Kurs

Datenbankgrundlagen und Modellierung

Sebastian Maneth, Universität Bremen
maneth@uni-bremen.de

Sommersemester 2023

12.6.2023

Vorlesung 7: Modellierung & UML: Einführung

Kurs **Datenbankgrundlagen und Modellierung**

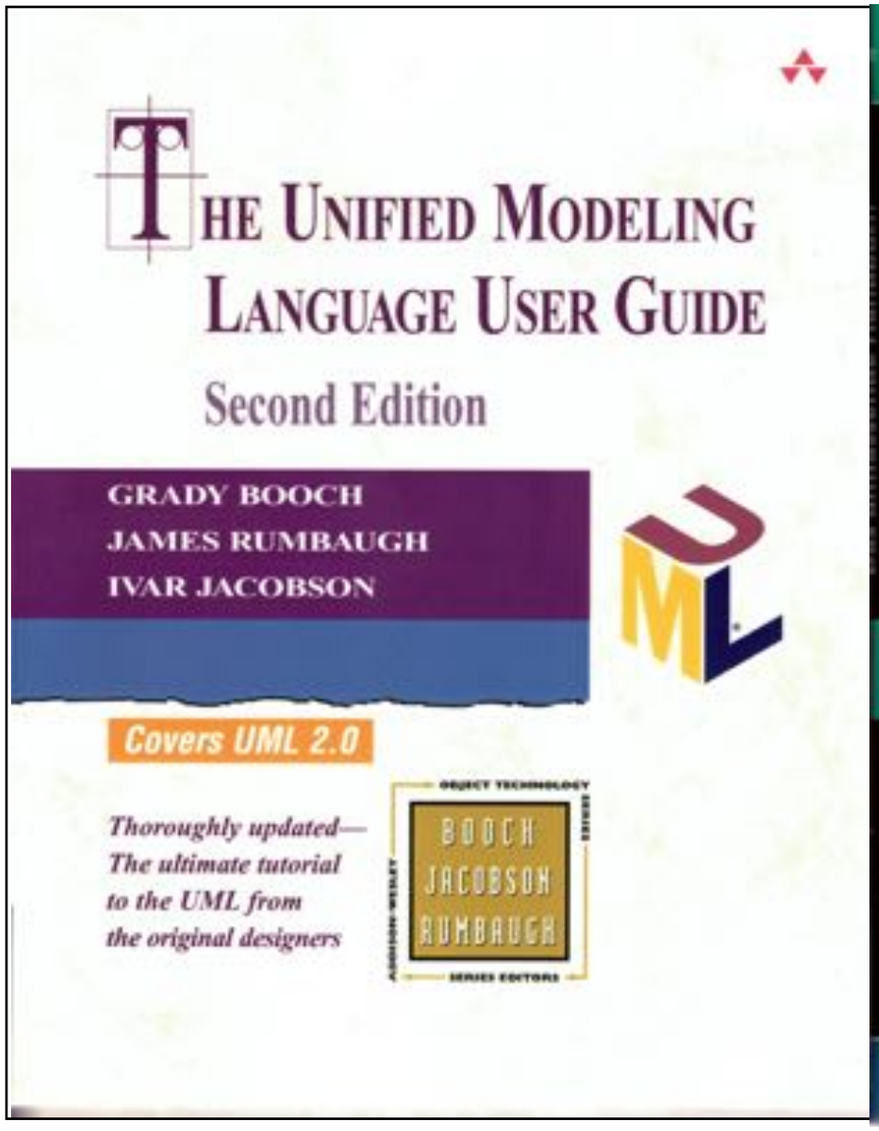
17.4. Vorlesung 1 — Intro
24.4. V2 — ER, SQL
1.5. keine Vorlesung
4.5. Fragestunden
8.5. V3 — SQL
10.5. Ü1
11.5. Fragestunden
15.5. V4 — SQL
17.5. Ü2
22.5. V5 — SQL & funct. dependencies
24.5. Ü3
~~29.5. V6 — funct. dependencies~~
31.5. Ü4
5.6. V6 — normal forms
7.6. Ü5

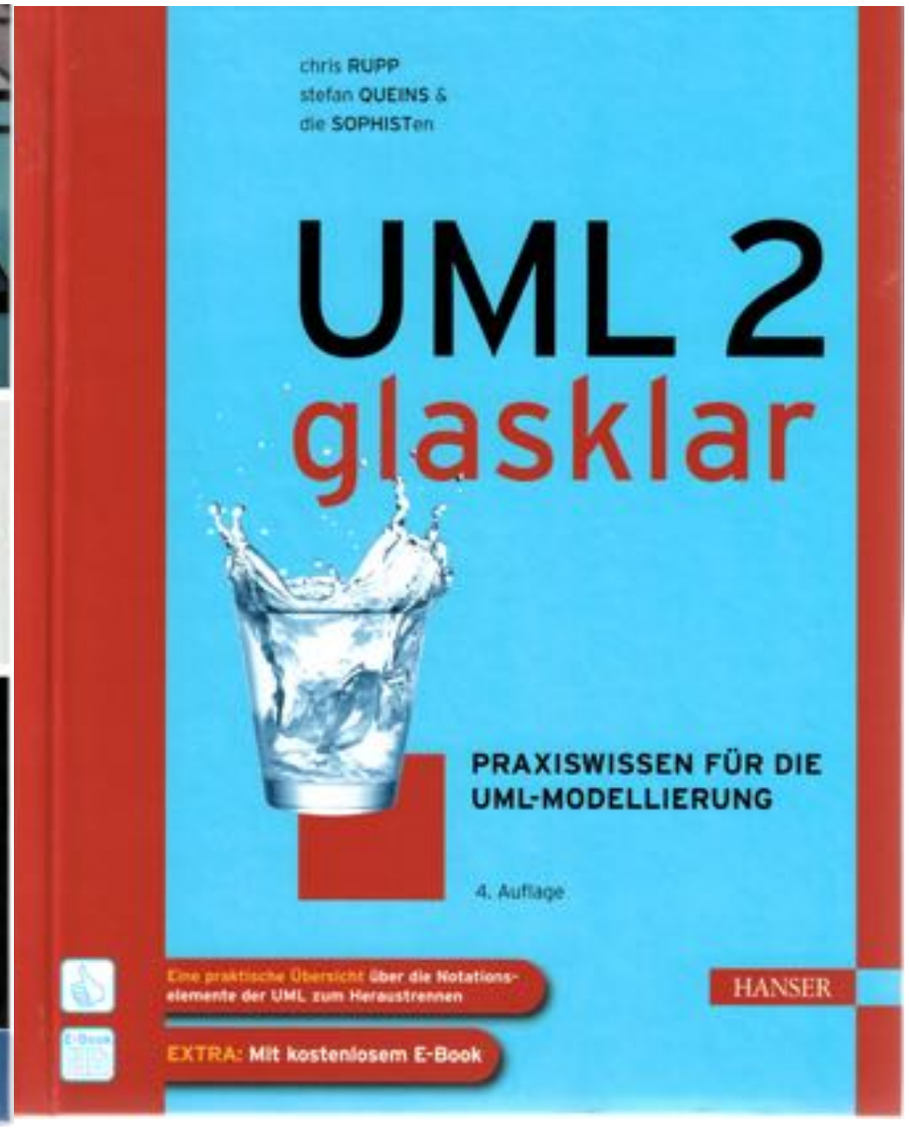
12.6. V7 — modelling Intro
14.6. Ü6
19.6. V8 — class diagrams
21.6. Ü7
26.6. V9 — state charts
28.6. Ü8
3.7. V10 — sequence diagrams
5.7. Ü9
10.7. V11 — Klausurvorbereitung

Literature

The lecture slides should be sufficient to master the contents of this course

Additionally, these textbooks may be helpful:





Contents of this Lecture

You will learn

— how to **model software**, esp. in an object oriented way

— how to (hierarchically) **relate classes**, using class diagrams

— how to **model time relationships** using state charts and sequence diagrams

Why do we model?

Why do we model?

Imagine you want to build a simple dog house.

Do you need a model?

Why do we model?

Imagine you want to build a simple dog house.

Do you need a model?
Probably not. Plan in your head, buy the wood and get going.

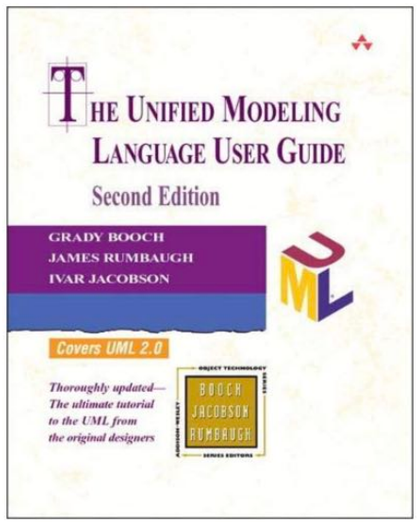
Why do we model?

Imagine you want to build a **skyscraper**.

What is a model?

*A model is a **simplification** of reality.*

From pages 6,7 of:

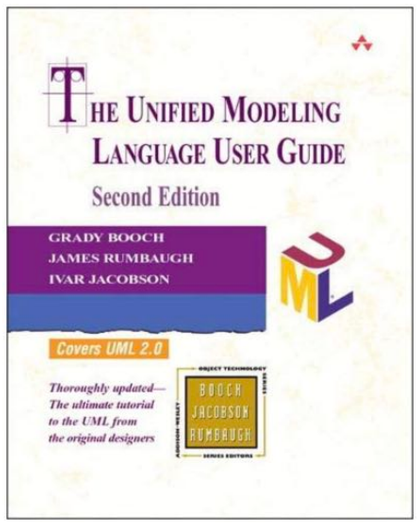


Why do we model?

*A model is a **simplification** of reality.*

*We build models so that we can better **understand** the system we are developing.*

From pages 6,7 of:



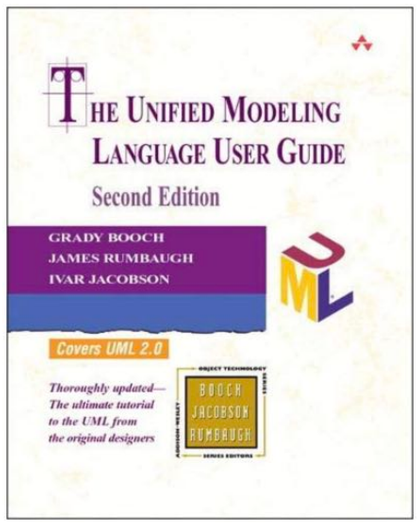
Why do we model?

*A model is a **simplification** of reality.*

*We build models so that we can better **understand** the system we are developing.*

We build models of complex systems because we cannot comprehend such a system in its entirety.

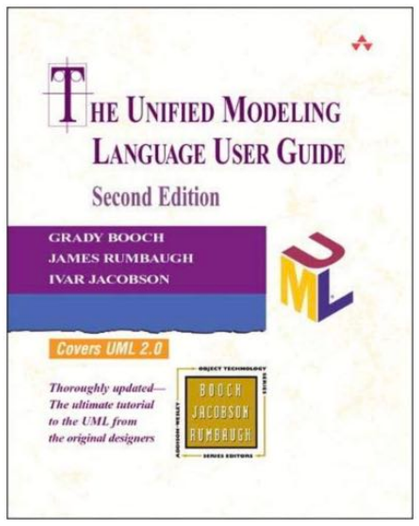
From pages 6,7 of:



About modelling

The choice of what models to create has a profound influence on how a problem is attacked and how a solution is shaped.

From pages 8,9 of:

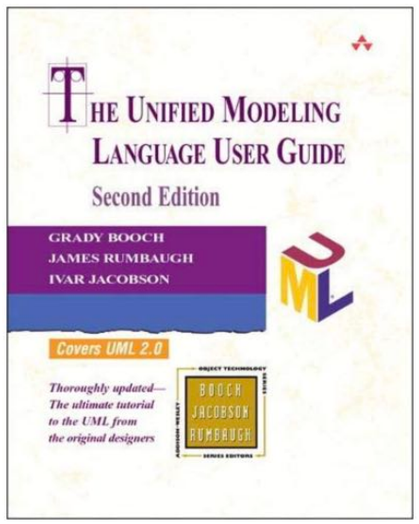


About modelling

The choice of what models to create has a profound influence on how a problem is attacked and how a solution is shaped.

Every model may be expressed at different levels of precision.

From pages 8,9 of:



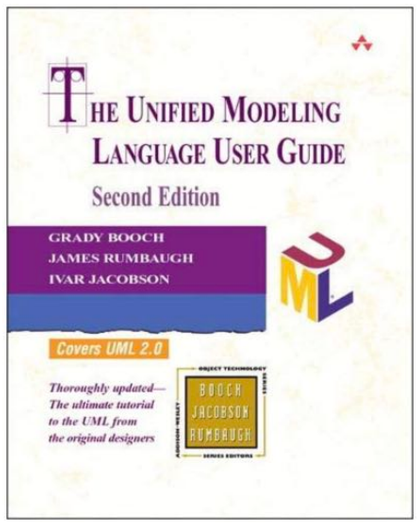
About modelling

The choice of what models to create has a profound influence on how a problem is attacked and how a solution is shaped.

Every model may be expressed at different levels of precision.

The best models are connected to reality.

From pages 8,9 of:



About modelling

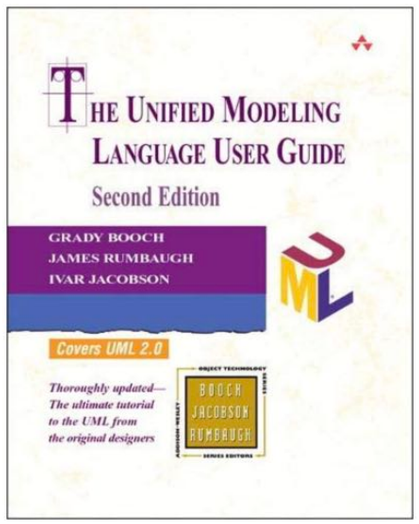
The choice of what models to create has a profound influence on how a problem is attacked and how a solution is shaped.

Every model may be expressed at different levels of precision.

The best models are connected to reality.

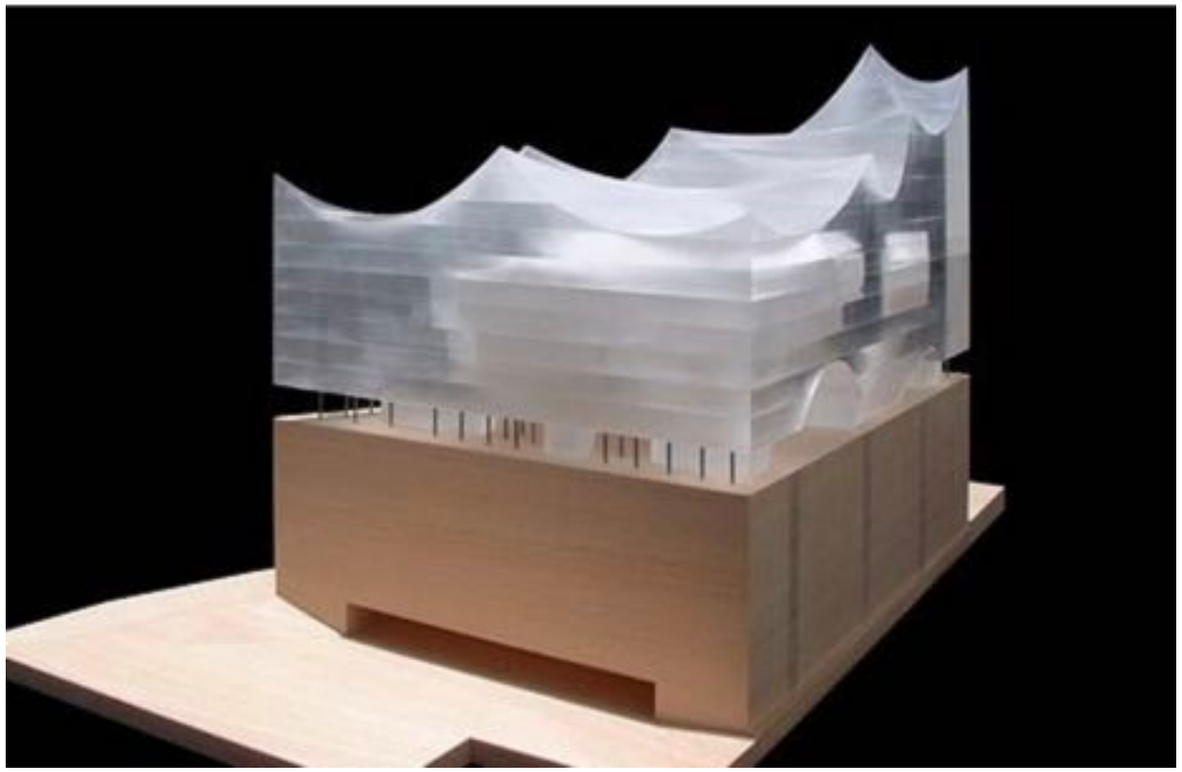
No single model or view is sufficient. Every nontrivial system is best approached via a small set of nearly independent models with multiple viewpoints.

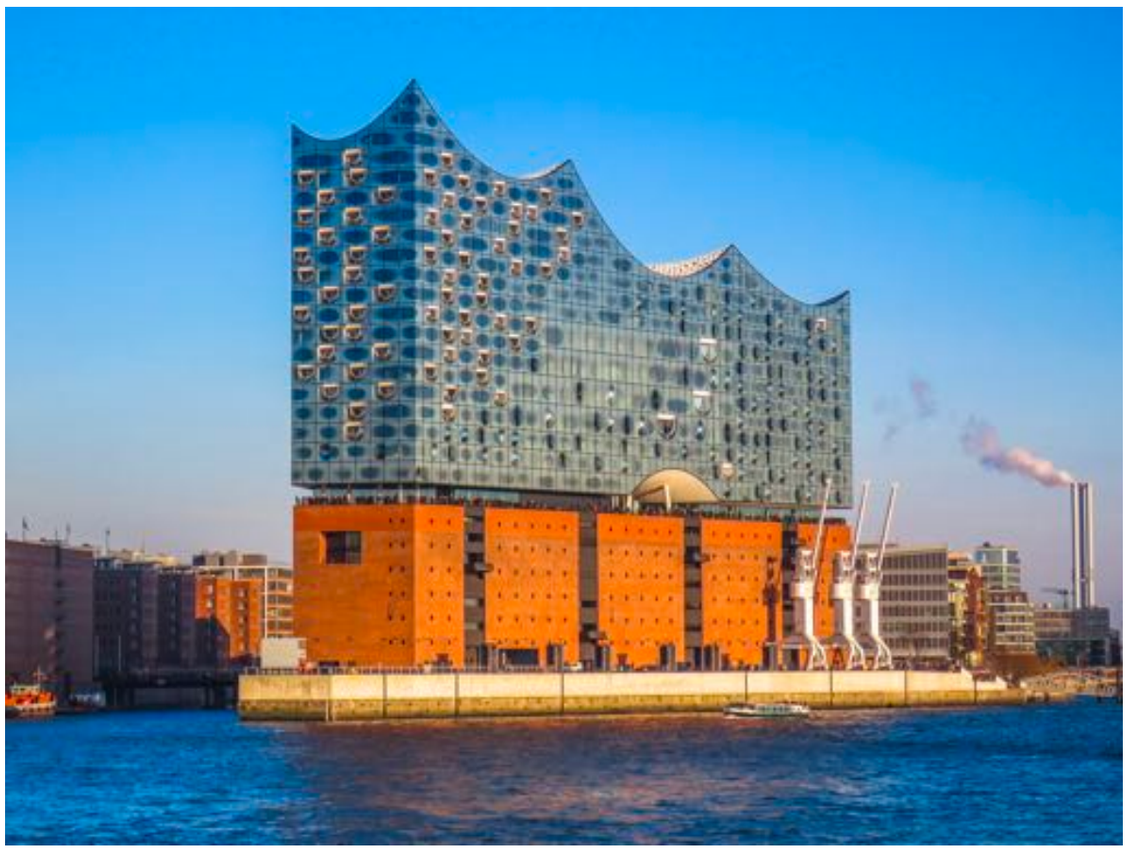
From pages 8,9 of:

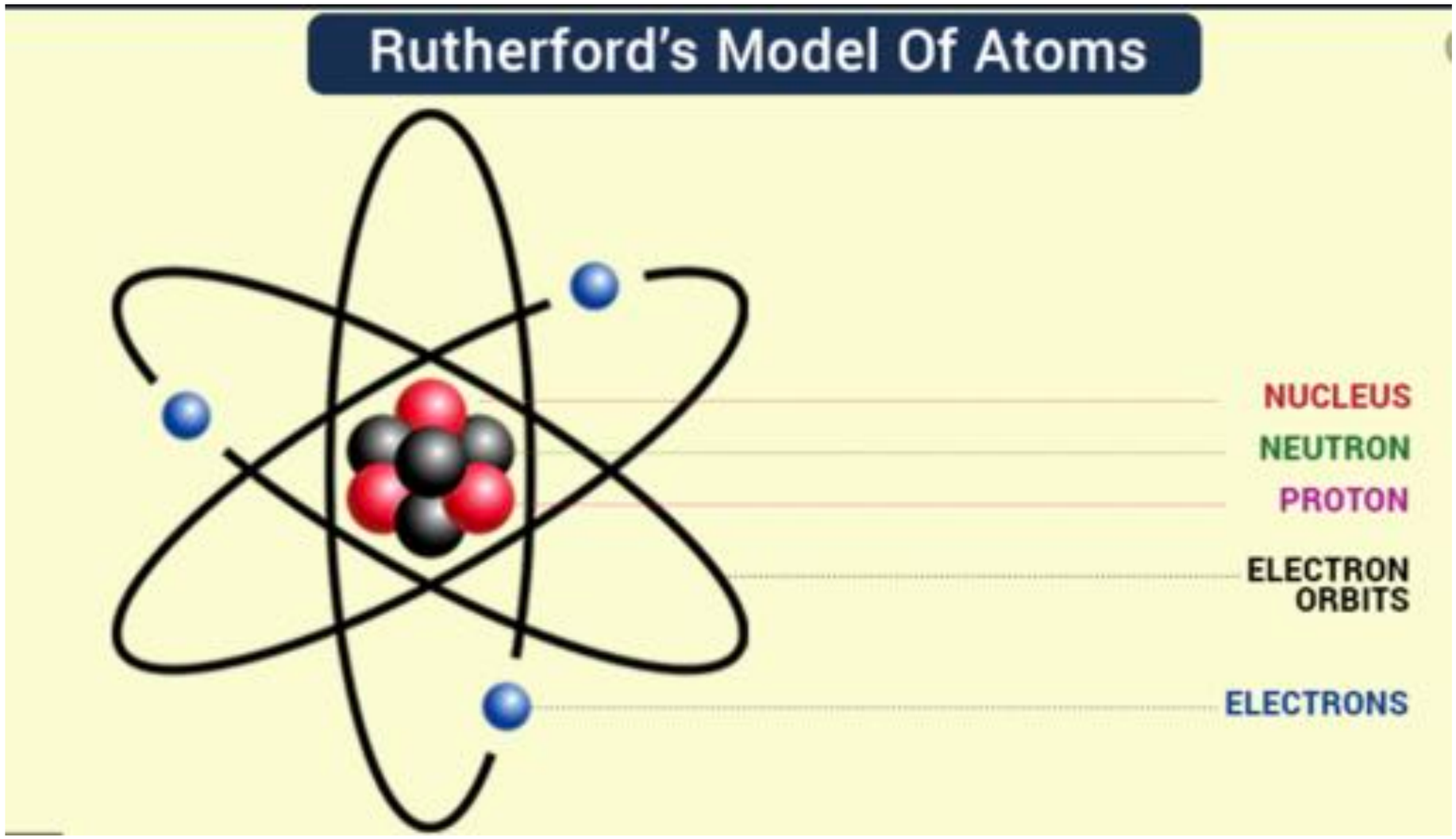


Examples

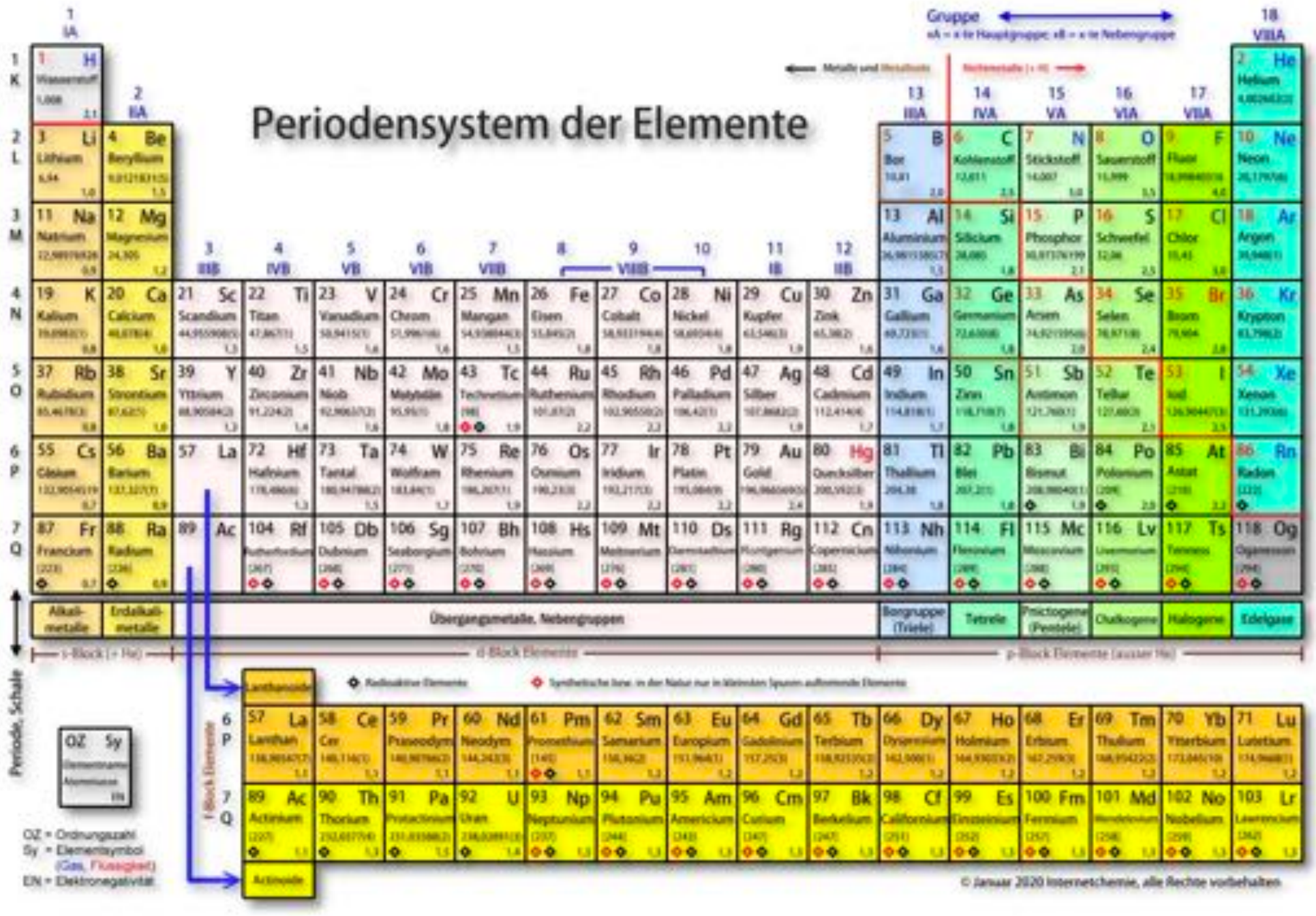
Examples: Architektur



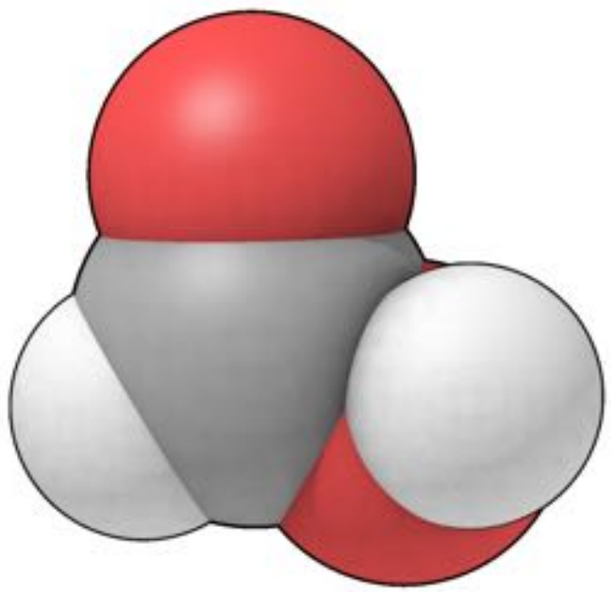



Rutherford's Model Of Atoms
NUCLEUS
NEUTRON
PROTON
ELECTRON ORBITS
ELECTRONS

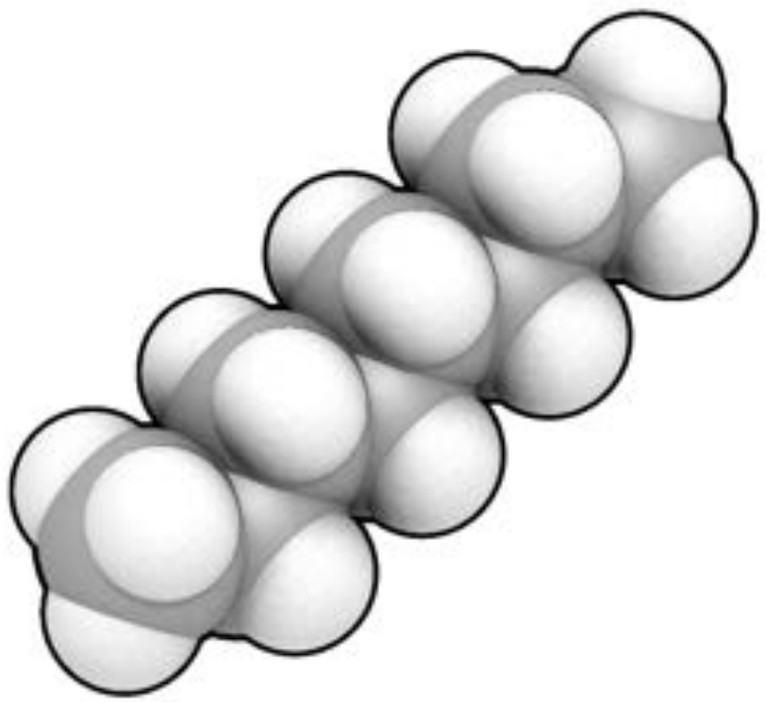
Examples: Atome



Kalottenmodell

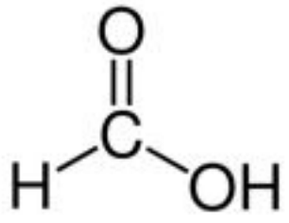


Kalottenmodell der Ameisensäure

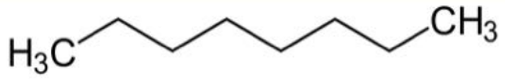


Kalottenmodell des Octans.

