HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

Objektorientierte Analyse und Design

Design (Software-Entwurf)

Prof. Dr.-Ing. Michael Uelschen
Hochschule Osnabrück

Sommersemester 2021

Objektorientierte Analyse und Design
Design (Software-Entwurf)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- 00 Organisatorisches
- 01 Einführung[1]
- 02 Anforderungsanalyse[3]
- 03 Design[4]
- 04 Entwurfsmuster[3]
- 05 Sonstiges[1]

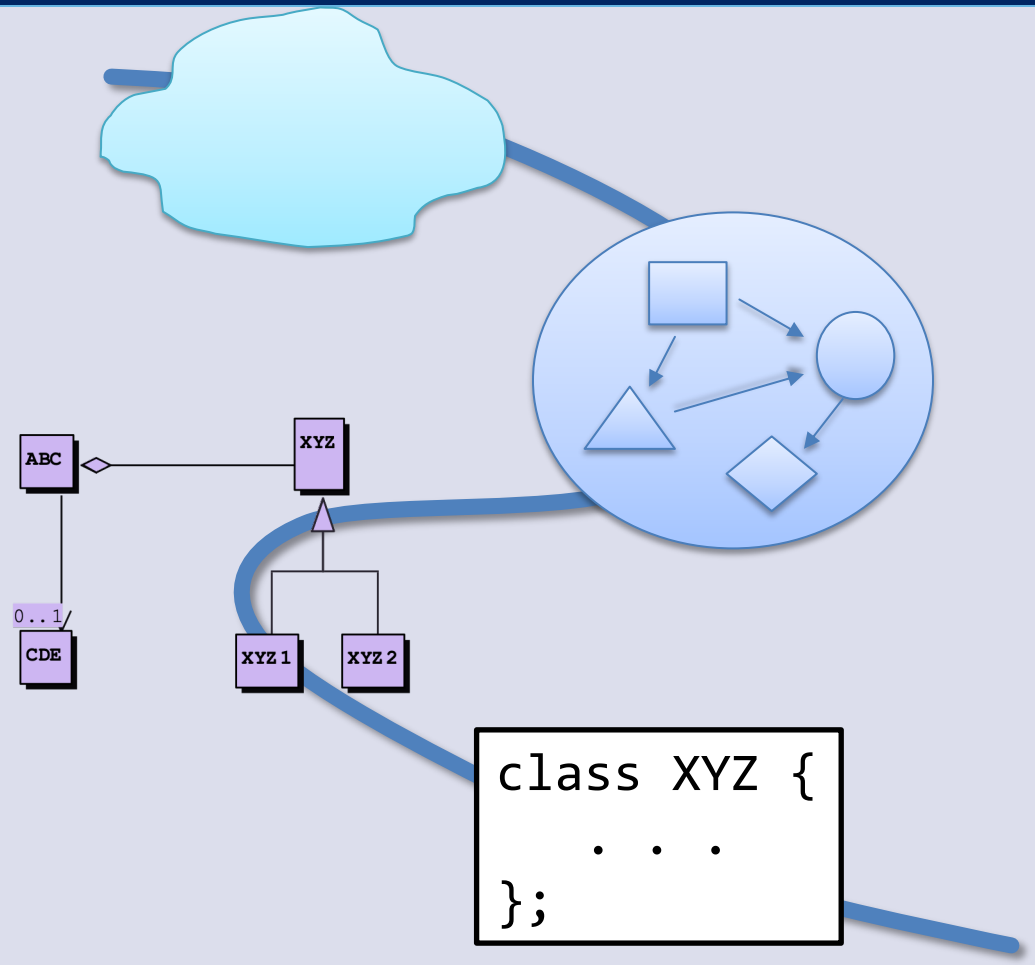
- Entwicklungsprozess
- **Statisches Modell**
  - Klassendiagramm (detailliert)
- **Dynamisches Modell**
  - Interaktionsdiagramm
    - Sequenzdiagramm
    - Timing-Diagramm
  - Zustandsdiagramm
- Sonstiges

Objektorientierte Analyse und Design

ENTWICKLUNGSPROZESS

Objektorientierte Analyse und Design
Vorgehensweise OOAD: Déjà-Vu

- **Gegeben**
  - „unstrukturiertes“ System
- **Analyse**
  - in Objekte und Beziehungen strukturiertes System und: „man weiß, was man will“.
- **Design**
  - zur Implementation gestaltetes System.
- **Implementation/Test,...**
  - Programmcode, lauffähiges System.



Objektorientierte Analyse und Design
Objektorientierte Gestaltung

- Bisher steht das Domänenmodell im Vordergrund, dass meist nicht genauso implementiert wird
  - Klassenmodell wird schrittweise in Richtung „sinnvoll programmierbar“ umgebaut.
  - In „sinnvoll“ gehen Erfahrungen und Randbedingungen ein (z. B. Web-Applikation)
  - Erfahrungen zum guten Design werden u. a. mit Design-Mustern dokumentiert.
  - Mit Design-Erfahrungen wird erstes Klassenmodell bei Erstellung besser (gibt dann nur ein zentrales Klassenmodell).

- Umwandlung der Klassen–struktur aus der **Analyse** in eine **implementierbare Klassenstruktur**, dabei:
  - Hinzunahme und Wegfall von Klassen (u. U. Hinzunahme von Kunst-und Hilfsklassen)
  - Änderung der Beziehungen zwischen den Klassen

Objektorientierte Analyse und Design
2 Schritte zur Gestaltung

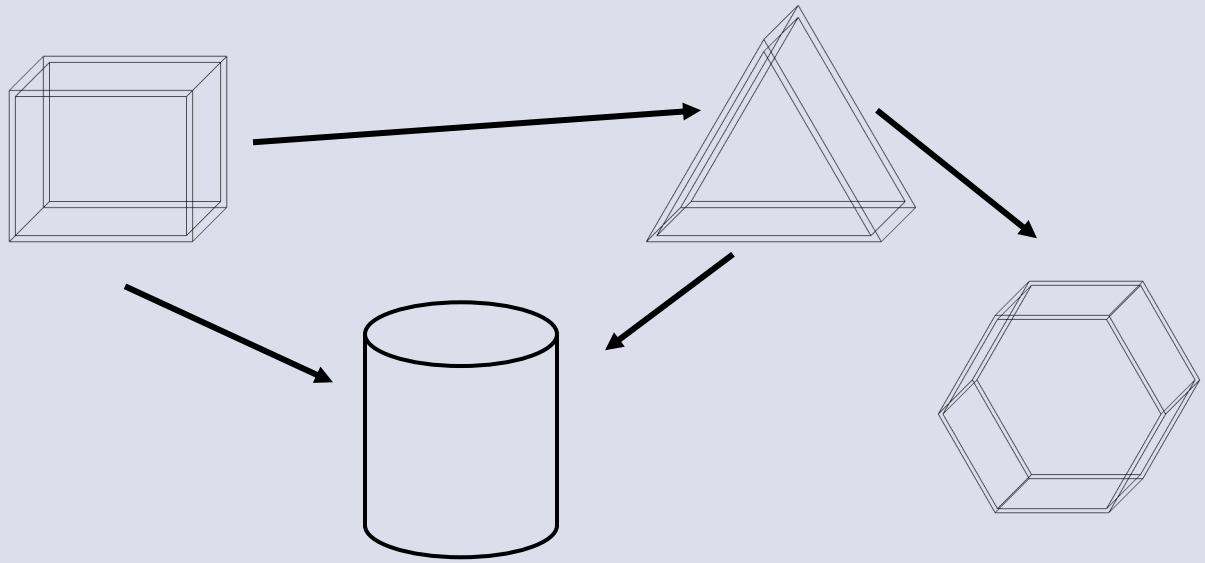
- Zwei Schritte der Gestaltung:
  1. Berücksichtigung der **Muster**, dadurch erste Anpassungen der Klassenstruktur (demnächst mehr)
  2. Versehen der Klassen mit **Methoden**, dieses erfolgt im Zusammenhang mit der Verteilung der Verantwortlichkeiten auf die Klassen.

