
    }
}
  
```

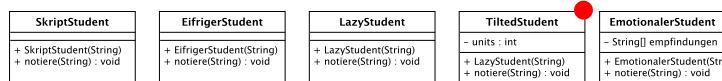
- String
- String
- frag
+ unte
+ geb
+ bea

Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme 62

Vielgestaltige Studenten



University of Applied Sciences



```

public class TiltedStudent extends Student {

    private int units;

    public TiltedStudent(String n) { super(n); }

    private boolean overflow() { return this.units++ >= 2; }

    public void notiere(String s) {
        if (overflow()) { super.notiere("Häh? Tilt ..."); }
        else           { super.notiere(s); }
    }
}
  
```

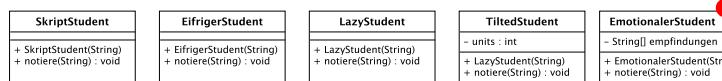
Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme

63

Vielgestaltige Studenten



University of Applied Sciences



```

public class EmotionalerStudent extends Student {

    private String[] empfindungen = {
        "Was für ein schöner Sonnenaufgang.", "Wieso immer ich?",  

        "Informatik ist so spannend!", "Wieso nur Informatik?",  

        "Ich hasse Klausuren.", "Gruppenarbeit ist toll. Das ist so dynamisch.",  

        "Objektorientierung ist super.", "Objektorientierung. Wie banal!"
    };

    public EmotionalerStudent(String n) { super(n); }

    public void notiere(String s) {
        Random r = new Random();
        super.notiere(empfindungen[r.nextInt(empfindungen.length)]);
    }
}
  
```

Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme

64

Eine exemplarische Kommunikation Dozent -> Student



University of Applied Sciences

```
Dozent d = new Dozent();
Student[] students = {
    new SkriptStudent("Max"),
    new EifrigerStudent("Maren"),
    new LazyStudent("Tessa"),
    new TiltedStudent("Moritz"),
    new EmotionalerStudent("Maya")
};

for (Student s : students) d.unterrichte(s);
d.gebeUnit(0);
d.gebeUnit(2);
d.gebeUnit(1);
d.gebeUnit(3);

for (Student s : students) {
    System.out.println(s);
}
```

Der Dozent spricht alle Objekte einheitlich als Student an. Aber jedes Objekt zeigt jetzt ein anderes Verhalten.

Notizen von: Max
- Ah, es gibt ein Skript.
Notizen von: Maren
- Ein Objekt hat ein Verhalten. !!!
- !!! Ein Objekt hat ein Verhalten. (Nacharbeiten !)
- Ein Objekt kann kommunizieren. !!!
- !!! Ein Objekt kann kommunizieren. (Nacharbeiten !)
- Ein Objekt hat einen Zustand. !!!
- !!! Ein Objekt hat einen Zustand. (Nacharbeiten !)
- Ein Objekt ist vielseitig. !!!
- !!! Ein Objekt ist vielseitig. (Nacharbeiten !)

Notizen von: Tessa
Jamaica, man!
- Ein Objekt hat
- Ein Objekt kann
- Ein Objekt hat
- Ein Objekt ist

Notizen von: Moritz
- Ein Objekt hat ein Verhalten.
- Ein Objekt kann kommunizieren.
- Häh? Tilt ...
- Häh? Tilt ...

Notizen von: Maya
- Objektorientierung ist super.
- Ich hasse Klausuren.
- Was für ein schöner Sonnenaufgang.
- Objektorientierung. Wie banal!

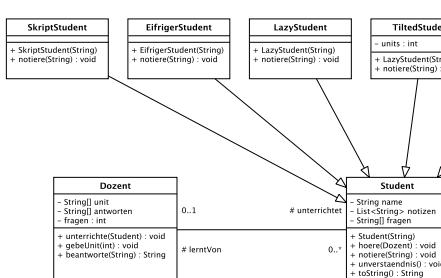
Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme

65

Eine exemplarische Kommunikation Dozent -> Student



University of Applied Sciences



Der Dozent spricht alle Objekte einheitlich als Student an. Aber jedes Objekt zeigt jetzt ein anderes Verhalten.

Den Großteil der Logik müssen wir also nicht anpassen. Den Dozenten interessiert es nicht einmal! Trotzdem funktioniert es.

Notizen von: Max
- Ah, es gibt ein Skript.
Notizen von: Maren
- Ein Objekt hat ein Verhalten. !!!
- !!! Ein Objekt hat ein Verhalten. (Nacharbeiten !)
- Ein Objekt kann kommunizieren. !!!
- !!! Ein Objekt kann kommunizieren. (Nacharbeiten !)
- Ein Objekt hat einen Zustand. !!!
- !!! Ein Objekt hat einen Zustand. (Nacharbeiten !)
- Ein Objekt ist vielseitig. !!!
- !!! Ein Objekt ist vielseitig. (Nacharbeiten !)

Notizen von: Tessa
Jamaica, man!
- Ein Objekt hat
- Ein Objekt kann
- Ein Objekt hat
- Ein Objekt ist

Notizen von: Moritz
- Ein Objekt hat ein Verhalten.
- Ein Objekt kann kommunizieren.
- Häh? Tilt ...
- Häh? Tilt ...

Notizen von: Maya
- Objektorientierung ist super.
- Ich hasse Klausuren.
- Was für ein schöner Sonnenaufgang.
- Objektorientierung. Wie banal!

Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme

66

Themen dieser Unit

Warum eigentlich OO?

- Beherrschung von Komplexität
- Kapselung
- Polymorphie
- Abstraktion

Objekte

- haben ein Verhalten
- haben einen (gekapselten) Zustand
- können kommunizieren
- sind unterschiedlich (aber ähnlich, bzw. polymorph)

Modellieren

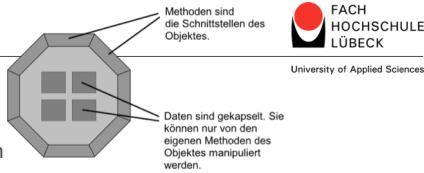
- Objekte schützen
- Objekte verknüpfen
- Objekte abstrahieren

Objekte schützen



Konzept der Kapselung

In der Objektorientierung betrachtet man Daten und Methoden als eine zusammengehörende Einheit. Die folgenden Begriffe sind dabei von Bedeutung:



Abstraktion

- Komplexer Sachverhalt der realen Welt
- wird auf das Wesentliche reduziert
- und vereinfacht dargestellt
- Datenfelder und Methoden eines Objekts repräsentieren diejenigen Daten und das Verhalten von Bedeutung für den Probletraum

Kapselung

- Objekt implementiert sein Verhalten in Schnittstellenmethoden
- Ein Objekt sollte (im Idealfall) nur über definierte Schnittstellenmethoden mit seiner Umwelt in Kontakt treten

Information Hiding

- Innere Daten eines Objekts sollen nach außen nicht direkt sichtbar sein
- Innere Eigenschaften eines Objekts sollen verborgen sein
- Ein Objekt sollten nichts von inneren Implementierungs-details eines anderen Objekts wissen müssen

Ein Objekt sollte also keine Kenntnisse über den inneren Aufbau anderer Objekte haben. Programmietechnische Änderungen innerhalb von Klassen (und daraus instantiierten Objekten) ziehen so keine Änderungen außerhalb der geänderten Klassen nach sich, solange die Schnittstellen gleich bleiben.

Information Hiding Zugriffsschutz für Methoden und Datenfelder

Objektorientierte Sprachen kennen
üblicherweise die folgenden
Zugriffsmodifikatoren

public

protected

private

Restriktivere Zugriffsrechte

Zusätzlich gibt es noch den impliziten Zugriffsmodifikator default, der gilt, wenn keiner der drei oberen gesetzt wird. Darüberhinaus gibt es noch ein paar mehr Feinheiten im Zusammenhang mit Packages, diese werden aber erst in der Unit 9 behandelt.

Zugriffsmodifikatoren UML

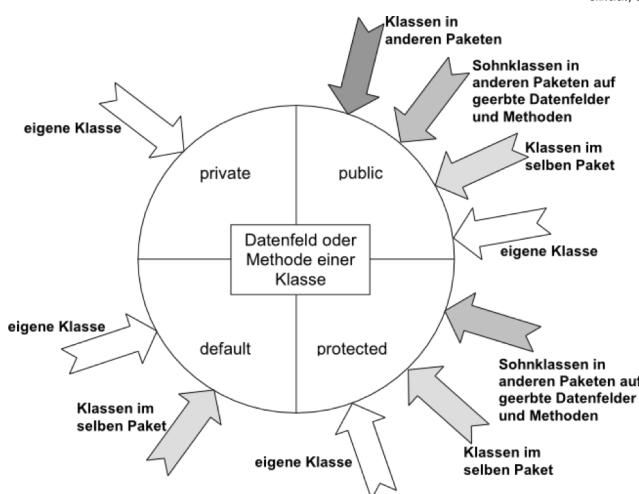
Um die Zugriffsmodifikatoren
 •public,
 •protected,
 •private und
 •package/default
 nicht immer in UML Diagrammen
 ausschreiben zu müssen,
 werden auch die folgenden
 abkürzenden Symbole +, #, -, ~
 genutzt.

Example

+ public_datenfeld : Type
protected_datenfeld : Type
- private_datenfeld : Type
~ package_datenfeld : Type

+ public_methode() : Type
protected_methode() : Type
- private_methode() : Type
~ package_methode() : Type

Zugriffsschutz im Überblick



Auf Besonderheiten im Zusammenhang mit Paketen und dem Zugriffsmodifikator default
 bitte Selbststudy Unit durcharbeiten.

Information Hiding



University of Applied Sciences

- Ein Ziel der Objektorientierung ist es,
- die Repräsentation der Daten und
- die Implementierung der Daten zu verbergen.
- Es soll kein Unbefugter die Daten verändern können.
- Nur Methoden des Objekts sollten auf die Daten des Objekts Zugriff haben.

Folgende Klasse ist zwar korrektes JAVA, befolgt aber nicht das Prinzip des Information Hiding.

