Informatik III

Objektorientierte Programmierung mit C++

Klaus Wieland

Dozent

Klaus Wieland

- Dipl.-Ingenieur Technische Informatik - Berufsakademie Stuttgart, 1992

klaus.wieland@lehre.dhbw-stuttgart.de

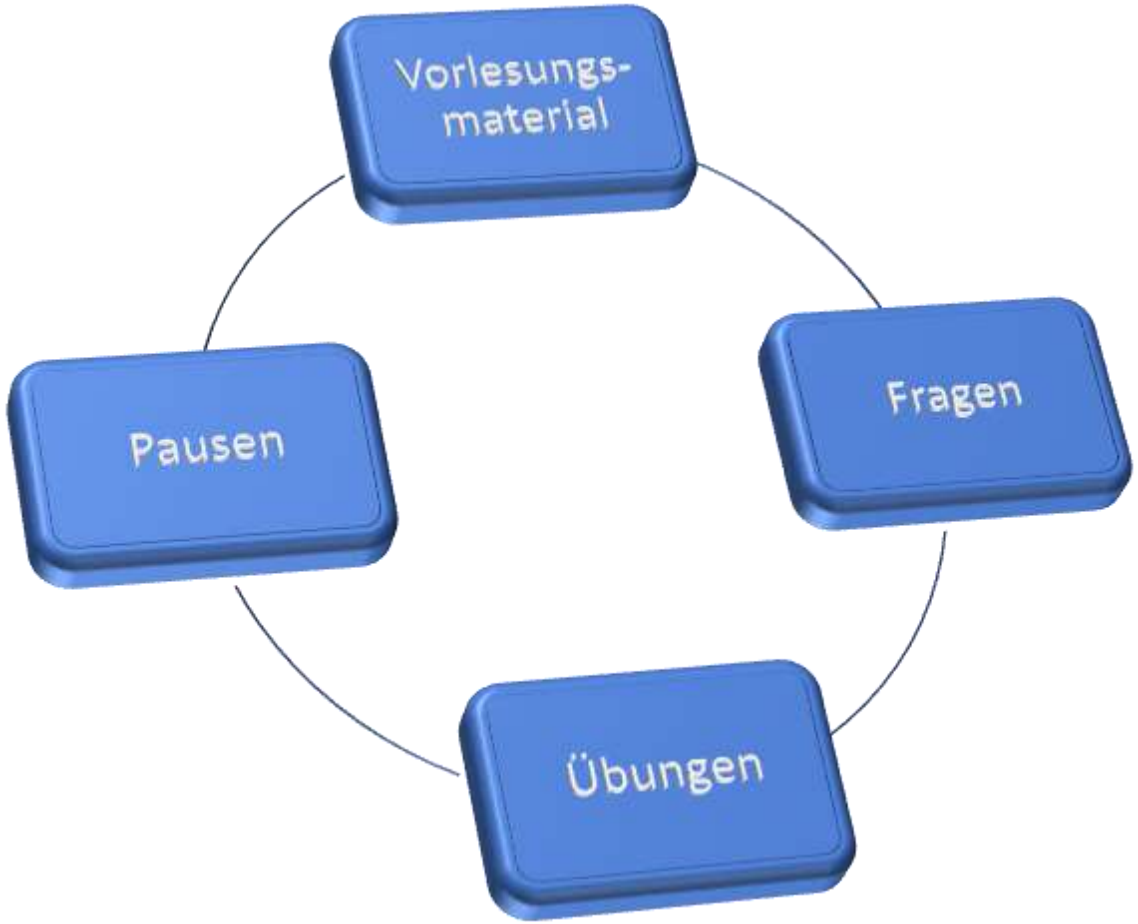
Verschiedene Rollen im Berufsleben

- Software Entwicklung, IT Beratung, Projektleiter, Architekt, Outsourcing Vertrieb, Management

Dozententätigkeit – DHBW Stuttgart/Mannheim

- Theoretische Informatik I - Grundlagen & Logik -
- Theoretische Informatik III - Compilerbau Labor -
- Informatik III - Objektorientierte Programmierung mit C++ -

Administratives



Themen der Vorlesung

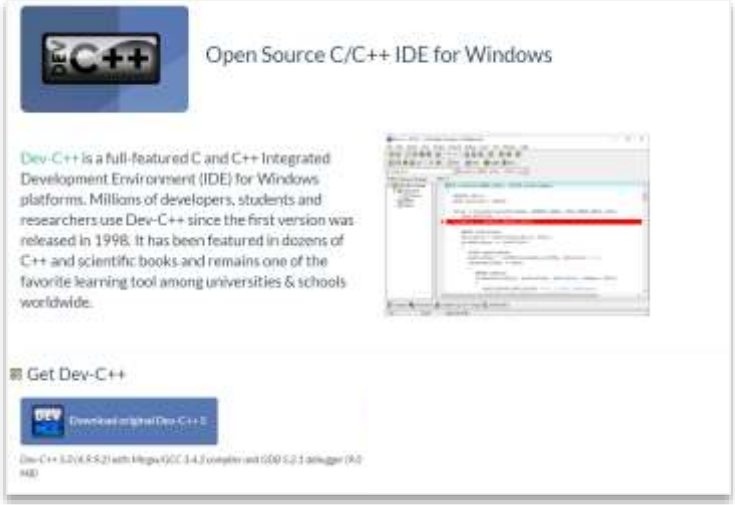
- **Grundlagen der objektorientierten Programmierung**
  - Paradigma und Grundbegriffe (Objekte, Klassen, Vererbung, Polymorphie)
  - UML / Objektorientierter Entwurf
- **Objektorientierte Programmierung in C++**
  - Von C zu C++
  - Die Grundbausteine von C++
  - Klassen, Objekte, Datenkapselung
  - Vererbung
  - Polymorphismus
  - Exception Handling
  - Templates
  - Klassenbibliothek

Wochenübersicht

Woche	Thema	Übung
1		
2	Intro & Admin / OOP Grundlagen / UML	
3	Einführung C++ / Klassen & Objekte	✓
4	Operatoren / Referenzen / Dynamischer Speicher	✓
5	Streams / Vererbung	✓
6	Virtuelle Funktionen / Polymorphie / Mehrfachvererbung	✓
7		
8	Exception Handling / Templates / Klassenbibliothek	✓
9		
10	Wiederholung / Prüfungsvorbereitung	
11		
12	PRÜFUNG	

Anforderung: C++ Compiler

- Dev C++
  - Home - Dev-C++ Official Website (bloodshed.net)
  - Plattform
    - Windows
- Microsoft Visual Studio
  - Kostenlose Entwicklungssoftware und Dienste - Visual Studio
  - Plattform:
    - Windows (**VS Community / VS Code**)
    - Apple (**VS Community für Mac / VS Code**)
    - Linux (**VS Code**)
- GNU C++ Compiler
  - Plattform:
    - Linux
    - Windows (Cygwin)



Literatur

- Bjarne Stroustrup: **Die C++ Programmiersprache**
- Torsten T Will : **C++: Das umfassende Handbuch zu Modern C++**
- Nicolai Josuttis: **Objektorientiertes Programmieren in C++**
- Nicolai Josuttis: **Die C++ Standardbibliothek**
- Scott Meyers: **Effektiv C++ programmieren**

Grundlagen OOP

Was bedeutet „*Objektorientiert*“?

1. *Alles ist ein Objekt*
2. *Objekte kommunizieren durch das Senden und Empfangen von Nachrichten (welche aus Objekten bestehen)*
3. *Objekte haben ihren eigenen Speicher (strukturiert als Objekte)*
4. *Jedes Objekt ist die Instanz einer Klasse (welche ein Objekt sein muss)*
5. *Die Klasse beinhaltet das Verhalten aller ihrer Instanzen (in der Form von Objekten in einer Programmliste)*
6. *Um eine Programmliste auszuführen, wird die Ausführungs-kontrolle dem ersten Objekt gegeben und das Verbleibende als dessen Nachricht behandelt*

Alan Kay: The Early History of Smalltalk (1993)

Im Gegensatz zur prozeduralen Programmierung, bei der Daten, Prozeduren und Funktionen getrennt betrachtet werden, fasst man sie bei der objektorientierten Programmierung zu einem Objekt zusammen. Objekte sind nicht nur passive Strukturen, sondern aktive Elemente, die durch Nachrichten anderer Objekte aktiviert werden. Objektorientierte Programme werden als kooperierende Sammlungen von Objekten angesehen.

Prof. Dr. Richard Lackes - Gabler Wirtschaftslexikon

OOP bedeutet für mich nur Messaging, lokales Beibehalten und Schützen und Verbergen des Prozesszustands sowie spätest-mögliche Bindung aller Dinge.

Alan Kay: Antwort auf eine Frage (2003)

Bezieht sich auf eine Technik oder Programmier-sprache, welche Objekte, Klassen und Vererbung unterstützt.

ISO/IEC 2382-15 (1999)

Objektorientierte Programmierung (OOP) ist ein Computerprogrammiermodell, das das Softwaredesign um Daten oder Objekte herum organisiert und nicht um Funktionen und Logik. Ein Objekt kann als Datenfeld mit eindeutigen Attributen und Verhalten definiert werden. OOP konzentriert sich auf die Objekte, mit denen das Programm interagieren soll, und nicht auf die Logik, die zu ihrer Manipulation erforderlich ist.

Nick Lewis - ComputerWeekly

Historie

Anfang der 60er Jahre
- Terminologie „Objekte“ im Sinne OOP am MIT
- Sketchpad Applikation (Objekte als Repräsentation von Oszilloskop Bildern)

1965 - Simula
- Erste Sprache mit Klassen und Instanzen

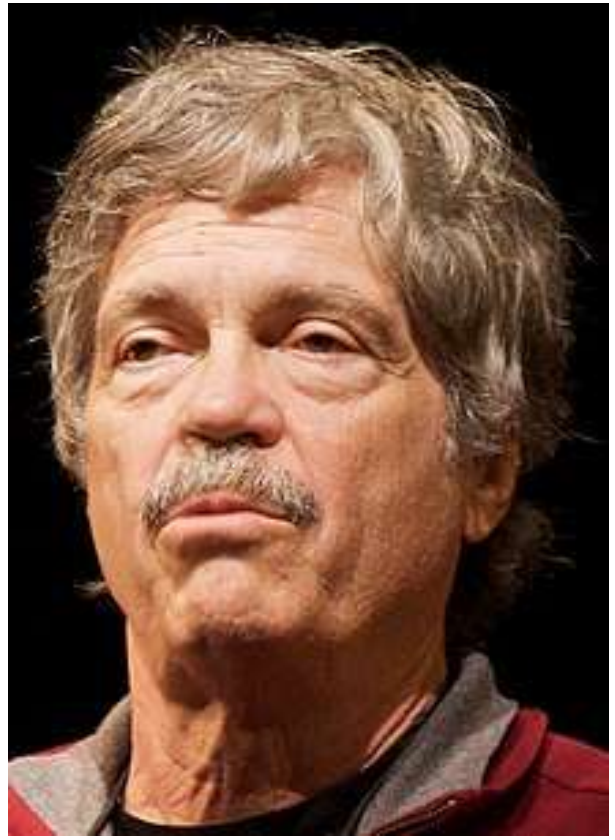
1967 - Begriff *„Object Oriented Programming”*
- Geprägt von Alan Kaye, inspiriert von biologischen Zellen bzw. Einzelrechnern im Netzwerk

1972 - Smalltalk
- Entwickelt von Alan Kaye u.a.
- Interpreter, Graphische Entwicklungsoberfläche

Ab Ende 70er Jahre
- Mehrere neue OOP Sprachen: C++, Eiffel, Oberon, Objective C

Beginn 90er Jahre – zunehmende Beliebtheit
- Auch durch die Beliebtheit von GUIs (Windows)

Heute
- Zusätzliche Sprachen: C#, VB.NET, Python, Ruby, …



Alan Kaye (1940-)

Paradigmenwechsel

Traditionelle (prozedurale) Programmierung

- Aufbrechen komplexer Operationen in kleine Schritte (Prozeduren und Module)
- Definition von Datenstrukturen

Beispiel:

- Für eine Bankapplikation benötigt man die Funktionen `Abheben` und `Einzahlen`. Zusätzlich dann noch die entsprechenden Datenstrukturen.