Strukturformel



Strukturformel



Examples: Keilschrift

Die **Keilschrift** ist die älteste Schrift der Welt. Sie wurde etwa **3.400 v. Chr.** erfunden und anschließend für die folgenden 3 ½ Jahrtausende genutzt.

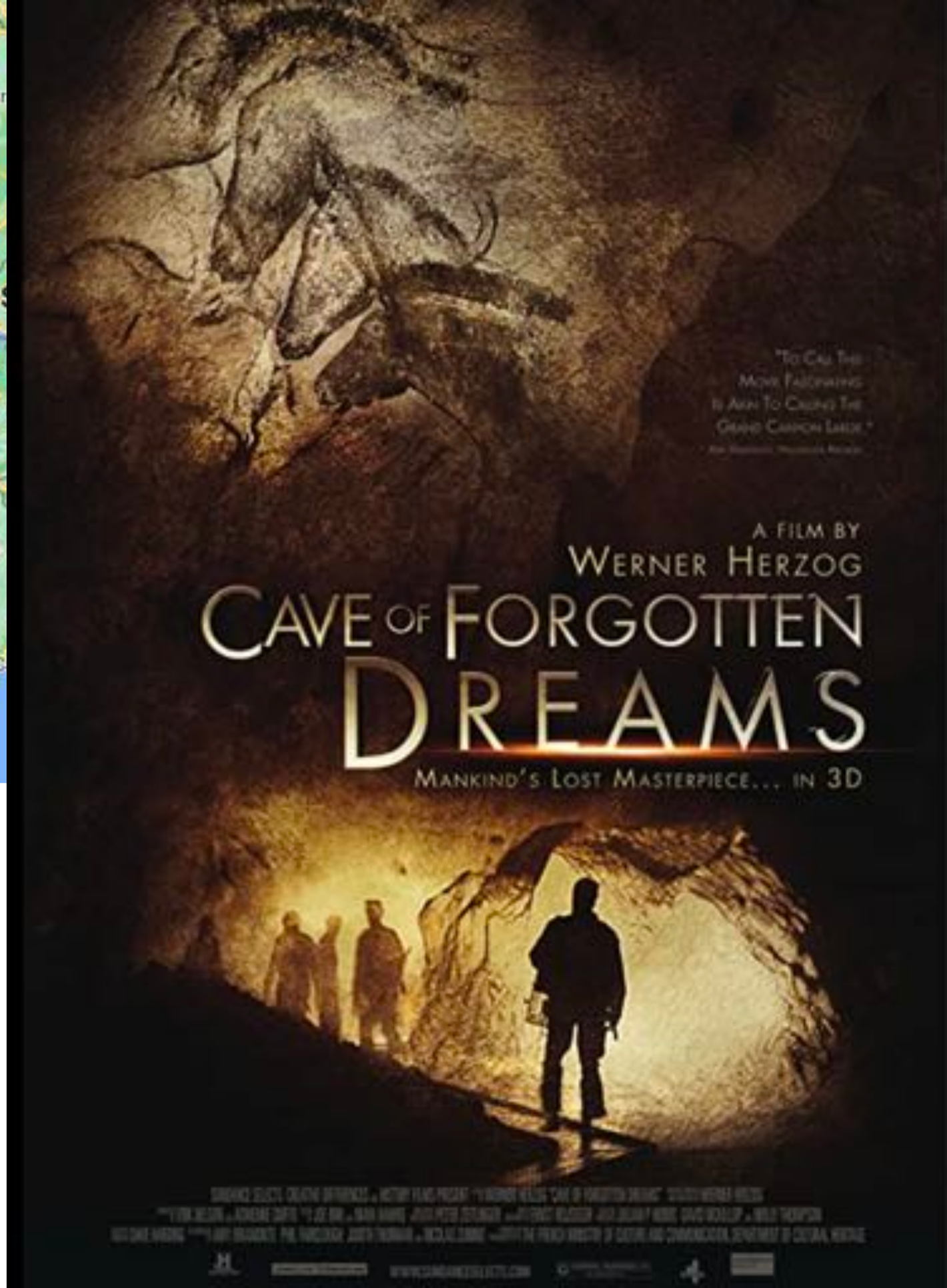
Ihr Ursprungsort liegt im Süden des heutigen Irak, ein Gebiet, das von den Griechen als “Mesopotamien” bezeichnet wurde, was etwa bedeutet “zwischen den Flüssen”.
Die Welt der Keilschrift erstreckte sich jedoch viel weiter in alle vier Himmelsrichtungen.
[…].

From
www.manuscript-cultures.uni-hamburg.de/cuneiform/
(accessed 14.4.2021)

Examples: Kunst

Wall painting with Horses, Chauvet Cave, France ca. **30,000 B.C.**

Excellent documentary
(7,4 on IMDB)



UML — Unified Modeling Language

UML — Unified Modeling Language

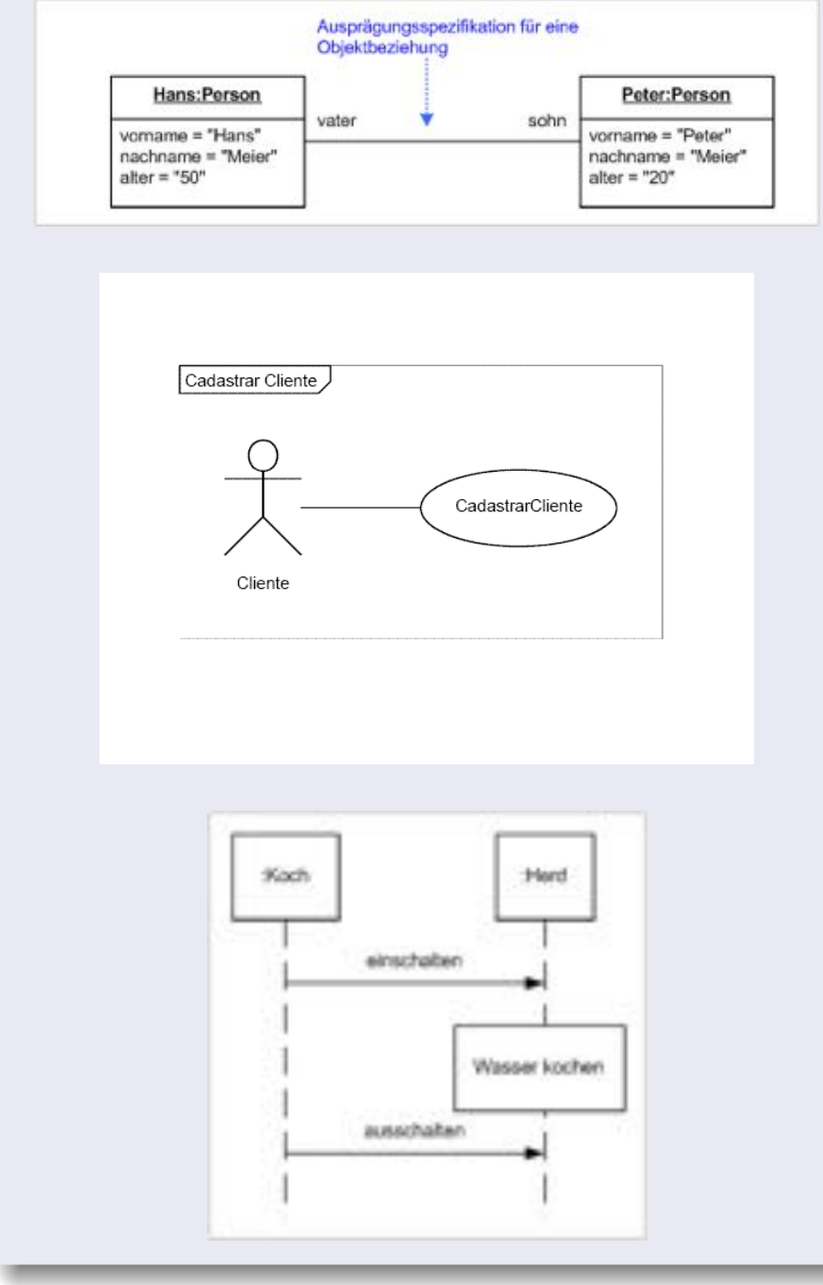
The next slides (in German) are from Norbert Fuhr's 2014/15 lecture on "Modelling"

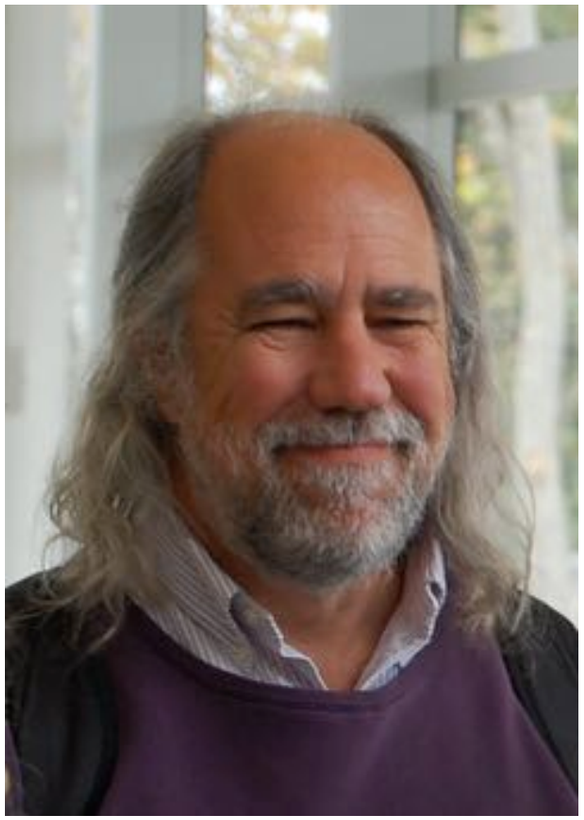
which in turn are based on Slides by Barbara König

UML: Einführung

UML = Unified Modeling Language

- Standard-Modellierungssprache für Software Engineering.
- Basiert auf objekt-orientierten Konzepten.
- Sehr umfangreich, enthält viele verschiedene Typen von Modellen.
- Entwickelt von Grady Booch, James Rumbaugh, Ivar Jacobson (1997).





Grady Booch

James Rumbaugh

Ivar Jacobson

UML und Objekt-Orientierung

Was bedeutet Objekt-Orientierung?

Grundidee

Die reale Welt besteht aus Objekten, die untereinander in Beziehungen stehen. Diese Sichtweise wird auch auf Modellierung und Softwareentwicklung übertragen.

UML und Objekt-Orientierung

Was bedeutet Objekt-Orientierung?

Grundidee

Die reale Welt besteht aus Objekten, die untereinander in Beziehungen stehen. Diese Sichtweise wird auch auf Modellierung und Softwareentwicklung übertragen.

Etwas genauer ...

Daten (= Attribute) werden zusammen mit der Funktionalität (= Methoden) in Objekten organisiert bzw. gekapselt. Jedes Objekt ist in der Lage Nachrichten (= Methodenaufrufe) zu empfangen, Daten zu verarbeiten und Nachrichten zu senden.

Diese Objekte können – einmal erstellt – in verschiedenen Kontexten wiederverwendet werden.

UML und Objekt-Orientierung

Geschichte der Objekt-Orientierung

- Entwicklung von objekt-orientierten Programmiersprachen:
  - 60er Jahre: Simula (zur Beschreibung und Simulation von komplexen Mensch-Maschine-Interaktionen)
  - 80er Jahre: C++
  - 90er Jahre: Java
- Verbreitung von objekt-orientierten Entwurfsmethoden:
  - 70er Jahre: Entity-Relationship-Modell
  - 90er Jahre: Vorläufer von UML:
    - Jacobson → OOSE (Object-Oriented Software Engineering), 1992
    - Rumbaugh → OMT (Object Modeling Technique) 1991
    - Booch Method 1992
  - Seit 1997: UML
  - Seit 2005: UML 2.0

* Latest Version: UML 2.5.1 2017

From https://de.wikipedia.org/wiki/Liste_objektorientierter_Programmiersprachen (accessed 21 March 2021)

ActionScript
ABAP Objects
Ada
Aleph
AppleScript
Beta
BlitzMax
Boo
C++ International Standard ISO/IEC 14882:**2020**
C#
Clarion
Cobol ISO 2002
Codesys
CFML (ColdFusion Markup Language)
Common Lisp (CLOS)
Component Pascal
D
Dylan
Eiffel
Fortran – ab Fortran 2003
FreeBASIC
Gambas
Genie
Go (Einbettung und Interfaces statt Vererbung)
Groovy 2003
IDL
incr Tcl
Io
Java SE 20: **2023**
JavaScript / ECMAScript
Lingo
Lotusscript
Modula-3
Modelica
NewtonScript
Oberon
Objective-C
Objective CAML
Object Pascal (Delphi)
Perl
PHP – ab Version 4
PowerBuilder
Python 2.0 2000
Ruby
R
S
Sather
Scala 2004
Scratch 2007
Snap!/BYOB 2011
Seed7
Self
Simula – die erste Programmiersprache mit Konzepten der Objektorientieru
Smalltalk – die erste konsequent objektorientierte Sprache
SuperCollider
Swift 2014
Vala
Visual Basic Classic (keine Vererbung)
Visual Basic for Applications (VBA; keine Vererbung)
Visual Basic .NET (VB.NET)
Visual Basic Script
Visual Objects
XBase
Xojo
XOTcl
Zonnon

UML und Objekt-Orientierung

Vorteile der objekt-orientierten Programmierung und Modellierung

- Leichte Wiederverwendbarkeit dadurch, dass Daten und Funktionalität zusammen verwaltet werden und es Konzepte zur Modifikation von Code (Stichwort: Vererbung) gibt.
- Nähe zur realen Welt: viele Dinge der realen Welt können als Objekte modelliert werden.
- Verträglichkeit mit Nebenläufigkeit und Parallelität: Kontrollfluss kann nebenläufig in den Objekten ablaufen und die Objekte können durch Nachrichtenaustausch bzw. Methodenaufrufe untereinander kommunizieren.

UML und Objekt-Orientierung

Vorteile der objekt-orientierten Programmierung und Modellierung

- Leichte Wiederverwendbarkeit dadurch, dass Daten und Funktionalität zusammen verwaltet werden und es Konzepte zur Modifikation von Code (Stichwort: Vererbung) gibt.
- Nähe zur realen Welt: viele Dinge der realen Welt können als Objekte modelliert werden.
- Verträglichkeit mit Nebenläufigkeit und Parallelität: Kontrollfluss kann nebenläufig in den Objekten ablaufen und die Objekte können durch Nachrichtenaustausch bzw. Methodenaufrufe untereinander kommunizieren.

==> do you know any **CONS** (disadvantages) of OOP?

UML und Objekt-Orientierung

Konzepte

- Klassen: definiert einen Typ von Objekten mit bestimmten Daten und bestimmter Funktionalität.

  Beispiel: die Klasse der VRR-Fahrkartenautomaten
- Objekte: Instanzen der Klasse.

  Beispiel: der Fahrkartenautomat am Duisburger Hauptbahnhof, Osteingang

UML und Objekt-Orientierung

Schema of a relation name

Konzepte

- Klassen: definiert einen Typ von Objekten mit bestimmten Daten und bestimmter Funktionalität.

  Beispiel: die Klasse der VRR-Fahrkartenautomaten
- Objekte: Instanzen der Klasse.

  Beispiel: der Fahrkartenautomat am Duisburger Hauptbahnhof, Osteingang

value (instance) of a relation name

Vorsicht

— object-orientation is **not** the only valid approach to programming!

— there are other paradigms that at times may be preferable!
(e.g., functional programming)

Vorsicht

— object-orientation is **not** the only valid approach to programming!

— there are other paradigms that at times may be preferable!
(e.g., functional programming)

— OOP is **good** for adding new types (classes)

— OOP is **not good** for adding new operations (across many types)

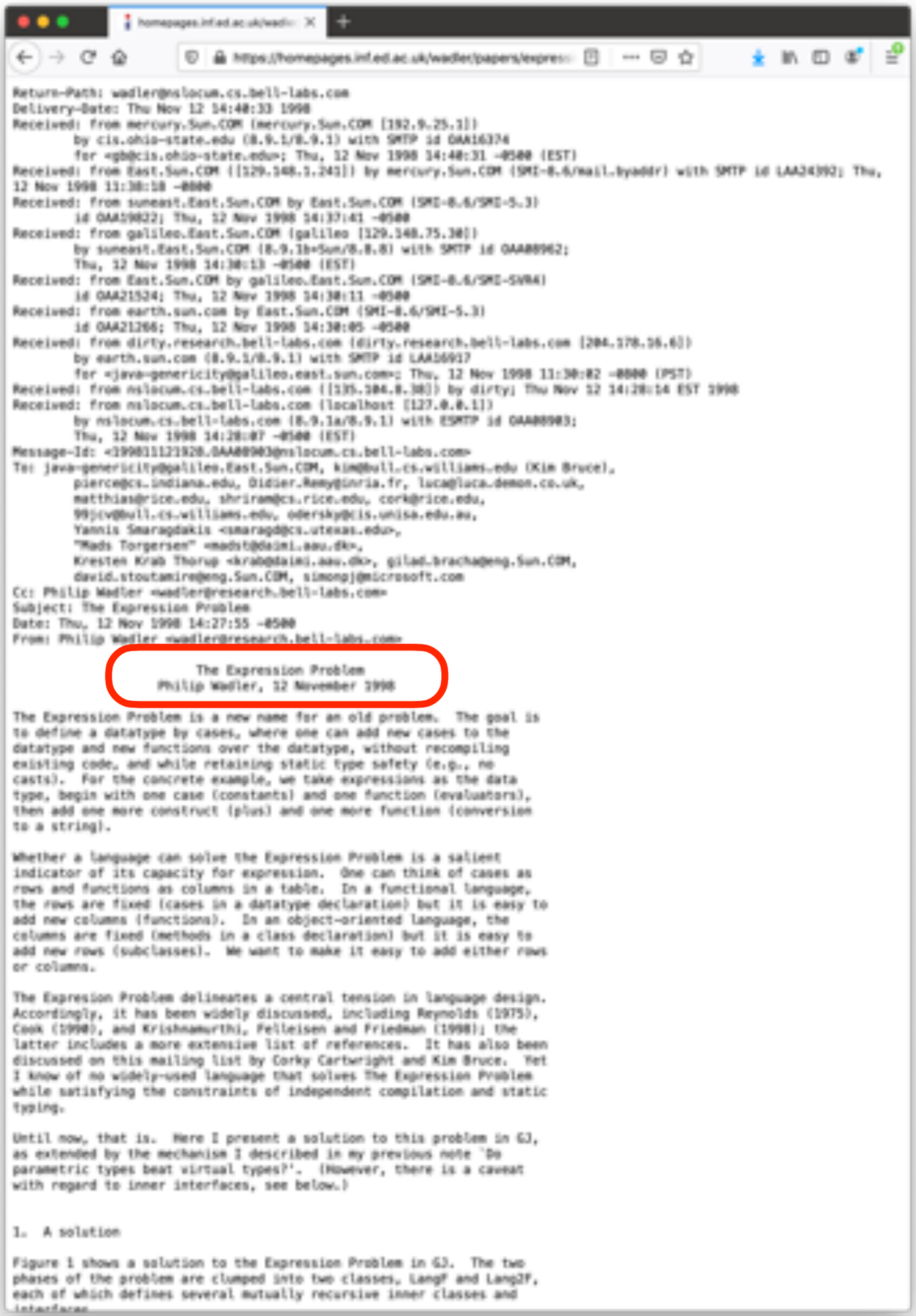
The next slides are by Prof. Clarke (Cornell University)
from his course “CS 3010 — Fall 2015”

Expression Problem

[Wadler 1998]:

- Start with an arithmetic *expression language*
- Add new forms of expressions
- Add new operations on expressions

The expression problem is: how well does your PL support this task?



homepages.inf.ed.ac.uk/wadler

https://homepages.inf.ed.ac.uk/wadler/papers/express

```
Return-Path: wadler@nslocum.cs.bell-labs.com
Delivery-Date: Thu Nov 12 14:40:33 1998
Received: from mercury.Sun.COM (mercury.Sun.COM [192.9.25.1])
        by cis.ohio-state.edu (8.9.1/8.9.1) with SMTP id OAA16374
        for <gb@cis.ohio-state.edu>; Thu, 12 Nov 1998 14:40:31 -0500 (EST)
Received: from East.Sun.COM ([129.148.1.241]) by mercury.Sun.COM (SMI-8.6/mail.byaddr) with SMTP id LAA24392; Thu,
12 Nov 1998 11:38:18 -0800
Received: from sunmail.East.Sun.COM by East.Sun.COM (SMI-8.6/SMI-5.3)
        id OAA19822; Thu, 12 Nov 1998 14:37:41 -0500
Received: from galileo.East.Sun.COM (galileo [129.148.75.30])
        by sunmail.East.Sun.COM (8.9.1b+Sun/8.8.8) with SMTP id OAA08962;
        Thu, 12 Nov 1998 14:38:13 -0500 (EST)
Received: from East.Sun.COM by galileo.East.Sun.COM (SMI-8.6/SMI-SVR4)
        id OAA21524; Thu, 12 Nov 1998 14:38:11 -0500
Received: from earth.sun.com by East.Sun.COM (SMI-8.6/SMI-5.3)
        id OAA21266; Thu, 12 Nov 1998 14:38:05 -0500
Received: from dirty.research.bell-labs.com (dirty.research.bell-labs.com [204.178.16.6])
        by earth.sun.com (8.9.1/8.9.1) with SMTP id LAA16917
        for <java-genericity@galileo.east.sun.com>; Thu, 12 Nov 1998 11:38:02 -0800 (PST)
Received: from nslocum.cs.bell-labs.com ([135.104.8.38]) by dirty; Thu Nov 12 14:28:14 EST 1998
Received: from nslocum.cs.bell-labs.com (localhost [127.0.0.1])
        by nslocum.cs.bell-labs.com (8.9.1a/8.9.1) with ESMTP id OAA08983;
        Thu, 12 Nov 1998 14:28:07 -0500 (EST)
Message-Id: <199811121928.OAA08983@nslocum.cs.bell-labs.com>
To: java-genericity@galileo.East.Sun.COM, kim@bull.cs.williams.edu (Kim Bruce),
        pierce@cs.indiana.edu, Didier.Remy@inria.fr, luca@luca.demon.co.uk,
        matthias@rice.edu, shriram@cs.rice.edu, cork@rice.edu,
        99jcv@bull.cs.williams.edu, odersky@cis.unisa.edu.au,
        Yannis Smaragdakis <smaragd@cs.utexas.edu>,
        "Mads Torgersen" <mads@daimi.aau.dk>,
        Kresten Krab Thorup <krab@daimi.aau.dk>, gilad.bracha@eng.Sun.COM,
        david.stoutamire@eng.Sun.COM, simonpj@microsoft.com
Cc: Philip Wadler <wadler@research.bell-labs.com>
Subject: The Expression Problem
Date: Thu, 12 Nov 1998 14:27:55 -0500
From: Philip Wadler <wadler@research.bell-labs.com>
```

The Expression Problem
Philip Wadler, 12 November 1998

The Expression Problem is a new name for an old problem. The goal is to define a datatype by cases, where one can add new cases to the datatype and new functions over the datatype, without recompiling existing code, and while retaining static type safety (e.g., no casts). For the concrete example, we take expressions as the data type, begin with one case (constants) and one function (evaluators), then add one more construct (plus) and one more function (conversion to a string).

Whether a language can solve the Expression Problem is a salient indicator of its capacity for expression. One can think of cases as rows and functions as columns in a table. In a functional language, the rows are fixed (cases in a datatype declaration) but it is easy to add new columns (functions). In an object-oriented language, the columns are fixed (methods in a class declaration) but it is easy to add new rows (subclasses). We want to make it easy to add either rows or columns.

The Expresion Problem delineates a central tension in language design. Accordingly, it has been widely discussed, including Reynolds (1975), Cook (1990), and Krishnamurthi, Felleisen and Friedman (1998); the latter includes a more extensive list of references. It has also been discussed on this mailing list by Corky Cartwright and Kim Bruce. Yet I know of no widely-used language that solves The Expression Problem while satisfying the constraints of independent compilation and static typing.

Until now, that is. Here I present a solution to this problem in GJ, as extended by the mechanism I described in my previous note `Do parametric types beat virtual types?'. (However, there is a caveat with regard to inner interfaces, see below.)

1. A solution

Figure 1 shows a solution to the Expression Problem in GJ. The two phases of the problem are clumped into two classes, LangF and Lang2F, each of which defines several mutually recursive inner classes and interfaces.

classes; so the change would not render invalid any existing Java programs. But this point requires further study. Also, I should note that since the changes have not been implemented yet, I have not actually run the proposed solution. (I did translate from GJ to Java by hand, and run that.)