```
class Person {  
    public String name;  
    public String nachname;  
    public int alter;  
  
    public void print() { ...  
        System.out.println(name);  
        System.out.println(nachname);  
        System.out.println(alter);  
    }  
}
```

Datenfelder des Objekts, sind von „außen“ zugreifbar und veränderbar.

```
Person p = new Person();  
p.name = "Max";  
p.nachname = "Mustermann";  
p.alter = 35;  
p.print();
```

Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme

73

Information Hiding (II)



University of Applied Sciences

„Objektorientierter“ wäre eine Realisierung, wie die folgende:

```
class Person {  
    private String name;  
    private String nachname;  
  
    public Person(String n, String nn) {  
        name = n; nachname = nn;  
    }  
  
    public void print() { ...  
        System.out.println(name);  
        System.out.println(nachname);  
    }  
  
    public String getName() {  
        return name;  
    }  
  
    public String getNachname() {  
        return nachname;  
    }  
}
```

- Somit kein direkter Zugriff mehr auf Datenfelder von Personenobjekten
- **private** ist ein sogenannter Zugriffsmodifikator

Da **name** und **nachname** als **private** deklariert wurden, können Sie nur innerhalb durch Objekte der Klasse **Person** geändert werden, nicht von außen.

Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme

74

Miniübung:



Gegeben ist folgende Klassendefinition.

```
class Auto {  
    private double fuel = 0.0;  
    private double kmstand = 0.0;  
  
    public Auto() {  
        this.fuel = 5.0;  
    }  
  
    public void tanke(double l) {  
        this.fuel += l;  
    }  
  
    public void fahre(double km) {  
        this.kmstand += km;  
        this.fuel -= 7.0 * km / 100;  
    }  
}
```

Geben Sie nun sinnvolle getter und setter Methoden an, um den Kilometerstand und den Tankstand auslesen und setzen zu können. Achten Sie auf sinnvolle Zugriffsmodifikatoren!

Tankstand

Kilometerstand

Miniübung:



Gegeben ist folgende Klassendefinition plus gerade vorgenommener Ergänzungen.

```
class Auto {  
    private double fuel = 0.0;  
    private double kmstand = 0.0;  
  
    public Auto() {  
        this.fuel = 5.0;  
    }  
  
    public void tanke(double l) {  
        this.fuel += l;  
    }  
  
    public void fahre(double km) {  
        this.kmstand += km;  
        this.fuel -= 7.0 * km / 100;  
    }  
}
```

Geben Sie nun eine sinnvolle Implementierung an, den Makrozustand hinsichtlich des Tankzustands (kaum noch Benzin) eines Autoobjekts zu bestimmen.

Kaum noch Benzin

Miniübung:



University of Applied Sciences

Gegeben ist folgende Klassendefinition plus gerade vorgenommener Ergänzungen.

```
class Auto {  
    private double fuel = 0.0;  
    private double kmstand = 0.0;  
  
    public Auto() {  
        this.fuel = 5.0;  
    }  
  
    public void tanke(double l) {  
        this.fuel += l;  
    }  
  
    public void fahre(double km) {  
        this.kmstand += km;  
        this.fuel -= 7.0 * km / 100;  
    }  
}
```

Geben Sie nun eine sinnvolle Implementierung an, den Makrozustand hinsichtlich des Wartungsstands zu bestimmen (alle 20.000km zur Inspektion).

Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme

77

Miniübung:



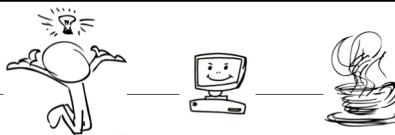
University of Applied Sciences

Geben Sie nun bitte die UML Notation der gerade definierten Klassen Auto und InspAuto an.

Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme

78

Miniübung:



Sie sollen nun Personen weiterhin wie folgt anlegen können.

```
Person p1 = new Person("Max", "Mustermann");
Person p2 = new Person("Maren", "Musterfrau");
Person p3 = new Person("Tessa", "Loniki");
```

Jedoch auf die einzelnen Namensbestandteile zielgerichtet zugreifen können.

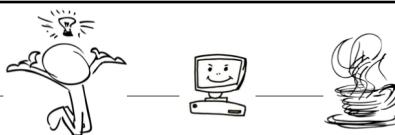
```
System.out.println(p2.getNachname());
System.out.println(p1.getVorname());
System.out.println(p3.getVorname() + " " + p3.getNachname());
```

Es soll folgendes auf der Konsole ausgegeben werden.

```
Musterfrau
Max
Tessa Loniki
```

Bitte geben Sie eine Implementierung für Person an, die entsprechende getter Methoden implementiert.

Miniübung:



Miniübung:



Sie sollen nun Personen weiterhin wie folgt anlegen können.

```
Person p1 = new Person("Max", "Mustermann");
Person p2 = new Person("Maren", "Musterfrau");
Person p3 = new Person("Tessa", "Loniki");
```

Jedoch nachträglich Nachnamen sinnvoll ändern können.

```
p2.setNachname("Mustermann");
System.out.println(p2);
```

Es soll dann folgendes auf der Konsole ausgegeben werden.

```
Maren Mustermann
```

Werden sinnlose Werte wie "" oder null als Nachname gesetzt, soll nichts im Objekt geändert werden. Die Methode soll aber false als Rückgabe liefern. Wird etwas geändert, soll sie true liefern.

```
p1.setNachname("") == false      => p1 bleibt Max Mustermann
p1.setNachname(null) == false    => p1 bleibt Max Mustermann
p1.setNachname("Müller") == true => p1 wird Max Müller
```

Miniübung:



Objekte verknüpfen



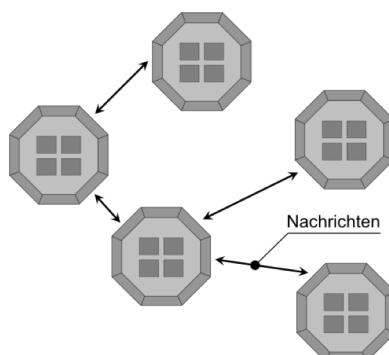
Quelle: Pixabay

Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme

83

Zusammenarbeit von Objekten Objektkommunikation

- Objektorientierte Systeme erbringen ihre Leistung durch das Zusammenwirken von Objekten
- in dem Nachrichten zwischen Objekten ausgetauscht werden
- (in JAVA entspricht dies Methodenaufrufen)



Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme

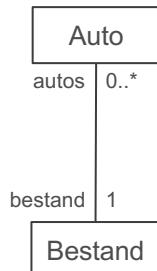
84

Assoziation zwischen Objekten



University of Applied Sciences

Assoziation in JAVA



Assoziationen sind erforderlich, damit Objekte miteinander kommunizieren können (hierzu benötigen sie eine Kenntnisbeziehungen von einander).

Programmiertechnisch, wird üblicherweise eine Assoziation mit Hilfe zweier Variablen erzeugt, die Referenzen zwischen den Objekten halten.

- Für die Konnektivitäten **0..1** (keine oder eine Verbindung) und **1** (genau eine Verbindung) kann dabei einfach eine Referenzvariable genutzt werden.
- Für Konnektivitäten **> 1** muss eine Datenstruktur gewählt werden, die mehr als einen Verweis aufnehmen kann. Üblicherweise wird hier eine Liste/Array genutzt.

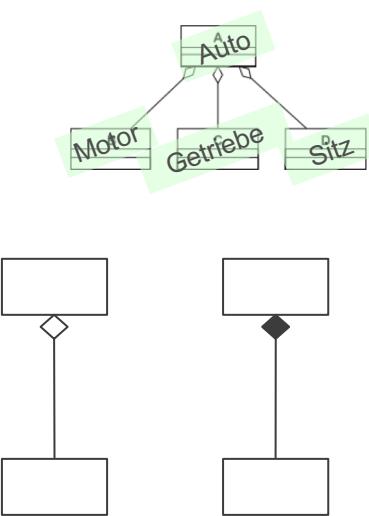
Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme

85

Zerlegungshierarchie



University of Applied Sciences



- Ein Objekt kann als Datenfelder andere Objekte haben
- Z.B. ein Auto besteht aus einem Motor, Getriebe und Sitzen (sowie weiteren Teilen)
- Man kann ein Objekt in seine Teilobjekte und diese wiederum in ihre Teilobjekte zerlegen (usw.).
- Bei dieser Zerlegung unterscheidet man Aggregationen und Kompositionen (kommt gleich)

Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme

86

Zerlegungshierarchie

Aggregation und Komposition (Spezialformen von Assoziationen)

FACH
HOCHSCHULE
LÜBECK
University of Applied Sciences

Aggregation in UML

Komposition in UML

- Bei einer Aggregation können die Bestandteile eines Objekts unabhängig von der Lebensdauer des Oberobjekts existieren
- **Teile können länger leben als das Ganze**
- **Beispiel:** Die Räder eines Autos können an andere Autos gebaut werden. Räder sind an ein Auto aggregiert (zugeordnet).

- Bei einer Komposition existieren die Bestandteile eines Objekts nur so lange wie auch das Oberobjekt existiert.
- **Teile können nicht länger leben als das Ganze**
- **Beispiel:** Die Seiten eines Buchs sind mit dem Buch untrennbar verbunden. Seiten und Buch sind komponiert.

Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme

87

Aggregation/Komposition in UML/JAVA

FACH
HOCHSCHULE
LÜBECK
University of Applied Sciences

Aggregation in UML

Aggregation in JAVA