Objektorientierten Programmierung

- Strukturierung des Systems nach Abstraktionen des Anwendungsbereiches / Realität

Beispiel:

- Für die Bankapplikation benötigt man die Objekte `Kunde`, `Konto`, `Geld`

Eine neue (eigentlich intuitive) Art zu denken und Probleme zu analysieren

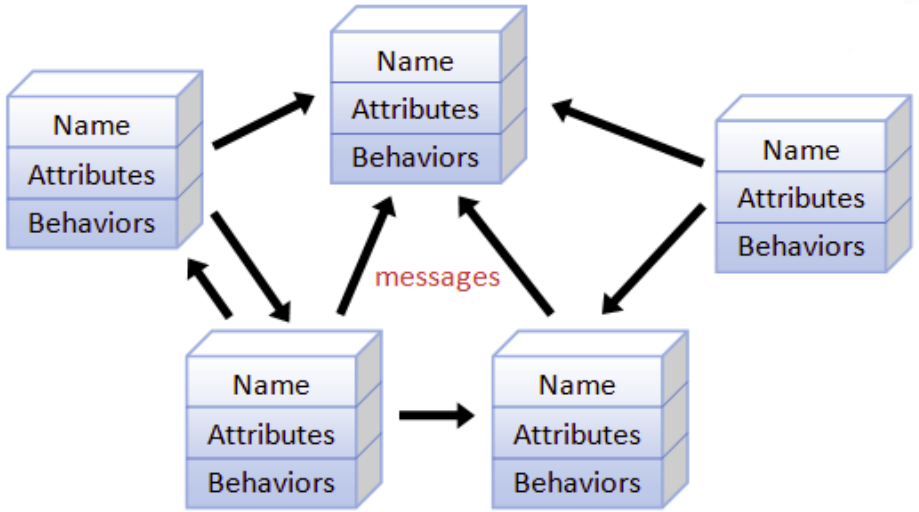
Objekte und ihre Fähigkeiten (I)

- Jedes Objekt hat Fähigkeiten
  - Attribute (Variablen)
  - Operationen (*„Behaviours“*)

- Jedes Objekt stellt Dienste anderen Objekten zur Verfügung
  - Ein Objekt agiert dabei als **Klient**, das andere als **Dienstleister**

- Das Anforderung erfolgt durch **Nachrichten** (*„messages“*)
  - *„Versteht“* das Objekt diese Nachricht, so erfüllt es die ihm anvertraute Aufgabe
  - Jedes Objekt versteht nur einen bestimmten Satz an Nachrichten – soweit eine entsprechende **Operation** vorhanden ist
  - Enthält eine Nachricht zusätzliche Informationen, die der Dienstleister für die Ausführung des gewünschten Dienstes benötigt, spricht man von **parametrisierten Nachrichten**

- Versteht das Objekt eine Nachricht nicht, so liegt ein Programmfehler vor



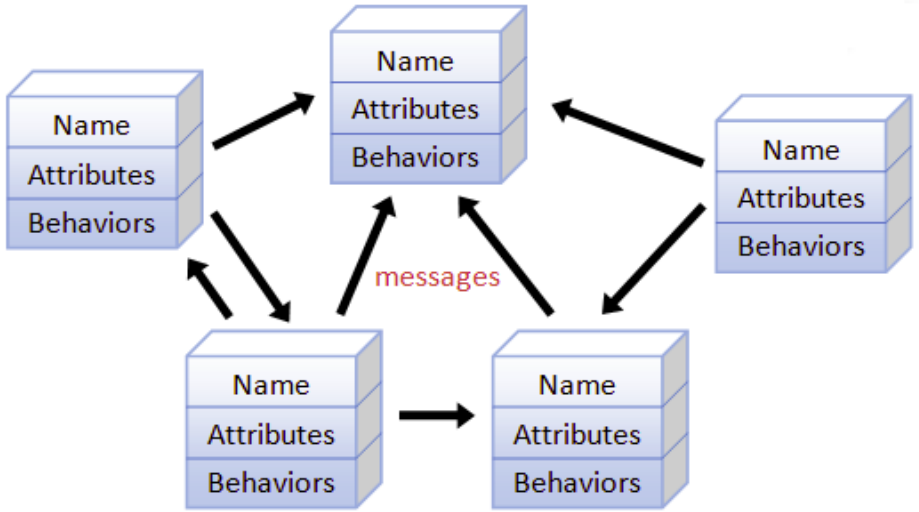
Ein Beispiel aus der realen Welt

Am Ende des Semesters fordert der Dozent einen Studenten auf ein C++ Programm zu schreiben.

- 2 Objekte: `Dozent` und `Student`
  - `Dozent` ist der **Klient**
  - `Student` ist der **Dienstleister**
- **Nachricht** vom Objekt `Dozent` an das Objekt `Student`: *„Schreibe ein C++ Programm“*
- Ende des Semesters = das Objekt `Student` hat die Fähigkeiten und eine entsprechende **Operation**
  - Nachricht wird verstanden und Operation ausgeführt

Objekte und ihre Fähigkeiten (II)

- Objekte kommunizieren nur durch Nachrichten miteinander und kennen nur ihre gegenseitigen Operationen
- Eine Operation ist die Spezifikation des Dienstes, d.h. die Schnittstellenbeschreibung
- Eine Nachricht ist die Anforderung eines Dienstes
- Eine Implementierung ist die Realisierung eines Dienstes
  - Die Implementierung ist nur dem jeweiligen Objekt bekannt (**Datenkapselung** – *„Encapsulations“*)
- Wenn zwei Objekte dieselbe Nachricht verstehen, kann die Implementierung trotzdem unterschiedlich sein
  - Damit auch das Ergebnis der Operation und das Verhalten des Objekts



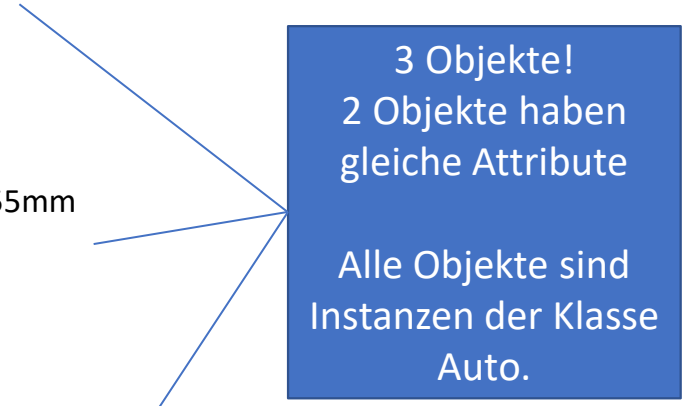
Klassen (I)

- Eine **Klasse** ist eine Schablone, mit der gleichartige Objekte beschrieben werden
  - Auch: Konstruktionsplan zur Erzeugung von Objekten
- Ein Objekt ist die Ausprägung einer Klasse (eine **Instanz** einer Klasse)
  - Von jeder Klasse können mehrere Objekte gleichzeitig existieren
  - Jedes Objekt gehört zu einer spezifischen Klasse
- Die Klasse verwaltet die Eigenschaften und Zustände ihrer Objekte
  - Attribute
  - Operationen/Implementierungen (Schnittstelle)
- Attribute können bei jedem Objekt andere Werte haben

Beispiel: Klasse `Auto` mit den Attributen

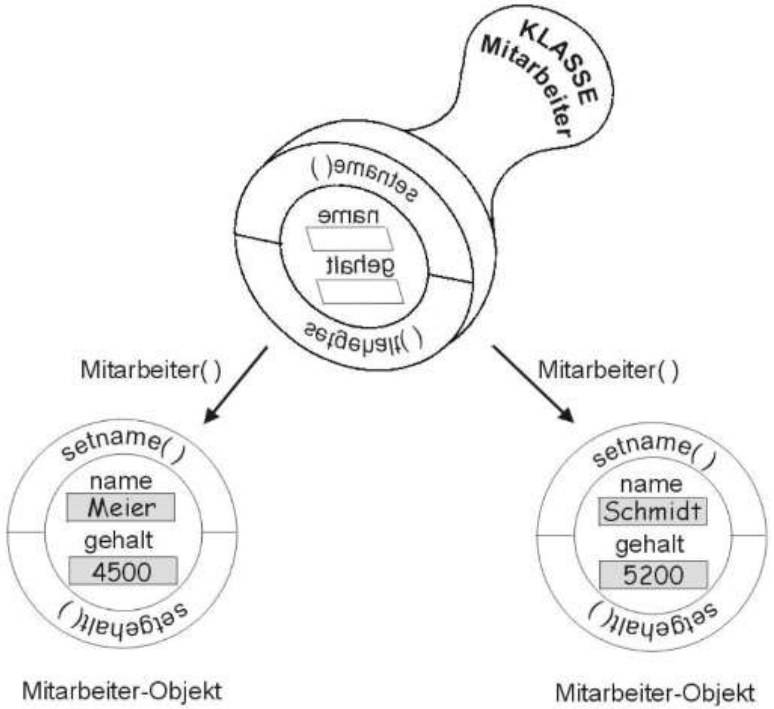
- 4 Reifen mit bestimmter Breite und Durchmesser
- 5 farbige Türen
- Einen Motor mit einer bestimmten Leistung
- 5 Sitze mit wählbaren Bezügen

- Objekt **„Auto1“**
  - 4 Reifen – 19 Zoll / 255mm
  - Fünf rote Türen
  - Motor mit 150 kW
  - 5 Ledersitze
- Objekt **„Auto2“**
  - 4 Reifen – 19 Zoll / 255mm
  - Fünf rote Türen
  - Motor mit 150 kW
  - 5 Ledersitze
- Objekt **„Auto3“**
  - 4 Reifen – 16 Zoll / 205mm
  - Fünf blaue Türen
  - Motor mit 90 kW
  - 5 Sitze mit Textilbezug



Klassen (II)

- Jedes Objekte ist allen anderen Objekten derselben Klasse in Form (Schnittstelle) und Verhalten (Implementierung) gleich
- Jedes Objekt hat zu jedem Attribut seiner Klasse einen eigenen Wert
  - Vom Zeitpunkt der Erzeugung bis hin zu seiner Zerstörung
  - Bei der Erzeugung eines Objekts werden die Attribute mit passenden Werten belegt – ansonsten ist der Wert undefiniert
  - Im Verlauf des Programms können sich diese Werte verändern
    - Z.B. wenn das Objekt seinen Zustand ändert (als Reaktion auf eine Anforderung eines Klienten)
- Die Beziehung zwischen Operation und Implementierung ist hingegen unveränderlich
  - Während der Laufzeit kann die Implementierung nicht verändert werden
  - Die Operationen und die Implementierungen werden durch die Klasse des Objekts beim Erzeugen des Objekts eindeutig festgelegt