3. Related work

It is instructive to compare this solution with previous solutions circulated by Corky Cartwright and Kim Bruce. Corky's solution requires contravariant extension -- that is, even though Lang2F.Exp extends LangF.Exp, one may use LangF.Exp in place of Lang2F.Exp and not conversely. This partly explains why fixpoints are required here: though LangF is a superclass of Lang2F, the classes Lang and Lang2 are unrelated. Short of complicating the language with contravariance, unrelated classes is the best we could hope for.

Kim's solution required a type to be parameterized over a type constructor (rather than another type). In terms of our example, it required Exp to be parameterized on Visitor. Here, instead of parameterizing Exp on Visitor, Exp refers to This.Vistor<R>. Although GJ supports parametization over types, it does not support parameterization over higher-order type constructors. However, virtual types (as simulated by GJ) in effect support higher-order type parameters for free. I'm grateful to Mads Torgersen and Kresten Krab Thorup for this insight, which they passed on when we discussed this problem at OOPSLA a few weeks ago. (Ironically, though, it looks like this solution won't work in Beta, which lacks interfaces or any other form of multiple supertyping; there also may be a problem in having a single expression type that allows visitors with different result types, like Integer and String.)

The solution presented here is similar to the Extended Visitor pattern described by Krishnamurthi et al. Their solution differs in that it is not statically typed; they cannot distinguish Lang.Exp from Lang2.Exp, and as a result must depend on dynamic casts at some key points. This isn't due to a lack of cleverness on their part, rather it is due to a lack of expressiveness in Pizza.

I am aware of two solutions to the expression problem, but both depend on special-purpose language extensions designed specifically for that problem. One appears in the Krishnamurthi et al. paper, the other in a master's thesis by a student of Martin Odersky. In contrast, the solution presented here arises from the general purpose mechanisms of type parameters and virtual types.

I'd be grateful for pointers to other solutions to the Expression Problem. How do Beta, Sather, Ocaml, and others fare?

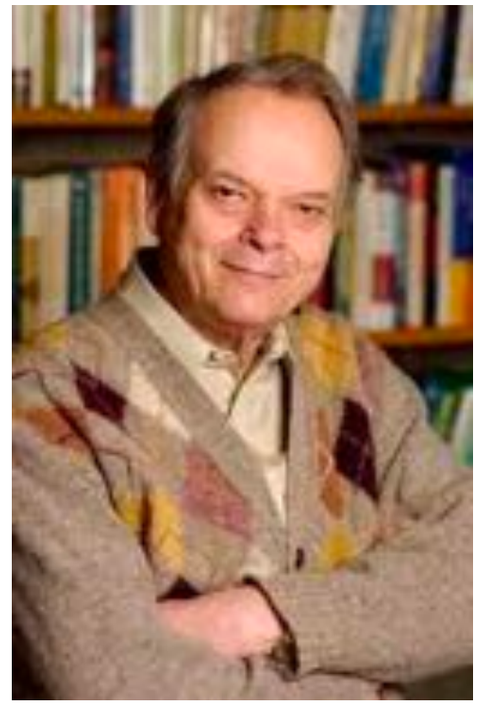
Cheers, -- P

References

W. R. Cook (1990). Object-oriented programming versus abstract data types. REX workshop on Foundations of Object-Oriented Languages, Springer-Verlag LNCS 489, 1990.

S. Krishnamurthi, M. Felleisen, and D. Friedman (1998). Synthesizing object-oriented and functional design to promote re-use. ECOOP 1998, Springer-Verlag LNCS 1445, July 1998.

J. C. Reynolds (1975). User-defined types and procedural data as complementary approaches to data abstraction. In S. A. Schuman, editor, New Directions in Algorithmic Languages, IFIP Working Group 2.1 on Algol, INRIA, 1975. Reprinted in D. Gries, editor, Programming Methodology, Springer-Verlag, 1978, and in C. A. Gunter and J. C. Mitchell, editors, Theoretical Aspects of Object-Oriented Programming, MIT Press, 1994.



Expression language

```
e ::= n | - e | e1 + e2 | ...
```

Operations:

- evaluate to integer value
- convert to string (e.g., for printing)
- determine whether zero occurs in expression
- …

How will you design code to implement language?

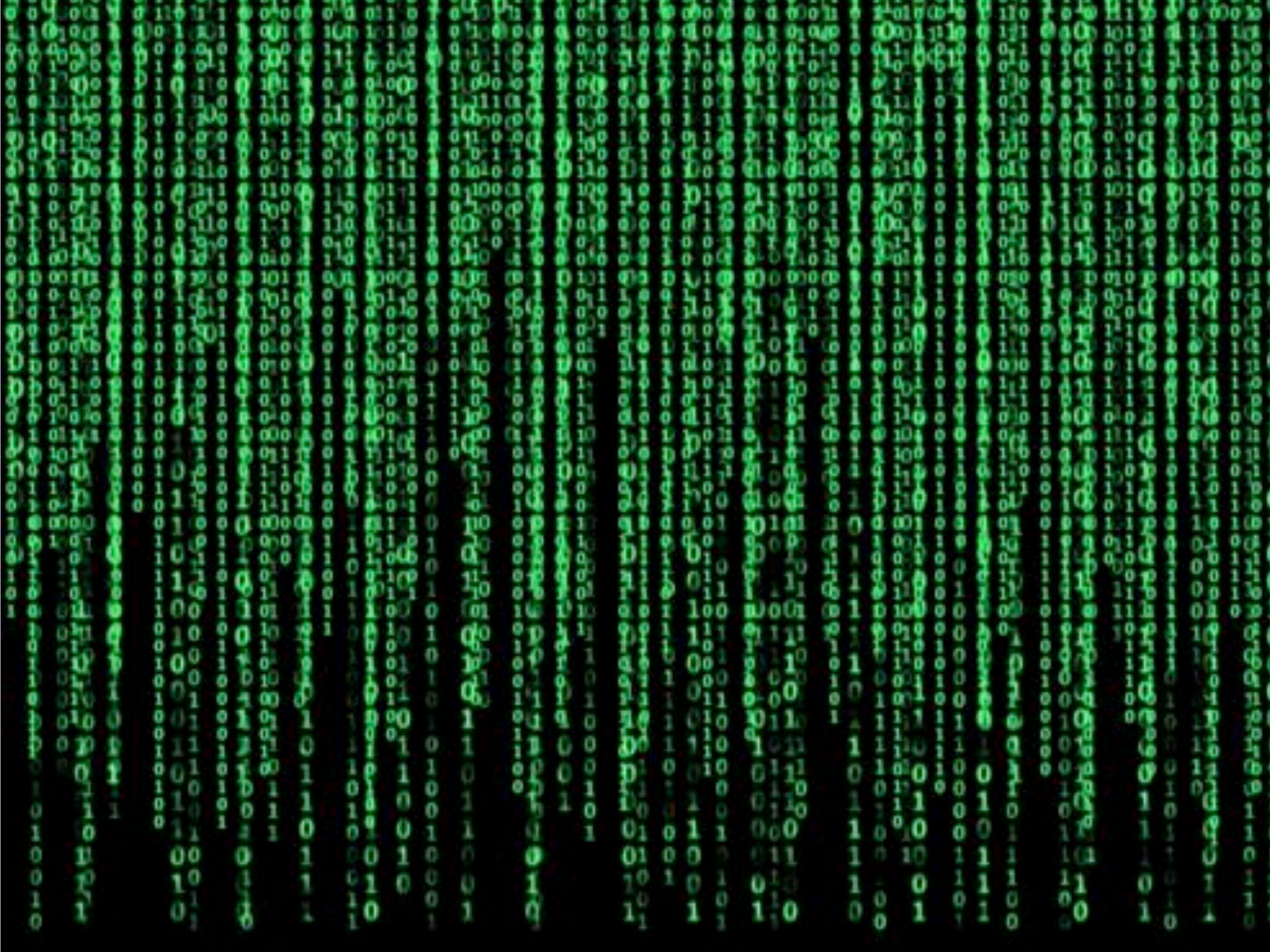
Expression language

```
e ::= n | - e | e1 + e2 | ...
```

Operations:

- evaluate to integer value
- convert to string (e.g., for printing)
- determine whether zero occurs in expression
- ...

How will you design code to implement language?
The answer depends on your perspective on The Matrix.



The Matrix

- Rows are forms of expressions: ints, additions, negations, ...
- Columns are operations to perform: eval, toString, hasZero, …

	eval	toString	hasZero	…
Int				
Add				
Negate				
…				

Implementation will involve deciding "what should happen" for each entry in the matrix *regardless of the PL*

Expression in FP

	eval	toString	hasZero	…
Int				
Add				
Negate				
…				

- In FP, decompose programs into functions that define operations
- Define a *variant type*, with one *constructor* for each expression form
- Fill out the matrix with one *function* per column
  - Function will pattern match on the forms
  - Can use a wildcard pattern if there is a default for multiple forms *(but maybe you shouldn't…)*

stage1.java

Expression Language in Java

```
interface Expr {
    int eval();
    String toString();
    boolean hasZero();
}
```

```
class Int implements Expr {
    private int i;
    public Int(int i) {
       this.i = i;
    }
    public int eval() {
       return i;
    }
    public String toString() {
       return Integer.toString(i);
    }
    public boolean hasZero() {
       return i==0;
    }
}
```

Expression in OOP

	eval	toString	hasZero	…
Int				
Add				
Negate				
…				

- In OOP, decompose programs into classes that define forms
- Define an *abstract class*, with an *abstract method* for each operation
  - In Java, an *interface* works for this
- Fill out the matrix with one *subclass* per row
  - Subclass will have method for each operation
  - Can use inheritance if there is a default for multiple forms
    (but maybe you shouldn't...)

FP vs. OOP

	eval	toString	hasZero	…
Int				
Add				
Negate				
…				

FP vs. OOP: first define a type, then…

- FP: express design by column
- OOP: express design by row

Extension in FP

	eval	toString	hasZero	noNegConstants
Int				
Add				
Negate				
Mult				

- Easy to add a new operation
  - Just write a new function
  - Don't have to modify existing functions
- Hard to add a new form
  - Have to edit all existing functions
  - But type-checker gives a todo list *if you avoid wildcard patterns*

Extension in OOP

	eval	toString	hasZero	noNegConstants
Int				
Add				
Negate				
Mult				

- Easy to add a new form
  - Just write a new class
  - Don't have to modify existing classes
- Hard to add a new operation
  - Have to modify all existing classes
  - But Java type-checker gives a todo list *if you avoid inheritance of methods*

…. back to UML!

UML — Unified Modeling Language

Today: Class Diagrams (Klassendiagramme)

1.) Attributes and Methods of a Class

2.) Associations between Classes

3.) Aggregation

4.) Composition

5.) Inheritance

Example: a bank

6.) Interfaces

UML: Einführung

Vokabular der UML (nach Booch/Rumbaugh/Jacobson)

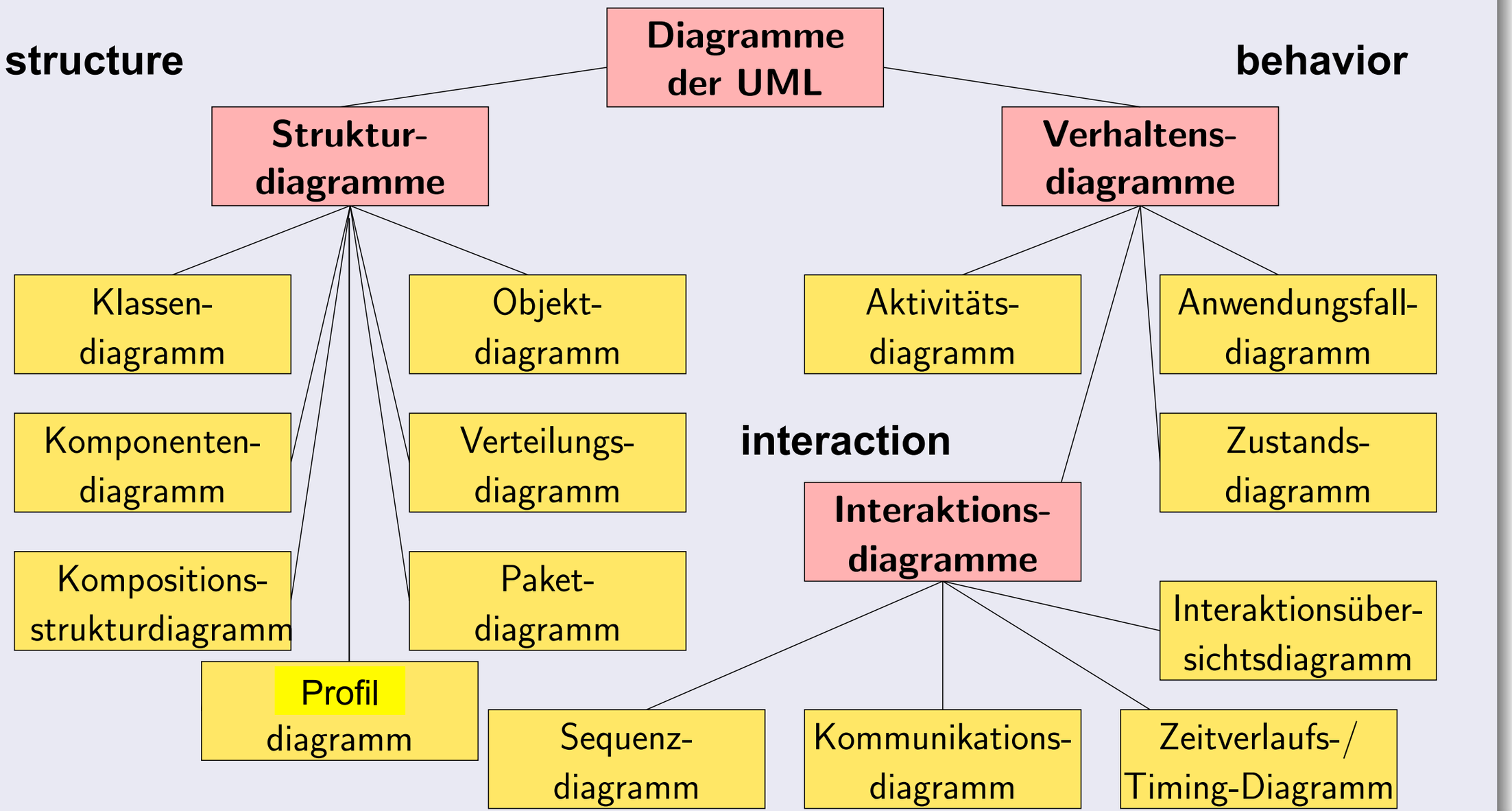
- Dinge (things)
- Beziehungen (relationships)
- Diagramme (diagrams)

Dinge

- Strukturen (structural things)
- Verhalten (behavioral things)
- Gruppen (grouping things)
- Annotationen (annotational things)

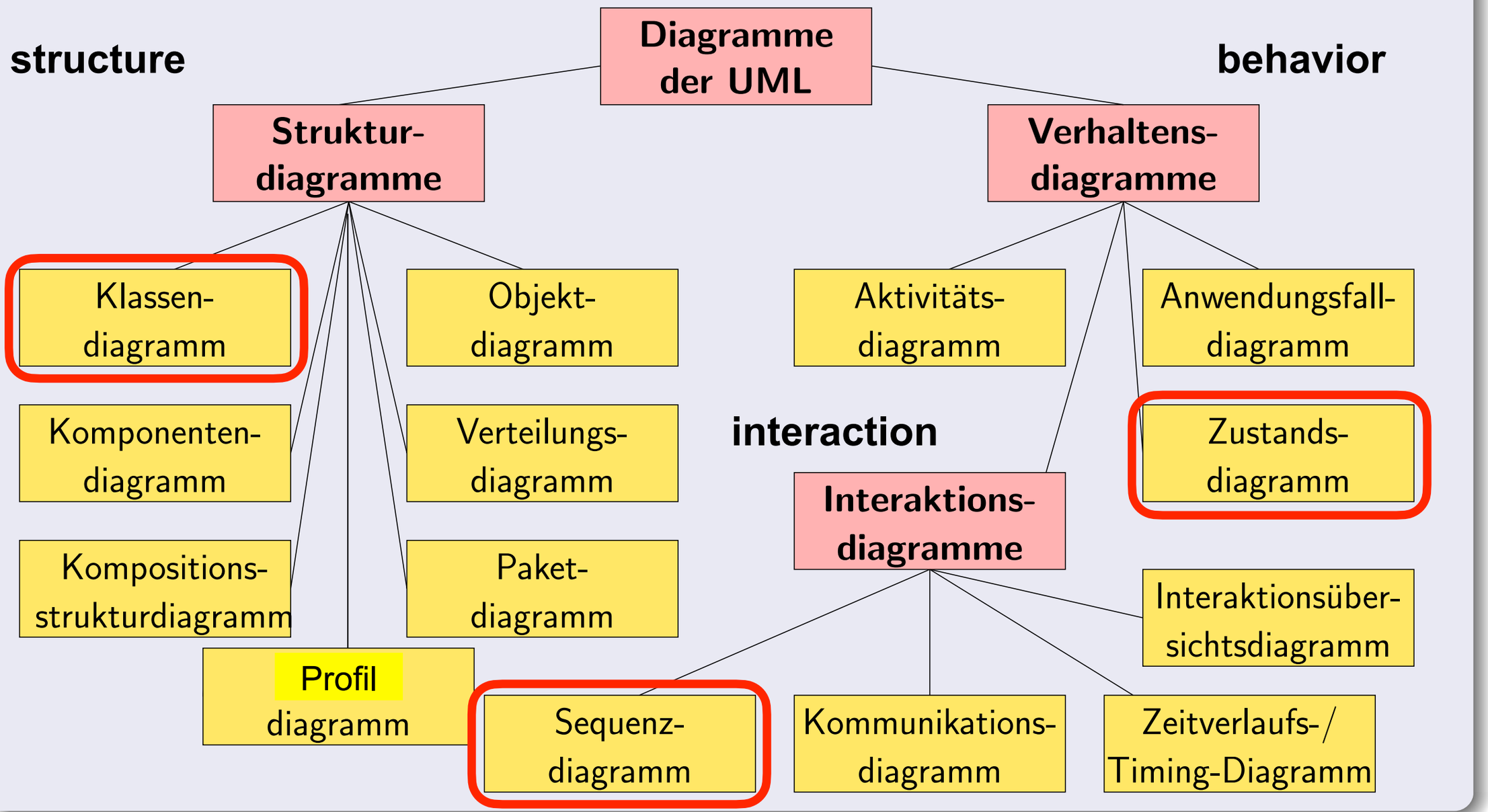
UML: Einführung

UML-Diagramme



UML: Einführung

UML-Diagramme



Klassendiagramme

Klassendiagramme

Disclaimer.

— these diagrams are meant to communicate with others about software

— omit complicated data type definitions etc in the **early phase** of modelling!

Klassendiagramme

Abbildung 2.2 Die einfachste Darstellung einer Klasse

Name der Klasse

The following Figures are from this book

— eine Klasse benötigt mindestens einen **Namen**

Klassendiagramme

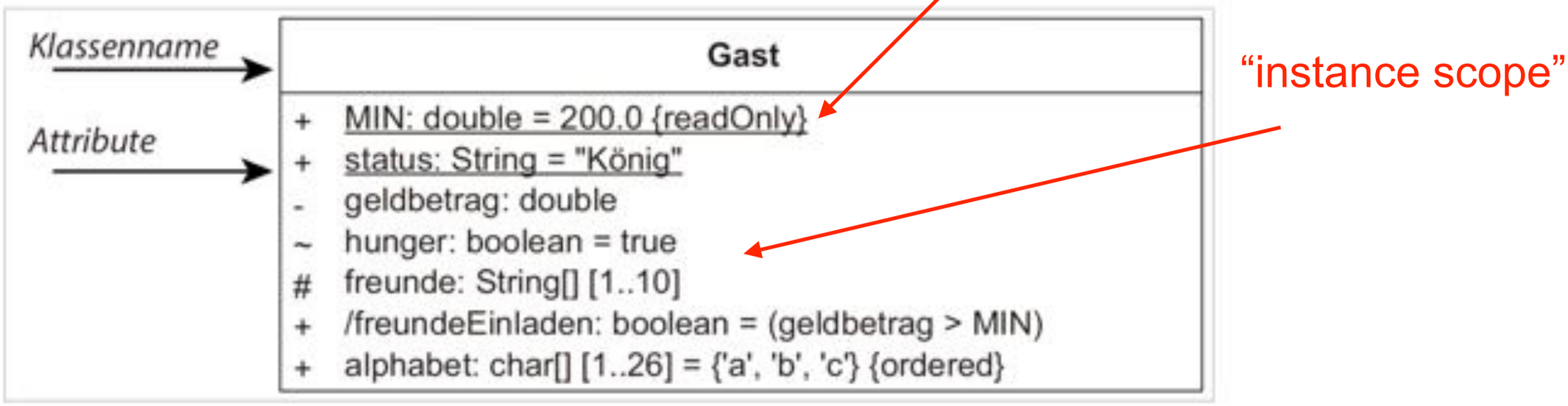


Abbildung 2.3 Attribute einer Klasse

— Instanzattribute: definieren den Zustand der aus dieser Klasse geblideten Objekte. Für jedes Objekt wird das Attribute separat erzeugt.

— Klassenattribute (unterstrichen): Nur einmal vorhanden und unabhängig von der Instanziierung von Objekten.

Klassendiagramme

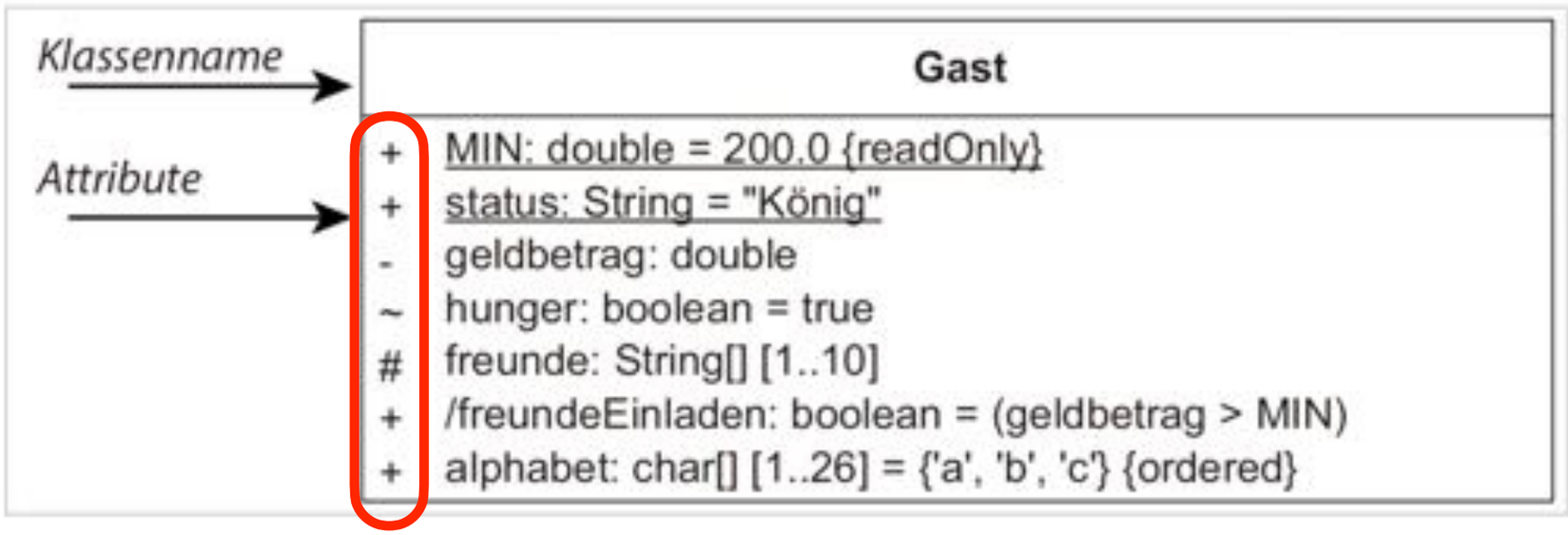


Abbildung 2.3 Attribute einer Klasse

[Sichtbarkeit] [/] Name [:Typ] [Multiplizität] [=Vorgabewert] [{Eigenschaftswert}]

+ (public) das Attribut ist für alle Klassen sichtbar

 # (protected) sichtbar fuer diese Klasse und all Klassen, die von ihr erben

- (private) nur sichtbar für diese Klasse

~ (package) nur sichtbar für die Klassen, die sich im gleichen Paket befinden

Klassendiagramme

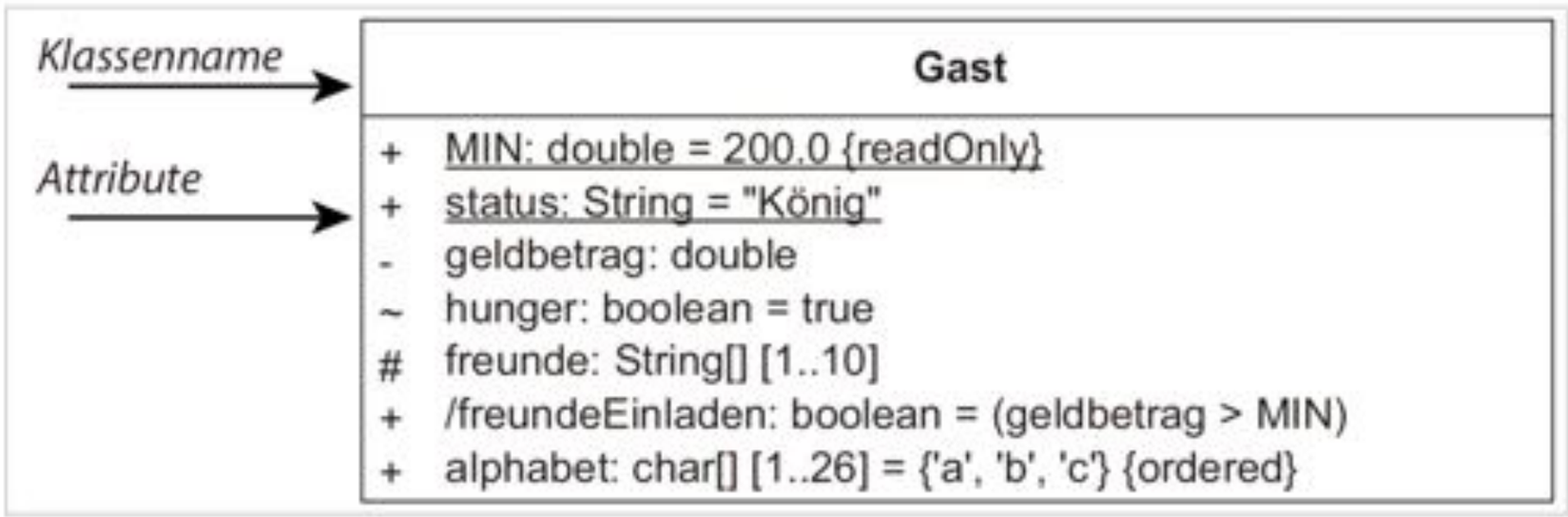


Abbildung 2.3 Attribute einer Klasse

[Sichtbarkeit] [/] Name [:Typ] [Multiplizität] [=Vorgabewert] [{Eigenschaftswert}]

abgeleitet

Der Wert des Attributes kann aus anderen Attributwerten abgeleitet werden.