Objektorientierte Analyse und Design

KLASSENDIAGRAMM (WELCOME BACK)

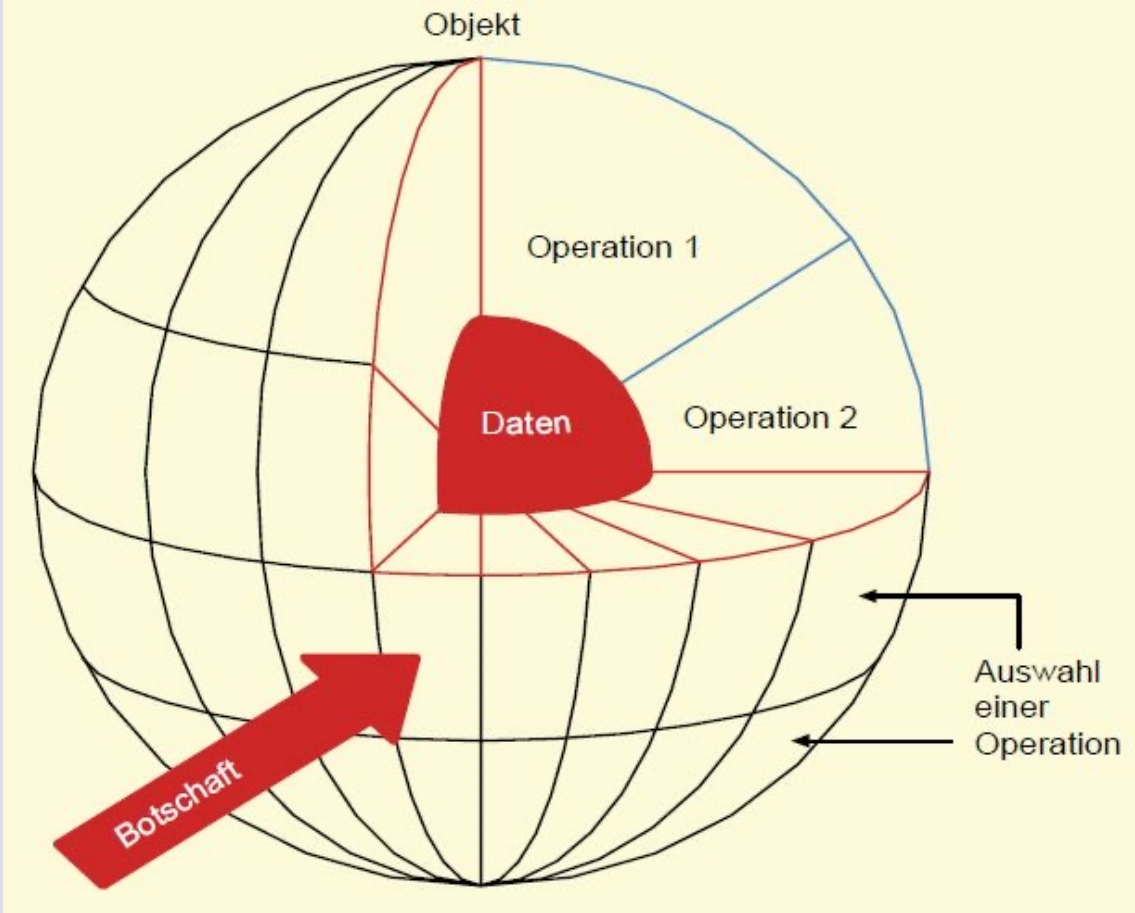
Objektorientierte Analyse und Design
Grundidee Objektorientierung

- Im Gegensatz zur prozeduralen Programmierung ist ein OO-Programm aus **Objekten** aufgebaut, die miteinander **in Beziehung stehen** und insbesondere einander **Nachrichten senden**.



Objektorientierte Analyse und Design
Was ist ein Objekt?

- Ein **Objekt** ist ein Gegenstand des Interesses, es kann ein Ding (**konkret**) oder ein Begriff (**abstrakt**) sein. Jedes Objekt besitzt eine **eigene Identität**.
- Es besteht aus inneren Daten (**Attributen**) und stellt über seine Schnittstelle (seine öffentlichen **Methoden**, Operationen) anderen Objekten Dienste zur Verfügung.

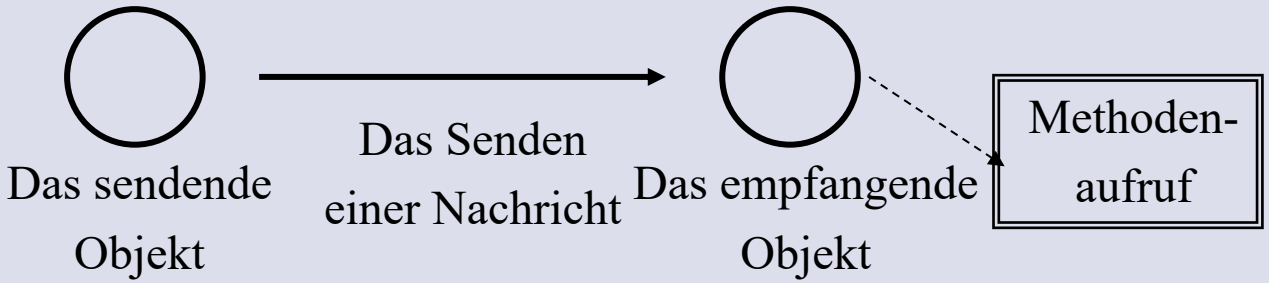


Objektorientierte Analyse und Design
Was ist eine Klasse?

- Gleichartige Objekte werden in einer **Klasse** zusammengefasst.
- Eine Klasse beschreibt eine **Schema zur Darstellung** von Objekten mit gleichen Eigenschaften und gleicher Funktionalität.
- Bei der Deklaration einer Klasse erfolgt die Festlegung für
  - der Schnittstelle (öffentliche Methoden),
  - der Daten (private bzw. geschützte Attribute),
  - u. U. der privaten oder geschützten Hilfsfunktionen.
- Von einer Klasse können i.a. beliebig viele **Ausprägungen** (Objekte der Klasse) angelegt werden.

Objektorientierte Analyse und Design
Was ist eine Beziehung?

- Objekte (bzw. Klassen) können zueinander **in Beziehung stehen** (Assoziation).
- Objekte können sich kennen oder sind aus anderen Objekten aufgebaut.
  - *Beispiel: StudentIn MusterstudentIn hört die Vorlesung OOAD.*
- Objekte können sich gegenseitig ihre Dienste anbieten oder die anderer Objekte anfordern.
- Das Senden einer Nachricht ist immer die Anforderung eines Dienstes und führt zur Ausführung einer **Methode beim Empfängerobjekt**.



Objektorientierte Analyse und Design
Vorteile der OO-Vorgehensweise

- Aufteilung eines Programmes in Komponenten (bzw. Module bzw. hier Objekte) unter gleichzeitiger Berücksichtigung von Daten und Prozeduren besitzt folgende Vorteile
- Verminderung und **Beherrschung der Komplexität** der Programme.
- Ermöglichung der **Wiederverwendung** bereits vorhandener Komponenten.
- Ganzheitliche Sichtweise, bessere **Modellierung der realen Welt**.