Klassen (III)

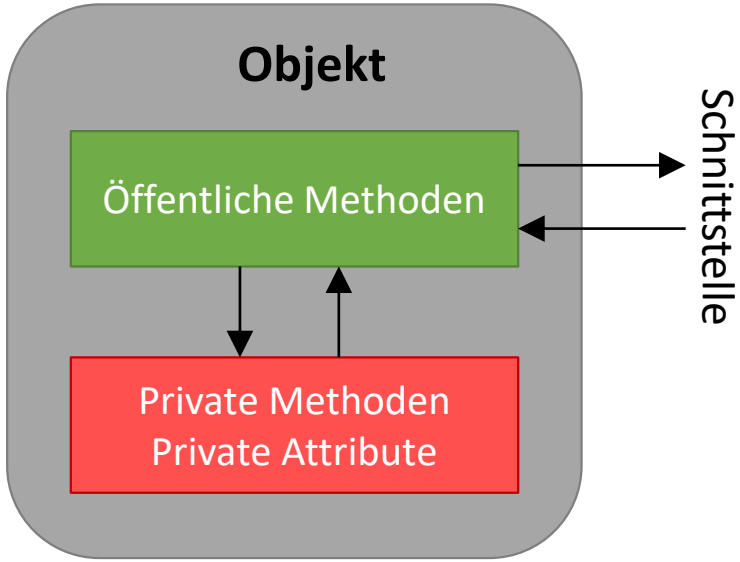
- In vielen Programmiersprachen werden Klassen wie Datentypen behandelt
  - Der Klassenname wird dabei wie jeder andere vordefinierte Datentyp benutzt
  - Die Klassendeklaration alleine belegt keinen Speicherplatz – sie legt nur die Struktur fest, nach der ein Compiler ein Objekt der Klasse erzeugt
- Typischerweise beginnen Klassennamen mit einem Großbuchstaben und sind Substantive
  - Bei Microsoft Windows beginnen Klassen meist mit einem **C** für Class. Die Attribute beginnen mit **m_** für Member und unterscheiden sich damit von lokalen Variablen und Parametern von Methoden

Allerdings unterscheiden sich die verschiedenen OOP Programmiersprachen und Bibliotheken zum Teil deutlich in diesen Punkten!

Datenkapselung („*Encapsulation*“)

Grundlegendes Konzept der OOP

- Kombination aus Daten und den Funktionen zur Manipulation der Daten (**Methoden**)
- Zugriff von außen nur über definierte Schnittstelle (**Data Hiding**)
- Vollständiger Zugriff „innerhalb“ des Objekts



Vorteile

- Vermeiden von Inkonsistenzen
  - Ein Objekts kann nicht in unerwarteter Weise von außen geändert werden
- Einfache Änderung der Implementierung möglich
  - Details der Implementierung sind anderen Klassen nicht bekannt
- Erhöhte Übersichtlichkeit
  - Nur die öffentliche Schnittstelle muss betrachtet werden

Vorsicht

Die verschiedenen Programmiersprachen implementieren dieses Konzept unterschiedlich (mehr oder minder strikt)

Beziehungen zwischen Objekten (I)

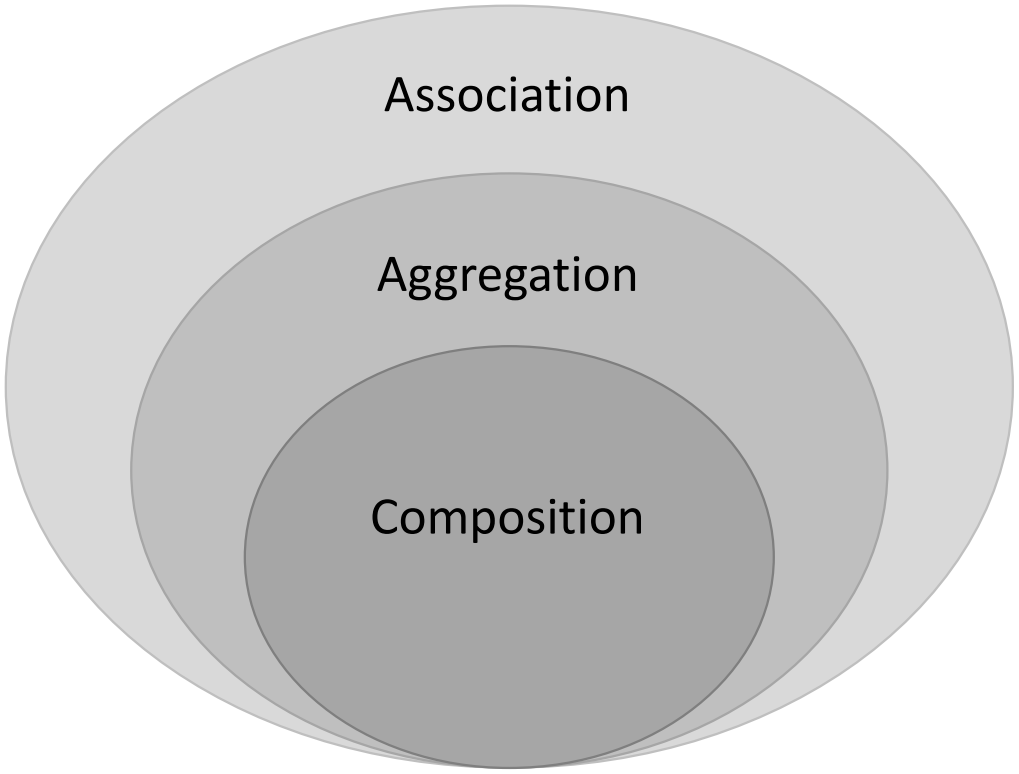
Konsequente Anwendung der objektorientierten Denkweise führt in der Regel zu Systemen mit vielen (kleinen) Objekten, die eine ganz bestimmte spezielle Aufgabe übernehmen

Die Funktionalität des Gesamt-Programms wird durch die richtige *„Verschaltung“* der Objekte untereinander erreicht

Mögliche Beziehungen:

- Assoziation
- Aggregation
- Komposition

- Spezialisierung/Vererbung



Beziehungen zwischen Objekten (II)

- Assoziation
  - Ein Objekt einer Klasse verwendet die Dienste eines Objekts einer anderen Klasse
- Aggregation & Komposition
  - Häufige Abstraktion im täglichen Leben: Teile/Ganzes-Beziehung
  - *”besteht aus“* bzw. *”ist Teil von“* (**has_a**)

Ein Auto besteht (u.a) aus 4 Rädern

- Aggregation
  - Teile existieren selbständig und können (gleichzeitig) zu mehreren Aggregat-Objekten gehören

Ein Auto besteht (u.a.) aus 4 Rädern
- Ein Rad ist Teil von höchstens einem Auto
- Es gibt aber auch Räder ohne Auto, die an andere Autos montiert werden können

- Komposition
  - starke Form der Aggregation
  - Teil-Objekt gehört zu genau einem Komposit-Objekt
    - Es kann nicht Teil verschiedener Komposit-Objekte sein
    - Es kann nicht ohne sein Komposit-Objekt existieren
  - Beim Löschen des Komposit-Objekts werden auch seine Teil-Objekte gelöscht

Ein Verzeichnis besteht aus beliebig vielen Dateien
- Dateien stehen immer in einem Verzeichnis
- Wird dieses gelöscht, so auch alle enthaltenen Dateien

Vererbung (I)

Grundlegendes Konzept der OOP:

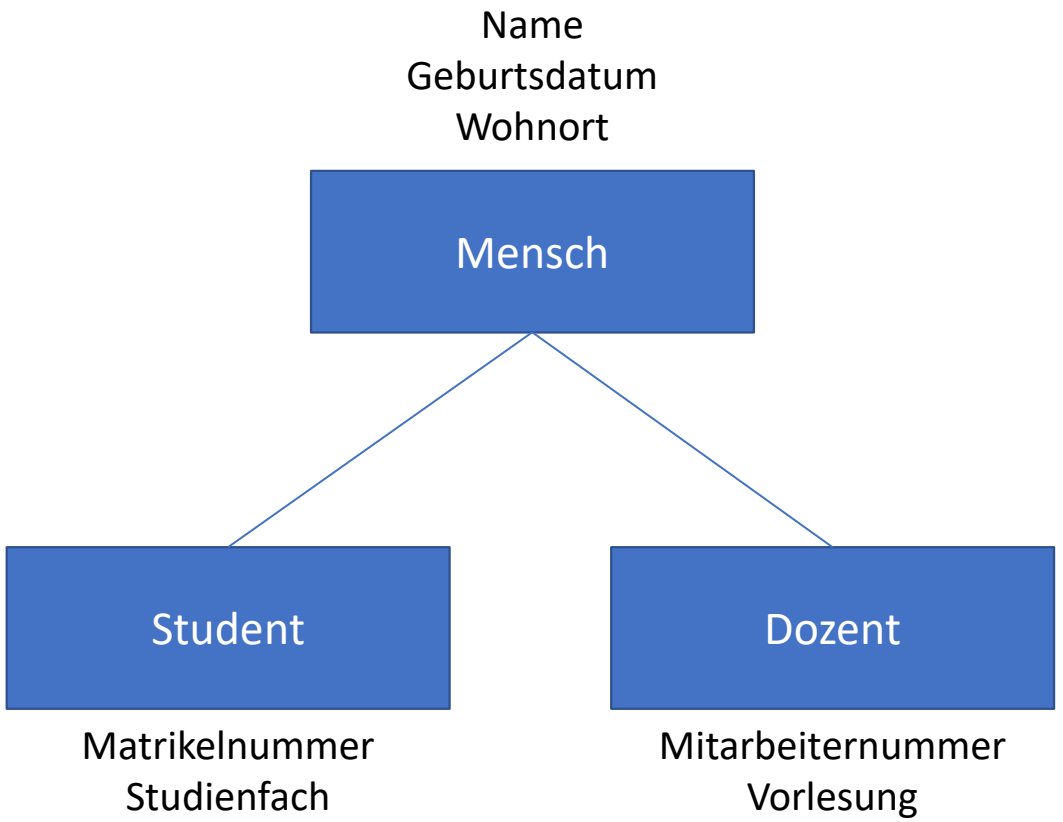
- Aufbauend auf existierenden Klassen durch Vererbung (**Inheritance**) neue Klassen schaffen
- Die neue Klasse erbt dabei alle Eigenschaften (Attribute, Operationen, Implementierungen) der ursprünglichen Klasse

- Die Klasse von der geerbt wird, nennt man **Basisklasse** (auch **Oberklasse**, **Elternklasse**)
- Die Klasse, die erbt, nennt man **abgeleitete Klasse** (auch **Unterklasse**, **Kind-Klasse**)

Typische Zielsetzung der Vererbung: **Spezialisierung**

1. Zusätzliche Attribute
2. Erweiterung der Schnittstelle der Basis-Klasse
3. Ändern der Implementierungen (**Überladen**)