Klassendiagramme

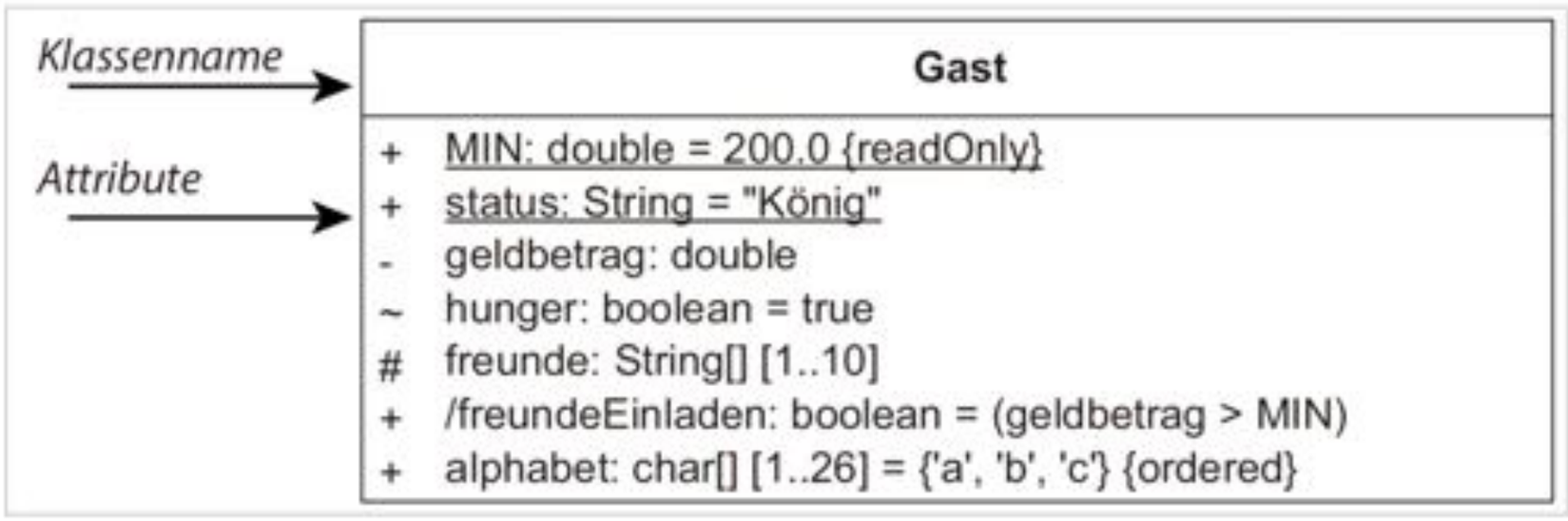


Abbildung 2.3 Attribute einer Klasse

[Sichtbarkeit] [/] Name [:Typ] [Multiplizität] [=Vorgabewert] [{Eigenschaftswert}]

beliebiger Name

beliebiger Datentyp

z.B.

```
[5]
[1..2]
[1..*]
[0..*] = [*]
```

Klassendiagramme

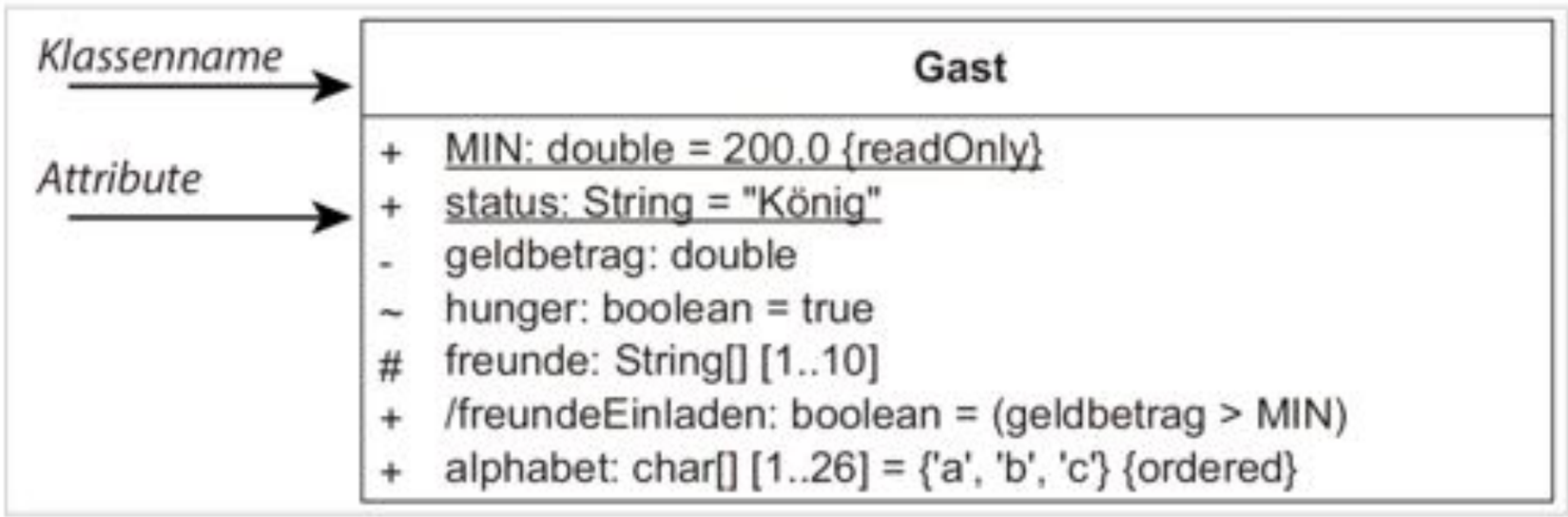


Abbildung 2.3 Attribute einer Klasse

[Sichtbarkeit] [/] Name [:Typ] [Multiplizität] [=Vorgabewert] [{Eigenschaftswert}]

beliebiger Name

beliebiger Datentyp

z.B.
```
[5]
[1..2]
[1..*]
[0..*] = [*]
```

z.B.
```
{id}
{readOnly}
{sequence}
{ordered}
{unique}
```

Klassendiagramme

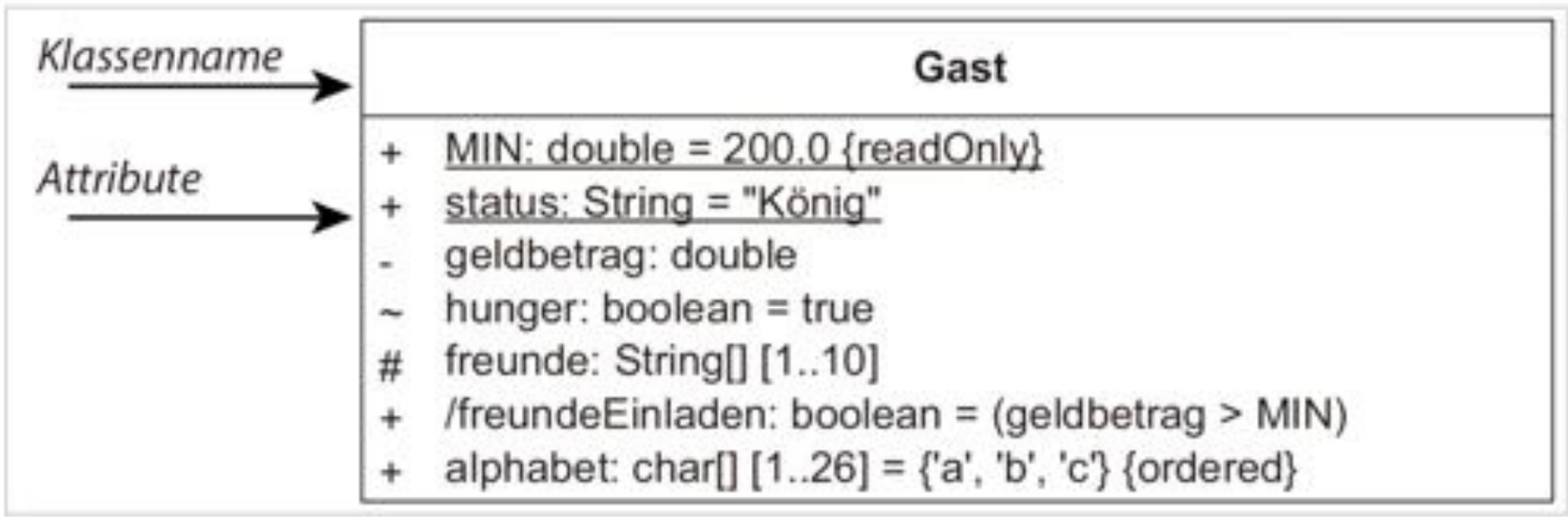


Abbildung 2.3 Attribute einer Klasse

[Sichtbarkeit] [/] Name [:Typ] [Multiplizität] [=Vorgabewert] [{Eigenschaftswert}]

beliebiger Name

beliebiger Datentyp

z.B.
```
[5]
[1..2]
[1..*]
[0..*] = [*]
```

z.B.
```
{id}
{readOnly}
{sequence}
{ordered}
{unique}
```

attribute is part of the
primary key (identifier) of the class

Klassendiagramme

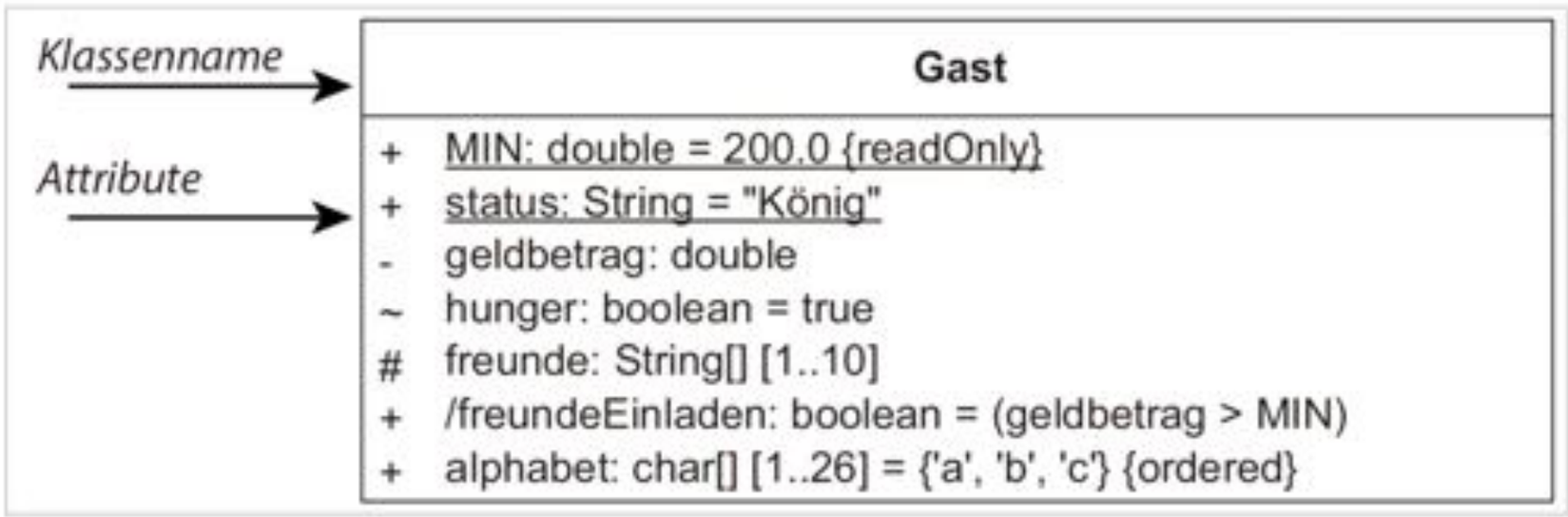


Abbildung 2.3 Attribute einer Klasse

[Sichtbarkeit] [/] Name [:Typ] [Multiplizität] [=Vorgabewert] [{Eigenschaftswert}]

beliebiger Name

beliebiger Datentyp

z.B.
```
[5]
[1..2]
[1..*]
[0..*] = [*]
```

z.B.
```
{id}
{readOnly}
{sequence}
{ordered}
{unique}
```

used for constants, e.g.,
```
pi: Real = 3.1415 {readOnly}
```

Klassendiagramme

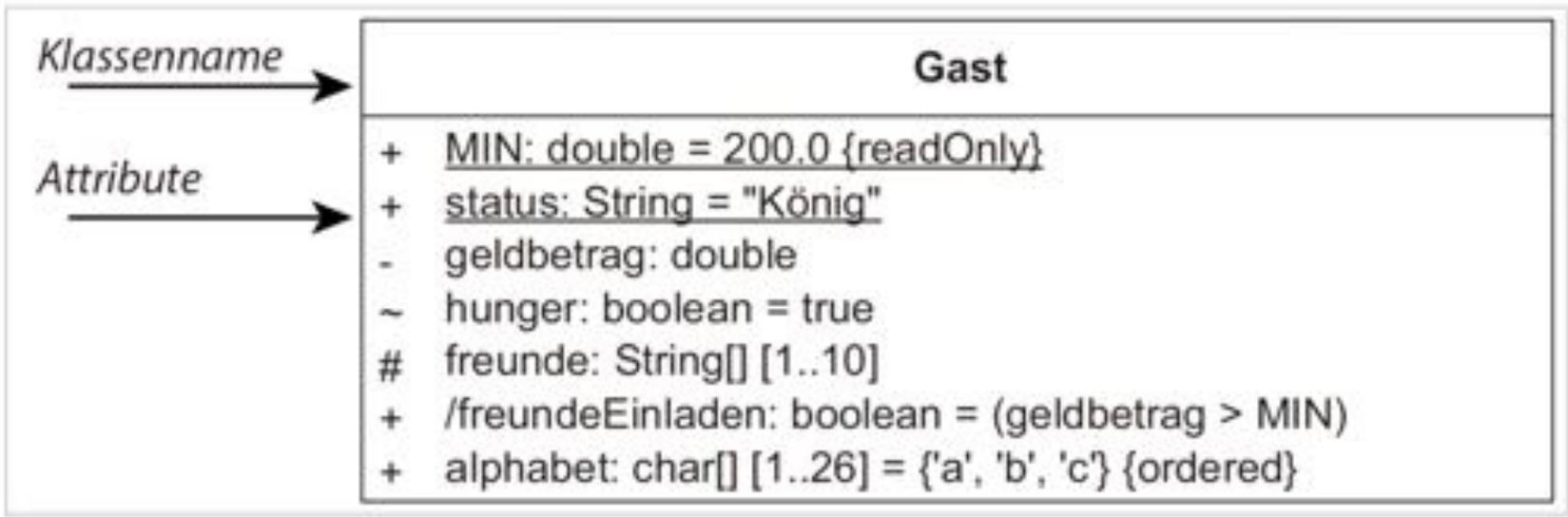


Abbildung 2.3 Attribute einer Klasse

[Sichtbarkeit] [/] Name [:Typ] [Multiplizität] [=Vorgabewert] [{Eigenschaftswert}]

beliebiger Name

beliebiger Datentyp

z.B.
```
[5]
[1..2]
[1..*]
[0..*] = [*]
```

z.B.
```
{id}
{readOnly}
{sequence}
{ordered}
{unique}
```

attribute must contain ordered sequences, e.g.
`[1,1,3,5,6,6,6,24]`

Klassendiagramme

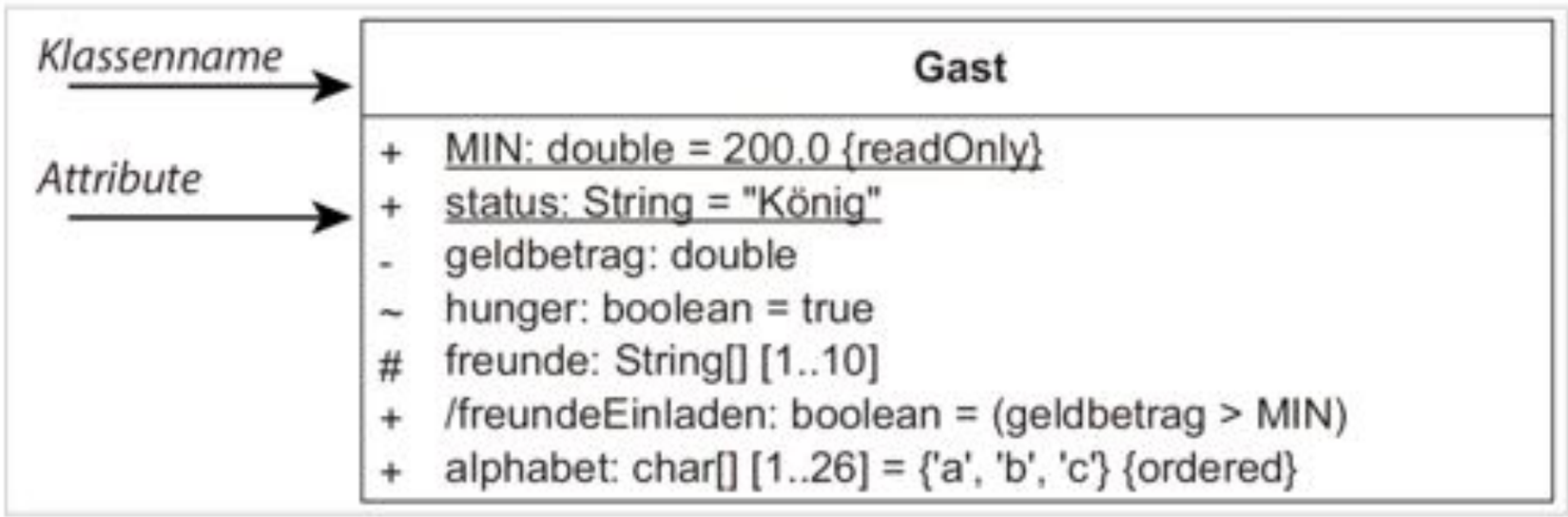


Abbildung 2.3 Attribute einer Klasse

[Sichtbarkeit] [/] Name [:Typ] [Multiplizität] [=Vorgabewert] [{Eigenschaftswert}]

beliebiger Name

beliebiger Datentyp

z.B.
```
[5]
[1..2]
[1..*]
[0..*] = [*]
```

z.B.
```
{id}
{readOnly}
{sequence}
{ordered}
{unique}
```

ordered sequences (no duplicates), e.g.
```
[1,3,5,6,24]
```

Klassendiagramme

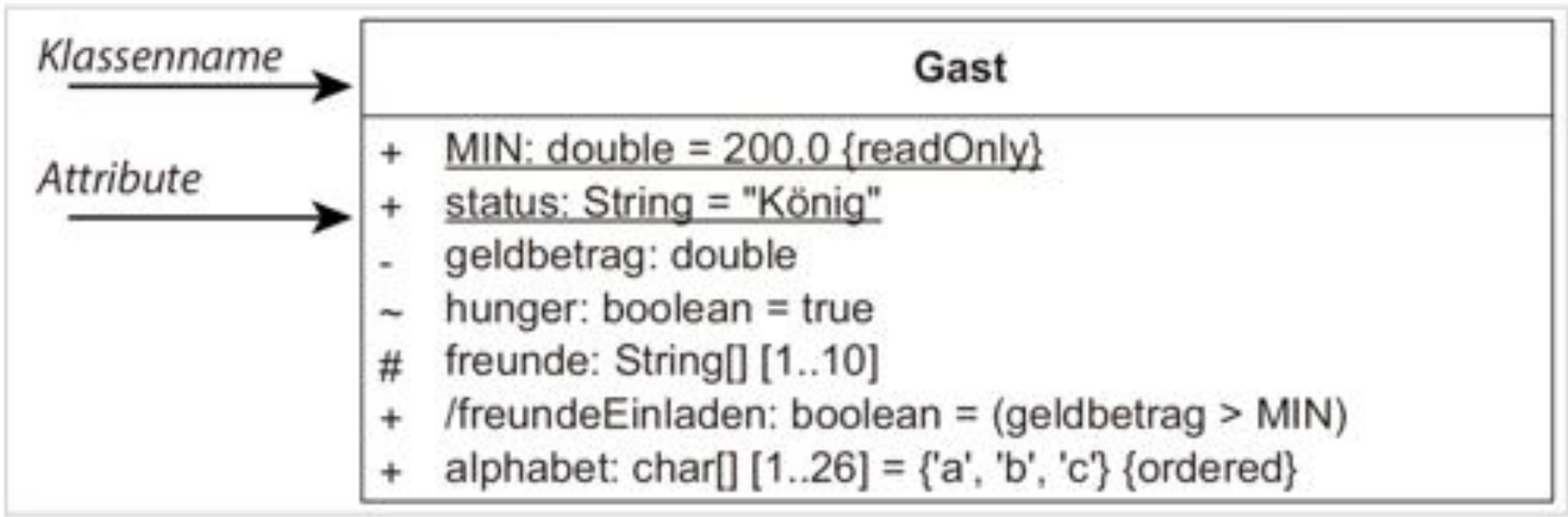


Abbildung 2.3 Attribute einer Klasse

[Sichtbarkeit] [/] Name [:Typ] [Multiplizität] [=Vorgabewert] [{Eigenschaftswert}]

beliebiger Name

beliebiger Datentyp

z.B.
```
[5]
[1..2]
[1..*]
[0..*] = [*]
```

z.B.
```
{id}
{readOnly}
{sequence}
{ordered}
{unique}
```

no duplicates, e.g.
```
[24,1,7,5,3,6]
```

Klassendiagramme

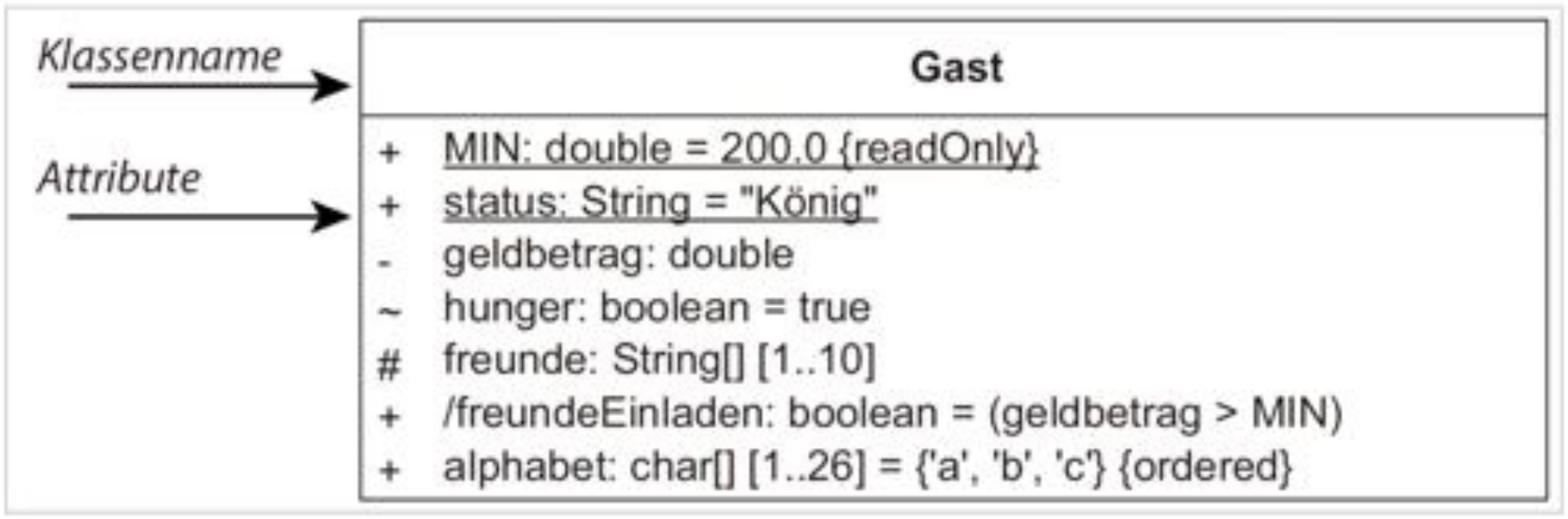


Abbildung 2.3 Attribute einer Klasse

[Sichtbarkeit] [/] Name [:Typ] [Multiplizität] [=Vorgabewert] [{Eigenschaftswert}]

beliebiger Name

beliebiger Datentyp

z.B.
```
[5]
[1..2]
[1..*]
[0..*] = [*]
```

z.B.
```
{id}
{readOnly}
{sequence}
{ordered}
{nonunique}
```

duplicates & any order, e.g.
```
[24,24,1,7,5,1,1]
```

Klassendiagramme

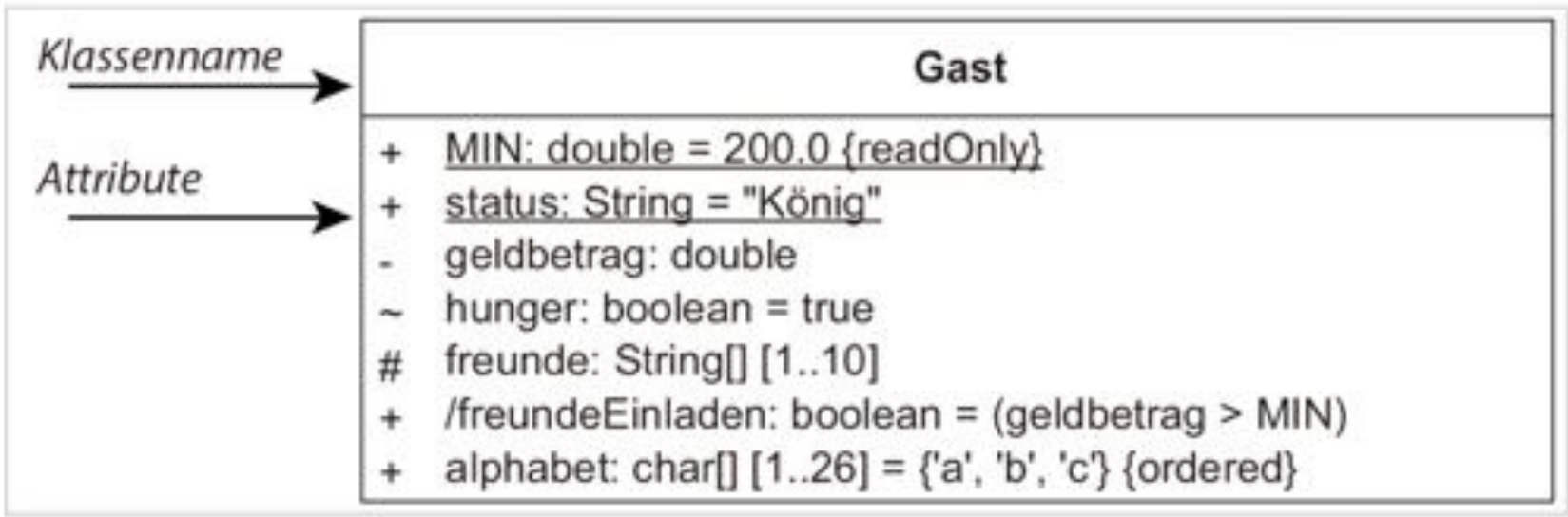


Abbildung 2.3 Attribute einer Klasse

[Sichtbarkeit] [/] Name [:Typ] [Multiplizität] [=Vorgabewert] [{Eigenschaftswert}]

beliebiger Name

beliebiger Datentyp

z.B.
```
[5]
[1..2]
[1..*]
[0..*] = [*]
```

z.B.
```
{id}
{readOnly}
{sequence}
{ordered}
{(non)unique}
{subsets <AttrName>}
```

values must be in the domain of `<AttrName>`

Klassendiagramme

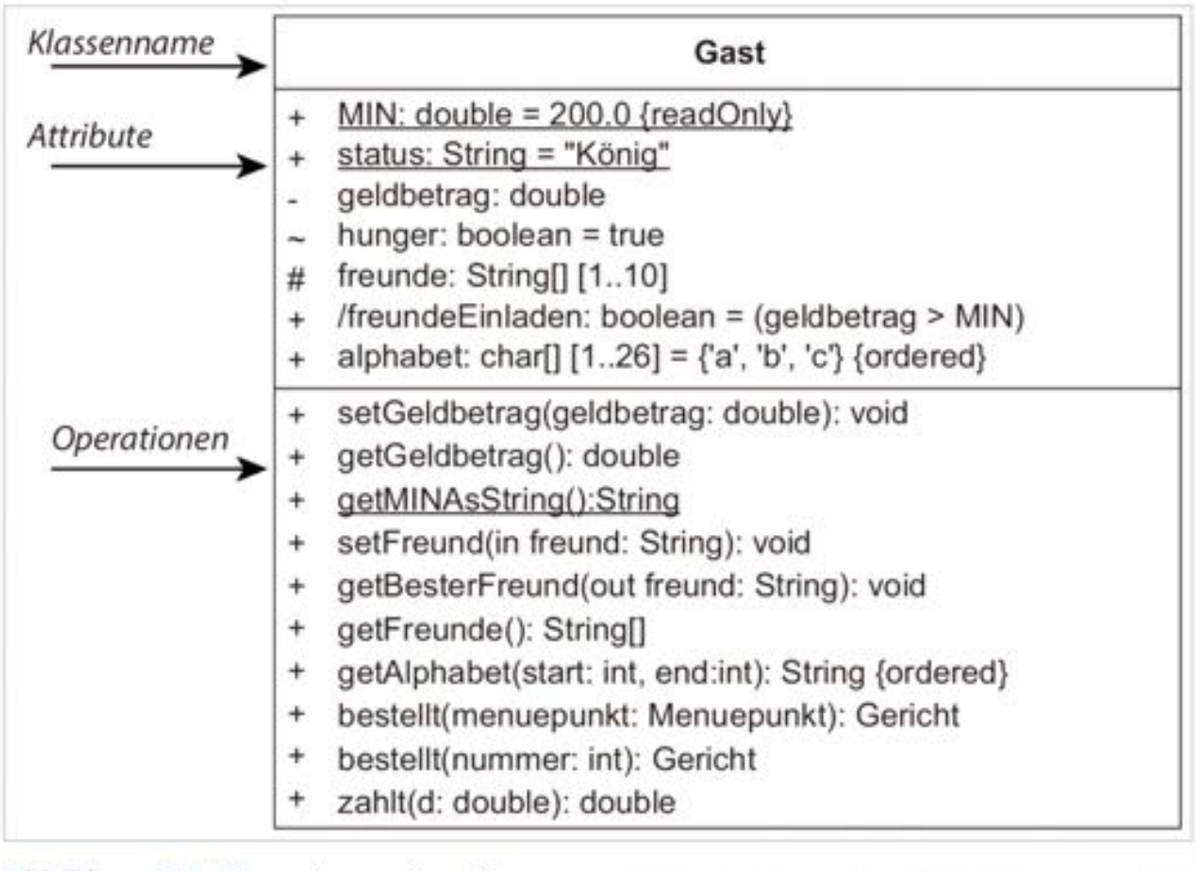


Abbildung 2.5 Operationen einer Klasse

aka
methods

[Sichtbarkeit] Name ([Parameter-Liste]) [:Rückgabetyp] [Multiplizität] [{Eigenschaft}]