Objektorientierte Analyse und Design
OO-Programmierung

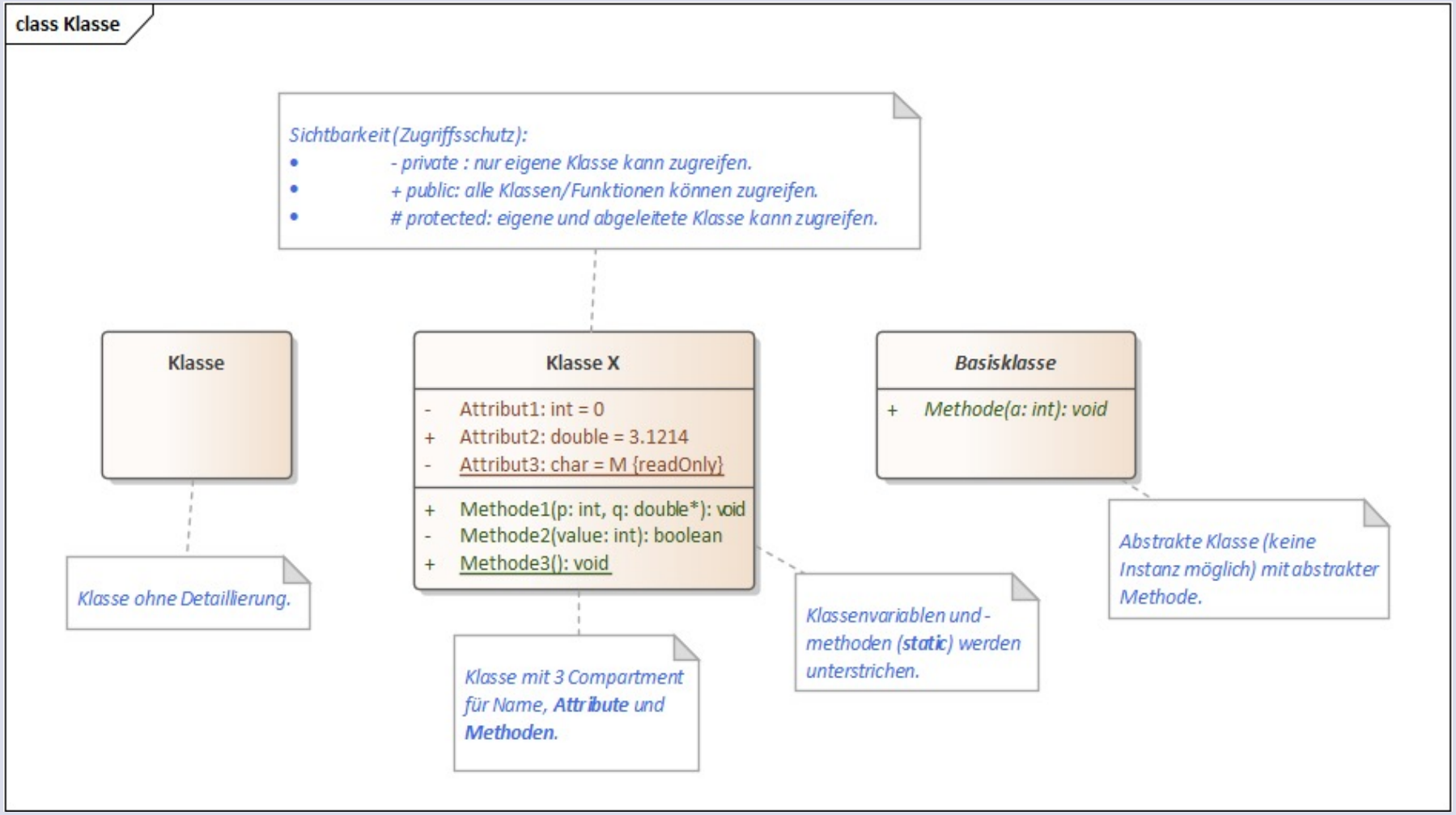
Bisher (vorherige Semester)

- Nach Vorgabe bereits gestalteter Klassen, d. h. nach Festlegung der Verantwortlichkeiten („Was sollen die Objekte tun?”)
  - ihre Beziehungen und die Beziehungen ihrer Objekte auffinden, und
  - diese Klassen in einer OOP-unterstützenden Sprache (C++, Java) zu implementieren.

Ab jetzt (Reststudium + Beruf)

- Finden der Klassen: Attribute und Methoden
- Finden der Beziehungen und des Zusammenspiels der Klassen
- Gestalten der Klassen
- Wünschenswert und hilfreich für diese Aufgaben ist:
  - Eine gute, übersichtliche, aussagekräftige Darstellung der Klassen.

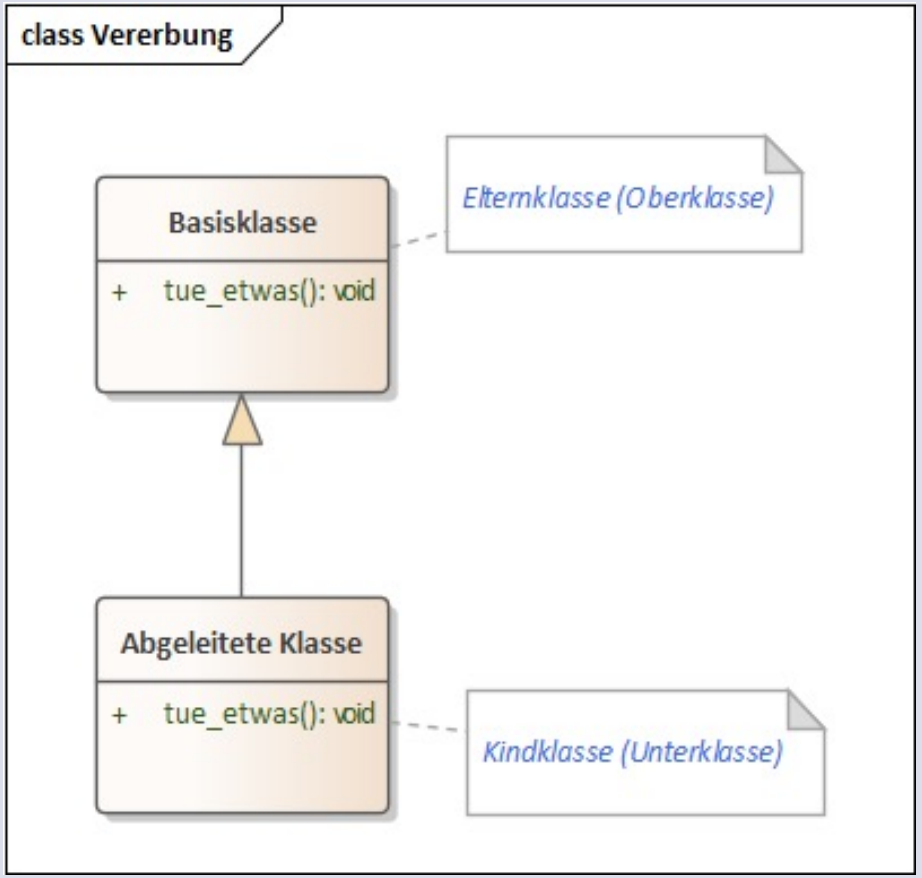
Objektorientierte Analyse und Design
UML: Klasse