Umgekehrt spricht man von **Verallgemeinerung** …

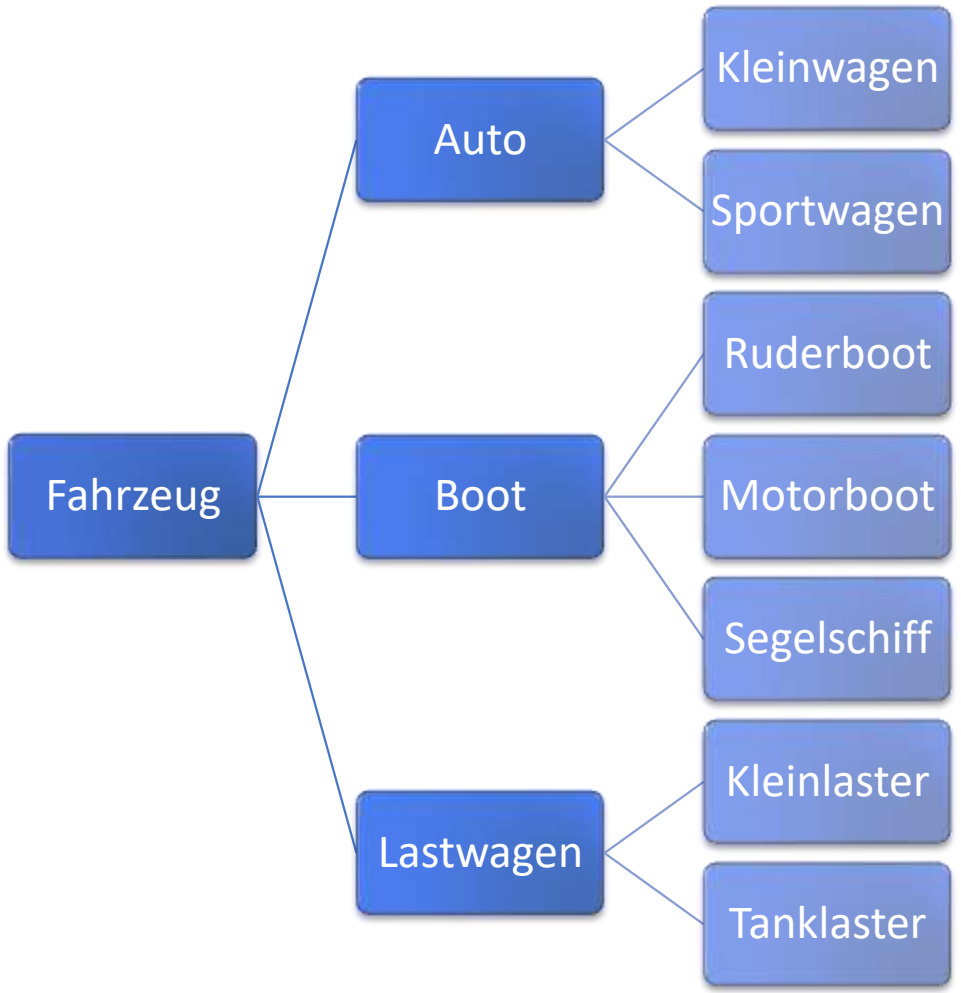


is_a Beziehung: `Student` ist ein `Mensch`

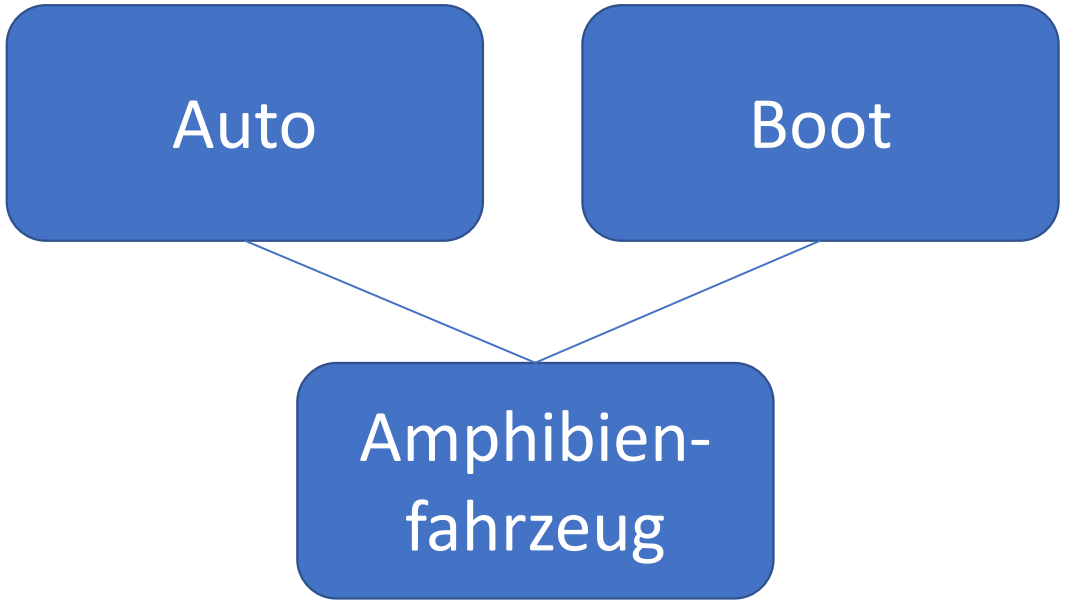
Vererbung (II)

Klassenhierarchie

- Abgeleitete Klasse als Basisklasse für weitere Klassen
- Eigenschaften werden dabei an alle Unterklassen vererbt
- Beispiel:
  - `Kleinwagen` hat alle Schnittstellen und alle Eigenschaften von `Auto` und damit auch von `Fahrzeug`
  - Aber: Eventuell sind Implementierungen von `Fahrzeug` von `Auto` umdefiniert wurden
- `Kleinwagen` **is_a** `Auto` **is_a** `Fahrzeug`
  - Ein Objekt der Klasse `Kleinwagen` kann auch an Stelle eines Objekts der Klasse `Fahrzeug` verwendet werden



Vererbung (III)



Mehrfachvererbung

- Abgeleitete Klasse besitzt mehrere Basisklassen
- Erfordert viel Sorgfalt beim Design
  - Umgang mit Schnittstellenkonflikten
  - Korrekte Modellierung der Beziehung
- Beispiel: `Apfelkuchen` als Erbe von `Apfel` und `Kuchen`
  - `Apfelkuchen` **is_a** `Kuchen`
  - ABER: `Apfelkuchen` **is_NOT_a** `Apfel`
  - Korrekt wäre eine **has_a** Beziehung (Aggregation)

ACHTUNG: Mehrfachvererbung wird nicht von allen OO Sprachen unterstützt - manche unterstützen es auch nur über Kunstgriffe!

Polymorphie

Polymorphie (*„Vielgestaltigkeit“*)

Grundlegendes Konzept der OOP:

- Objekte verschiedener Klassen werden über eine gemeinsamen Schnittstelle angesprochen
- Die Objekte führen aber unterschiedliche Implementierungen aus

Eng verbunden mit der Vererbung

- Ein Objekt der abgeleiteten Klasse kann an Stelle eines Objekts der Basisklasse verwendet werden
  - Beispiel: `Kleinwagen` **is_a** `Auto` **is_a** `Fahrzeug`
- Überladen von Implementierungen in der abgeleiteten Klasse

Strukturiert / Modular / Objektorientiert

Strukturierte Programmierung	Modulare Programmierung	Objektorientierte Programmierung
Verständlicher, übersichtlicher Code	Besonders geeignet für große Projekte	Aus Modulen werden Objekte
Wartbarer und erweiterbarer Code	Wiederverwendbarkeit universeller Module (hohe Sicherheit der fehlerfreien Funktionsweise)	Objektorientierung orientiert sich am Konzept, wie wir Menschen die Umwelt wahrnehmen
Wiederverwendung von Algorithmen	Module können einfach und beliebig zu großen Programmen zusammengefügt werden	Hohes Maß der Wiederverwendbarkeit von Softwarekomponenten durch Klassen
Wiederverwenden von Code durch allgemeingültige Funktionen oder Makros anstelle von Copy-and-Paste	Ist eine Erweiterung der strukturierten Programmierung	Beliebige Wiederverwendbarkeit definierter Programmlogik. Objektressourcen stehen in vielfältigen Objektklassen zur freien Verwendung

Strukturiert, modular und objektorientiert schließen sich gegenseitig *nicht* aus, sondern sind Evolutionsschritte.

Zusammenfassung

- Bei der **Objektorientierte Programmierung** (**OOP**) wird die Software Architektur an den Grundstrukturen desjenigen Bereichs der Wirklichkeit ausgerichtet, der die gegebene Anwendung betrifft.

- Daten und Code werden dabei in **Objekten** zusammengefasst (**Datenkapselung**)
  - Der Zugriff auf die Dienste eines Objekts erfolgt über Nachrichten.
  - Ein Objekt reagiert nur auf Nachrichten die zu seiner Schnittstelle passt.

- **Klassen** dienen dabei als Schablonen für die Struktur von Objekten.
  - Ein Objekt ist die **Instanz** einer Klasse.

- Über **Vererbung** können Klassen von anderen Klassen abgeleitet werden.
  - Dabei erbt die Klasse die Eigenschaften (Attribute und Operationen) der **Basisklasse**.
  - Typischerweise erfolgt dabei eine Spezialisierung, in der die abgeleitete Klasse zusätzliche Attribute einführt, die Schnittstelle erweitert oder die Implementierung modifiziert (**Überladen**).
  - Abgeleitete Klassen können wieder als Basisklasse für andere Klassen dienen (**Klassenhierarchien)**.

- **Polymorphie** erlaubt die Ansprache verschiedener Objekte über eine gemeinsame Schnittstelle wobei unterschiedliche Aktionen ausgelöst werden können.

Unified Modelling Language (UML)

Historie

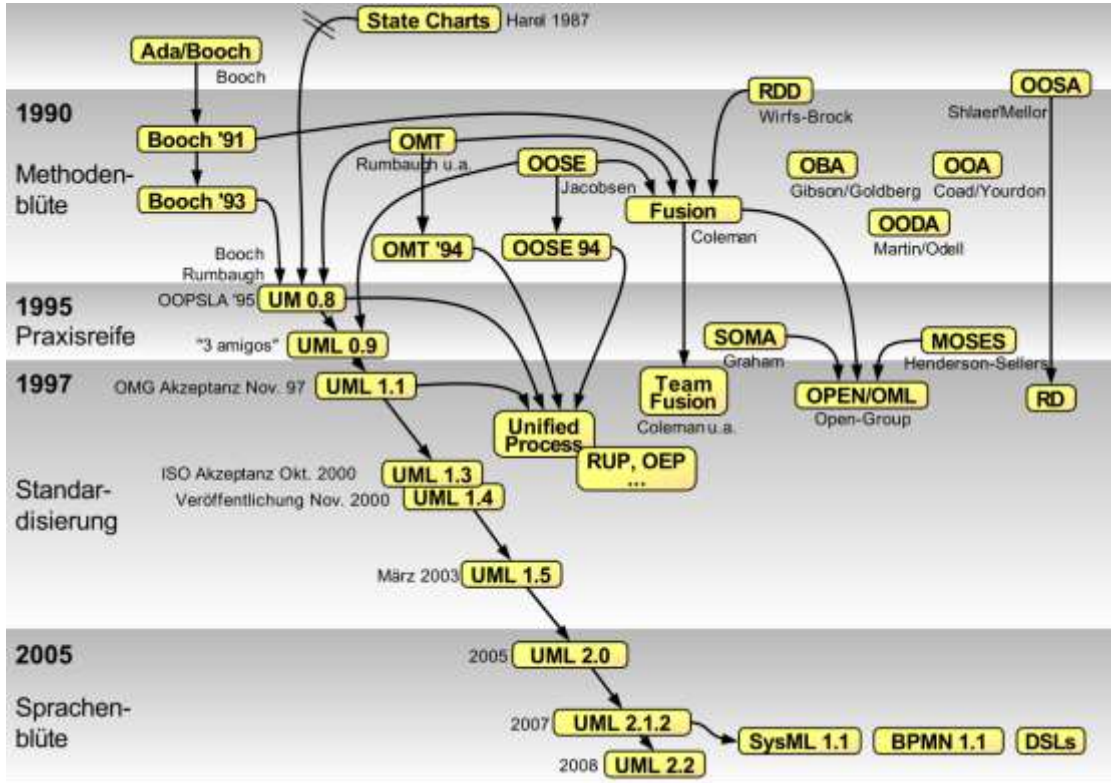
Ursprünglich verschiedene Modellierungssprachen und Methoden