```
class Auto {
    Lenkrad hat_ein;
    ...
}

class Lenkrad {
    ...
}
```

```
Auto auto = new Auto();
Lenkrad lenkrad = new Lenkrad();
auto.hat_ein = lenkrad;
```

Programmiertechnisch, wird üblicherweise eine Aggregation/Komposition mit Hilfe einer Variablen erzeugt, die eine Referenz auf das Teilobjekt enthält. Da JAVA nur Referenztypen kennt, geht dies in JAVA sehr einfach (siehe oben). Solch eine Variable wird auch **Referenzvariable** (ergänzend zu Instanz- und Klassenvariable genannt).

Kompositionen werden in der Regel genauso umgesetzt, aber beim Löschen wird auch das Komposit mitgelöscht.

Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme

88

Multiplizitäten



University of Applied Sciences

Multiplizität	Beschreibung
1	Genau eine Verbindung
0..1	Höchstens eine Verbindung
0..*	Beliebig viele Verbindungen
1..*	Mindestens eine Verbindung
n..m	Mindestens n höchstens m Verbindungen. Eher ungewöhnlich, nur zu nutzen wenn die Obergrenze zweifelsfrei feststeht, z.B. die Anzahl an Reifen an einem PKW hätte die Multiplizität 0..4. Häufig nutzt man in solchen Fällen dennoch die Multiplizität 0..*.

Assoziationen erhalten neben einem Namen auch Anzahlangaben (Multiplizitätsangaben). Dies gibt an mit wievielen Objekten der gegenüberliegenden Assoziationsseite je ein Objekt der Ausgangsseite verbunden ist.

Letztlich entscheiden diese Angaben, ob zum Verwalten der Kenntnisbeziehungen zwischen Objekten eine einfache Referenzvariable oder eine Collection über den Typ des Assoziationspartners genutzt werden muss.

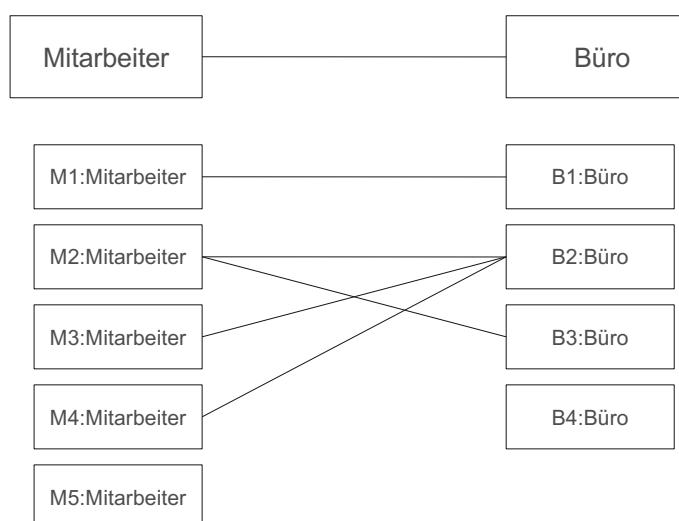
Prof. Dr. rer. nat. Nane Kratzke
 Praktische Informatik und betriebliche Informationssysteme

89

Multiplizitäten Beispiel



University of Applied Sciences



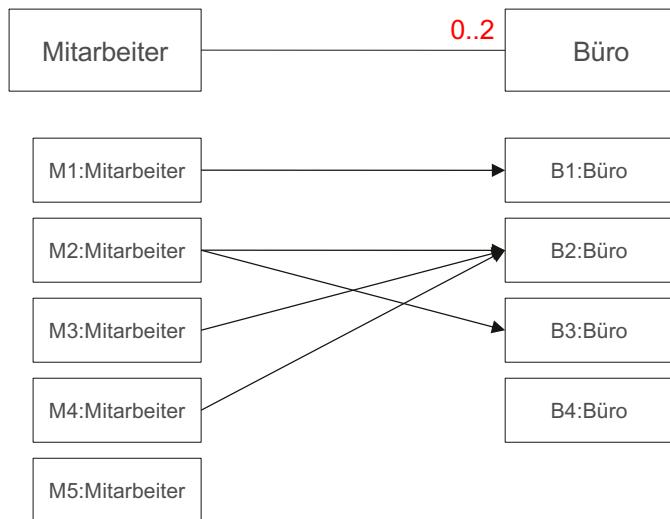
Prof. Dr. rer. nat. Nane Kratzke
 Praktische Informatik und betriebliche Informationssysteme

90

Multiplizitäten Beispiel Sicht der Mitarbeiter



University of Applied Sciences



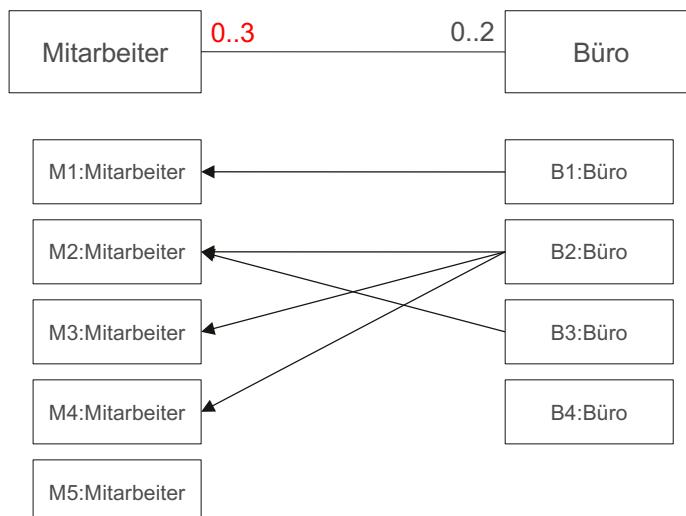
Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme

91

Multiplizitäten Beispiel Sicht der Büros



University of Applied Sciences



Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme

92

Multiplizitäten Beispiel

Angabe der Multiplizitäten

FACH HOCHSCHULE LÜBECK
 University of Applied Sciences

Konkrete Multiplizitäten > 1 werden üblicherweise verallgemeinert.

Prof. Dr. rer. nat. Nane Kratzke
 Praktische Informatik und betriebliche Informationssysteme 93

Transformationsregeln von Assoziationen

FACH HOCHSCHULE LÜBECK
 University of Applied Sciences

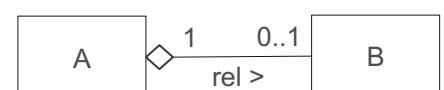
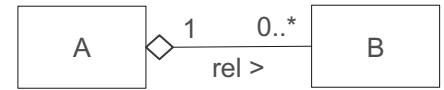
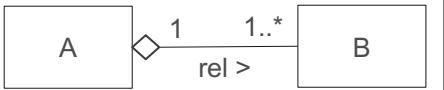
<pre>class A { B b; ... }</pre>	<pre>class B { A a; ... }</pre>
<pre>class A { List b; ... }</pre>	<pre>class B { List<A> a; ... }</pre>

Prof. Dr. rer. nat. Nane Kratzke
 Praktische Informatik und betriebliche Informationssysteme 94

Transformationsregeln von Aggregationen/Kompositionen

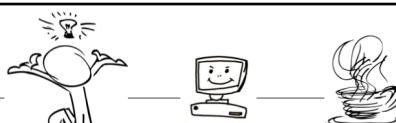


 University of Applied Sciences

 <pre>class A { B rel; ... }</pre>	 <pre>class A { B rel; ... }</pre>
 <pre>class A { List rel; ... }</pre>	 <pre>class A { List rel; ... }</pre>

Prof. Dr. rer. nat. Nane Kratzke
 Praktische Informatik und betriebliche Informationssysteme 95

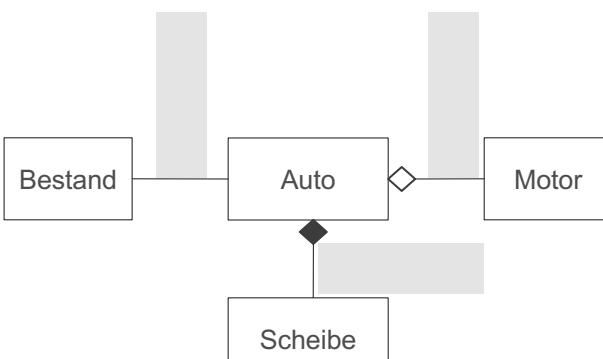
Miniübung:





 University of Applied Sciences

Gegeben ist folgendes UML Diagramm. Welche Arten von Kenntnisbeziehungen sind zwischen den Klassen definiert worden?

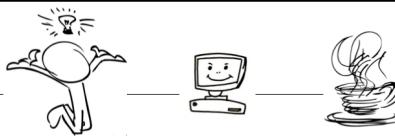


```

graph LR
    Bestand --- Auto
    Motor --- Auto
    Scheibe --- Auto
    
```

Prof. Dr. rer. nat. Nane Kratzke
 Praktische Informatik und betriebliche Informationssysteme 96

Miniübung:



Studierende sollen wie folgt angelegt und ausgegeben werden können.