Klassendiagramme

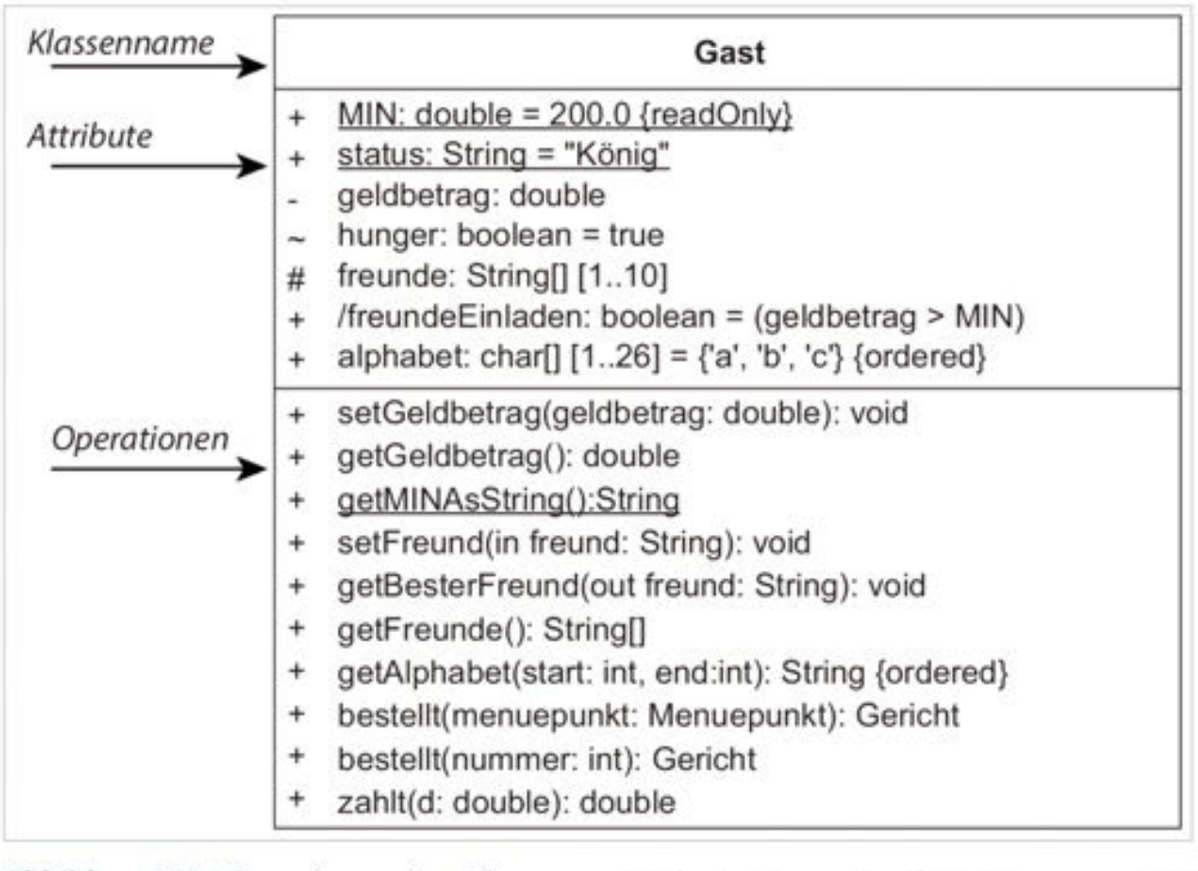


Abbildung 2.5 Operationen einer Klasse

aka
methods

[Sichtbarkeit] Name ([Parameter-Liste]) [:Rückgabetyp] [Multiplizität] [{Eigenschaft}]

Jeder **Parameter** hat diesen Aufbau: **[Übergabemodus] Name :Typ[Multiplizität][=Vorgabewert] [{Eigenschaft}]**

Klassendiagramme

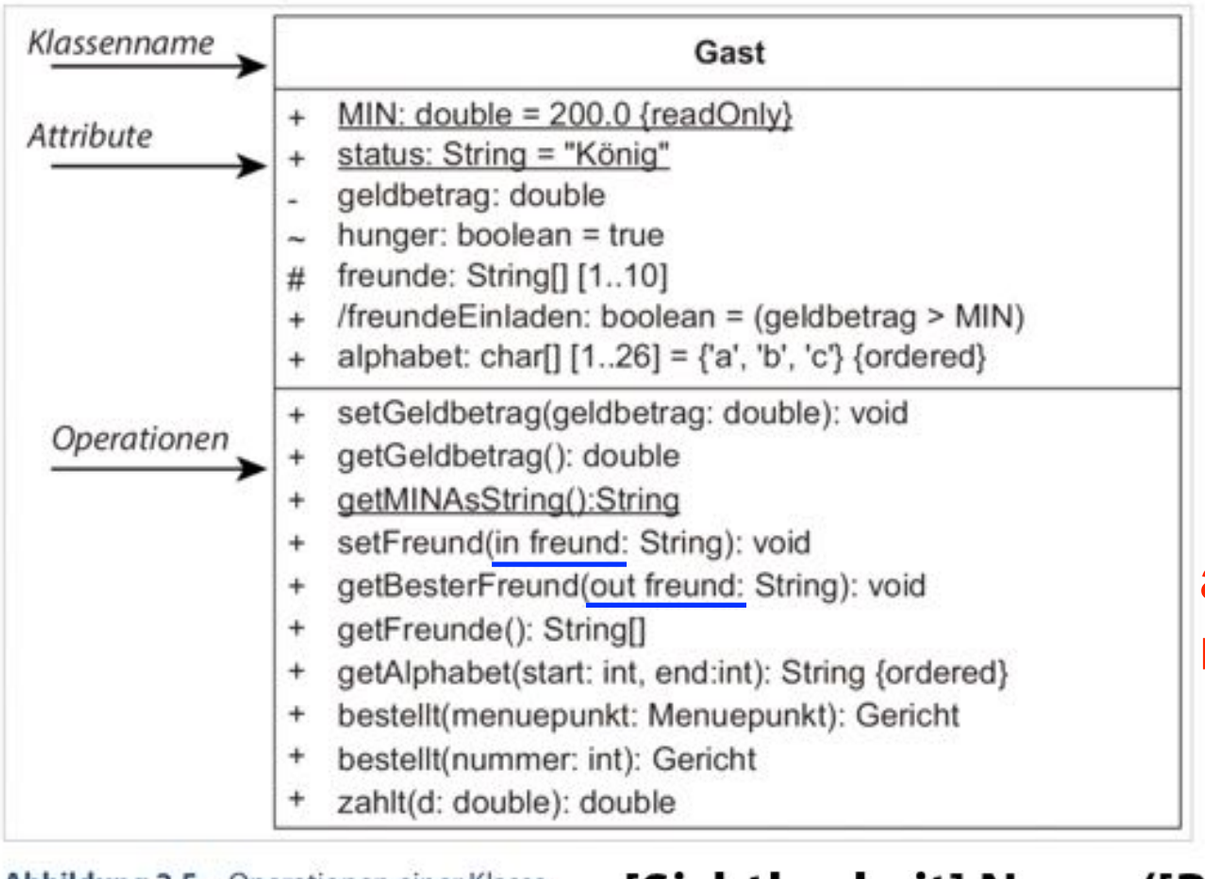


Abbildung 2.5 Operationen einer Klasse

aka methods

[Sichtbarkeit] Name ([Parameter-Liste]) [:Rückgabetyp] [Multiplizität] [{Eigenschaft}]

Jeder **Parameter** hat diesen Aufbau: **[Übergabemodus] Name :Typ[Multiplizität][=Vorgabewert] [{Eigenschaft}]**

— **`in`** darf nur gelesen werden
— **`out`** darf nur geschrieben werden (nicht gelesen)
— **`inout`** dart gelesen und geschrieben werden

Klassendiagramme

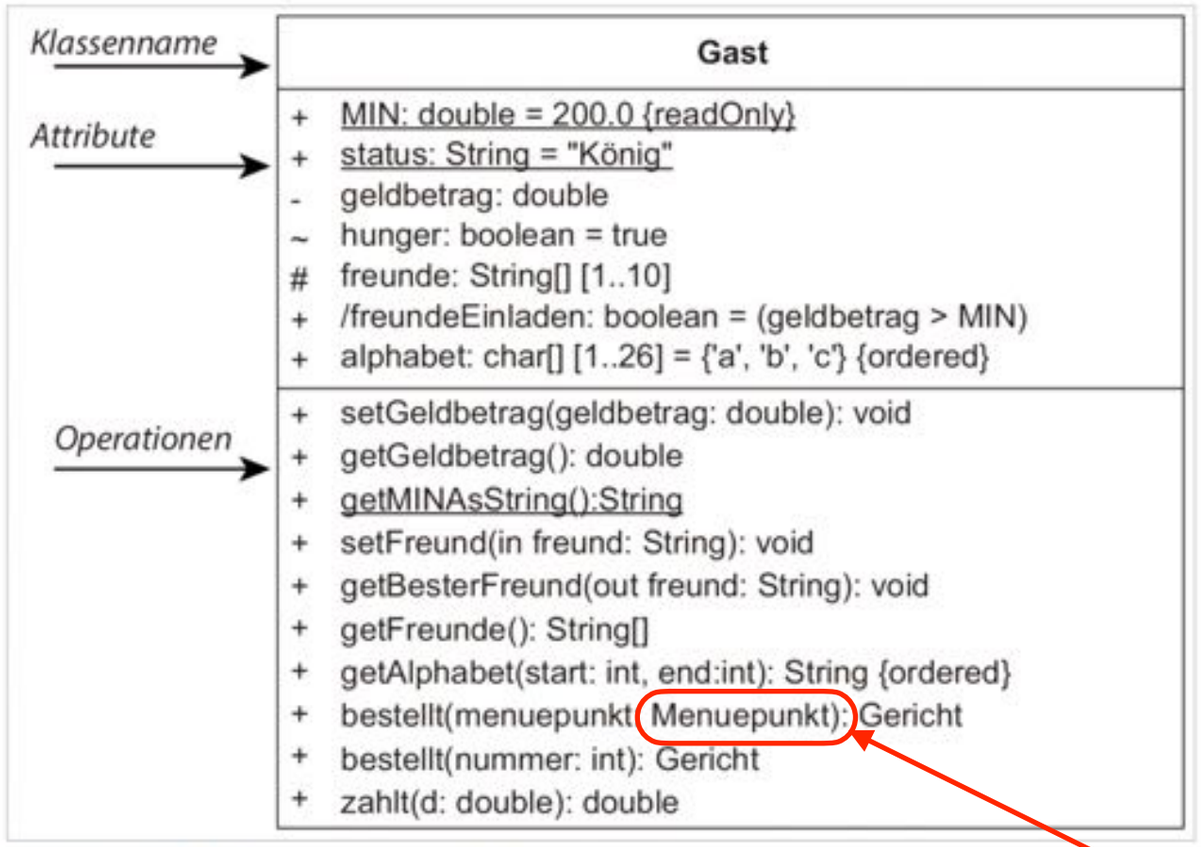


Abbildung 2.5 Operationen einer Klasse

Dieser enumeration-Datentyp wird im gleichen Diagram definierit

Klassendiagramme

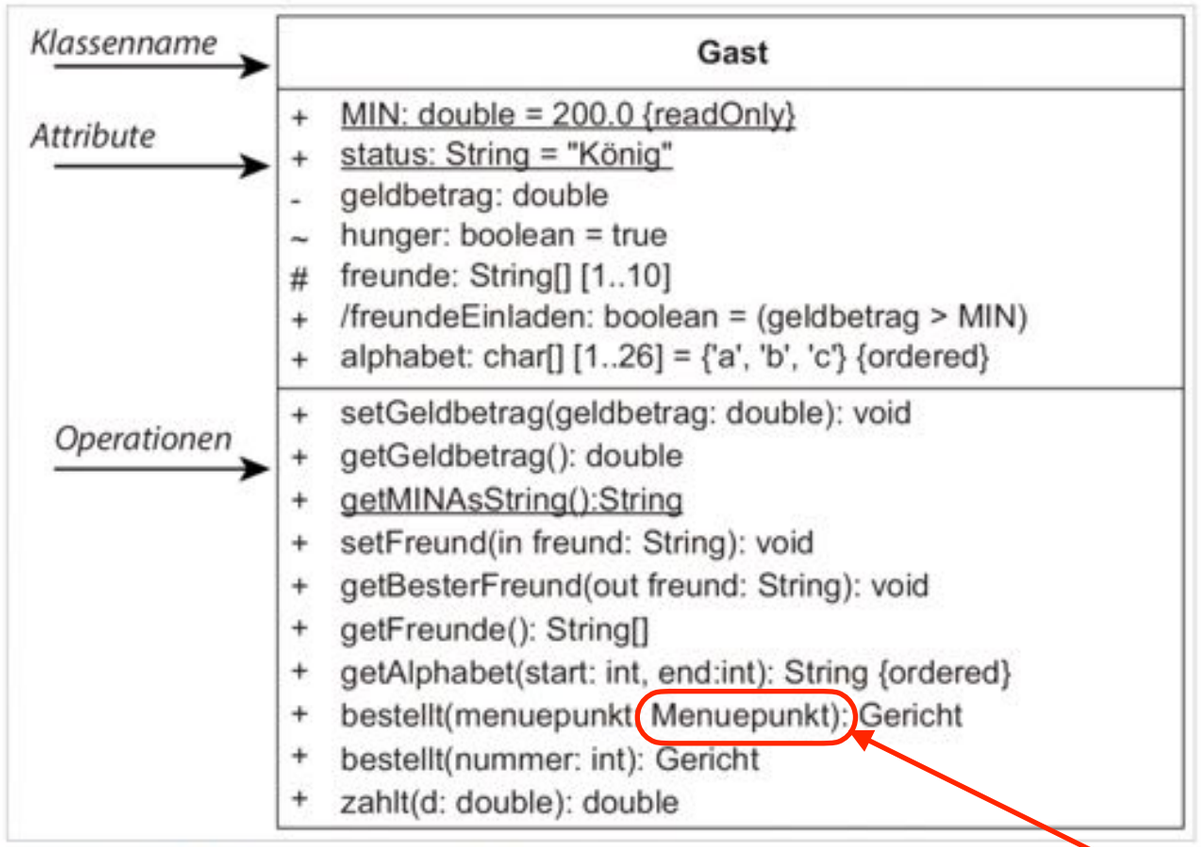
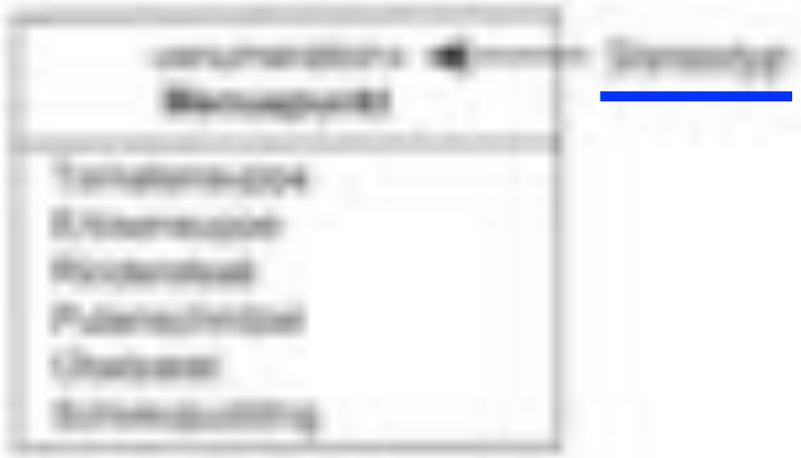


Abbildung 2.5 Operationen einer Klasse



Dieser enumeration-Datentyp wird im gleichen Diagram definierit

Andere gebräuchliche **Stereotypen**

```
<<script>>
<<source>>
<<executable>>
<<document>>
<<file>>
<<library>>
<<form>>
```

Klassendiagramme

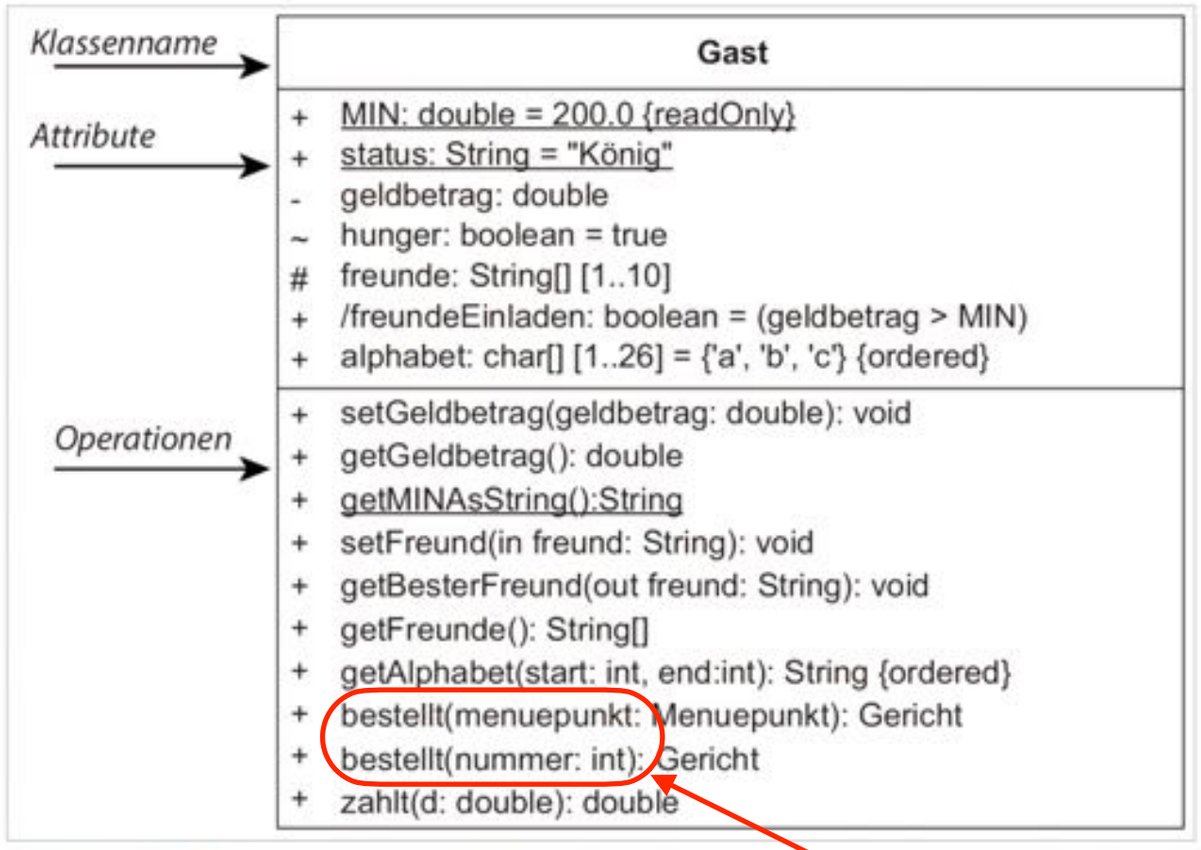


Abbildung 2.5 Operationen einer Klasse

Überladung der “bestellt” Methode

Assoziation zwischen Klassen

Assoziation zwischen Klassen

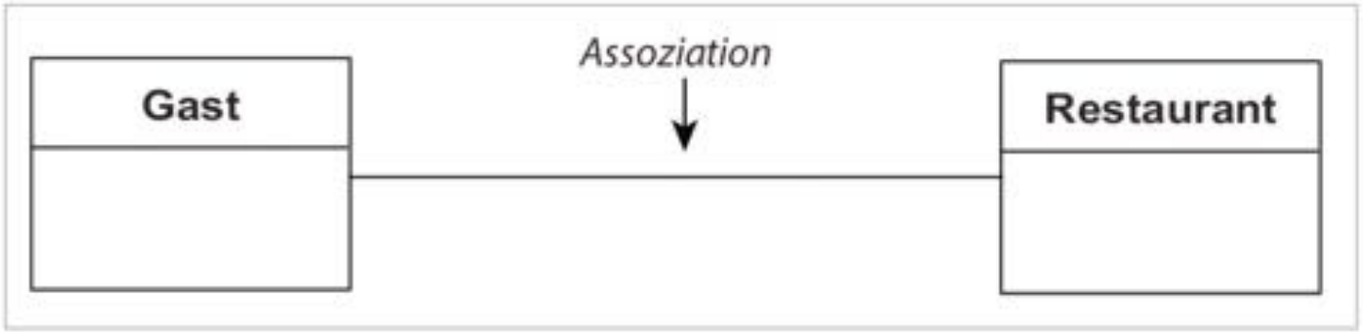


Abbildung 2.7 Assoziation

Eine Assoziation spezifiziert eine **semantische Beziehung** zwischen Klassen.

Assoziation zwischen Klassen

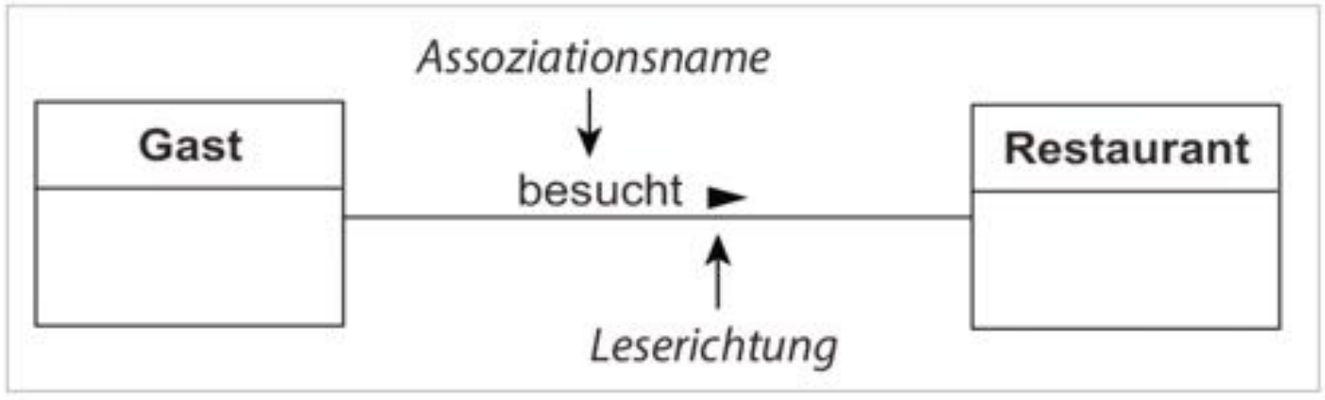


Abbildung 2.8 Assoziation mit Namensangabe und Leserichtung

Eine Assoziation spezifiziert eine **semantische Beziehung** zwischen Klassen.

— es gibt Gäste, die Restaurants besuchen
— es gibt Restaurants, die von Gästen besucht werden

similar to Entity-Relationship Diagrams

Assoziation zwischen Klassen

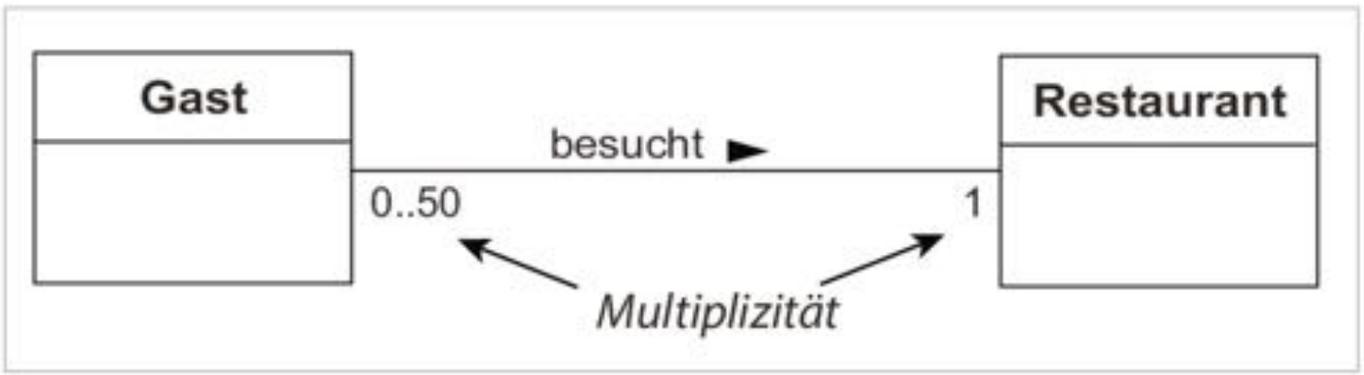


Abbildung 2.9 Assoziation mit Multiplizitätsangabe

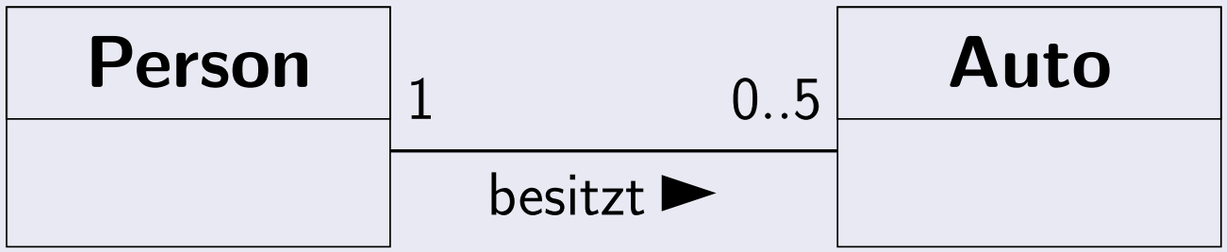
Eine Assoziation spezifiziert eine **semantische Beziehung** zwischen Klassen.