- Booch Method (Grady Booch - *„Object-Oriented Analysis and Design with Applications“*)
- OMT (Object Modeling Technique) von James Rumbaugh
- RDD (Responsibility Driven Design) von Wirfs-Brock

„3 Amigos“

- Grady Booch, James Rumbaugh und Ivar Jacobson
- Zusammenführung verschiedener Notationssysteme

Daraus entstand UML …



Unified Modeling Language (I)

- Entwicklung und Normierung durch die **O**bject **M**anagement **G**roup (www.omg.org)
- ISO Standard
- Dominierende Sprache für die Modellierung von Softwaresystem
- Informelle Dokumentation verwendet oft UML Elemente

Unified Modeling Language (II)

- Grafische Modellierungssprache
- Semantik
  - Bezeichner für die Modellierung
  - Beziehungen zwischen den Begriffen
- Notation (Visual Representation)
  - Unterteilung in verschiedene Sichten/Diagramme
  - Sicht repräsentiert einen Aspekt des Modells (Modelle sind typischerweise komplex)
  - Üblich ist Einsatz einer Teilmenge - die für das aktuelle Projekt sinnvoll ist
  - Grenzen zwischen Sichten verlaufen nicht scharf – Mischung ist möglich
  - Erstellung mittels spezieller Software oder per Hand (Papier und Stift)

Aufteilung der Sichten in zwei Hauptgruppen

- Strukturelle Sicht
  - Klassendiagramm
  - Kompositionsstrukturdiagramm (Montagediagramm)
  - Komponentendiagramm
  - Verteilungsdiagramm
  - Objektdiagramm
  - Paketdiagramm
  - Profildiagramm
- Verhaltenssicht
  - Aktivitätsdiagramm
  - Use Case Diagramm (Anwendungsfalldiagramm)
  - Interaktionsübersichtsdiagramm
  - Kommunikationsdiagramm
  - Sequenzdiagramm
  - Zeitverlaufsdiagramm
  - Zustandsdiagramm

Use Case Diagramm (I)

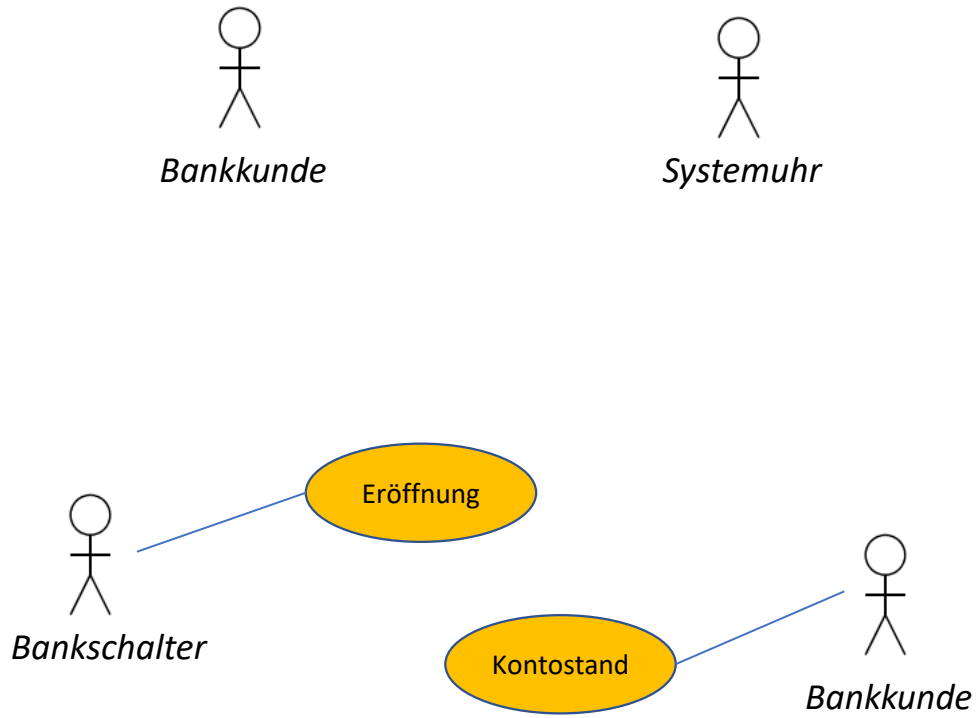
- Stellt das erwartete Verhalten eines Systems dar
- Spezifiziert die Anforderungen an ein System
- Keine Ablaufbeschreibung!

Ziel: möglichst einfach zu zeigen, welche Fälle der Anwendung es gibt

Akteure	Als Strichmännchen dargestellt. Aktive Teilnehmer / Personen oder Systeme.
Anwendungsfälle	Als Ellipsen dargestellt.
Assoziationen	Gekennzeichnet durch Linien. Zwischen Akteuren und Anwendungsfällen.
Systemgrenzen	Durch Rechtecke gekennzeichnet.
include-Beziehungen	Gestrichelte Linie mit *<<include>>* zum inkludierten Anwendungsfall der für den aufrufenden Anwendungsfall notwendig ist.

Use Case Diagramm (II)

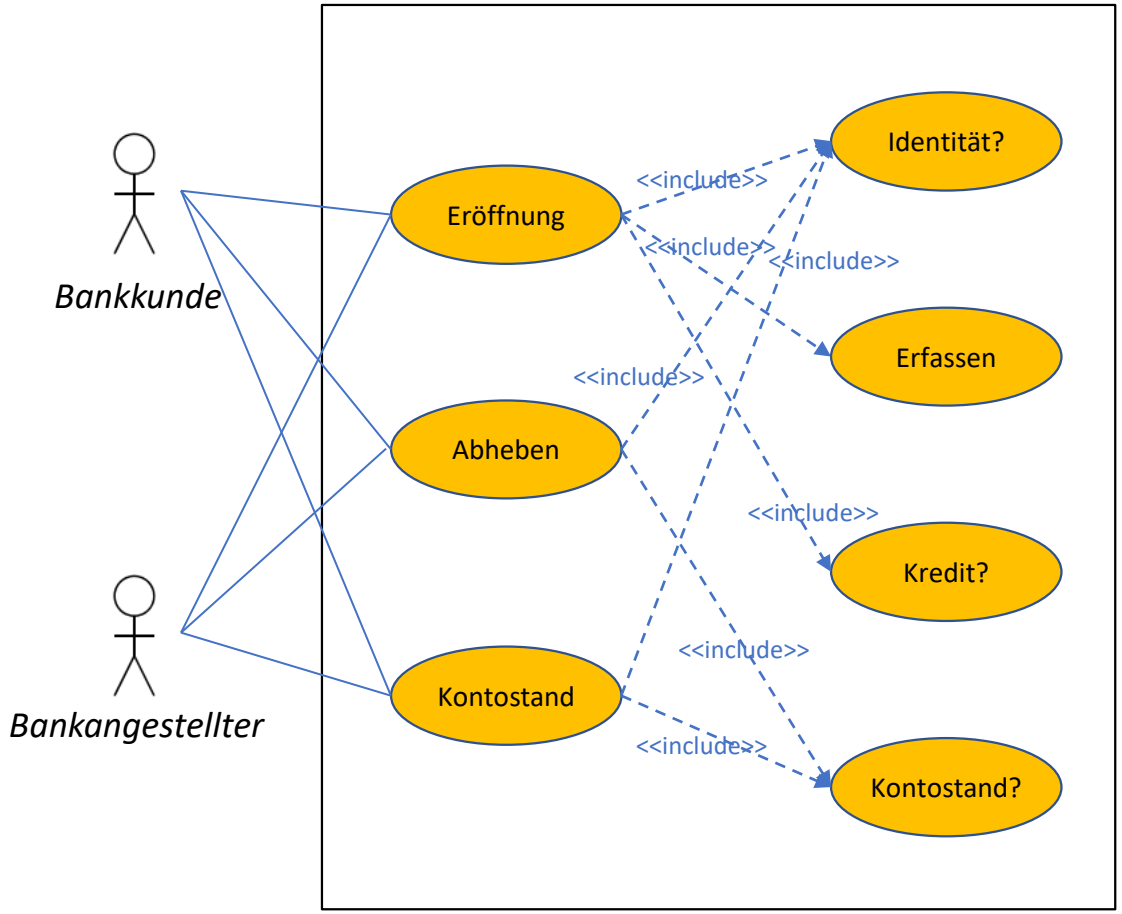
- Akteure sind aktive Teilnehmer – setzen Prozesse in Gang oder halten Prozesse am Laufen
  - Von Menschen übernehmbare Rollen, die direkt interaktiv mit dem System arbeiten
  - Andere Systeme, die über eine Verbindungen kommunizieren
  - Interne Komponenten, die kontinuierlich laufen (Systemuhr)
- Use Cases dokumentieren typischen Prozeduren aus der Sicht der aktiven Teilnehmer (Akteure)
  - Aufzählung einzelner Schritte, die zu einem Vorgang gehören
  - Zusammengefasst in grafischer Form, in der nur noch die Akteure, die zusammengefassten Prozeduren und Beziehungen zu sehen sind



Use Case Diagramm – Beispiel (I)