```
Student s = new Student("Max", "Mustermann", 123456);  
System.out.println(s);
```

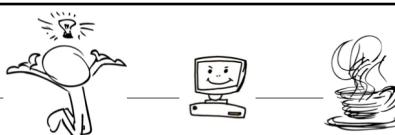
```
Max Mustermann (MatrNr.: 123456)
```

Termine sollen wie folgt angelegt und ausgegeben werden können.

```
Termin t = new Termin(16, 15, 17, 45, "Übung VProg", "18-1.18");  
System.out.println(t);
```

```
16:15h bis 17:45h : Übung VProg in 18-1.18
```

Miniübung:



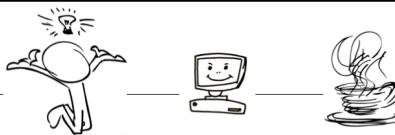
Studierenden können ferner Termine wie folgt zugeordnet werden.

```
Student s = new Student("Max", "Mustermann", 123456);  
Termin t1 = new Termin(14, 30, 16, 00, "Vorlesung VProg", "18-0.01");  
Termin t2 = new Termin(16, 15, 17, 45, "Übung VProg", "18-1.18");  
s.insertTermin(t1);  
s.insertTermin(t2);  
s.insertTermin(t1); // Termin versehentlich doppelt eingegeben.
```

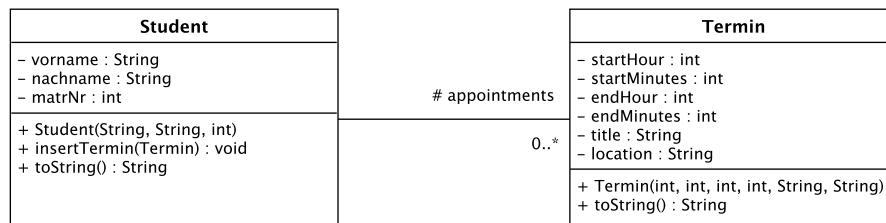
Werden nun Studierende ausgegeben, sollen auch die Termine mit ausgegeben werden, die einem Studierenden zugeordnet sind.

```
System.out.println(s);  
  
Max Mustermann (MatrNr.: 123456)  
- 14:30h bis 16:00h : Vorlesung VProg in 18-0.01  
- 16:15h bis 17:45h : Übung VProg in 18-1.18
```

Miniübung:

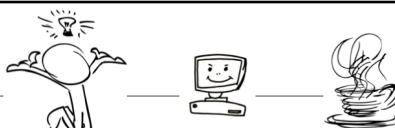


Um sie zu unterstützen, ist Ihnen folgendes UML-Diagramm gegeben.



Implementieren Sie nun bitte **Student** und **Termin**.

Miniübung:



Miniübung:





FACH
HOCHSCHULE
LÜBECK
University of Applied Sciences

Prof. Dr. rer. nat. Nane Kratzke
 Praktische Informatik und betriebliche Informationssysteme

101

Miniübung:





FACH
HOCHSCHULE
LÜBECK
University of Applied Sciences

Gegeben sei folgendes UML Diagramm:

- Implementieren Sie die gezeigten Klassen inklusive ihrer Assoziationen.
- Bestimmen sie die TOP 10 der besten Kunden eines jeden Shops.
- marketing() schreibt 5% aller insgesamt vorhandenen Kunden an und schlägt 25% aller Produkte vor.
- Kunden reagieren auf 10% aller vorgeschlagenen Produkte (Methode mail()) mit einem Kauf.

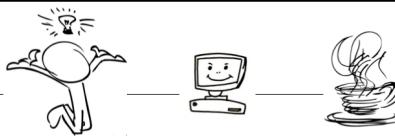
```

classDiagram
    class Shop {
        -String name
        +Shop(String)
        +registerCustomer(): boolean
        +marketing(): void
        +toString(): String
        +topCustomers(): List<Customer>
    }
    class Customer {
        -name : String
        -street : String
        -town : String
        -postCode : String
        +ALL : List<Customer> { static }
        +Customer(String, String, String, String, String)
        +buyList<Product>, Shop): boolean
        +mailList<Product>, Shop): void
        +getOrders(): List<Order>
        +getValue(): double
        +getPostCode(): String
        +toString(): String
    }
    class Order {
        #customer | 1
        #orders | 1
        0..* "products"
        +Order(Customer, List<Product>, Shop)
        +getProducts(): List<Product>
        +getShop(): Shop
        +getCustomer(): Customer
        +getValue(): double
    }
    class Product {
        -name : String
        -double : price
        +ALL : List<Product> { static }
        +Product(String, double)
        +getPrice(): double
        +getName(): String
    }
    Shop "0..* --> " Customer : customers
    Customer "*" "0..* --> " Order : "# orders"
    Order "*" "1..* --> " Product : "# products"
    
```

Prof. Dr. rer. nat. Nane Kratzke
 Praktische Informatik und betriebliche Informationssysteme

102

Miniübung:



Testen Sie ihre Implementierung mit folgendem Setting:

```
// Wir erzeugen 100 Kunden
for (int i = 1; i <= 100; i++) {
    new Customer("Max Mustermann " + i, "Beispielstr. " + (i % 27), "Luebeck", 26500 + (i % 43) + "");
}

// Wir erzeugen 1000 Produkte mit zufaelligen Preisen
for (int i = 1; i <= 1000; i++) { new Product("Testprodukt " + i, Math.random() * i); }

// Wir erzeugen 3 Shops
List<Shop> shops = Arrays.asList(new Shop("Amazon"), new Shop("Otto"), new Shop("EBay"));

// 1. Marketing Runde
for (Shop shop : shops) { shop.marketing(); System.out.println(shop); }

// 2. Marketing Runde
for (Shop shop : shops) { shop.marketing(); System.out.println(shop); }

// 3. Marketing Runde
for (Shop shop : shops) { shop.marketing(); System.out.println(shop); }
```

Objekte abstrahieren



Abstraktion zur Bildung von Hierarchien

- Information Hiding ist ein effizientes Mittel um Komplexität zu beherrschen
- Ein weiteres Mittel ist die Bildung von **Hierarchien**
- Die Objektorientierung kennt im Kern zwei Hierarchieformen:

Vererbungshierarchie

- Kind of-Hierarchie
- Is a-Hierarchie
- Anordnung von Klassen in Kategorieebenen(-bäumen)

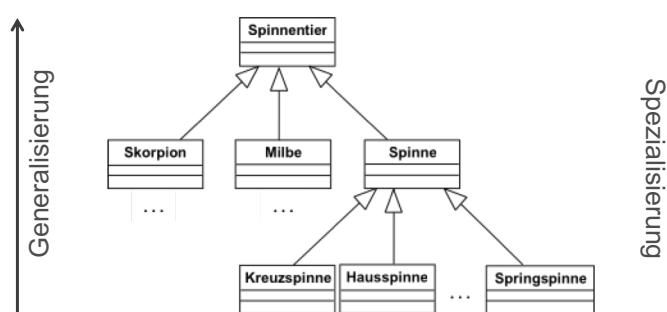
Zerlegungshierarchie

- Part of-Hierarchie
- Betrachtung von zusammengesetzten Objekten in Form von Aggregationen
- Kompositionen

Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme

105

Vererbungshierarchien (I)



Darstellung von Vererbungshierarchien in UML:

Pfeil bedeutet bspw. Skorpion ist Unterklasse von Spinnentier

Kann auch so gelesen werden: Skorpion (spezieller) ist ein Spinnentier (genereller), daher auch der Name „is a-Hierarchie“

Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme

106

Vererbungshierarchien (II)

FACH
HOCHSCHULE
LÜBECK
University of Applied Sciences

```

class Spinnentier {
    ...
}

class Milbe extends Spinnentier {
    ...
}

```

Darstellung von Vererbungshierarchien in UML

Ausdrücken einer Vererbung in JAVA (nur der markierte Ausschnitt)

Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme
107

Klassen sind Datentypen für Referenzen

FACH
HOCHSCHULE
LÜBECK
University of Applied Sciences

Ist beispielsweise folgendes UML Diagramm gegeben, so ergibt sich daraus, das Studenten und Dozenten Personen sind. SkriptStudenten, EifrigerStudenten und LazyStudenten sind Studenten und damit ebenfalls Personen.

Ein EifrigerStudent kann damit generell als Person, spezifischer als Student oder auch sehr spezifisch als EifrigerStudent angesprochen (referenziert) werden.