— **Ein** Gast besucht (zu einem Zeitpunkt) genau **ein** Restaurant
— **Ein** Restaurant wird (zu einem Zeitpunkt) von **keinem bis fünfzig** Gästen besucht.

similar to Entity-Relationship Diagrams

Klassen- und Objektdiagramme

- Multiplizitäten: an beide Enden der Assoziationen können Multiplizitäten in Form von Intervallen $m..n$ (oder einfach nur m für $m..m$) angegeben werden. Hier besitzt eine Person bis zu fünf Autos. Ein Auto ist im Besitz genau einer Person.



Falls die Multiplizität größer als eins ist, muss dies in der Implementierung durch eine Liste (oder Menge oder Array) von Referenzen realisiert werden.

Assoziation zwischen Klassen

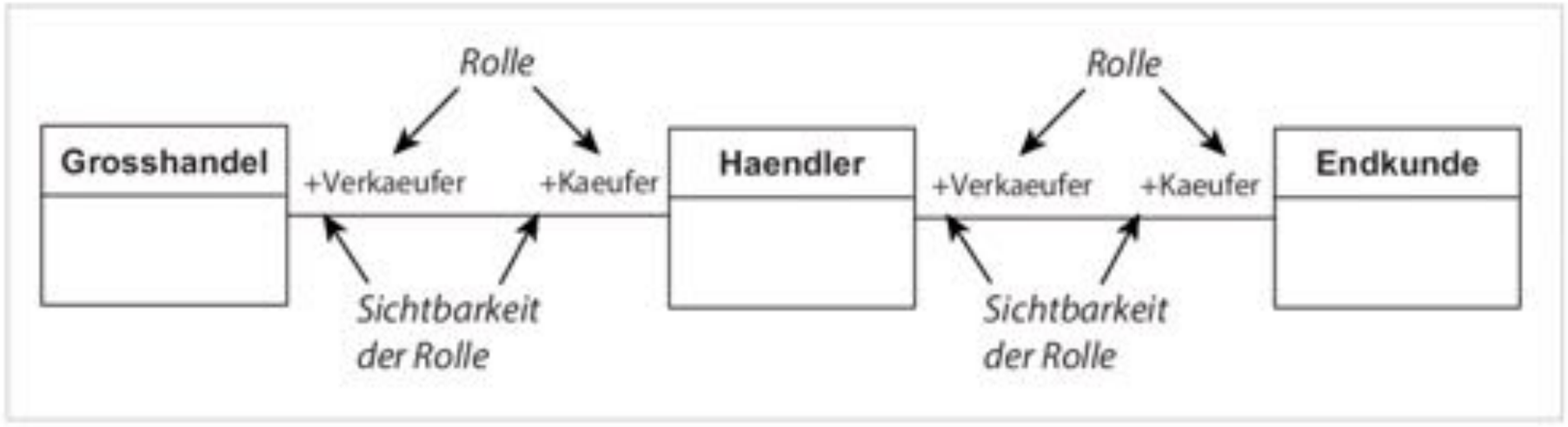
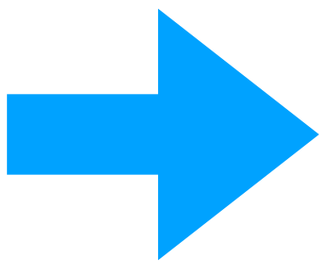


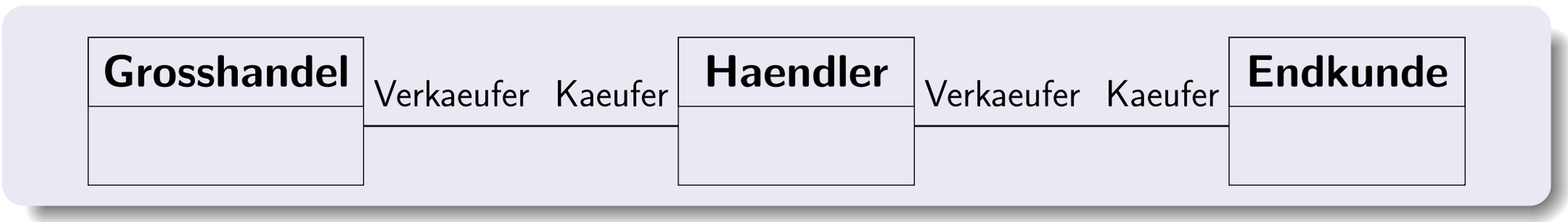
Abbildung 2.10 Assoziationen mit Rollenangaben



similar to Entity-Relationship Diagrams

Klassen- und Objektdiagramme

- Rollen: Klassen können in verschiedenen Assoziationen verschiedene Rollen spielen. Rollen werden auch an den Assoziationen notiert (und können alle Bestandteile eines Attributs enthalten).



Assoziation zwischen Klassen

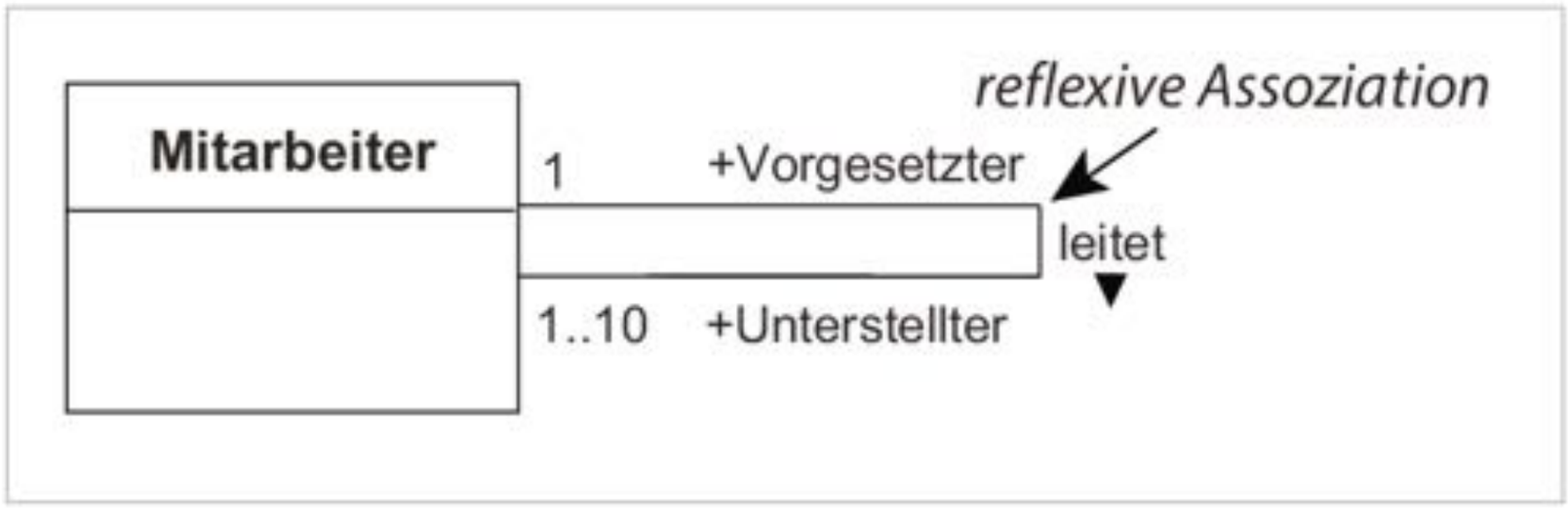


Abbildung 2.20 Reflexive Assoziation

similar to Entity-Relationship Diagrams

Assoziation zwischen Klassen

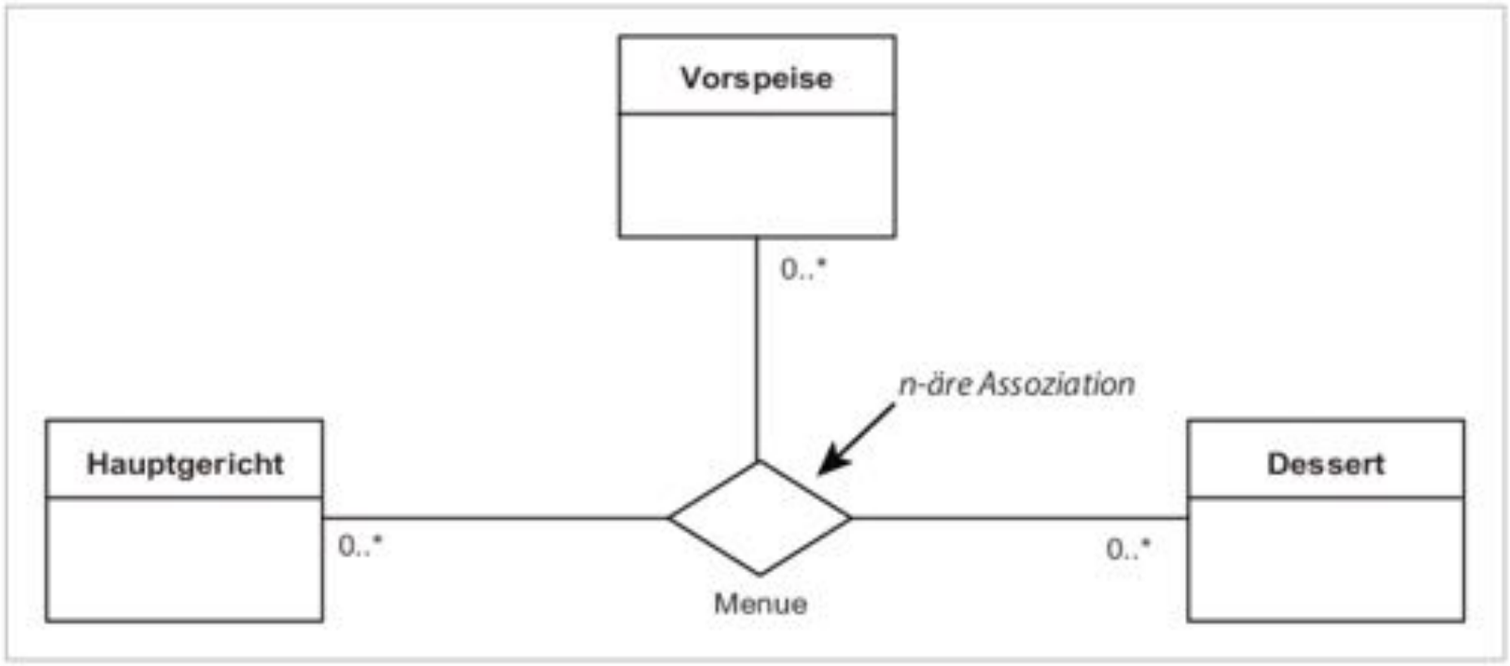


Abbildung 2.21 n-äre Assoziation zwischen Klassen

similar to Entity-Relationship Diagrams

Aggregation

Aggregation

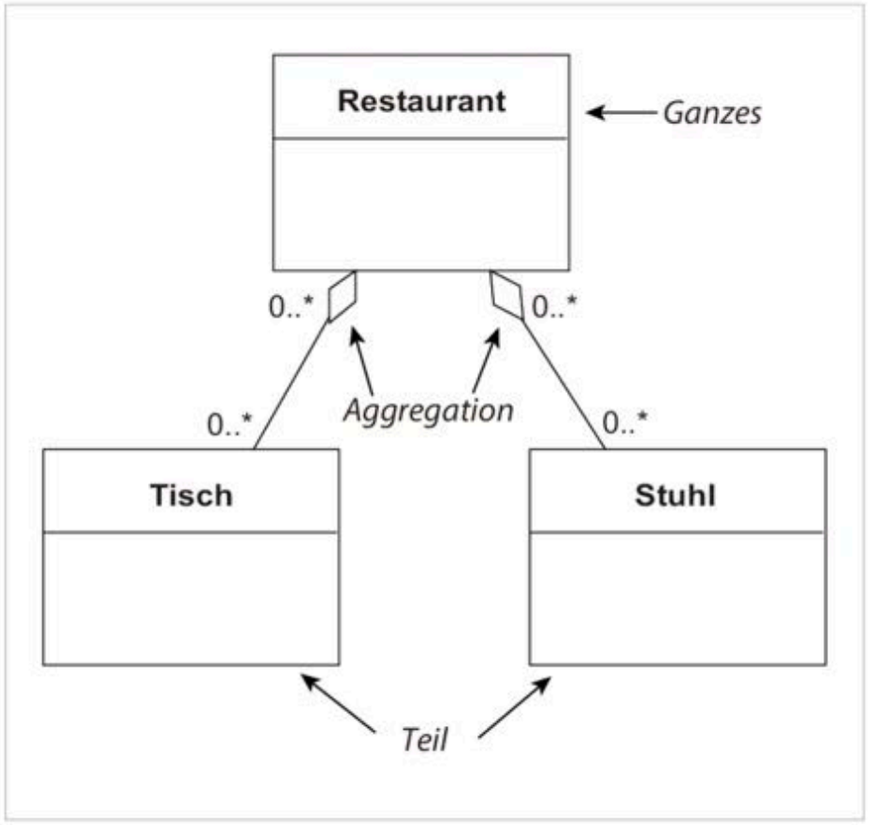


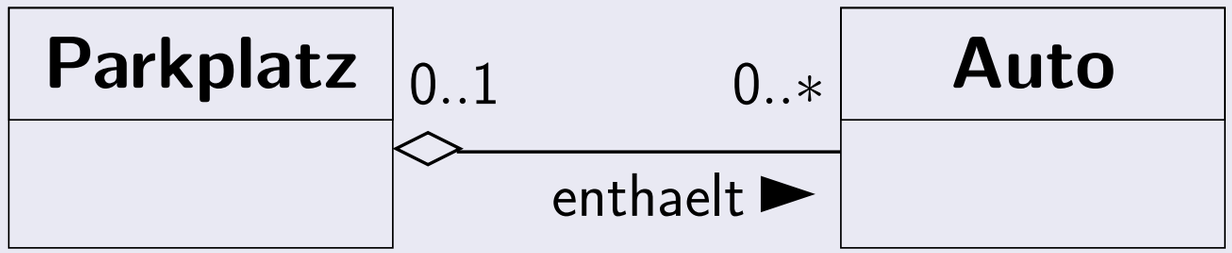
Abbildung 2.27 Aggregation

Aggregation = binäre Ganz-Teile-Beziehung

Klassen- und Objektdiagramme

Beispiel für eine Aggregation

Ein Parkplatz “enthält” mehrere Autos.



Klassen- und Objektdiagramme

Die stärkste Relation ist die sogenannte Komposition.

Komposition

Es gibt eine Komposition zwischen den Klassen **A** und **B**, wenn

Instanzen der Klasse **A** Instanzen der Klasse **B** als Teile enthalten *und* die Lebenszeit der Teile wird vom "Ganzen" kontrolliert. Das heißt, die Teile können (müssen) gelöscht werden, sobald die Instanz der Klasse **A** gelöscht wird.

Komposition

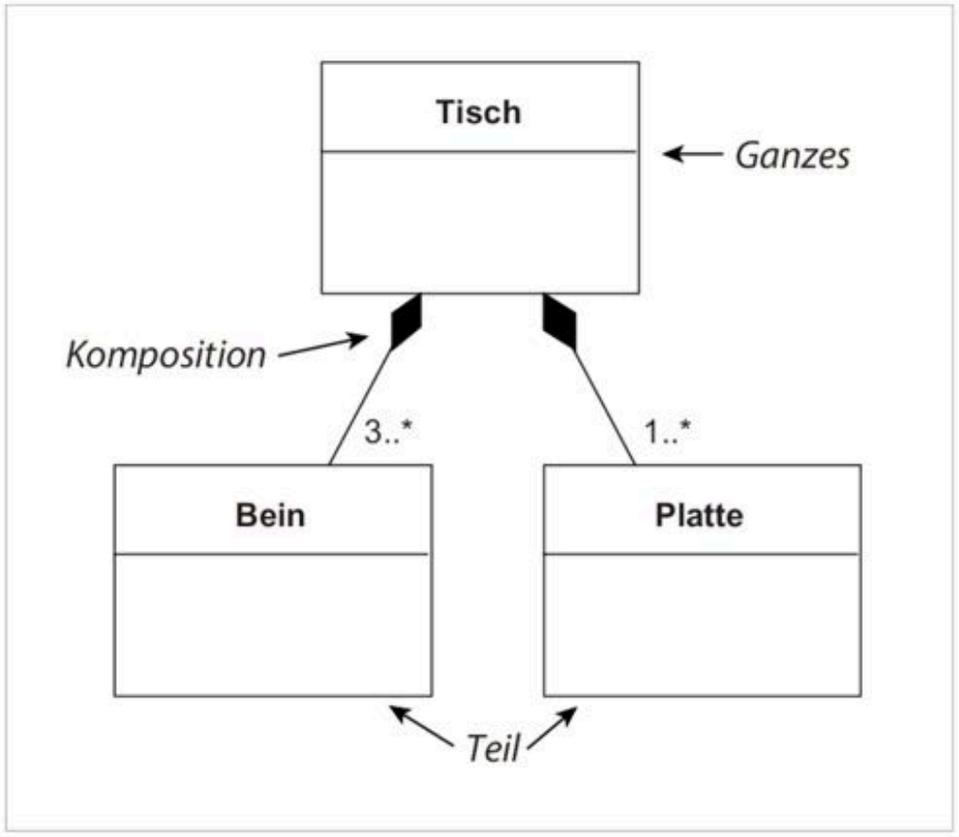


Abbildung 2.29 Komposition

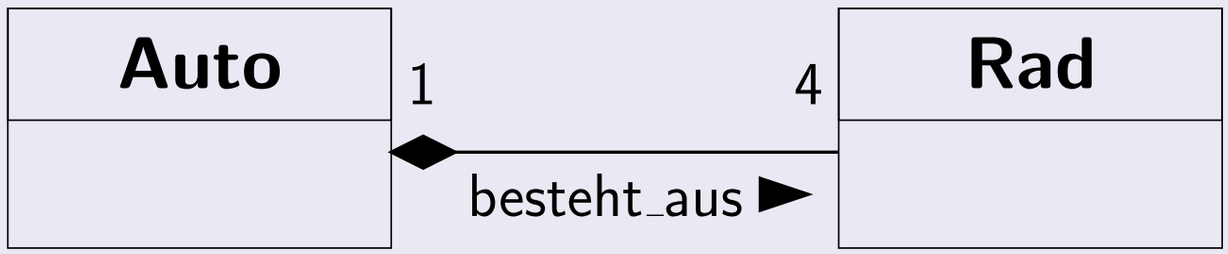
Komposition = “starke” Form der Aggregation

— Beine und Platte werden zerstört, sobald der Tisch zerstört wird.
— Der Tisch ist verantwortlich für die Erstellung der Beine und der Platte.

Klassen- und Objektdiagramme

Beispiel für eine Komposition

Ein Auto besteht aus vier Rädern.



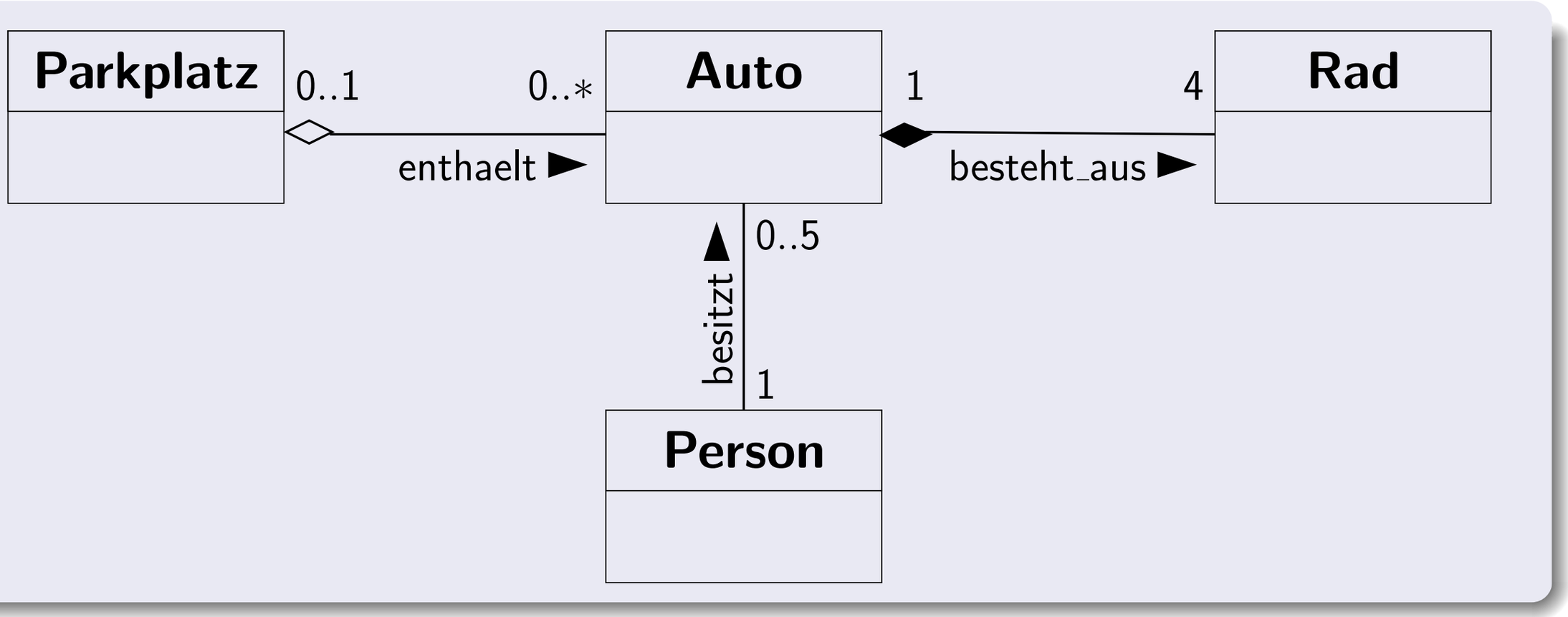
Die Räder werden zerstört, sobald das Auto zerstört wird.

Bemerkung: In diesem Fall muss die Multiplizität, die an der schwarzen Raute steht, immer 0 oder 1 sein. Jedes Teil kann höchstens zu einem Ganzen gehören.

Klassen- und Objektdiagramme

In Klassendiagrammen befinden sich normalerweise nicht nur zwei Klassen mit einer Assoziationen, sondern verschiedene Klassen eines Programms oder Moduls, mit ihren Beziehungen untereinander.

Beispiel:



Generalisierung / Spezialisierung

Generalisierung / Spezialisierung

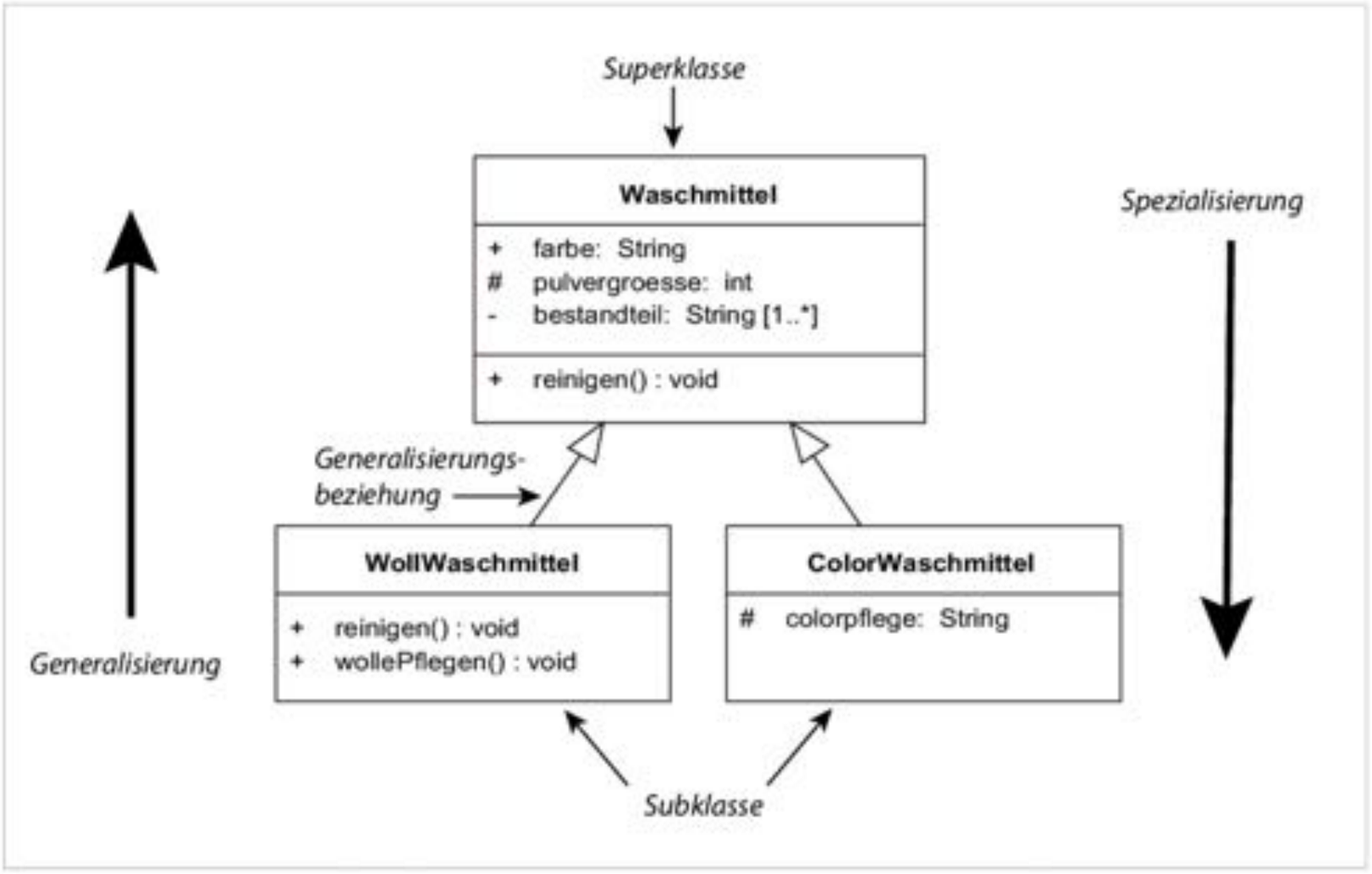
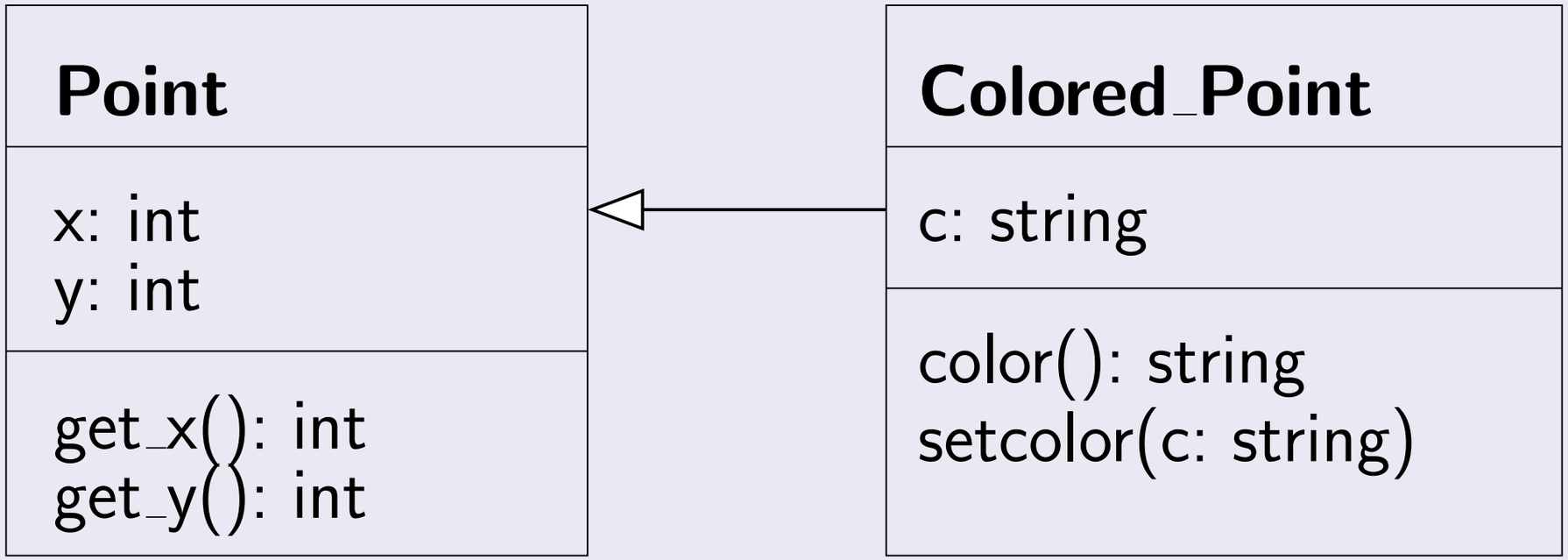


Abbildung 2.33 Generalisierung

Graphische Darstellung von Generalisierung/Spezialisierung



Die Klasse **Colored_Point** spezialisiert die Klasse **Point**.
Umgekehrt generalisiert **Point** die Klasse **Colored_Point**.