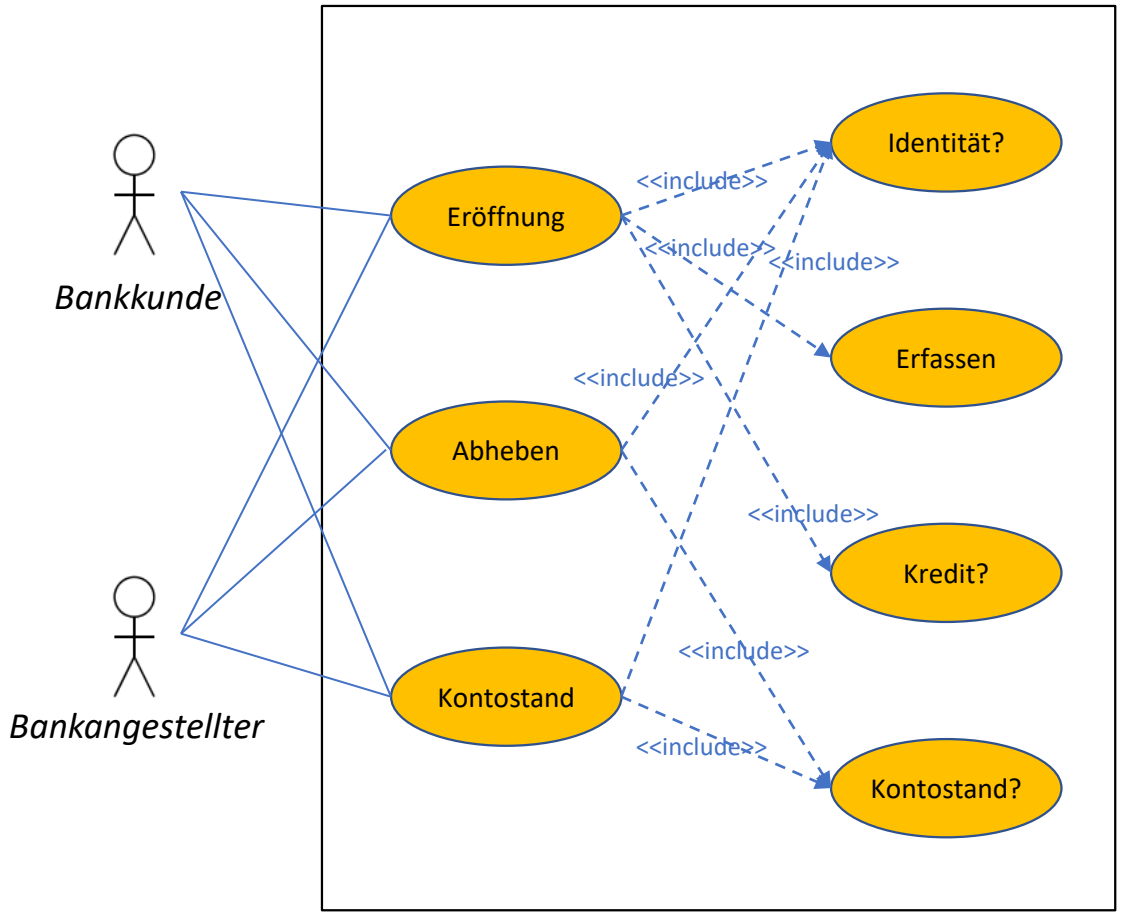
Aus welchen für die Nutzer sichtbaren Schritten bestehen einzelne typische Abläufe bei dem Umgang mit Bankkunden?

- Eröffnung eines Kontos
  - Feststellung der Identität
  - Persönliche Angaben erfassen
  - Kreditwürdigkeit überprüfen
- Geld abheben
  - Feststellung der Identität
  - Überprüfung des Kontostandes
  - Abbuchung des abgehobenen Betrages
- Auskunft über den Kontostand
  - Feststellung der Identität
  - Überprüfung des Kontostandes



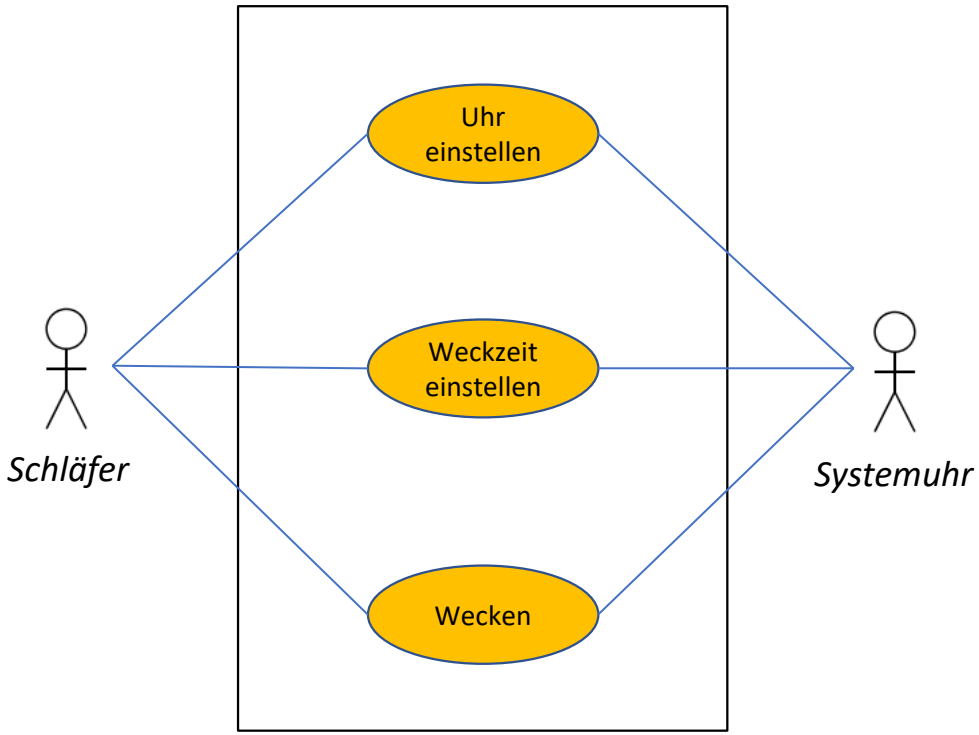
Use Case Diagramm – Beispiel (II)

- Eine glatte Linie bedeutet, dass die mit dem Prozedur von diesem Akteur angestoßen bzw. durchgeführt wird (oder er beteiligt ist)
- Gestrichelte Linien repräsentieren Beziehungen zwischen mehreren Prozeduren
  - Damit können Gemeinsamkeiten hervorgehoben werden
- Wichtig: Pfeile repräsentieren keine Flussrichtungen von Daten
- Eventuell gibt es Pfeile bei den glatten Linien
- Teilweise wird das *<<include>>* weggelassen



Use Case Diagramm – Beispiel (III)

- Ein Wecker hat intern einen Akteur — die Systemuhr
  - Aktualisiert laufend die Zeit
  - Muss eine Neu-Einstellung der Zeit bzw. die Weckzeit erfahren
- Das Auslösen des Wecksignals wird von der Systemuhr als Akteur vorgenommen
  - Diese Prozedur führt (hoffentlich) dazu, dass der Schläfer geweckt wird



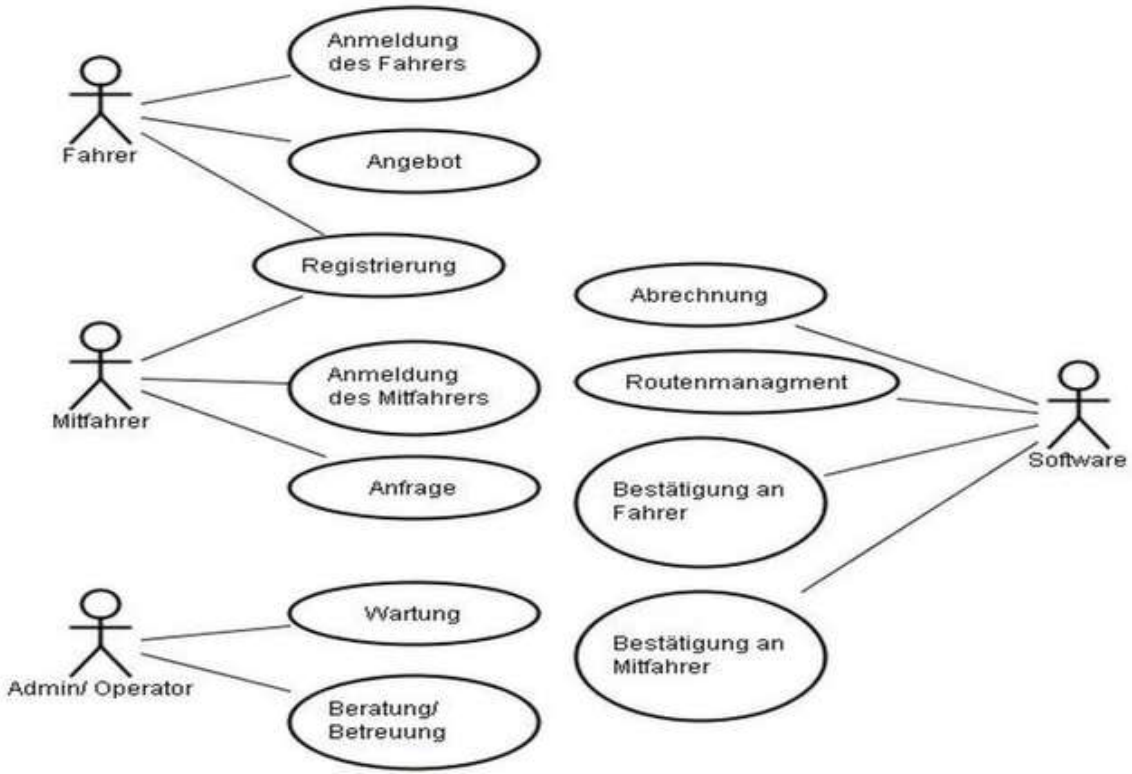
Übung

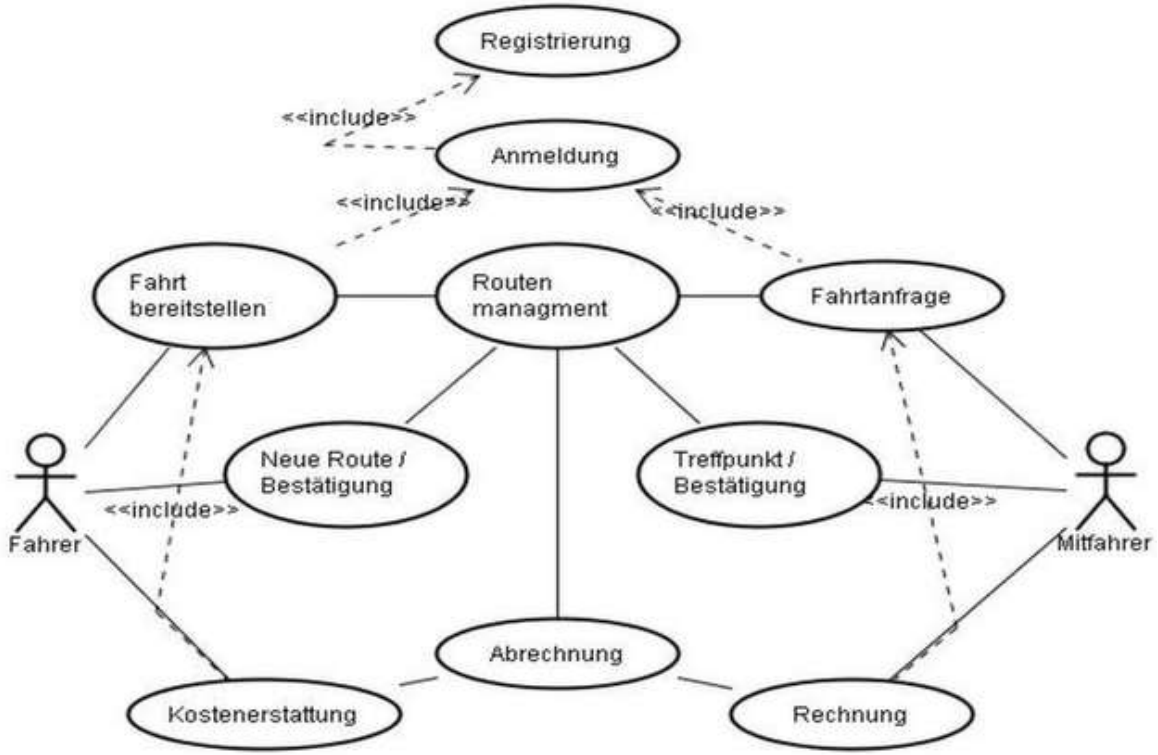
Erstellen Sie ein Use Case Diagramm für das System **Geldautomat**.

Bedenken Sie dabei alle möglichen Zustände und Abläufe, sowie alle potentiell beteiligten Akteure!