```

Person
  ^
  |
  +--> Student
  +--> Dozent
  |
  +--> SkriptStudent
  +--> EifrigerStudent
  +--> LazyStudent

```

```
EifrigerStudent s = new EifrigerStudent("Max");
```

```
Student t      = new EifrigerStudent("Moritz");
```

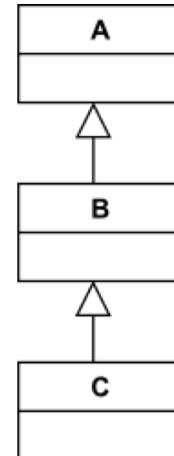
```
Person p      = new EifrigerStudent("Tessa");
```

Referenztyp Objekttyp

Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme
108

Besonderheiten bei der Vererbung

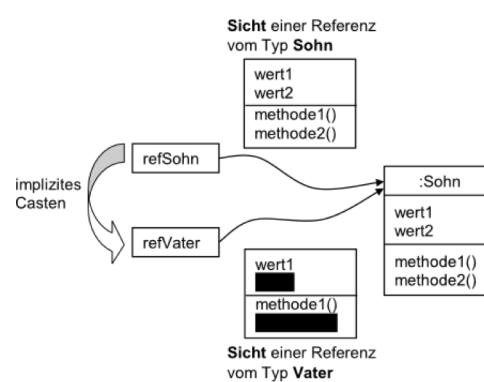
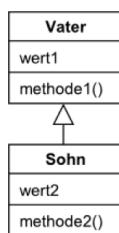
- Für den Einsatz der Vererbung muss man Kenntnisse über die Typkonvertierungen haben
- Wichtig: Ein Sohnobjekt ist immer vom Typ der eigenen Klasse, als auch vom Typ der Vaterklasse, der Vatervaterklasse, etc.
- Somit kann ein Objekt durchaus mehrere Typen haben.



Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme

109

Implizites „Upcasten“



```

Sohn s = new Sohn();
Vater v = s;
  
```

Die Referenz vom Typ Sohn sieht das gesamte Objekt, die vom Typ Vater sieht nur die Vateranteile

s.wert1
s.wert2
s.methode1()
s.methode2()

v.wert1
v.wert2
v.methode1()
v.methode2()

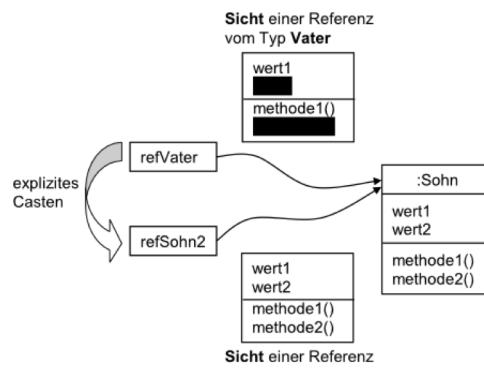
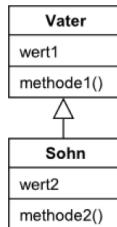
Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme

110

Explizites „Downcasten“



University of Applied Sciences



```

Sohn s = new Sohn();
Vater v = s;
Sohn s2 = (Sohn)v;
  
```

Eine explizite Typkonvertierung (cast) von Referenzen muss immer dann erfolgen, wenn bei einer Zuweisung eine Referenzvariable vom Typ Vater auf ein Objekt der Klasse Sohn zeigt und einer Referenzvariablen vom Typ Sohn zugewiesen wird.

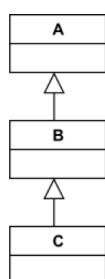
Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme

111

Casting im Überblick Zulässige implizite und explizite Type Casts

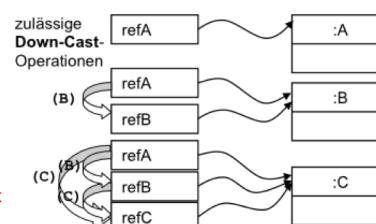


University of Applied Sciences



Wenn oben stehende Klassenhierarchie gilt, dann sind die neben stehenden Cast Operationen zulässig

Funktioniert eine explizite Cast Operation zur Laufzeit nicht, wird eine Exception vom Typ ClassCastException geworfen. Implizite Casts können bereits zur Kompilierzeit geprüft werden.



Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme

112

Miniübung:

FACH HOCHSCHULE LÜBECK
University of Applied Sciences

	B b = new C();	Ja, impliziter Upcast
	A a = b;	Ja, impliziter Upcast
	Object o = b;	Ja, impliziter Upcast
	B b2 = new B(); C c = (C)b2;	Nein, expliziter Downcast aber b2 vom Typ B nicht C
	C c = (C)b;	Ja, expliziter Downcast und b vom Typ C
	D d = (D)b;	Nein, expliziter Cast aber b vom Typ C nicht D

Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme 113

Abstrakte Klassen

FACH HOCHSCHULE LÜBECK
University of Applied Sciences

SkriptStudent + SkriptStudent(String) + notiere(String) : void	EifrigerStudent + EifrigerStudent(String) + notiere(String) : void	LazyStudent + LazyStudent(String) + notiere(String) : void	TiltedStudent - units : int + TiltedStudent(String) + notiere(String) : void	EmotionalerStudent - String[] empfindungen + EmotionalerStudent(String) + notiere(String) : void
-----------------------------------------------------------------------------	---------------------------------------------------------------------------------	-------------------------------------------------------------------------	----------------------------------------------------------------------------------------------	------------------------------------------------------------------------------------------------------------------

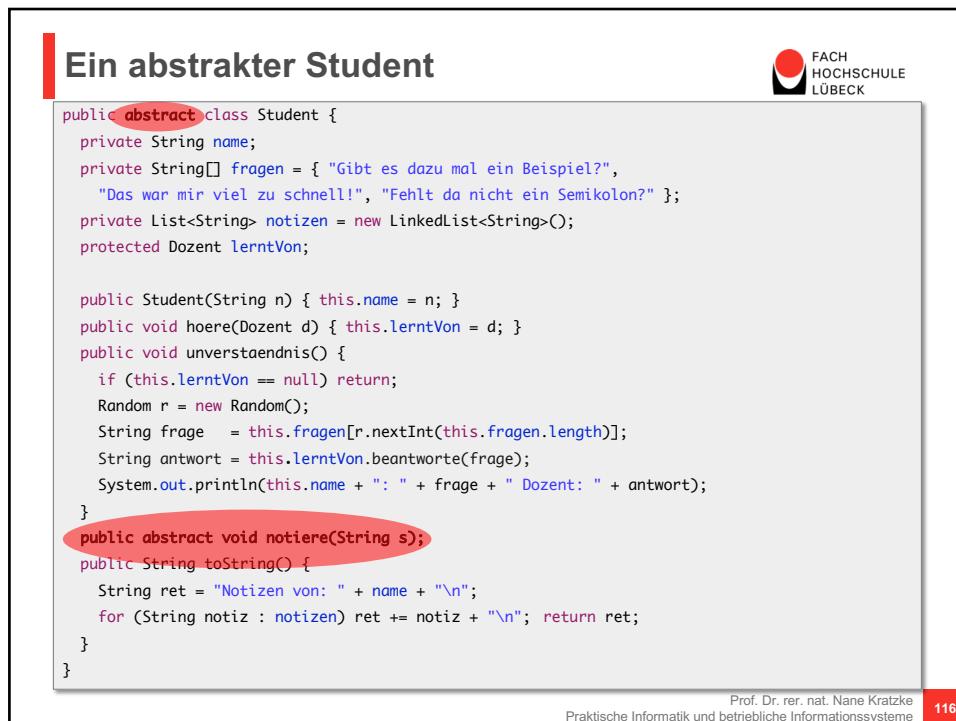
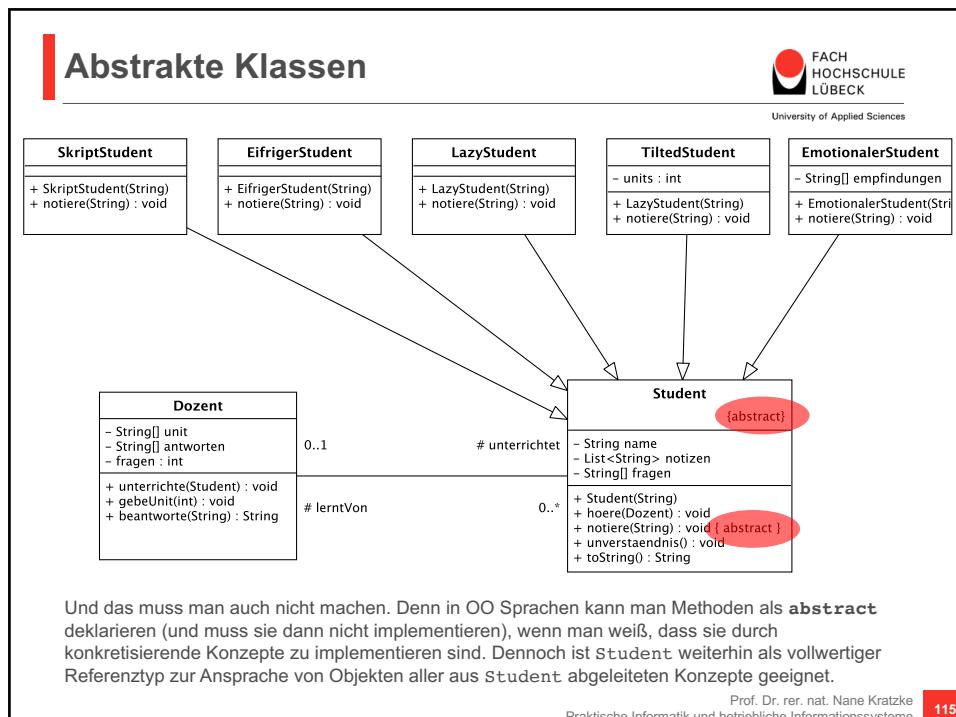
Dozent
- String[] unit
- String[] antworten
- Fragen : int
+ unterrichte(Student) : void
+ gebeUnit(int) : void
+ beantworte(String) : String

Student
- String name
- List<String> notizen
- String[] Fragen
+ Student(String)
+ hoere(Dozent) : void
+ notiere(String) : void
+ unverstaendnis() : void
+ toString() : String

In unserem Polymorphie Beispiel haben diverse Spezialisierungen des generellen Konzepts Student jeweils das notiere() Verhalten (Methode) neu implementiert. Die ursprüngliche notiere() Implementierung wird gar nicht mehr genutzt.

Es stellt sich daher die Frage, wieso diese dann überhaupt implementieren?

Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme 114



Abstrakte Klassen



University of Applied Sciences

- In Basisklassen kann nur die Schnittstelle (Signatur/Methodenrumpf) einer Methode festgelegt werden, aber nicht die Implementierung
- Solche Methoden nennt man abstrakte Methoden
- Eine Klasse mit mindestens einer abstrakten Methode nennt man abstrakte Klasse
- Abstrakte Klassen und Methoden sind mit dem Schlüsselwort **abstract** zu versehen
- Von abstrakten Klassen können keine Objekte instantiiert werden
- Abstrakte Methoden werden üblicherweise dazu genutzt, um Logik zwar vorzusehen, ansprechbar zu machen, aber noch nicht implementieren zu müssen.
- Sie stellen eine Art Pluginmöglichkeit für nachträglich zu ergänzenden Code dar (bspw. für Extension Points).

Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme

117

Finale Methoden und finale Klassen

Das Gegenstück zu abstract



University of Applied Sciences

- Finale Methoden können in einer Subklasse nicht überschrieben werden
- Finale Klassen sind Klassen, von denen man zwar Objekte instantiiert kann, aber keine weiteren Klassen ableiten kann
- Hierzu nutzt man in JAVA das Schlüsselwort final

Deklaration finaler Methoden