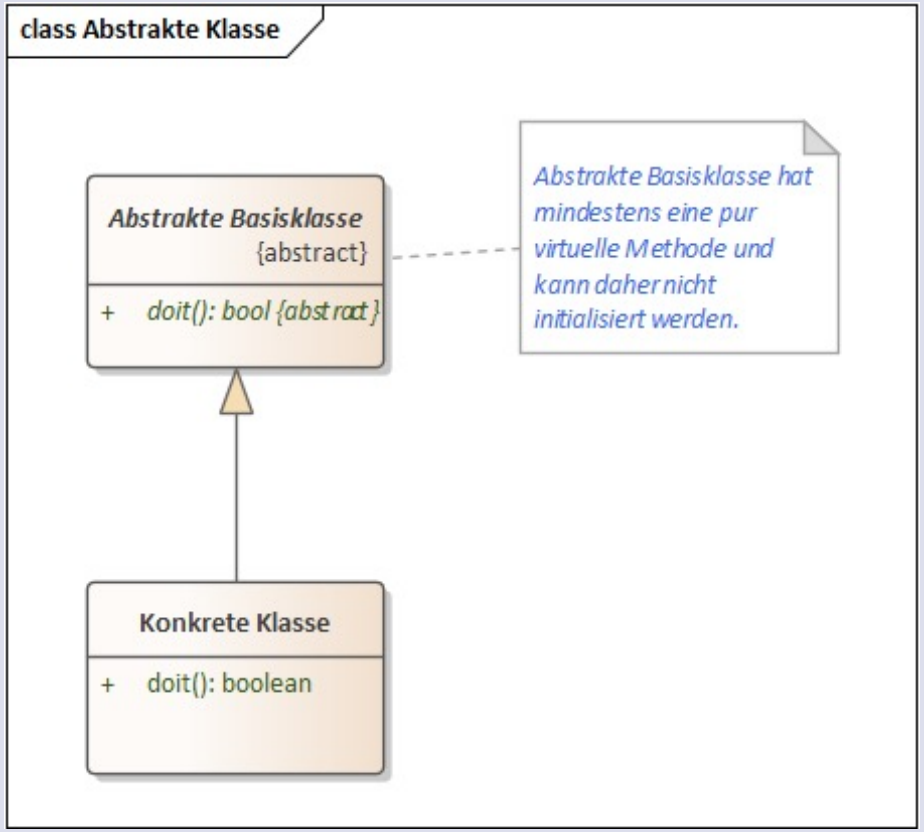


Objektorientierte Analyse und Design
Vererbung

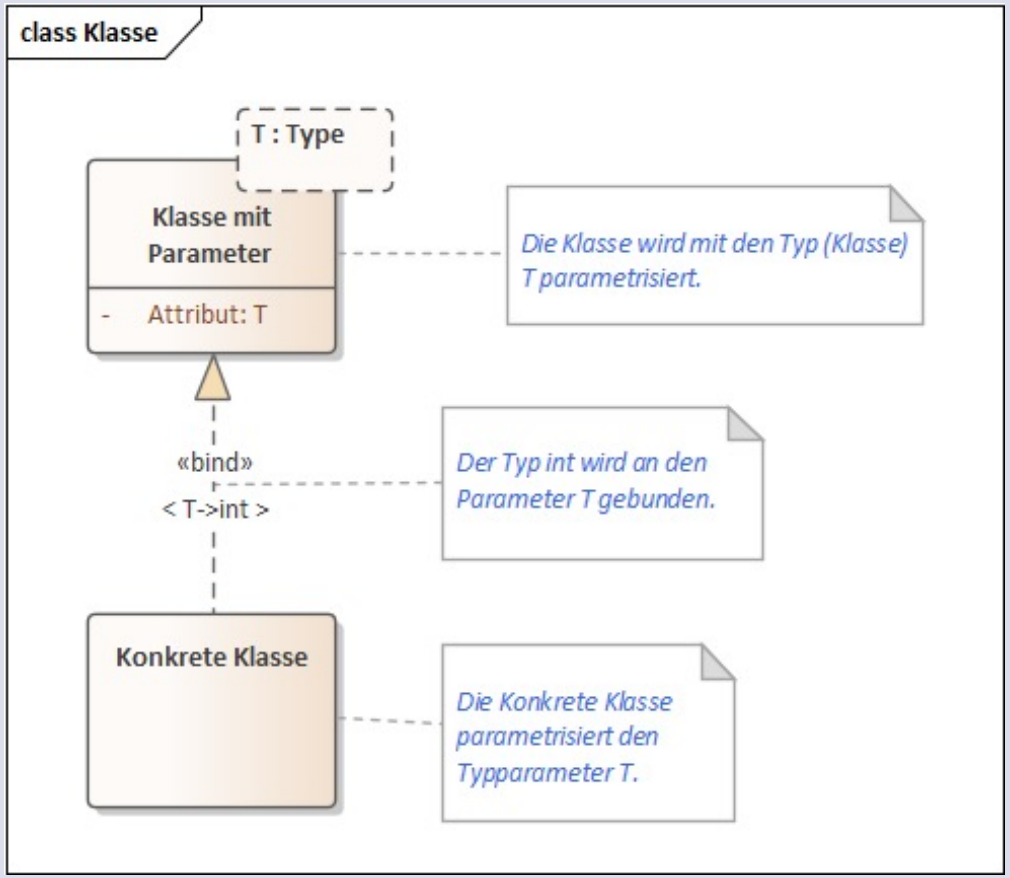
- Analysemodell wird auf erste Optimierungen geprüft.
- Wenn verschiedene Klassen große Gemeinsamkeiten haben, kann Vererbung genutzt werden
  - Variante 1: Abstrakte Klasse mit möglichen Attributen, einigen implementierten und mindestens einer nicht-implementierten Methode
  - Variante 2: Interface ausschließlich mit abstrakten Methoden (haben später noch Bedeutung)
- **Liskovsches Prinzip** für überschreibende Methoden der erbenden Klassen berücksichtigen:
  - Vorbedingung gleich oder abschwächen
  - Nachbedingungen gleich oder verstärken
- Vererbung…
  - ist Hilfsmittel nicht Ziel der Objektorientierung
  - reduziert den Codierungsaufwand
  - erschwert Wiederverwendung

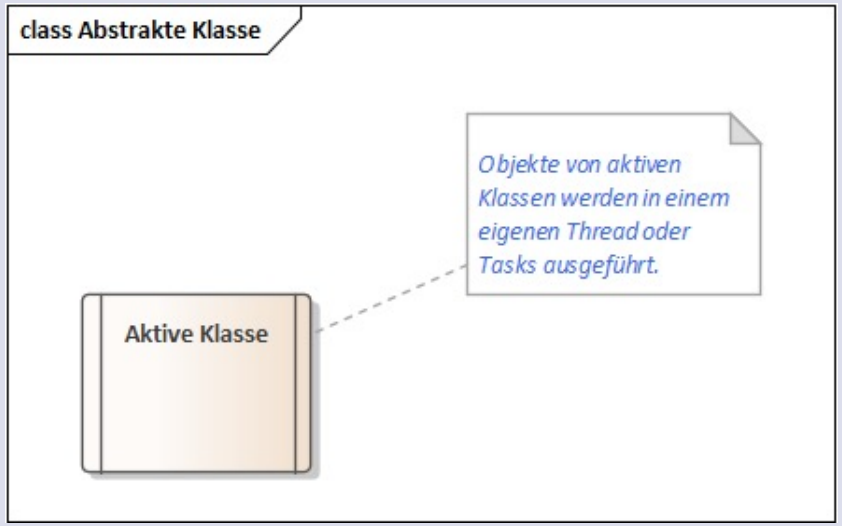
Objektorientierte Analyse und Design
UML: Vererbung und Abstrakte Klasse





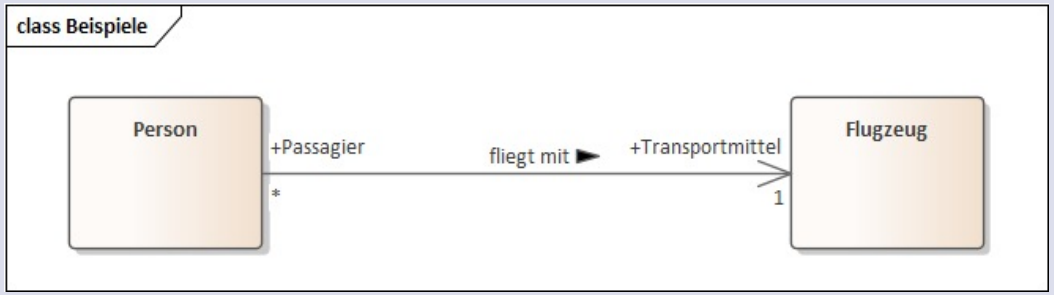
Objektorientierte Analyse und Design
UML: Parametrisierte Klasse





Objektorientierte Analyse und Design
UML: Beziehung (Assoziation)

- Zwischen den Objekten in einer Objektwelt können **vielfältige Beziehungen** bestehen.
- Die Beziehung unter Objekten heißt **Assoziation**.
- Eine Assoziation zwischen zwei Klassen beschreibt eine allgemeine Beziehung, die zwischen den Objekten der einen und den Objekten der anderen Klasse bestehen kann.