Klassen- und Objektdiagramme

Bemerkungen:

- In einer solchen Situation nennt man **Point** Superklasse und **Colored_Point** Subklasse.
- Da die Subklasse die Attribute, Methoden und Assoziationen der Superklasse erbt (und diesen noch weitere hinzufügen kann), spricht man auch von Vererbung. Außerdem kann man in der Subklasse Methoden der Superklasse überschreiben und durch neue ersetzen.

Generalisierung / Spezialisierung

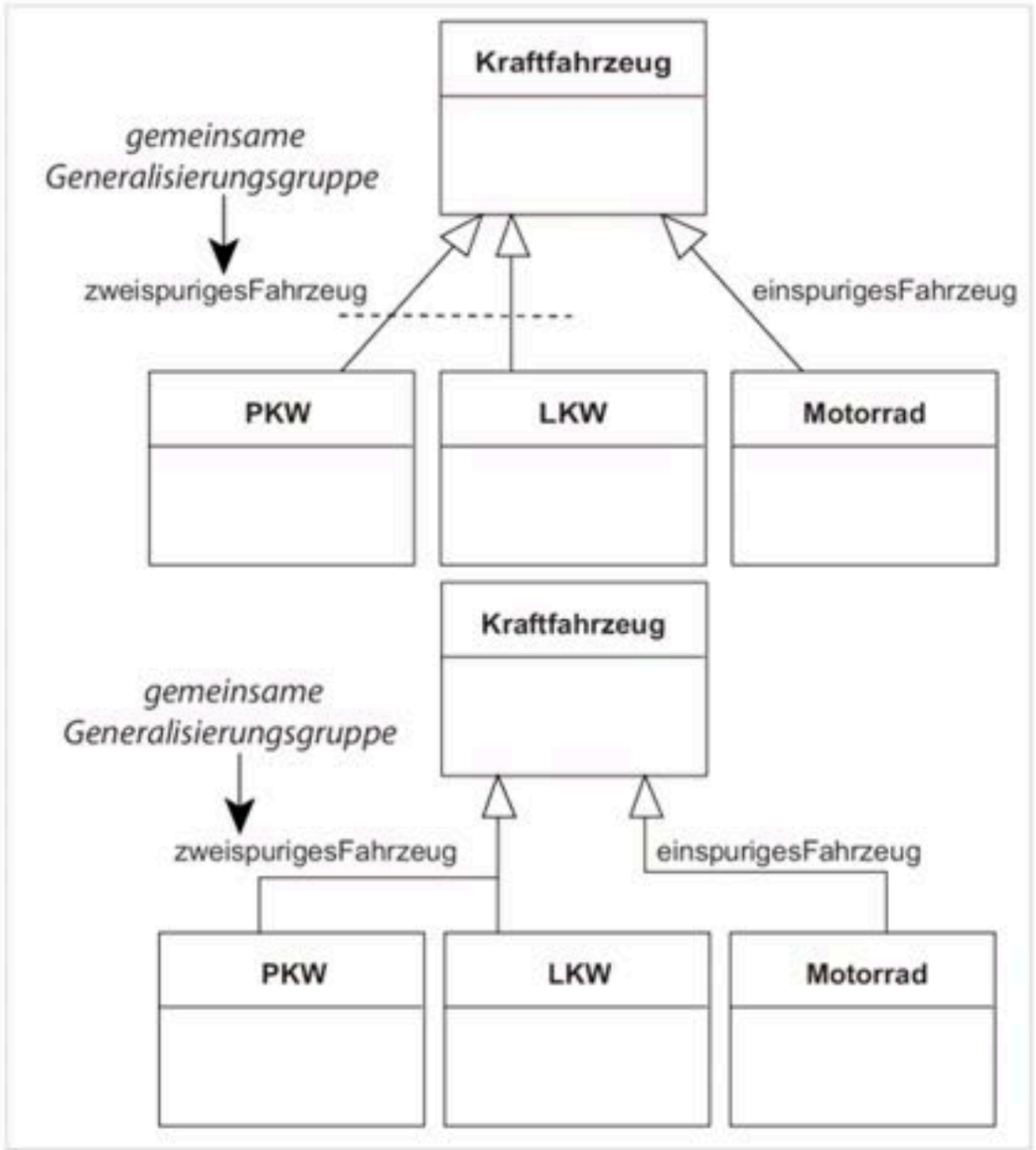
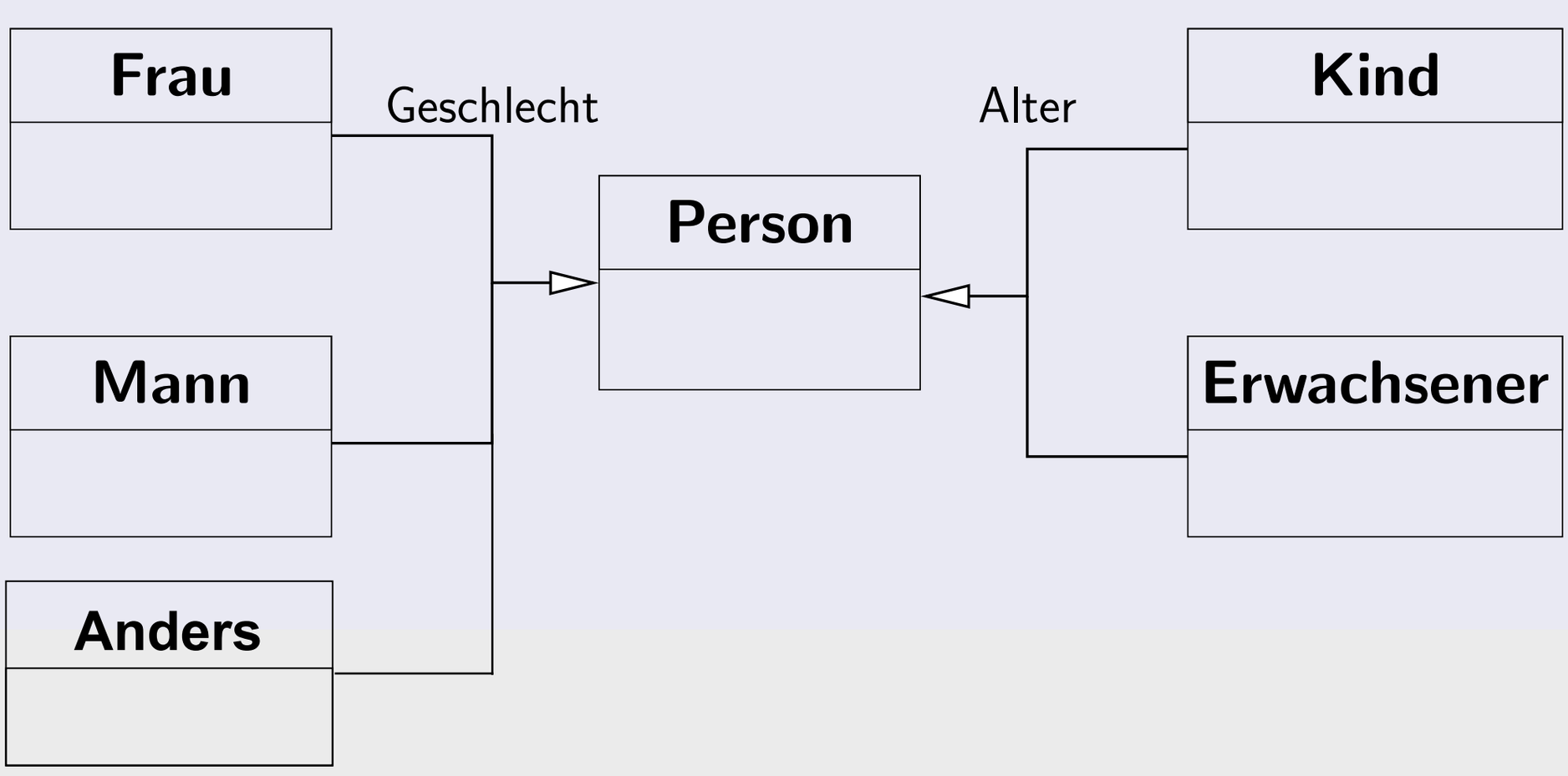


Abbildung 2.35 Gemeinsame Generalisierungsgruppen

Klassen- und Objektdiagramme

Generalisierungsgruppen

Klassen können in unterschiedlicher Weise spezialisiert bzw. unterteilt werden. Daher können Generalisierungen zu Gruppen zusammengefasst werden.



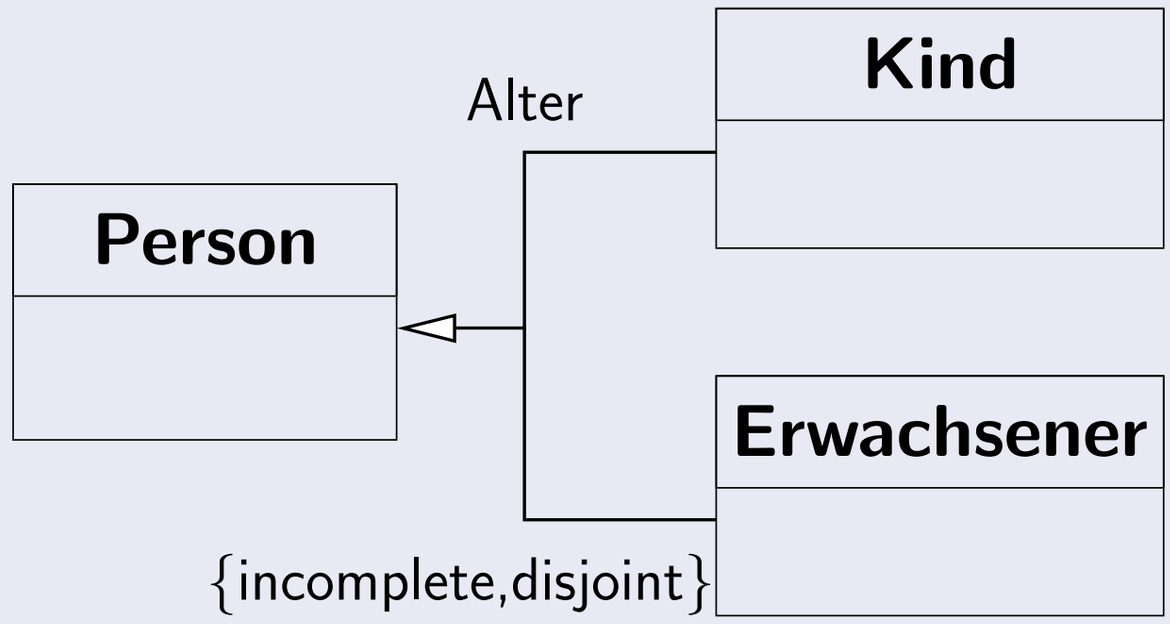
Klassen- und Objektdiagramme

Den Generalisierungsgruppen können Eigenschaften (in geschweiften Klammern) zugeordnet werden.

- complete/incomplete:
  - complete: die Generalisierungsgruppe ist vollständig, d.h., sie überdeckt alle Instanzen der Klasse.
  - incomplete: die Generalisierungsgruppe ist unvollständig.
- disjoint/overlapping:
  - disjoint: die spezialisierenden Klassen besitzen keine gemeinsamen Instanzen (keine Überlappung).
  - overlapping: die spezialisierenden Klassen können gemeinsame Instanzen besitzen.

Klassen- und Objektdiagramme

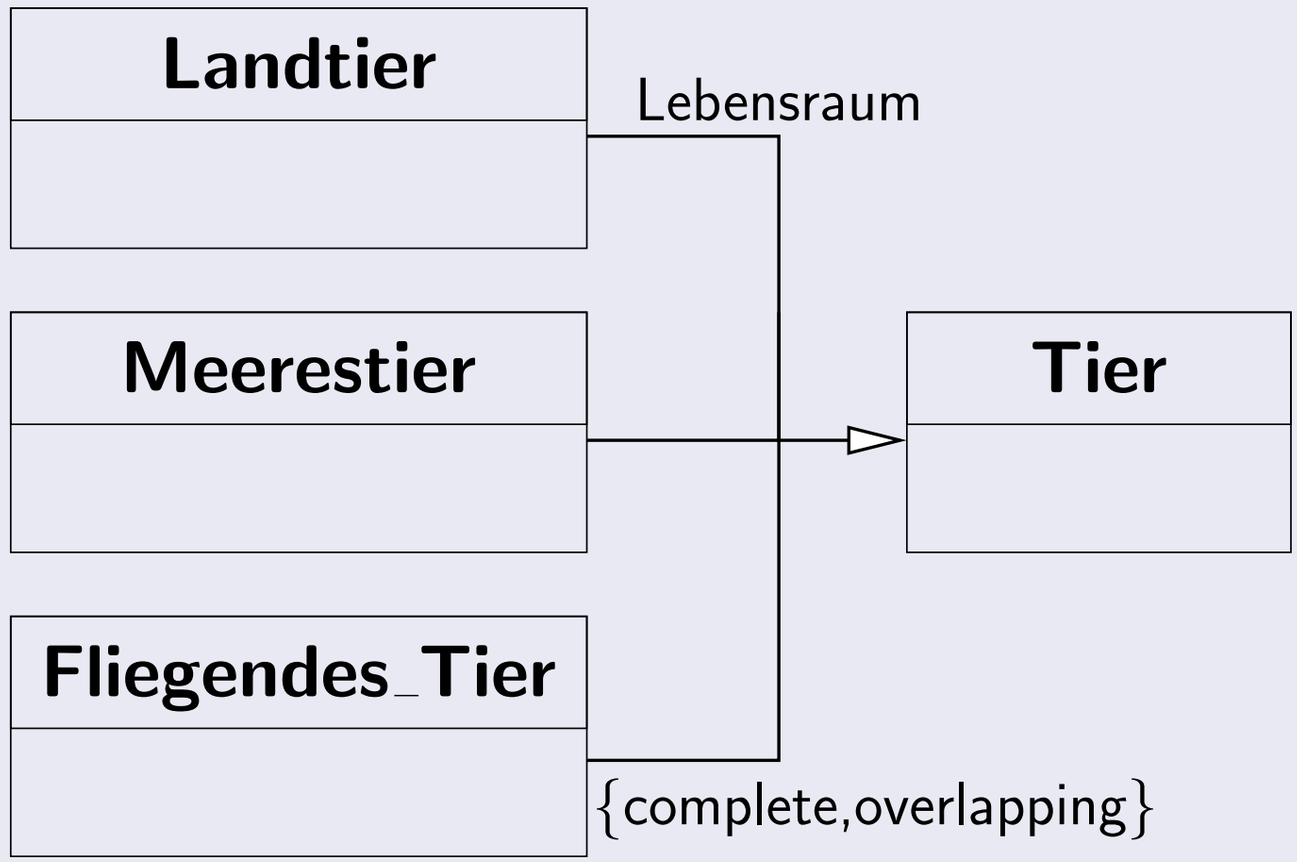
{incomplete,disjoint}



Warum unvollständig? $\rightsquigarrow$ es fehlt eine Klasse Jugendlicher.

Klassen- und Objektdiagramme

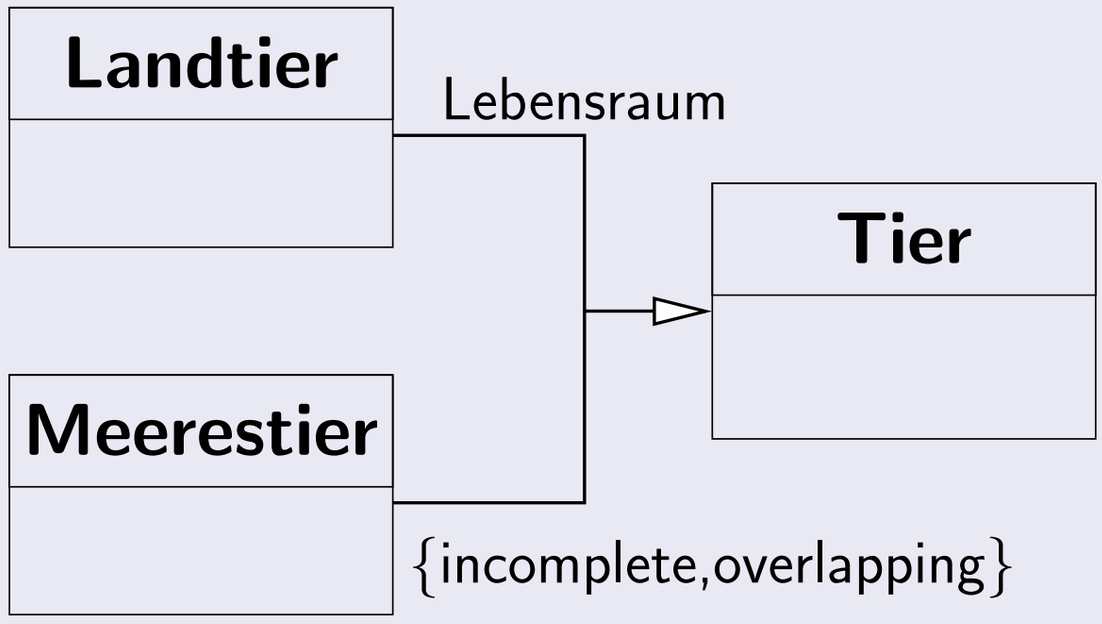
{complete,overlapping}



Schildkröten sind sowohl Land- als auch Meerestiere.

Klassen- und Objektdiagramme

{incomplete,overlapping}

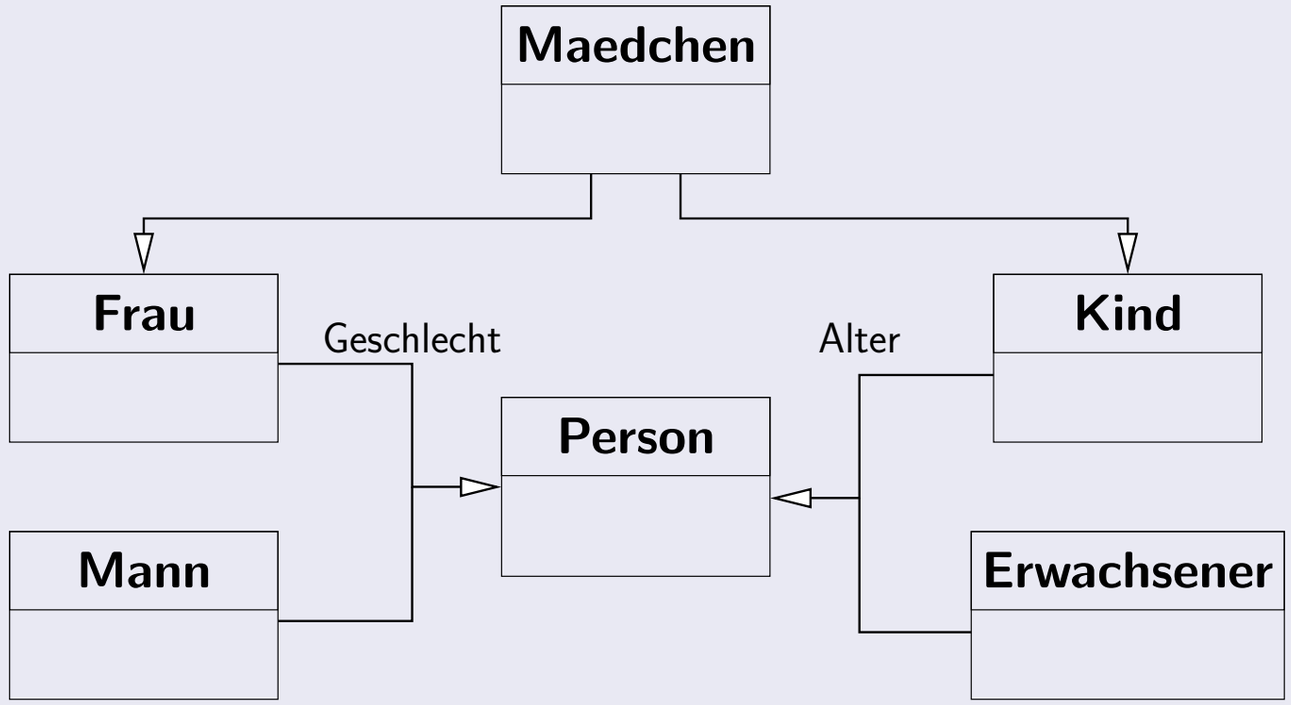


Fliegende Tiere fehlen.

Klassen- und Objektdiagramme

Mehrfachvererbung

Es ist auch möglich, dass eine Klasse von mehreren Klassen erbt, d.h., eine Spezialisierung verschiedener Klassen ist. Dies bezeichnet man als Mehrfachvererbung.



Ein Mädchen ist sowohl ein Kind als auch ein weiblicher Mensch (= Frau).

Klassen- und Objektdiagramme

Basierend auf diesen graphischen Notation sollte man ein objekt-orientiertes System modellieren, bevor es implementiert wird. Dabei stellen sich insbesondere folgende Fragen:

- Welche Objekte und Klassen werden benötigt?
- Welche Merkmale haben diese Klassen und welche Beziehungen bestehen zwischen Ihnen?
- Wie sollen die Klassen eingesetzt werden?
- Welche Methoden stellen diese Klassen zur Verfügung? Wie wirken diese Methoden zusammen?
- In welchen Zuständen können sich Objekte befinden und welche Nachrichten werden wann an andere Objekte geschickt?

Beispiel: Modellierung einer Bank

Ein größeres Beispiel für objekt-orientierte Modellierung und Programmierung: Wir modellieren eine Bank.

Folgende Anforderungen werden gestellt:

- Eine Bank
  - hat mehrere Kunden
  - und mehrere Angestellte
  - und führt eine Menge von Konten.
- Konten können
  - Giro- oder Sparkonten sein. (Ein Sparkonto wirft höhere Zinsen ab, darf aber nicht ins Minus absinken.)
  - in Euro oder in Dollar geführt werden.

Beispiel: Modellierung einer Bank

- Auf den Konten sollen folgende Operationen ausgeführt werden können:
  - Einzahlen
  - Abheben
  - Verzinsen
  - Umbuchen
- Außerdem sollen alle Objekte (inklusive der Bank) ihre Darstellung ausdrucken können. Dazu hat jedes Objekt eine eigene print-Methode.

  Das bezeichnet man auch als Overloading: eine Methode gleichen Namens kann bei Objekten verschiedener Klassen aufgerufen werden und erzielt dort unterschiedliche Effekte.

Beispiel: Modellierung einer Bank

Welche Klassen gibt es?

Beispiel: Modellierung einer Bank

Welche Klassen gibt es?

— Bank

Beispiel: Modellierung einer Bank

Welche Klassen gibt es?

- Bank
- Konto

Beispiel: Modellierung einer Bank

Welche Klassen gibt es?

- Bank
- Konto

gibt es Unterklassen?

Beispiel: Modellierung einer Bank

Welche Klassen gibt es?

- Bank
- Konto
  - Girokonto
  - Sparkonto

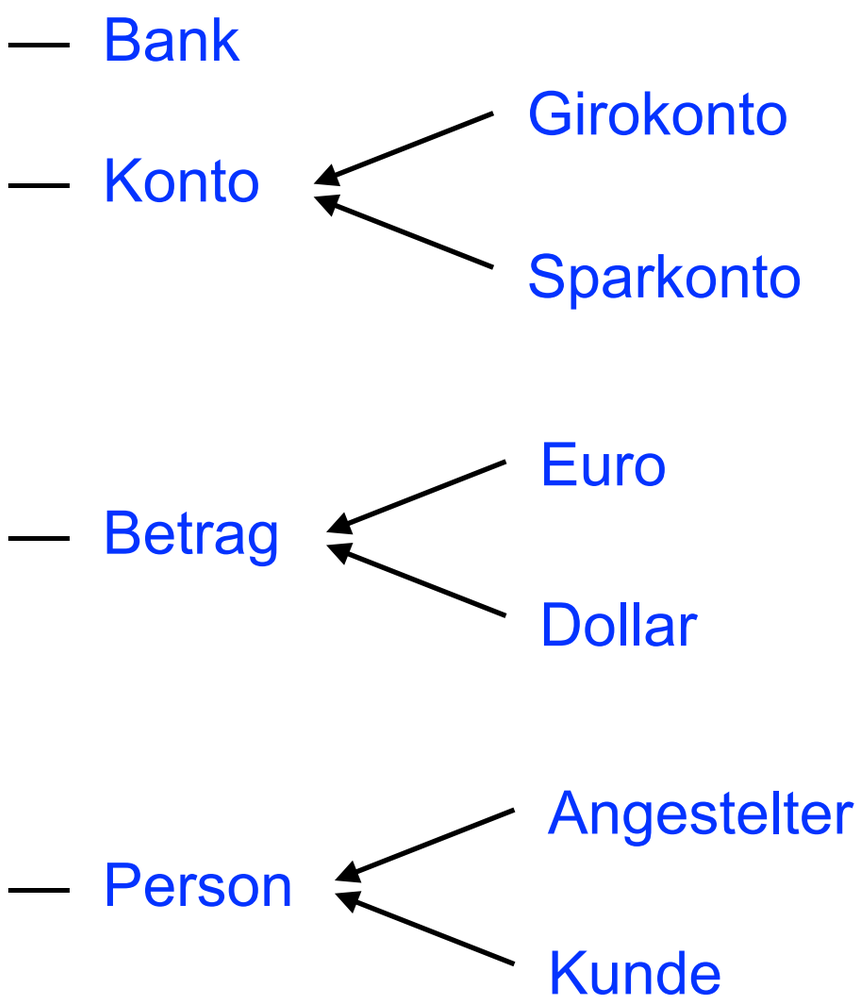
Beispiel: Modellierung einer Bank

Welche Klassen gibt es?

- Bank
- Konto
  - Girokonto
  - Sparkonto
- Betrag
  - Euro
  - Dollar

Beispiel: Modellierung einer Bank

Welche Klassen gibt es?



Beispiel: Modellierung einer Bank

Klasse Bank:

Bank
name: string
neuen_kunden_anlegen() angestellten_einstellen() konten_verzinsen() print()

Methoden: neuen Kunden anlegen; neuen Angestellten einstellen; alle Konten verzinsen

Beispiel: Modellierung einer Bank

Klasse Bank:

Bank
name: string
neuen_kunden_anlegen() angestellten_einstellen() konten_verzinsen() print()

Methoden: neuen Kunden anlegen; neuen Angestellten einstellen; alle Konten verzinsen

Außerdem: Eine Bank besteht (in Kompositionsrelation) aus einer Menge von Konten, die verschwinden, wenn die Bank verschwindet. Außerdem gibt es Aggregationen mit einer Liste von Kunden und von Angestellten.