```
class C {  
    public void aenderbareMethode() { ... }  
    public final void finaleMethode() { ... }  
}
```

Deklaration finaler Klassen

```
final class C {  
    ...  
}
```

Meist sind es konzeptionelle Gründe des Designs um finale Methoden und Klassen zu nutzen, häufig Sicherheitsgründe um z.B. zu verhindern das Trojanische Pferde von Hackern eingeschleust werden können (ein abgeleitetes Objekt kann überall dort stehen, wo auch ein (vertrauenswürdiges) Vaterobjekt stehen kann).

Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme

118

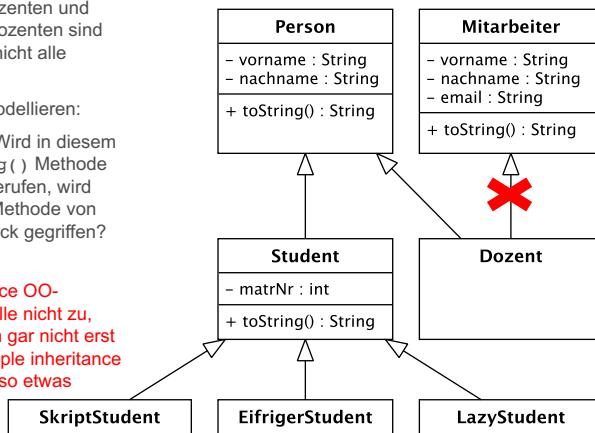
Schnittstellen

Nun zu diesem Problem: Dozenten und Studenten sind Personen. Dozenten sind aber auch Mitarbeiter. Aber nicht alle Studenten sind Mitarbeiter.

Man könnte dies wie folgt modellieren:

Es bleibt aber ein Problem. Wird in diesem Beispiel bspw. die `toString()` Methode eines Dozentenobjekts aufgerufen, wird dann auf die `toString()` Methode von Person oder Mitarbeiter zurückgegriffen?

Java ist eine single inheritance OO-Sprache und lässt solche Fälle nicht zu, um oben stehendes Problem gar nicht erst entstehen zu lassen (in multiple inheritance Sprachen, bspw. C++, kann so etwas jedoch auftreten).



Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme

119

Schnittstellen

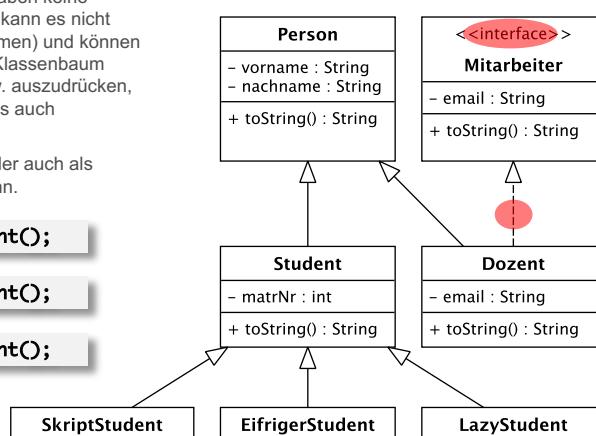
Bei solchen Problemen bietet sich der Einsatz von Schnittstellen an.

Schnittstellen sind sozusagen voll abstrakte Klassen (d.h. abstrakte Klassen haben keine implementierten Methoden, damit kann es nicht zum multiple inheritance Fall kommen) und können daher an beliebiger Stelle einen Klassenbaum „hinzugemischt“ werden, um bspw. auszudrücken, dass ein Dozent sowohl Person als auch Mitarbeiter ist.

Also als Person, als Mitarbeiter oder auch als Dozent angesprochen werden kann.

```

Mitarbeiter m = new Dozent();
Person p     = new Dozent();
Dozent d   = new Dozent();
  
```



Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme

120

Ein Dozent ist Person und Mitarbeiter

```

public interface Mitarbeiter {
    private String email;
    public String toString();
}

public class Dozent extends Person implements Mitarbeiter {
    private String email;
    public String toString() {
        return super.toString() + " Email: " + this.email;
    }
}

class Person {
    - vorname : String
    - nachname : String
    + toString() : String
}

class Mitarbeiter {
    <<interface>>
    - email : String
    + toString() : String
}

class Dozent {
    - email : String
    + toString() : String
}

class LazyStudent {
}

```

Hinweis: Eine Klasse kann beliebige viele Schnittstellen implementieren (**implements**) aber nur eine Klasse erweitern (**extends**).

Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme

121

Neu in Java 8: Default Implementierungen in Schnittstellen

```

public interface Person {
    public String getName();
    default String sayHello() {
        return "Hi, my name is " + getName();
    }
}

public class Dozent implements Person {
    private String name = "Max Mustermann";
    public String getName() {
        return this.name;
    }
}