Use Case Diagramm - Beispiele

Call-Car Center Projekt



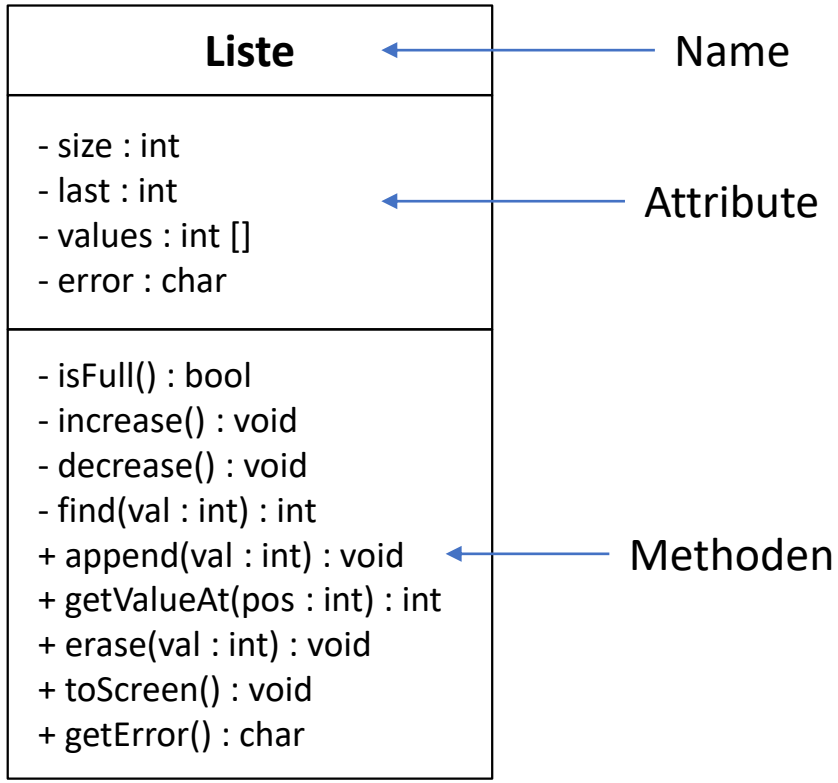


Klassendiagramm (I)

- Eine Klasse wird als Rechteck mit 3 Bereichen dargestellt
  - Der Namen der Klasse
  - Die Attribute der Klasse
  - Die Methoden der Klasse
- Existieren keine Methoden, ist der unterste Bereich optional

- Attribute
  - [*<Sichtbarkeit>*] *<name>* [:*<Typ>*]
- Methoden
  - [*<Sichtbarkeit>*] *<name>* [(*{Parameter}*)] [:*<Rückgabetyp>*]

- Sichtbarkeit der Attribute wird gekennzeichnet mit
  - - für ***private*** (Zugriff nur von der Klasse selbst)
  - + für ***public*** (öffentlich - unbeschränkter Zugriff)
  - # für ***protected*** (Zugriff von der Klasse und Unterklassen)
  - ~ für ***package*** (innerhalb des Pakets - nicht in C++)

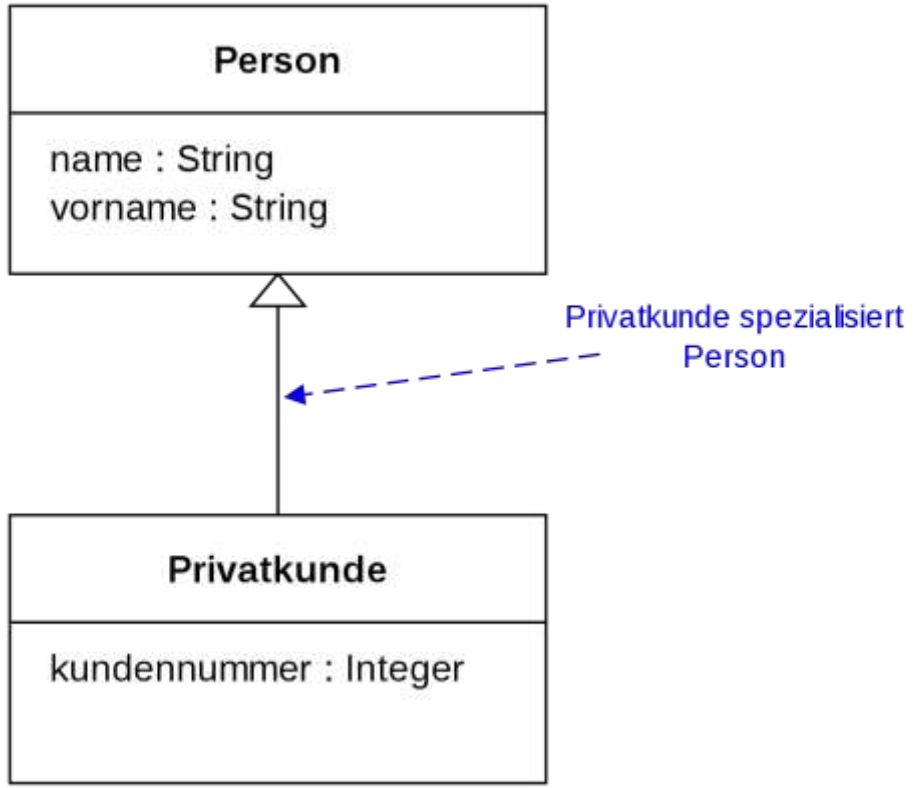


Klassendiagramm (II)

Beziehung: **Spezialisierung** (Vererbung)

- Gerichtete Beziehung
- Exemplare der spezielleren Klasse sind auch Exemplare der generelleren Klasse (is_a)

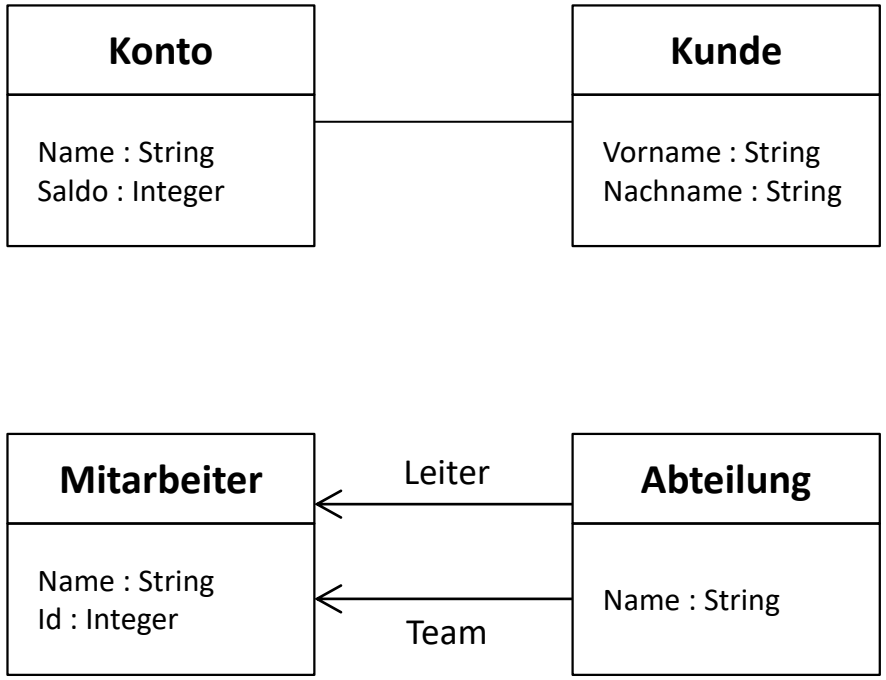
Dargestellt als durchgezogene Linie mit einer geschlossenen, nicht ausgefüllten Pfeilspitze



Klassendiagramm (III)

Beziehung: **Assoziation**

- Dargestellt durch eine durchgezogene Linie
- Optional kann über Pfeile am Linienende eine mögliche Navigationsrichtung dargestellt werden
  - Ausgehend vom Objekt der Klasse `Abteilung` kann der Leiter (Objekt der Klasse `Mitarbeiter`) bestimmt werden
  - Ausgehend vom Objekt der Klasse `Abteilung` können die Mitarbeiter des Teams bestimmt werden
- Navigation kann auch in beide Richtungen gehen
- Ohne Pfeile keine Aussagen über Navigierbarkeit
- An den Enden sind häufig **Multiplizitäten** (*Multiplicity*) vermerkt
  - Manchmal kann man statt dessen auch den Begriff Komplexitätsgrade finden



Multiplizitäten

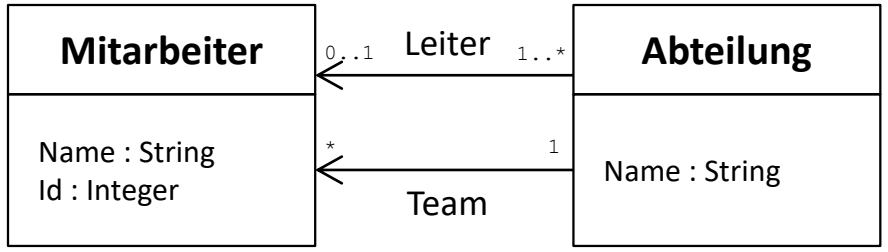
Intervall mit unterer und oberer Grenze

<untereGrenze>`..`*<obereGrenze>*

- Regeln
  - Die untere Grenze muss kleiner oder gleich der oberen Grenze sein
  - `*` steht für als obere Grenze für unbeschränkt
  - `0..*` wird oft auch als `*` abgekürzt
  - Sind beide Grenzen gleich, kann auch nur die obere Grenze angegeben werden
  - Beide Grenzen mit `0` oder `*` ist nicht erlaubt
- Seit UML 2.0 nicht mehr erlaubt:
  - Multiplizität bestehend aus mehreren Intervallen (durch Komma getrennt)
  - Z.B. `0..6,9..*` (alle Zahlen ohne 7 und 8)
- Spezifizieren jeweils aus der Sicht eines einzelnen Objekts, wie viele konkrete Beziehungen zu Objekten der anderen Klasse existieren können

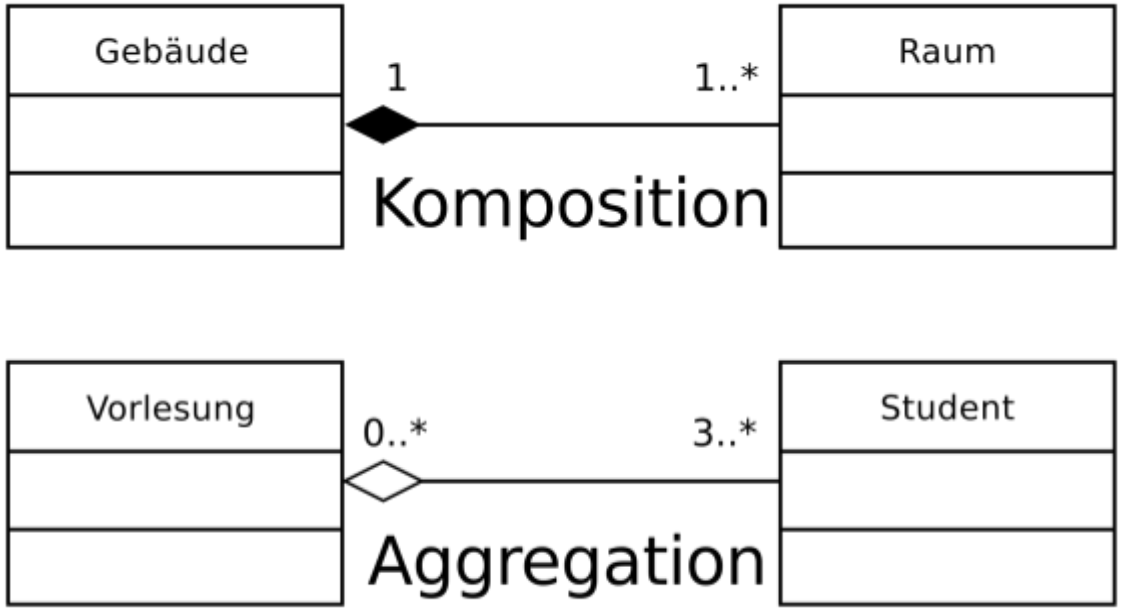
Beispiele

`1..3`	(≙ 1 oder 2 oder 3)
`6..7`	(≙ 6 oder 7)
`0..1`	(≙ 0 oder 1 – auch **optional**)
`1..*`	(alle Zahlen größer gleich 1)
`1`	(≙ `1..1` – nur die Zahl 1)