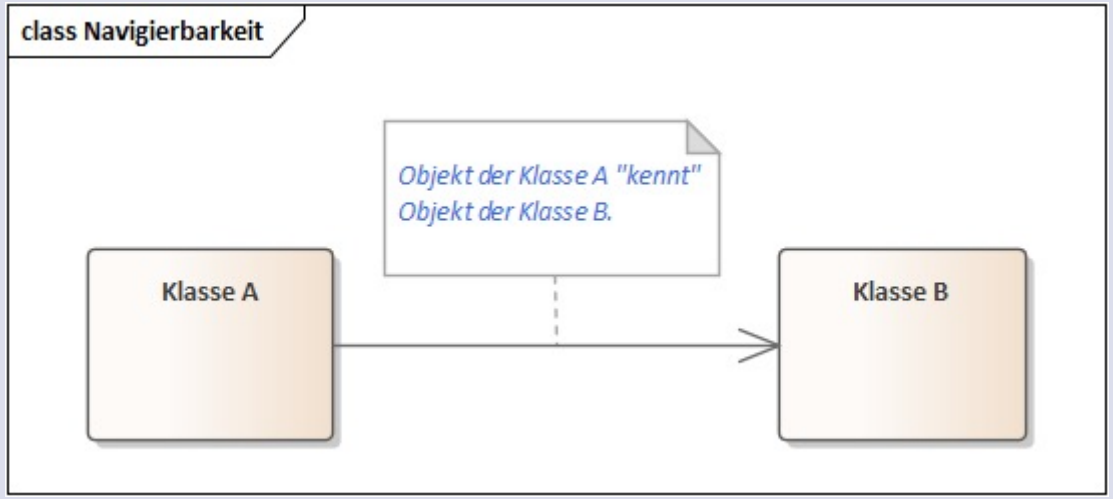


Objektorientierte Analyse und Design
UML: Multiplizität

- Die **Multiplizität** (Kardinalität) einer Assoziation gibt an, wie viele Objekte der betreffenden Klasse mit einem Objekt der gegenüberliegenden Klasse in Beziehung stehen.
- Mögliche Werte sind (z. B.):
  - `*` 0 bis beliebig viele
  - `1..*` 1 bis beliebig viele
  - `0..1` einer oder keiner
  - `2..7` zwei bis sieben
  - `2,4,6` zwei, vier oder sechs
- Die Multiplizitätsangabe einer Assoziation kann fehlen; in diesem Fall beträgt in der Regel die Multiplizität 1.

Objektorientierte Analyse und Design
UML: Navigierbarkeit

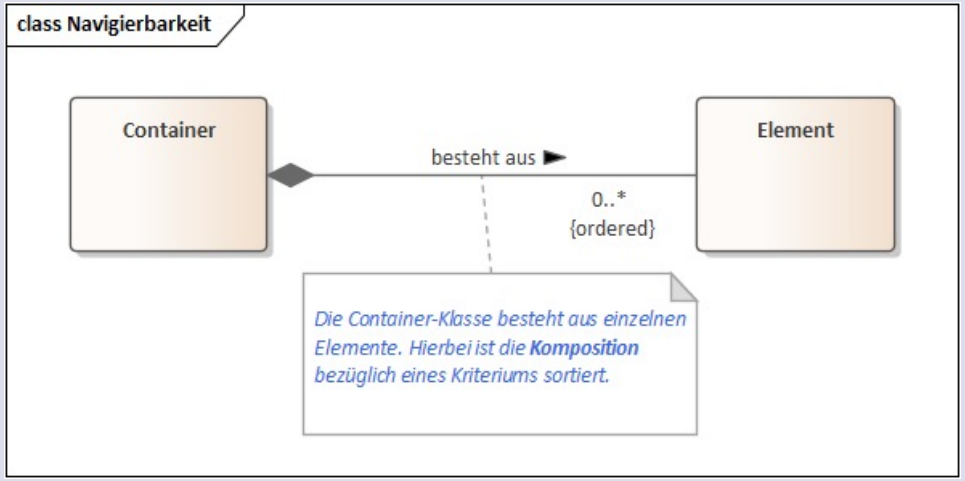
- Bei den bisherigen Assoziationen geht die Navigierbarkeit von einer Klasse aus (unidirektional).
- Grundsätzlich ist die Navigierbarkeit der Beziehung auch von beiden Klassen aus denkbar, wenn auch etwas komplizierter realisierbar. Man spricht dann von bidirektionalen Assoziationen.



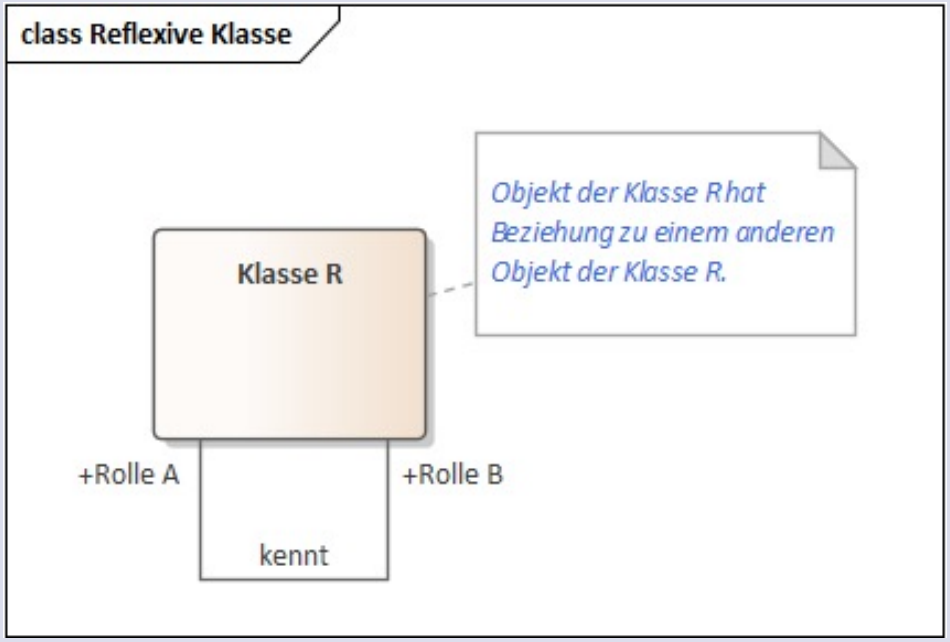
Objektorientierte Analyse und Design
UML: Aggregation und Komposition

- **Aggregation**: eine Ganzes-Teile-Beziehung, die Einzelteile sind jedoch auch ohne ein Gesamtobjekt existenzfähig.
  - Beispiel: Eisenbahnzug-Waggon
- **Komposition** (starke Aggregation): wie die Aggregation, die Einzelteile sind jedoch ohne ein Gesamtobjekt nicht existenzfähig oder verlieren ihre Eigenart.
  - Beispiel: Rechnung-Rechnungsposition

- Bemerkung: Es ist oft schwierig, zwischen Assoziation, Aggregation und Komposition zu entscheiden. Regel: Im Zweifelsfalle die schwächere Beziehungsart auswählen!