Dozent d = new Dozent();
System.out.println(d.sayHello());

```

Seit Java 8 können nun default Implementierungen in Schnittstellen vorgesehen werden. Werden diese nicht überschrieben, erben die eine Schnittstelle implementierenden Klassen diese. Default Methoden können allerdings nicht direkt auf Datenfelder eines Objekts zugreifen, sondern nur mittels der Schnittstelle bekannte Methoden. So werden viele Probleme der Mehrfachvererbung umgangen.

Default Methoden sind immer automatisch public.

Warnung: Default Methoden ermöglichen Mehrfachvererbung, und ziehen damit alle Probleme der Mehrfachvererbung mit sich.

Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme

122

Das war viel Theorie ...

Jetzt noch ein kleines Beispiel zu Vererbung



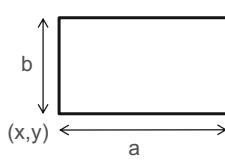
Quelle: Pixabay

Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme

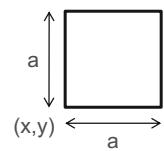
123

Veranschaulichung an einem Beispiel Fächenberechnung von Figuren

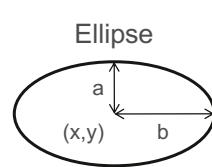
Rechteck



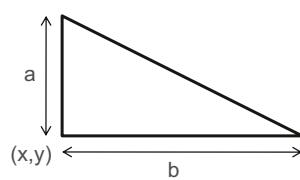
Quadrat



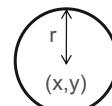
Ellipse



Rechtwinkliges Dreieck



Kreis



Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme

124

Flächenberechnung von Figuren

Was haben alle Figuren gemeinsam?

FACH HOCHSCHULE
 LÜBECK
University of Applied Sciences

Rechteck 	Quadrat 	Ellipse
Rechtwinkliges Dreieck 	Kreis 	Einen Bezugspunkt Einen Bezugspunkt

Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme 125

Flächenberechnung von Figuren

Was haben viele Figuren gemeinsam?

FACH HOCHSCHULE
 LÜBECK
University of Applied Sciences

Rechteck 	Quadrat 	Ellipse
Rechtwinkliges Dreieck 	Kreis 	Zwei Längenangaben Zwei Längenangaben

Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme 126

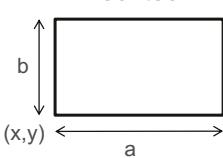
Flächenberechnung von Figuren

Welche Figuren sind Spezialfälle anderer Figuren



FACH
HOCHSCHULE
LÜBECK
University of Applied Sciences

Rechteck

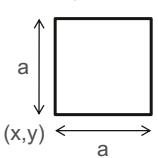


b

(x,y)

a

Quadrat

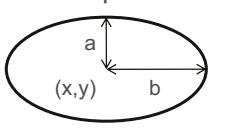


a

(x,y)

a

Ellipse

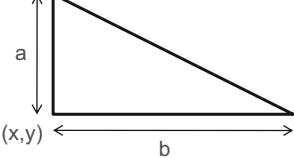


a

(x,y)

b

Rechtwinkliges Dreieck

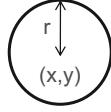


a

(x,y)

b

Kreis



r

(x,y)

Ein Quadrat ist ein spezielles Rechteck

Ein Kreis ist eine spezielle Ellipse

Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme

127

Flächenberechnung von Figuren



FACH
HOCHSCHULE
LÜBECK
University of Applied Sciences

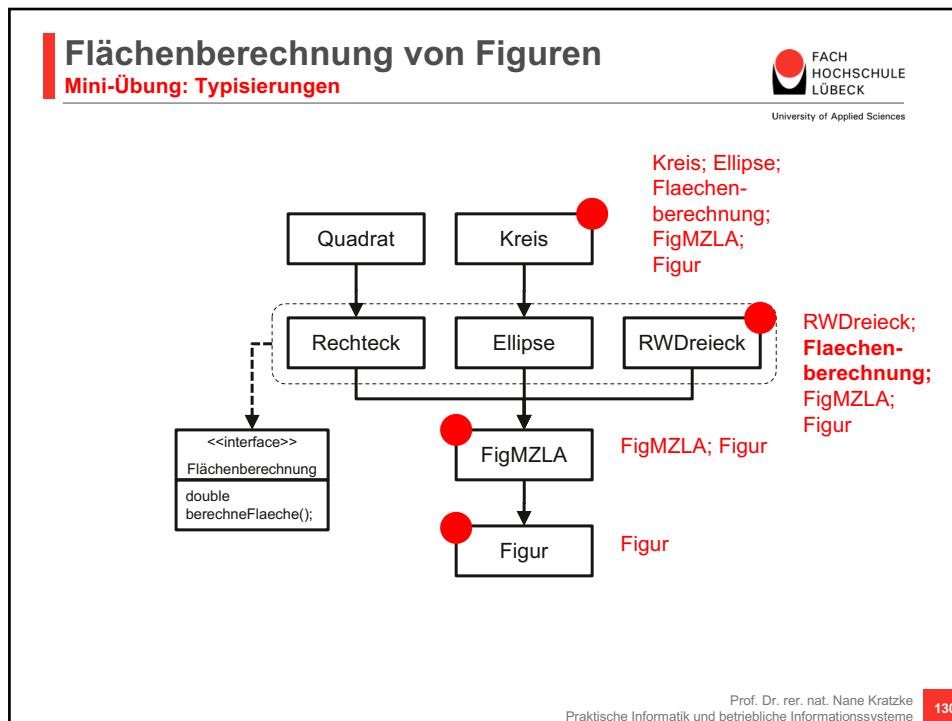
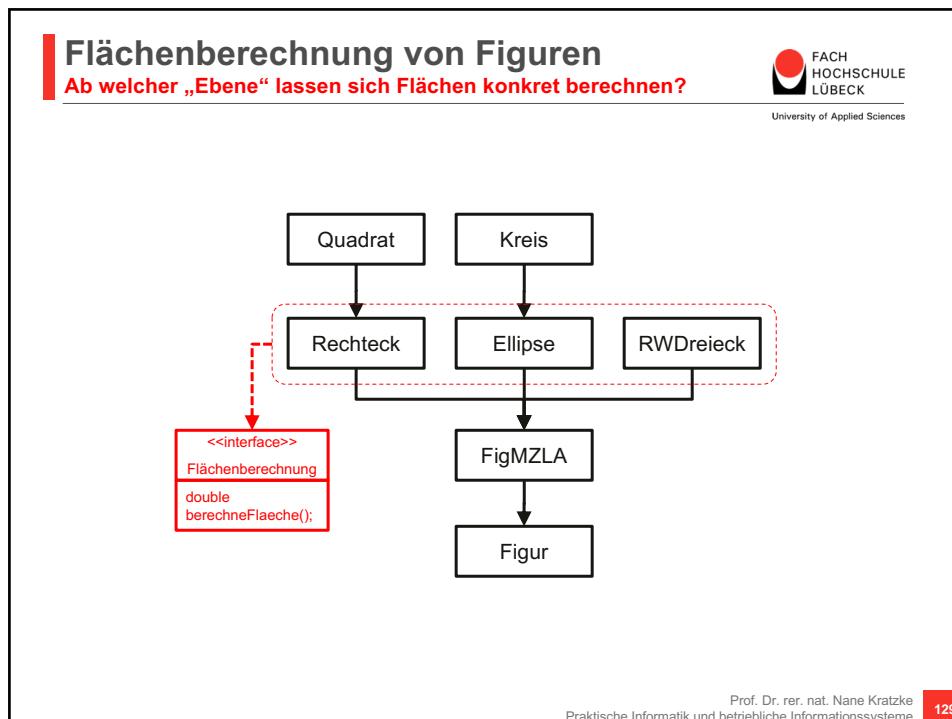
- Es gibt Figuren.
- Figuren mit zwei Längenangaben sind Figuren.
- Rechteck, Ellipse und rechtwinkliges Dreieck sind Figuren mit zwei Längenangaben.
- Ein Quadrat ist ein Rechteck.
- Ein Kreis ist eine Ellipse.

```

graph TD
    Quad[Quadrat] --> Rec[Rechteck]
    Kreis[Kreis] --> Ellipse[Ellipse]
    Rec --- Ellipse
    Rec --- RWD[Rechtwinkliges Dreieck]
    Ellipse --> FigMZLA[FigMZLA]
    RWD --> FigMZLA
    FigMZLA --> Fig[Figur]
  
```

Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme

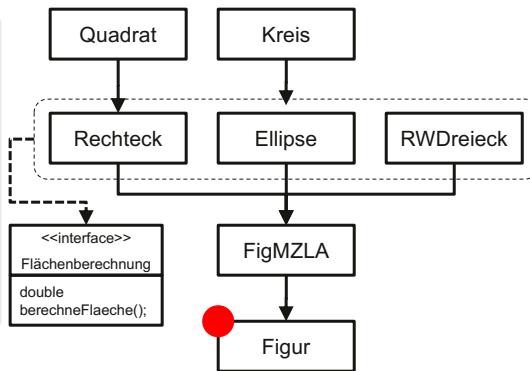
128



Flächenberechnung von Figuren

Einführung von Figuren

```
public class Figur {
    protected int x = 0;
    protected int y = 0;
    public Figur(int x, int y)
    {
        this.x = x;
        this.y = y;
    }
}
```

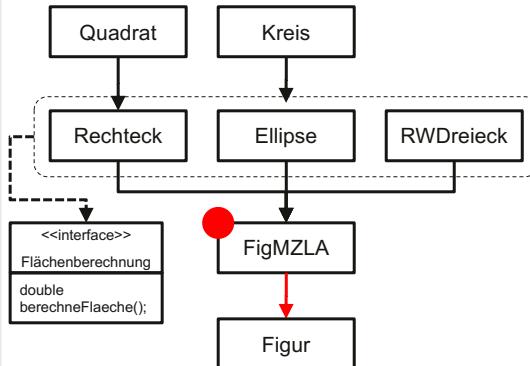


Flächenberechnung von Figuren

Einführung von Figuren mit zwei Längenangaben

```
public class FigMZLA
    extends Figur
{
    protected int A = 0;
    protected int B = 0;

    public FigMZLA(int x,
                   int y,
                   int a,
                   int b)
    {
        super(x,y);
        this.A = a;
        this.B = b;
    }
}
```



Flächenberechnung von Figuren

Einführung einer Flächenberechnungsschnittstelle