Beispiel: Modellierung einer Bank

Klasse Konto:

Konto
nummer: string zins: float kontostand: Betrag
einzahlen(wert: Betrag) abheben(wert: Betrag) umbuchen_auf(kto: konto, wert: Betrag) print() verzinsen()

Attribute: Kontonummer, Zins

Methoden: Kontostand abfragen, einzahlen, abheben, umbuchen eines Betrags auf ein anderes Konto, Konto verzinsen

Beispiel: Modellierung einer Bank

Klasse Konto:

Konto
nummer: string zins: float kontostand: Betrag
einzahlen(wert: Betrag) abheben(wert: Betrag) umbuchen_auf(kto: konto, wert: Betrag) print() verzinsen()

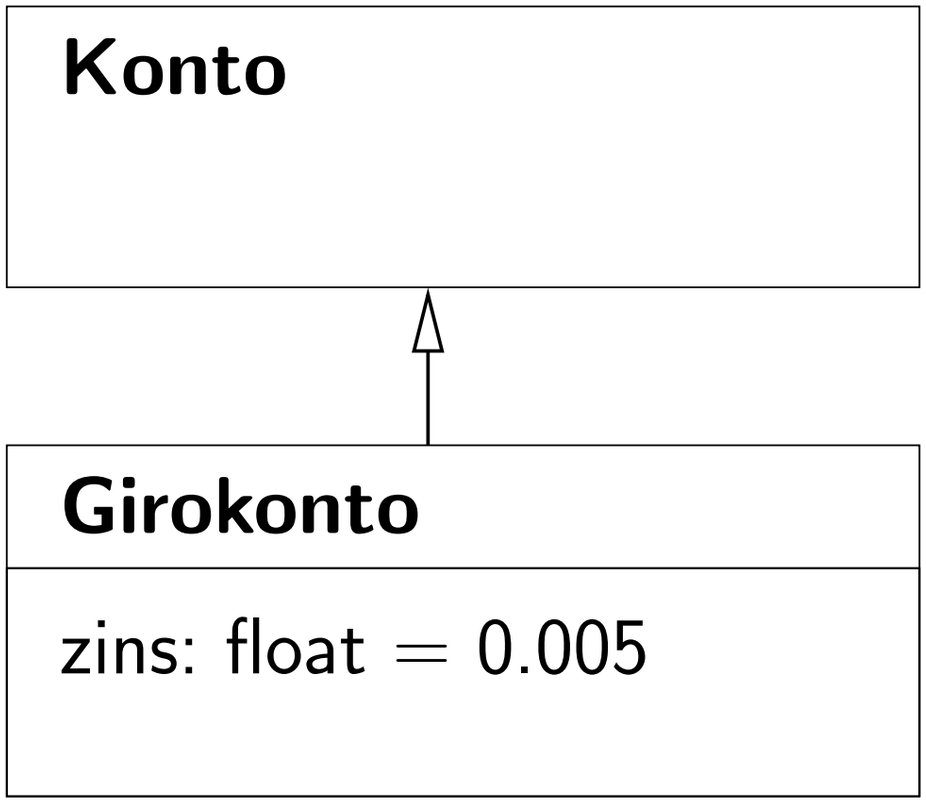
Attribute: Kontonummer, Zins

Methoden: Kontostand abfragen, einzahlen, abheben, umbuchen eines Betrags auf ein anderes Konto, Konto verzinsen

Außerdem: Konto steht in einer Kompositionsrelation mit Betrag, d.h., jedes Konto enthält einen Betrag (siehe Klassendiagramm).

Beispiel: Modellierung einer Bank

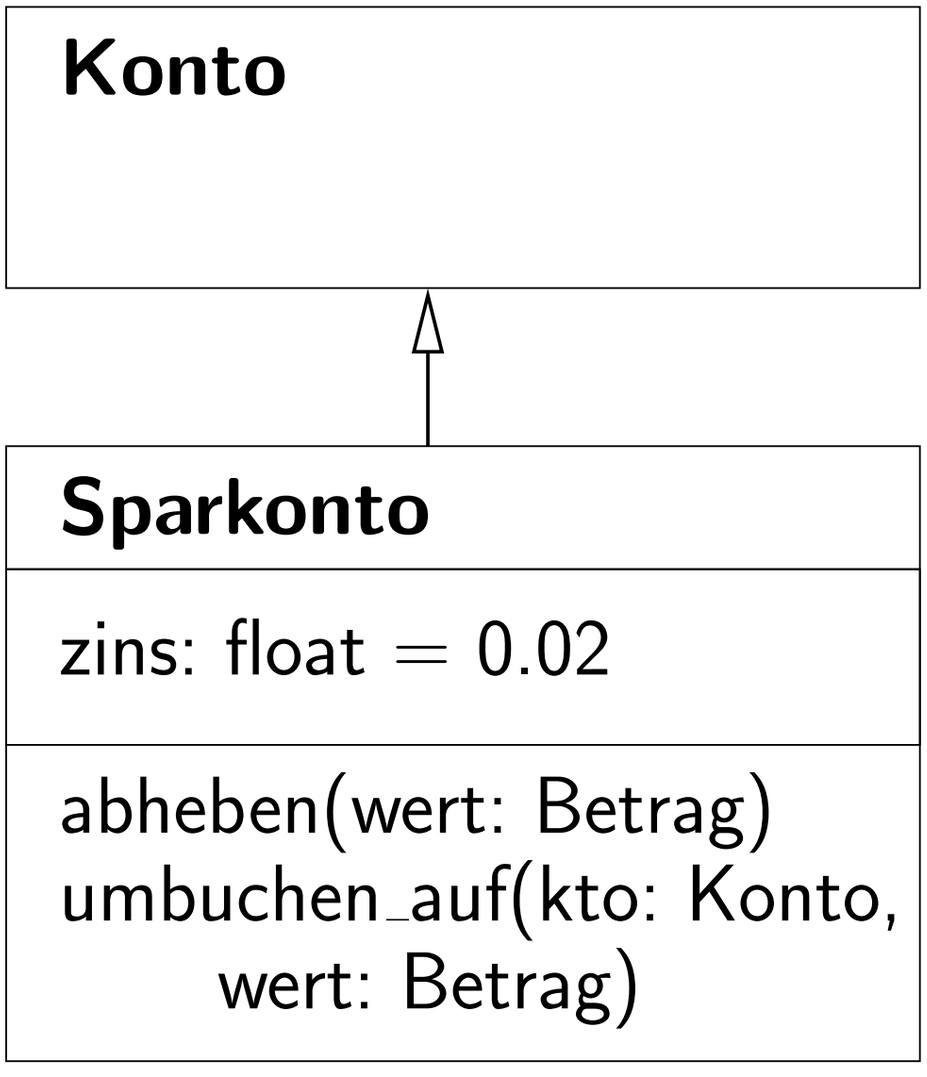
Klasse Girokonto:



Die Klasse Girokonto wird von Konto abgeleitet. Typischerweise ist der Zins bei Girokonten niedriger als bei Sparkonten.

Beispiel: Modellierung einer Bank

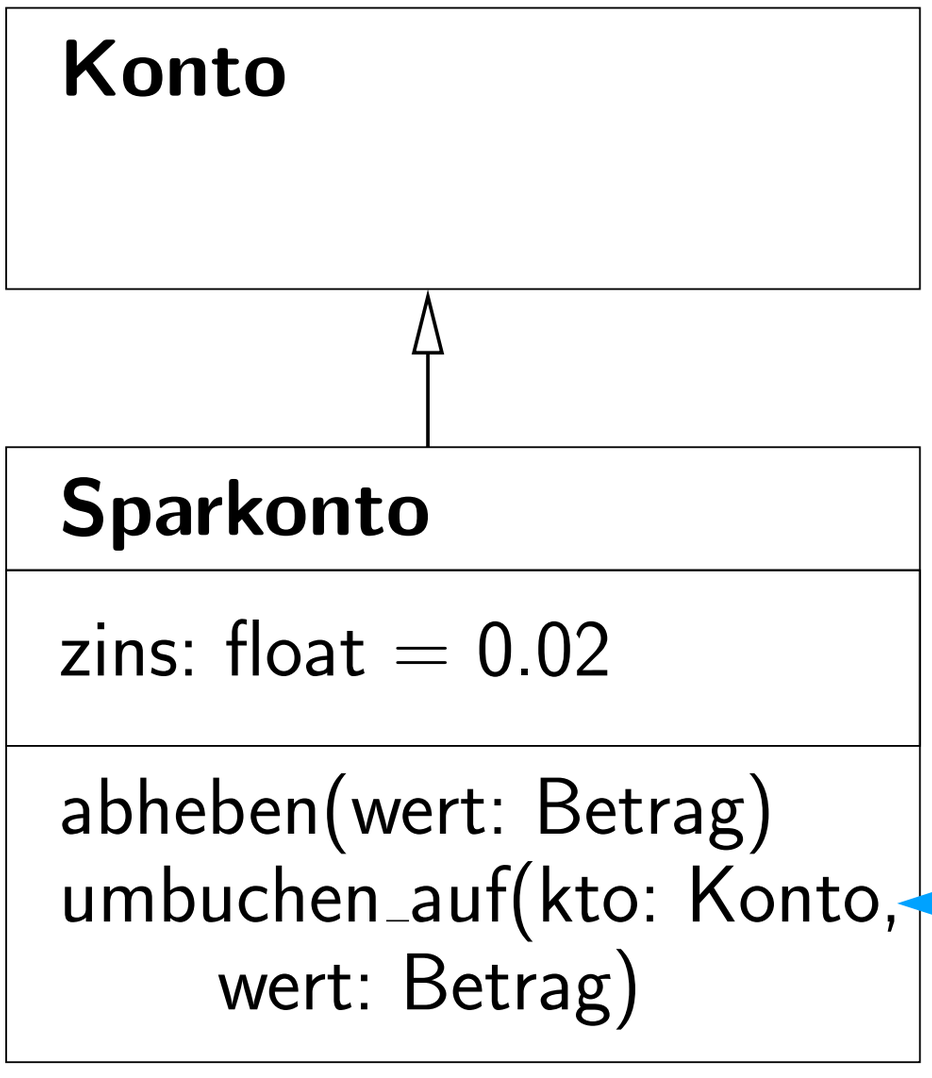
Klasse Sparkonto:



Bei der Klasse Sparkonto muss – wie beim Girokonto – ein neuer Anfangswert für den Zins gesetzt werden.

Beispiel: Modellierung einer Bank

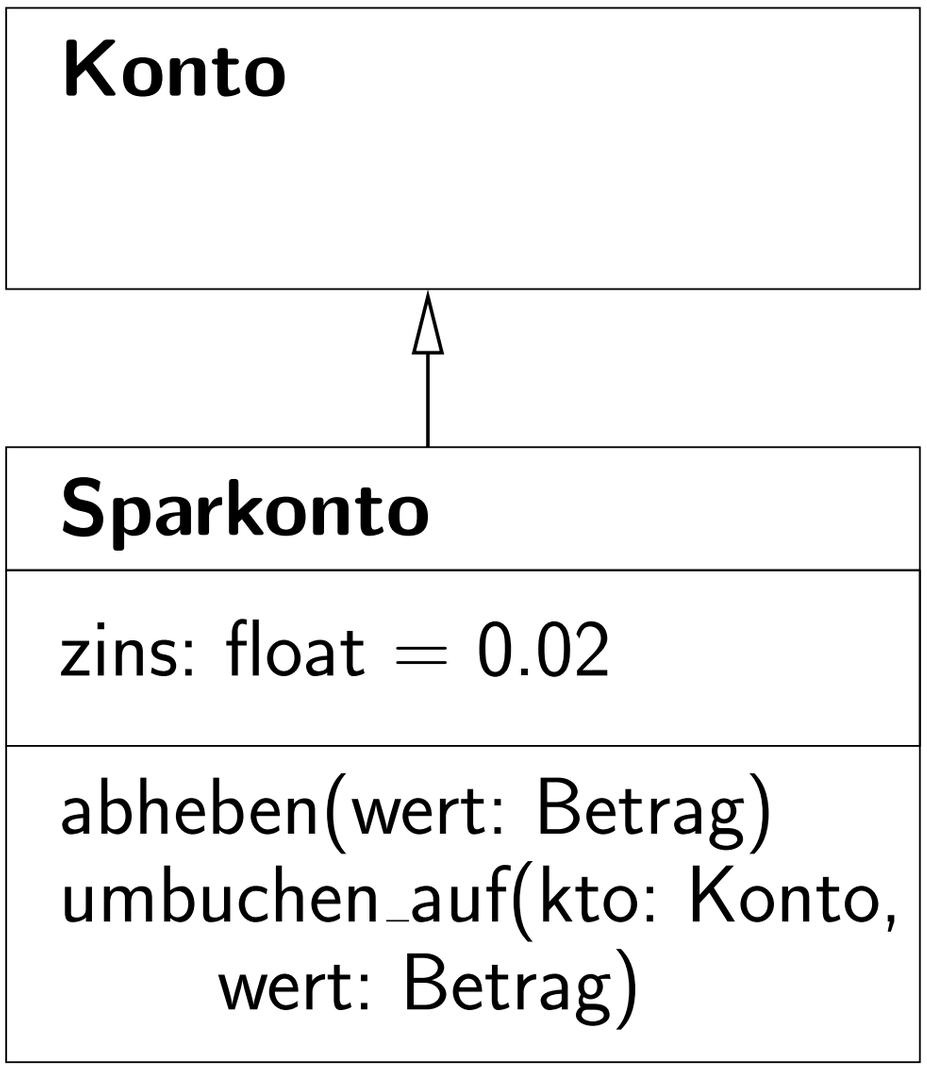
Klasse Sparkonto:



Bei der Klasse Sparkonto muss – wie beim Girokonto – ein neuer Anfangswert für den Zins gesetzt werden.

Beispiel: Modellierung einer Bank

Klasse Sparkonto:



Bei der Klasse Sparkonto muss – wie beim Girokonto – ein neuer Anfangswert für den Zins gesetzt werden.

Außerdem: es muss darauf geachtet werden, dass das Konto nicht ins Minus abgleitet. Dazu werden die entsprechenden Methoden überschrieben (in der Implementierung muss die Bedingung entsprechend getestet werden).

Beispiel: Modellierung einer Bank

Klasse Betrag:

Betrag
wert: float
negativ(): bool plus(wert: Betrag) minus(wert: Betrag) print() mult(faktor: float)

Methoden: Test, ob Konto im Minus; Betrag addieren, subtrahieren; Multiplikation mit einer Gleitpunktzahl (zum Verzinsen!)

Beträge müssen im allgemeinen in einer Währung angegeben werden. Daher werden von der Klasse Betrag die Unterklassen Euro und Dollar abgeleitet.

Beispiel: Modellierung einer Bank

Klasse Betrag:

Betrag
wert: float
negativ(): bool plus(wert: Betrag) minus(wert: Betrag) print() mult(faktor: float)

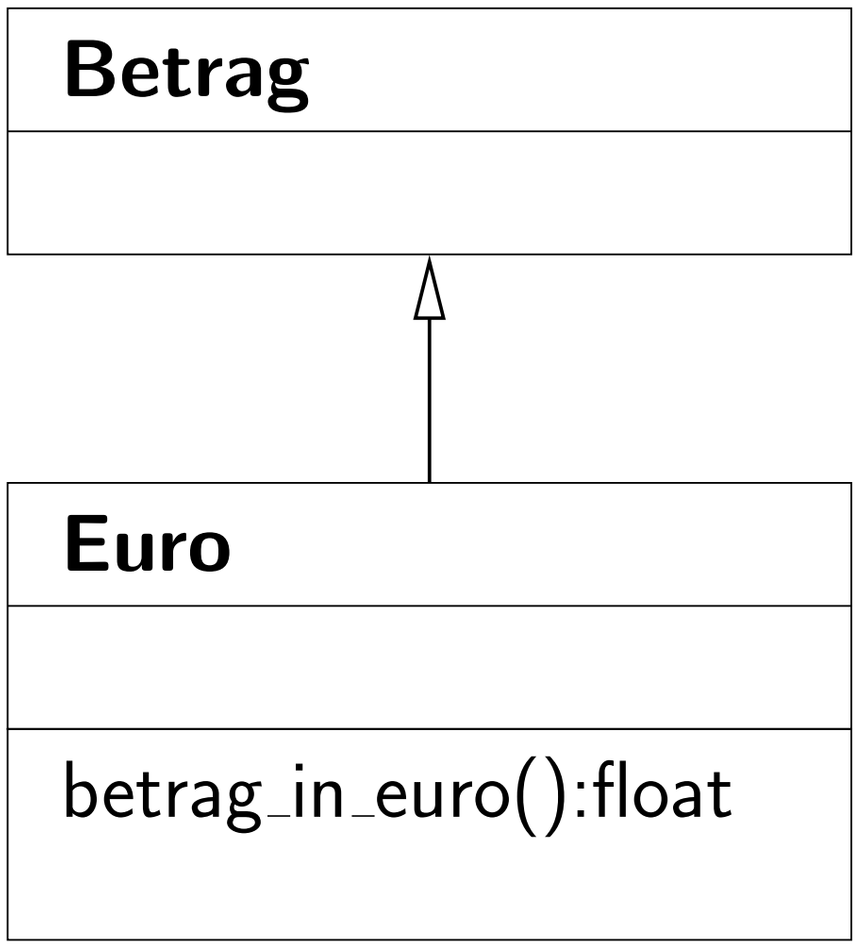
Alternative:
Unterklasse `nichtNegativerBetrag`
Wird von Klasse `Sparkonto` benutzt.

Methoden: Test, ob Konto im Minus; Betrag addieren, subtrahieren; Multiplikation mit einer Gleitpunktzahl (zum Verzinsen!)

Beträge müssen im allgemeinen in einer Währung angegeben werden. Daher werden von der Klasse Betrag die Unterklassen Euro und Dollar abgeleitet.

Beispiel: Modellierung einer Bank

Klasse Euro:

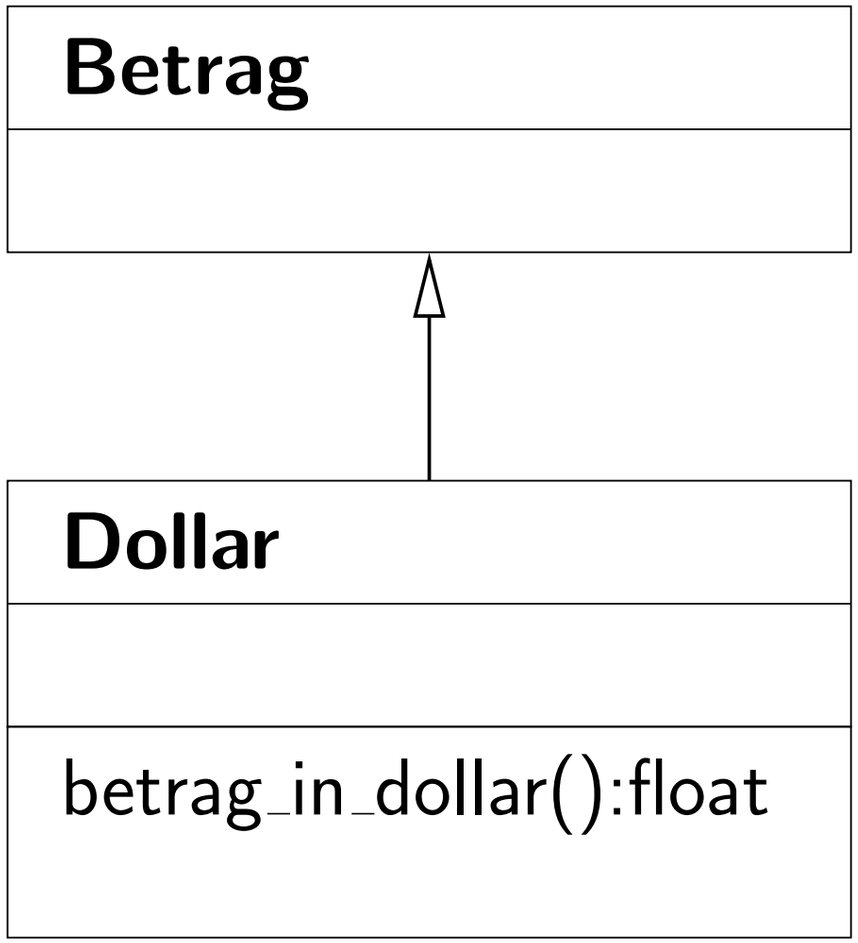


Von Betrag wird zunächst die Klasse Euro abgeleitet.

Methoden: Betrag in Euro ausgeben

Beispiel: Modellierung einer Bank

Klasse Dollar:

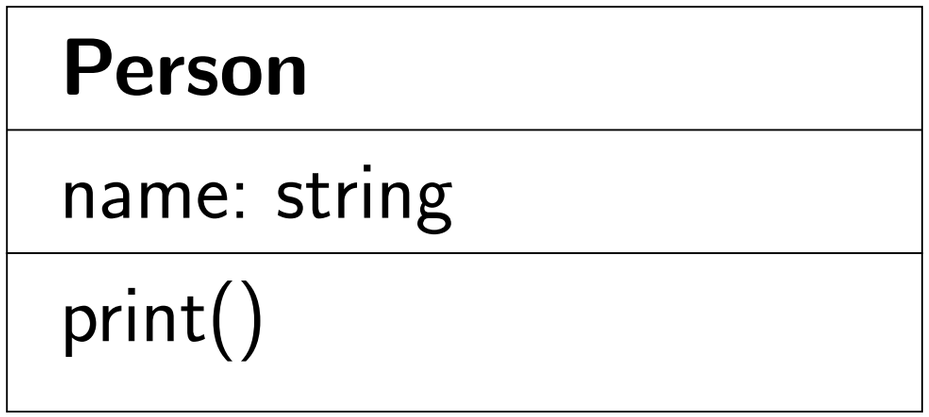


Von Betrag wird außerdem die Klasse Dollar abgeleitet.

Methoden: Betrag in Dollar ausgeben

Beispiel: Modellierung einer Bank

Klasse Person:



Methoden: nur die print-Methode, weitere Methoden werden in den Unterklassen definiert

Außerdem: Person steht in einer Assoziationsrelation mit einer Liste von Konten, die entweder dieser Person gehören (Kunde) oder auf die diese Person Zugriff hat (Angestellte/r).

Beispiel: Modellierung einer Bank

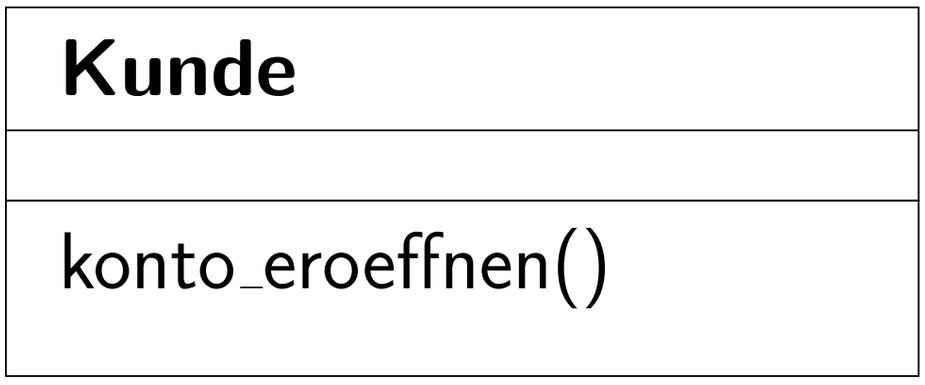
Klasse Angestellter (erbt von Person):

Angestellter
zugriff_auf_konto(kto: Konto)

Methoden: Zugriff auf ein Konto erlangen

Beispiel: Modellierung einer Bank

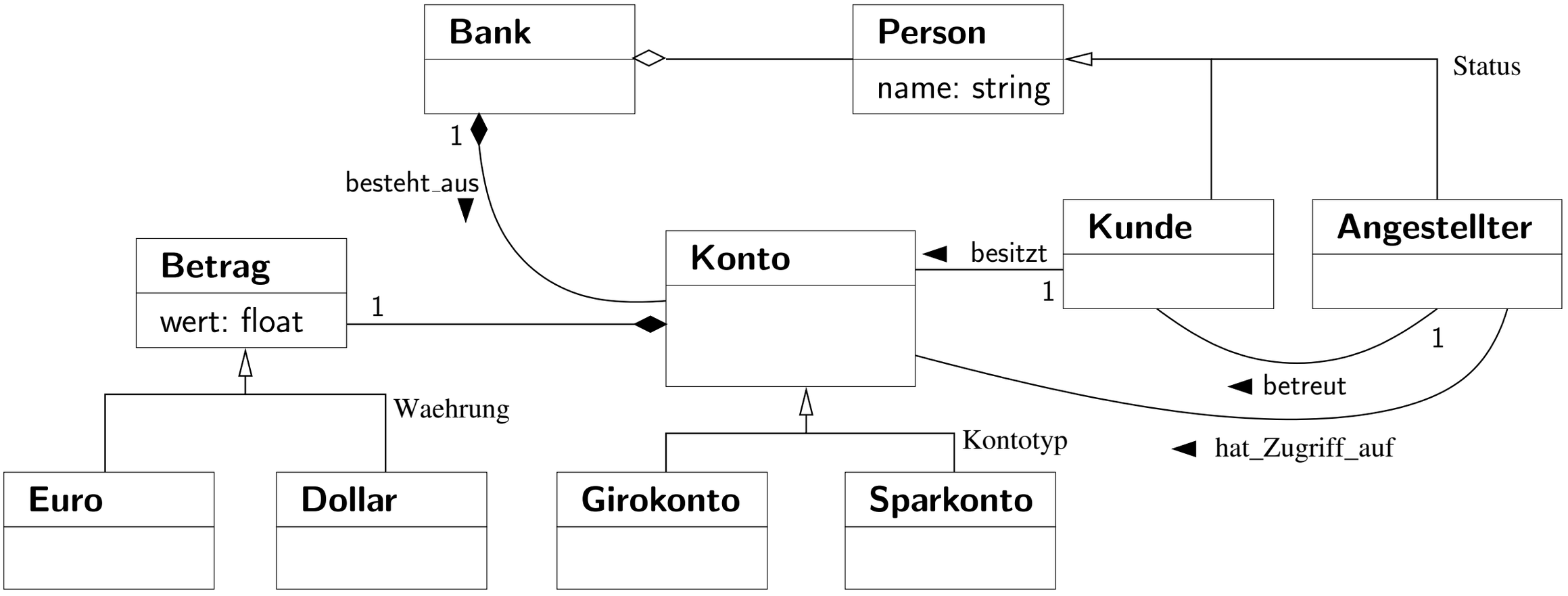
Klasse Kunde (erbt von Person):



Methoden: Konto eröffnen (hier könnte noch ein Parameter übergeben werden, der beschreibt, ob das Konto ein Giro- oder Sparkonto sein soll und ob es in Euro oder Dollar geführt werden soll)

Außerdem: Ein Kunde steht in einer Assoziationsrelation mit seinem Betreuer.

Beispiel: Modellierung einer Bank



Klassen- und Objektdiagramme

Abstrakte Klasse

Eine Klasse heißt abstrakt, wenn sie selbst keine Instanzen haben kann. Dazu wird die Eigenschaft {abstract} unter dem Klassennamen angegeben. Manchmal wird stattdessen auch der Klassenname *kursiv* geschrieben.

Beispiel: in dem Bank-Beispiel möchte man beispielsweise nie eine Instanz der Klasse Person bilden, sondern nur von Kunde oder Angestellter.

Kurs
Datenbankgrundlagen und Modellierung

Sebast… …rsität Bremen
…remen.de

…mersemester 2023

12.6.2023

…lesung 7: Modellierung & UML: Einführung