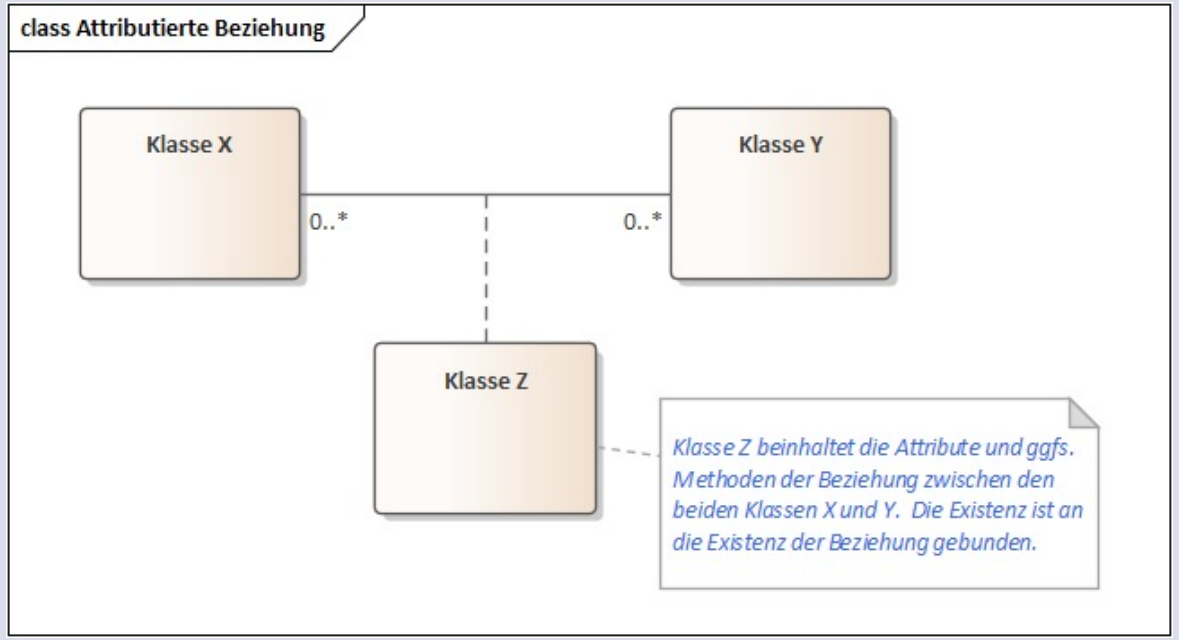


Objektorientierte Analyse und Design
UML: Reflexive Beziehung

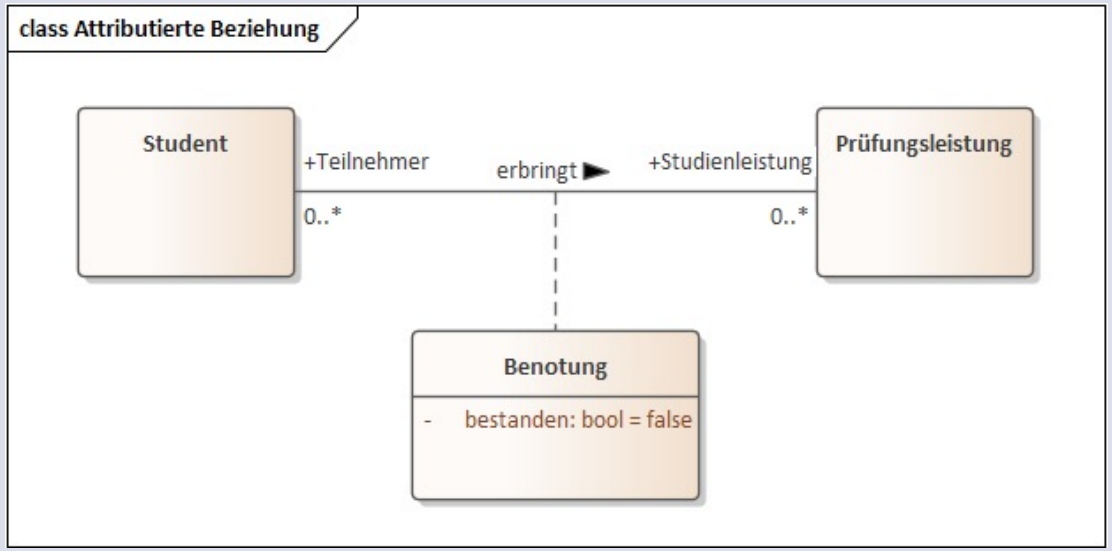


Objektorientierte Analyse und Design
UML: Attributierte Beziehungen

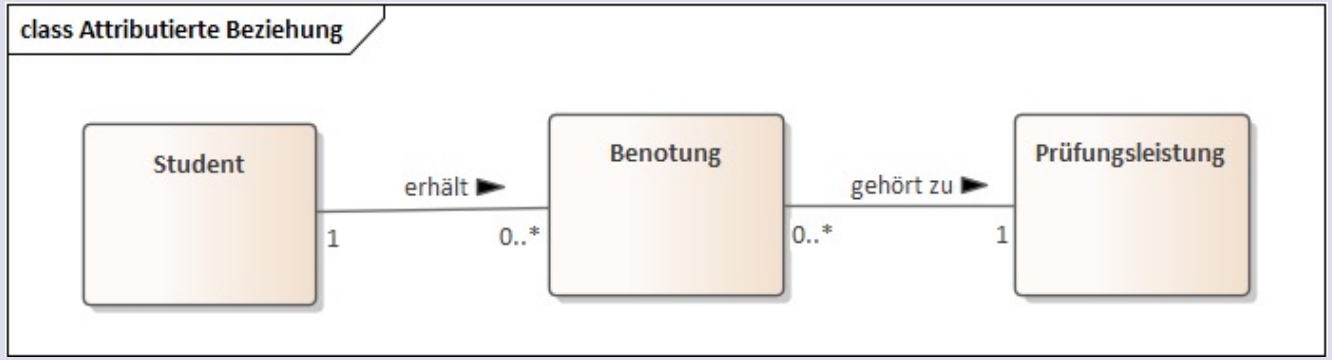
- Häufig kommen *-* Assoziationen vor. Diese können durch eine weitere Klasse aufgelöst werden: **Assoziationsklasse**.
- Weitere **Attribute** in der Assoziationsklasse können die Beziehung charakterisieren.