```

classDiagram
    class Flaechenberechnung {
        double berechneFlaeche();
    }
    class Quadrat
    class Kreis
    class Rechteck
    class Ellipse
    class RWDreieck
    class FigMZLA {
        <<interface>>
        Flaechenberechnung
        double berechneFlaeche();
    }
    class Figur

    Quadrat --> Rechteck
    Kreis --> Ellipse
    Kreis --> RWDreieck
    Rechteck --> FigMZLA
    Ellipse --> FigMZLA
    RWDreieck --> FigMZLA
    FigMZLA --> Figur
  
```

The diagram illustrates a software architecture for calculating the area of figures. It features a UML class diagram with the following components:

- Flächenberechnung Interface:** A class with a single method `double berechneFlaeche();`.
- Concrete Classes:** `Quadrat`, `Kreis`, `Rechteck`, `Ellipse`, and `RWDreieck`.
- FigMZLA Class:** An interface represented by a dashed box containing the code for the `Flächenberechnung` interface.
- Figur Class:** A final class that receives calculations from the `FigMZLA` interface.
- Relationships:** `Quadrat` and `Kreis` inherit from their respective base classes (`Rechteck` and `Ellipse`). All three base classes (`Rechteck`, `Ellipse`, and `RWDreieck`) implement the `FigMZLA` interface, which in turn implements the `Figur` class.

Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme

133

Flächenberechnung von Figuren

Implementierungen von Rechteck

```

classDiagram
    class Rechteck {
        <<class>>
        int x, y, a, b;
        double berechneFlaeche();
    }
    class Quadrat
    class Kreis
    class FigMZLA {
        <<interface>>
        Flaechenberechnung
        double berechneFlaeche();
    }
    class Figur

    Quadrat --> Rechteck
    Kreis --> Ellipse
    Kreis --> RWDreieck
    Rechteck --> FigMZLA
    Ellipse --> FigMZLA
    RWDreieck --> FigMZLA
    FigMZLA --> Figur
  
```

This slide focuses on the implementation of the `Rechteck` class. It includes:

- Diagram of a Rectangle:** A rectangle with vertices labeled `(x,y)` at the bottom-left and side lengths `a` and `b`.
- Implementation Code:** A code snippet for the `Rechteck` class that extends `FigMZLA` and implements the `Flächenberechnung` interface.
- Calculation Formula:** The formula $a * b$ is shown for calculating the area of a rectangle.
- Relationships:** Similar to the first slide, `Quadrat` and `Kreis` inherit from their respective base classes (`Rechteck` and `Ellipse`). All three base classes (`Rechteck`, `Ellipse`, and `RWDreieck`) implement the `FigMZLA` interface, which in turn implements the `Figur` class.

Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme

134

Flächenberechnung von Figuren

Implementierung von Ellipse

FACH HOCHSCHULE LÜBECK
 University of Applied Sciences

Ellipse

Berechnung der Fläche einer Ellipse mit Radien a und b?

$$\pi * a * b$$

```

public class Ellipse
    extends FigMZLA
    implements Flaechenberechnung {
    public Ellipse(int x,
                  int y,
                  int a,
                  int b) {
        super(x, y, a, b);
    }

    public double
    berechneFlaeche() {
        return Math.abs(
            Math.PI * A * B);
    }
}
    
```

Quadrat Kreis

Rechteck Ellipse RWDreieck

<<interface>>
 Flaechenberechnung
 double berechneFlaeche();

FigMZLA

Figur

Prof. Dr. rer. nat. Nane Kratzke
 Praktische Informatik und betriebliche Informationssysteme 135

Flächenberechnung von Figuren

Implementierung eines rechtwinkligen Dreiecks

FACH HOCHSCHULE LÜBECK
 University of Applied Sciences

Rechtwinkliges Dreieck

Berechnung eines rechtwinkligen Dreiecks mit den Seitenlängen a und b?

$$a * b / 2$$

```

public class RWDRDreieck
    extends FigMZLA
    implements Flaechenberechnung {
    public RWDRDreieck(int x,
                       int y,
                       int a,
                       int b) {
        super(x, y, a, b);
    }

    public double
    berechneFlaeche() {
        return Math.abs(
            A * B / 2.0);
    }
}
    
```

Quadrat Kreis

Rechteck Ellipse RWDRDreieck

<<interface>>
 Flaechenberechnung
 double berechneFlaeche();

FigMZLA

Figur

Prof. Dr. rer. nat. Nane Kratzke
 Praktische Informatik und betriebliche Informationssysteme 136

Flächenberechnung von Figuren

Implementierung von Quadrat

FACH HOCHSCHULE LÜBECK
University of Applied Sciences

Quadrat

```
public class Quadrat
extends Rechteck
{
    public Quadrat(int x,
                   int y,
                   int a)
    { super(x, y, a, a);
    }
}
```

Berechnung der Fläche eines Quadrats mit der Seitenlänge a?

$$a^2$$

Und wo erfolgt die Flächenberechnung?
In der Klasse Rechteck

Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme 137

Flächenberechnung von Figuren

Implementierung von Kreis

FACH HOCHSCHULE LÜBECK
University of Applied Sciences

Kreis

```
public class Kreis
extends Ellipse
{
    public Kreis(int x,
                int y,
                int r)
    { super(x, y, r, r);
    }
}
```

Berechnung der Fläche eines Kreises mit dem Radius r?

$$\pi r^2$$

Und wo erfolgt die Flächenberechnung?
In der Klasse Ellipse

Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme 138

Erzeugung von Objekten

Verfolgen von Konstruktoraufrufen

```

public class Kreis extends Ellipse
{
    public Kreis(int x, int y, int r) {
        super(x, y, r, r);
    }
}

public class Ellipse extends FigMZLA
{
    public Ellipse(int x, int y, int a, int b) {
        super(x, y, a, b);
    }
}

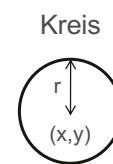
public class FigMZLA extends Figur
{
    public FigMZLA(int x, int y, int a, int b) {
        super(x, y); this.A = a; this.B = b;
    }
}

public class Figur
{
    public Figur(int x, int y) {
        this.X = x; this.Y = y;
    }
}

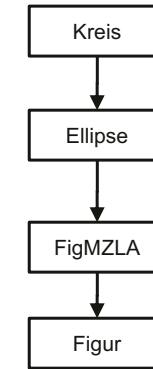
```



University of Applied Sciences



Kreis



Praktische Informatik und betriebliche Informationssysteme

139

Arbeiten mit Objekten

Verfolgen von Methodenaufrufen

```

Kreis k = new Kreis(5, 5, 10);

double flaeche = k.berechneFlaeche();

public class Kreis extends Ellipse
{
    public Kreis(int x, int y, int r) {
        super(x, y, r, r);
    }
}

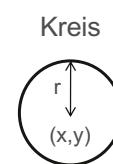
public class Ellipse extends FigMZLA
    implements Flaechenberechnung
{
    public Ellipse(int x, int y, int a, int b) {
        super(x, y, a, b);
    }

    public double berechneFlaeche() {
        return Math.abs(Math.PI * this.A * this.B);
    }
}

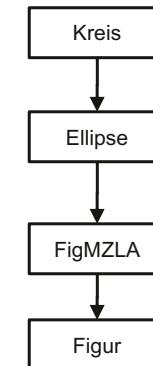
```



University of Applied Sciences



Kreis



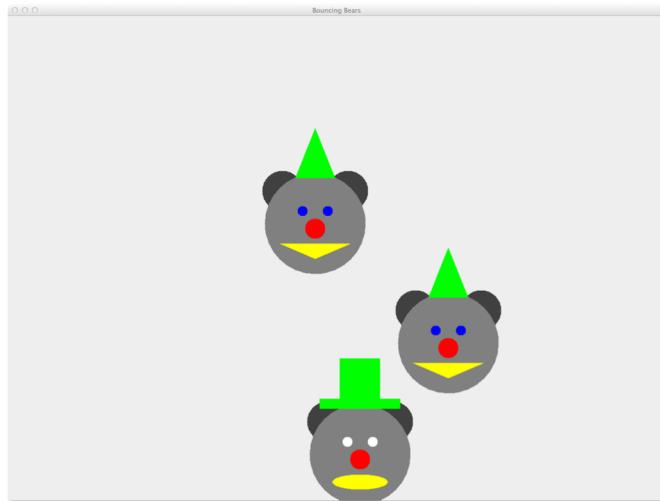
Praktische Informatik und betriebliche Informationssysteme

140

Das alles erweitern wir noch zu ...



University of Applied Sciences



Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme

141

Zusammenfassung



University of Applied Sciences

- Grundsatz der Objektorientierung: Denken in Objekten
 - Klassen sind Baupläne
 - Objekte sind konkrete Ausprägungen dieser Baupläne
 - Objekte kommunizieren miteinander (Methoden) um ein Problem zu lösen
- Objekte haben ein **Verhalten** (Methoden)
- Objekte haben einen (gekapselten) **Zustand** (Datenfelder)
- Objekte können **kommunizieren** (Methodenaufrufe entlang ihrer Assoziationen)
 - Assoziationen
 - Part-of-Hierarchien
- Objekte sind vielgestaltig (**polymorph**)
 - Abstraktion entlang von
 - Vererbungshierarchien (is a-Hierarchien)



Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme

142