Klassendiagramm (IV)

- Beziehung: **Aggregation** / **Komposition**
- Komposition wird dargestellt durch eine ausgefüllte Raute
  - Multiplizität an der Raute ist dabei immer 1 (`1..1`)
  - Teile können nicht ohne das Ganze existieren
  - Das Gebäude besteht aus mindestens einem Raum und jeder Raum gehört zu genau einem Gebäude
- Aggregation durch eine nicht ausgefüllte Raute.
  - Multiplizität an der Raute ist immer `0..*`
  - Teile können ohne das Ganze existieren
  - Jede Vorlesung besteht aus mindestens 3 Studenten, ein Student kann keine oder mehrere Vorlesungen besuchen.
- Raute ist immer auf der Seite des Ganzen (der Beginn der **has_a** Beziehung)



Von Kakashi-Madara (talk) - max limper, Gemeinfrei, https://commons.wikimedia.org/w/index.php?curid=24355827

Übung

Stellen Sie das System ***„Auto“*** als UML Klassendiagramm inkl. Multiplizitäten und Navigationsrichtungen dar:

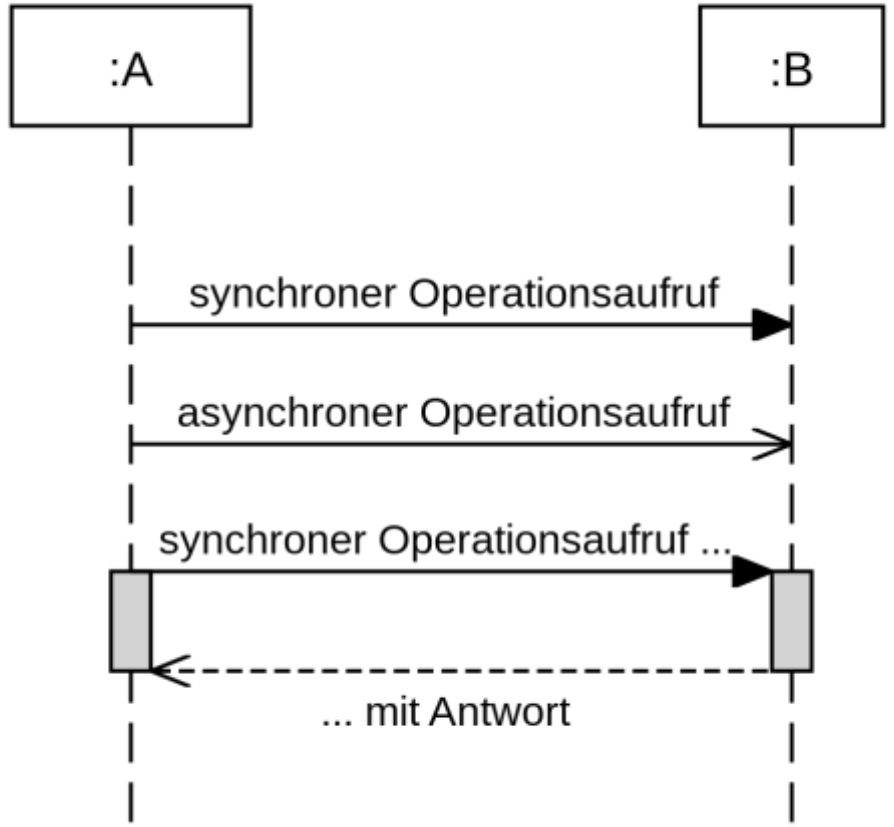
- Klassen: `Auto`, `Sportwagen`, `Motor`, `Räder`, `Fahrer`
- `Sportwagen` ist eine Spezialisierung von `Auto`
- Jedes `Auto` hat einen `Motor` und vier `Räder`
- `Motoren` und `Räder` können ausgebaut werden
- Jedes `Auto` kann maximal einen `Fahrer` haben, `Fahrer` können mehrere `Autos` fahren
- Der `Fahrer` kann über das `Auto` bestimmt werden

Sequenzdiagramme

- Stellt Interaktion grafisch dar
- Beschreibt den Austausch von Nachrichten mittels Lebenslinien
- Stellt nur einen Weg durch einen Entscheidungs-baum dar

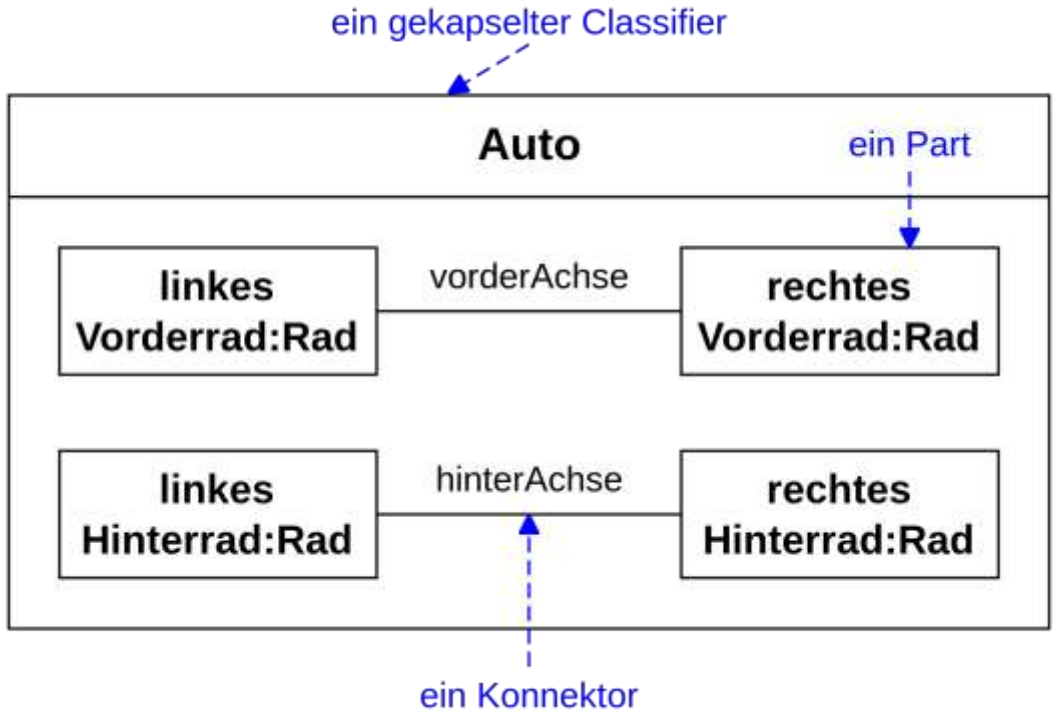
- Zwei Rechtecke stellen die Kommunikations-partner dar
- Gestrichelte senkrechte Linie ist die Lebenslinie (Zeitlinie) – gelesen von oben nach unten
  - Dargestellt wird der zeitliche Ordnung der Ereignisse, nicht der präzise Zeitpunkt
- Operationsaufrufe sind Nachrichten

- Bei C++: typischerweise synchrone Aufrufe

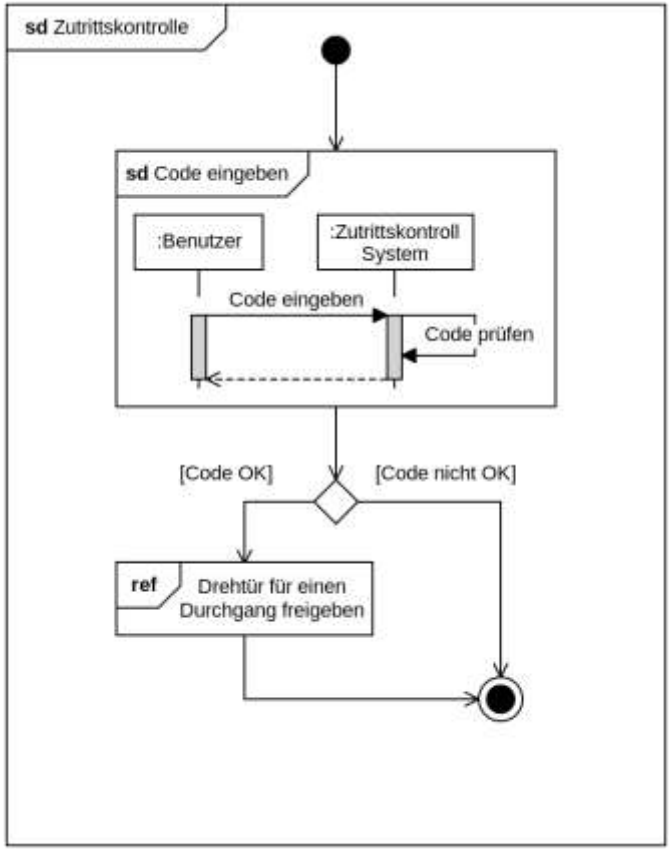


Weitere Diagramme

Kompositionsstrukturdiagramm



Interaktionsübersichtsdiagramm



Pragmatische Ansätze

- Einsatz von Teilmengen der Diagramme ist üblich
  - Was ist für das aktuelle Projekt sinnvoll?
- Grenzen zwischen Sichten verlaufen nicht scharf – Mischung ist möglich
  - Wenn damit eine besonders treffende Aussage zu einem Modell erreicht wird
- Bei größeren Projekten
  - Vermeiden Sie, alle Details in ein großes Diagramm zu integrieren
  - Es ist sinnvoller, mehrere Ebenen zu haben
  - Fokussieren sie entweder auf die Übersicht oder Details in eingeschränkten Bereichen

Zusammenfassung

- Die **Unified Modelling Language** (UML) ist eine grafische Modellierungssprache für Software Systeme.
  - UML definiert dabei sowohl die Semantik (Bezeichner/Begriffe und zugehörige Beziehungen) als auch die Notation (visuelle Repräsentation) in Form von Sichten.
- Es gibt 14 verschiedene Diagrammtypen (Sichten) unterteilt in zwei Hauptgruppen: **Strukturelle Sichten** und **Verhaltenssichten**.
- Mit einem **Use Case Diagramm** können die Anforderungen an ein System und damit das zu erwartende Verhalten dokumentiert werden.
- Mit dem **Klassendiagramm** werden die verwendeten Klassen und Beziehungen zwischen diesen Klassen dokumentiert.
  - Über die **Multiplizität** wird spezifiziert, wieviel konkrete Beziehungen es zwischen einem Objekt einer Klasse und Objekten der anderen Klasse existieren .