Objektorientierte Analyse und Design
Auflösung attributierter Beziehungen

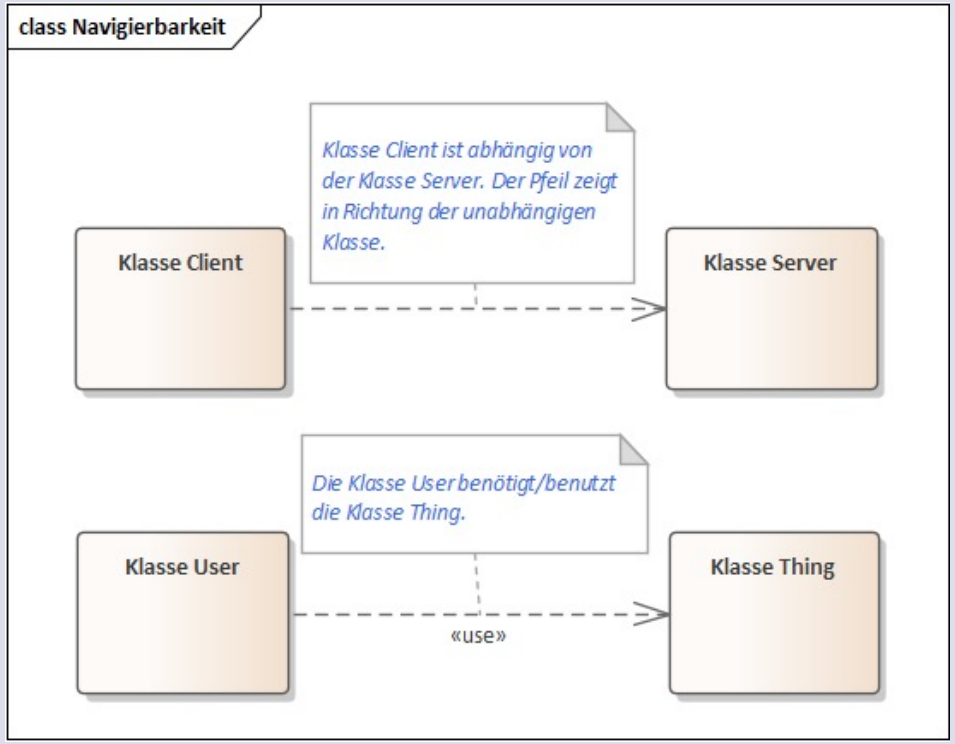


- Assoziationsklassen können oftmals in zwei gewöhnliche Assoziationen aufgelöst werden.



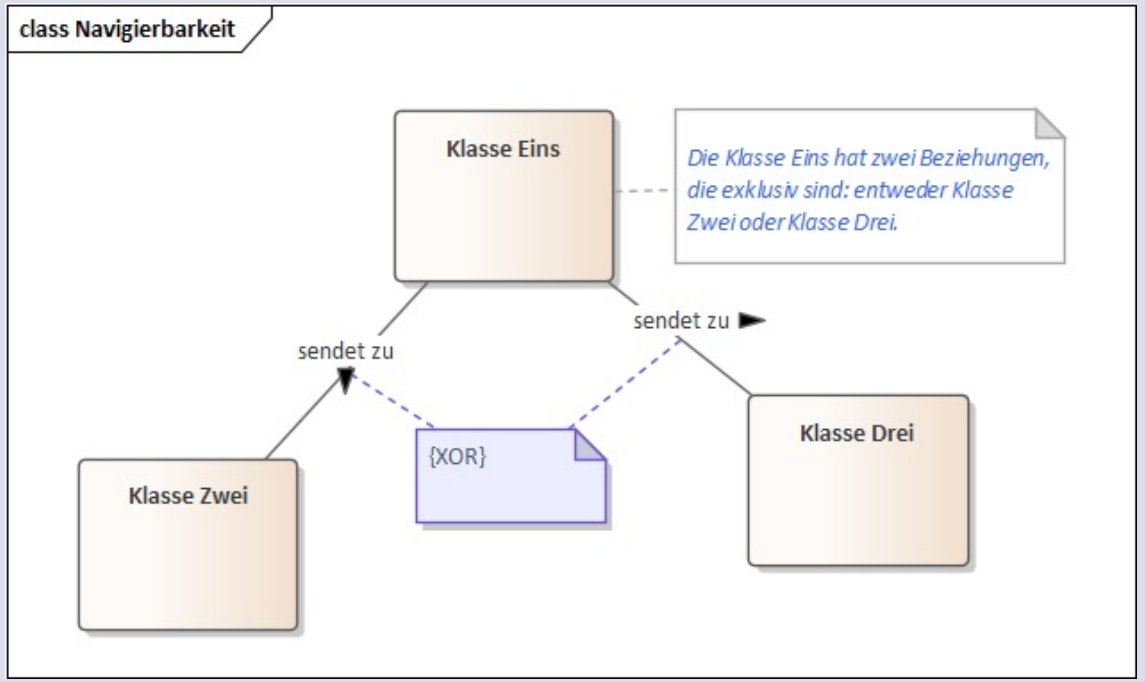
Objektorientierte Analyse und Design
UML: Abhängigkeiten

- Die **Abhängigkeit** einer Klasse A von einer anderen Klasse B ist eine spezielle Ausprägung einer Beziehung:
  - eine Methode von A ein lokales B-Objekt enthält,
  - eine Methode von A als Parameter ein B-Objekt bekommt,
  - eine Methode von A auf ein globales B-Objekt zugreift,
- ein Objekt der Klasse B verändert wird, so muss die Klasse A kontrolliert werden.



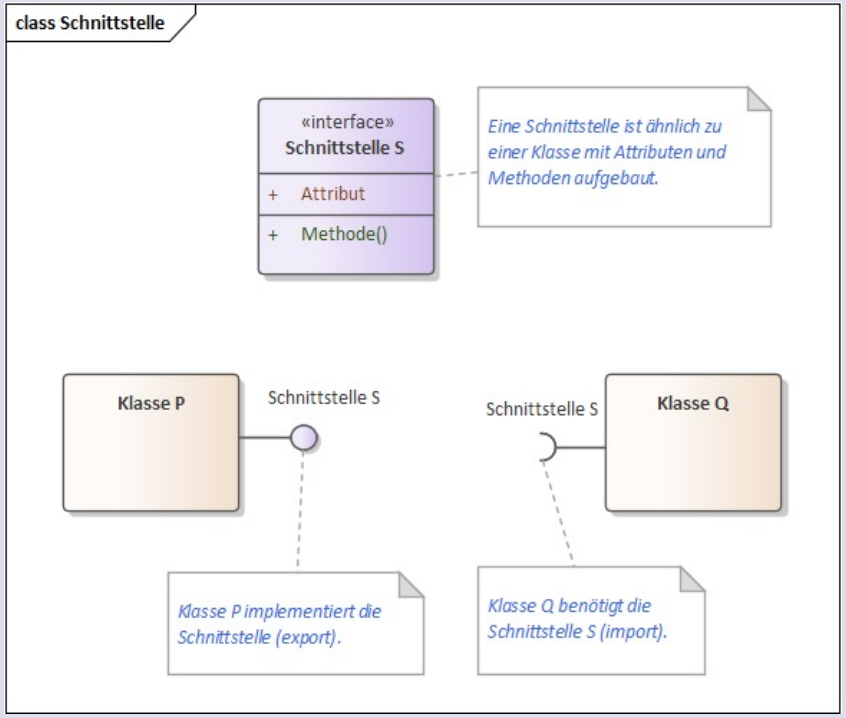
Objektorientierte Analyse und Design
UML: Abhängigkeiten

- Abhängigkeiten dienen einem bestimmten Zweck, der als Stereotyp charakterisiert werden kann.
  - **<<use>>** Klasse A nutzt B zur Implementierung seiner Methoden.
  - **<<permit>>** Klasse A darf private Elemente von B nutzen.
  - **<<create>>** Klasse A erzeugt Elemente von B.



Objektorientierte Analyse und Design
UML: Schnittstelle (Interface)

- Schnittstellen sind **Spezifikationen** des externen Verhaltens von Klassen.
- Diese enthalten eine Anzahl von Deklarationen für Operationen und Attributen.
- Die Klassen, die diese **Schnittstelle bereitstellen**, müssen diese Operationen und Attribute implementieren.
- Schnittstellen werden ähnlich wie Klassen notiert, tragen jedoch das Stereotyp **<<interface>>**.

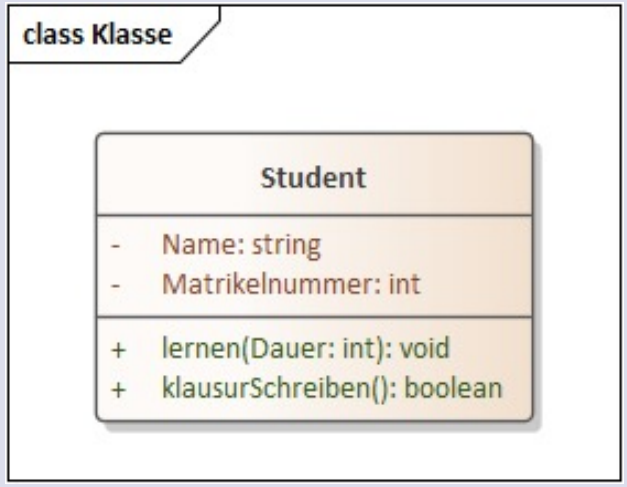


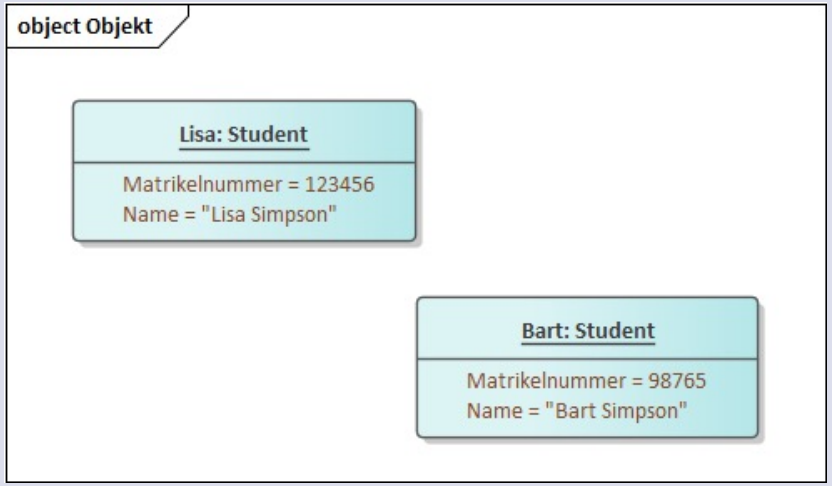
Objektorientierte Analyse und Design
UML: Schnittstelle (Interface)

- Über Schnittstellen wird einer neu zu schreibenden Klasse eine **Funktionalität** verordnet.
- Mehrere Klassen können dieselbe Schnittstelle realisieren.
- Mit Hilfe von Schnittstellen kann die Fähigkeit einer Klasse weitergegeben werden, **ohne Implementierungsdetails** offen zu legen. Die Schnittstelle dient dabei als Referenztyp.
- Eine Klasse kann mehrere Schnittstellen implementieren.
- Schnittstellen können von anderen Schnittstellen erben.
- Unterschiede zur **abstrakten Klasse**:
  - Abstrakte Klasse kann Realisierungen enthalten.
  - Eine davon abgeleitete Klasse gehört thematisch dazu.

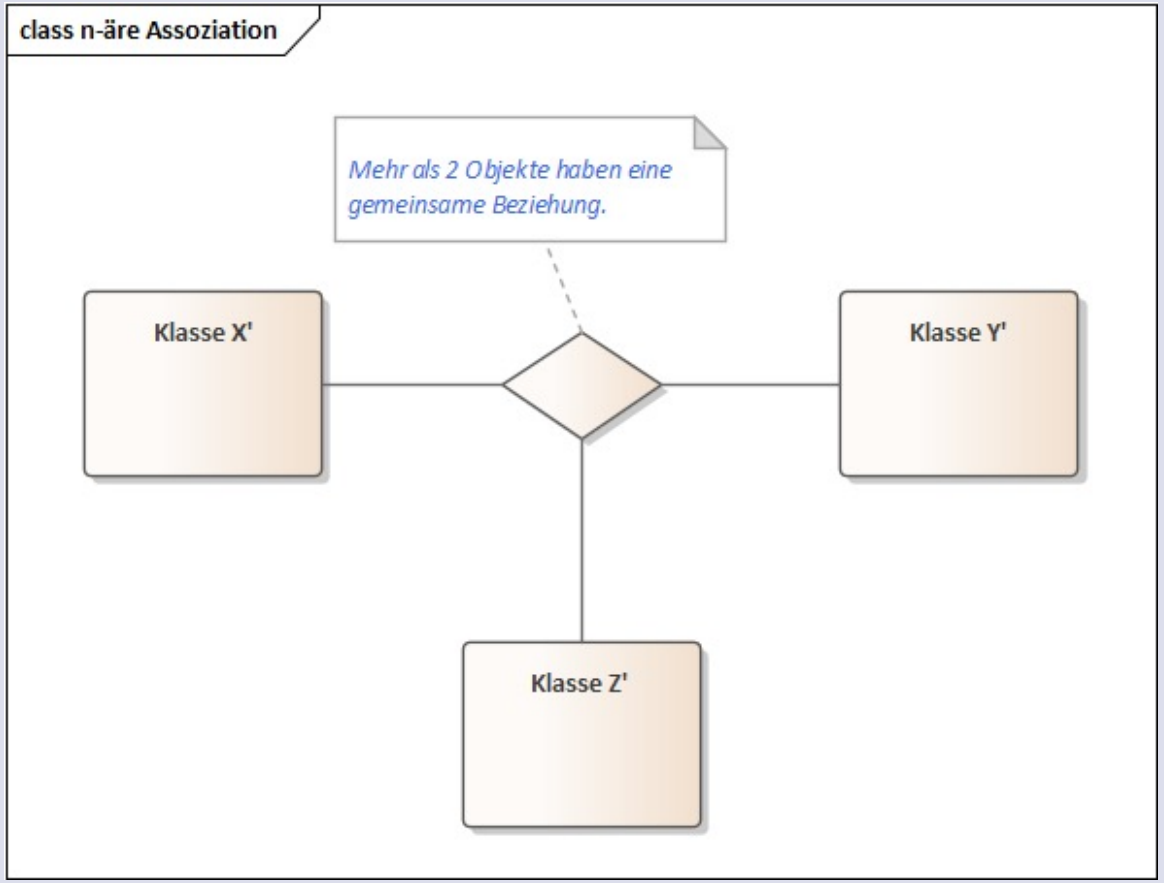
Objektorientierte Analyse und Design
UML: Objektdarstellung

- **Klasse** ist die Gesamtheit aller Ausprägungen (Instanzen, **Objekte**).
- **Objektdiagramm** stellt Moment–aufnahme eines Systems dar.
- Häufig nur „interessante“ Attribute.





Objektorientierte Analyse und Design
n-äre Beziehung



Objektorientierte Analyse und Design
Darstellungstiefe

- Nicht in jeder Phase sind alle Angaben im Klassendiagram erforderlich.
- In der **Analyse**
  - Attribute und Methoden nur mit Namen
- Im **Design**
  - Attribute mit Typangabe (wie oben)
  - Methoden nur mit Rückgabetyp
- Zur **Implementierung**
  - Attribute mit Typangabe (hinter dem Namen, abgetrennt durch ":")
  - Methoden mit Rückgabetyp und Parametern
  - statische Attribute zusätzlich mit Anfangswert

Objektorientierte Analyse und Design

INTERAKTIONSDIAGRAMM

Objektorientierte Analyse und Design
Interaktionsdiagramm

- Es gibt zwei Arten von UML-Interaktionsdiagrammen:
  - **Kommunikationsdiagramm**: hierarchische Nummerierung der Methodenaufrufe
  - **Sequenzdiagramm**: Die Objekte werden durch Lebenslinien mit Kennzeichnung der aktiven Zeiten dargestellt.
- Sequenzdiagramme beschreiben, wie Objekte bei anderen Objekten **Methoden aufrufen**.
- Mit Hilfe des **Klassenmodell**s lässt sich mit **Sequenzdiagramm**en validieren, ob die im **Aktivitäts–diagramm** beschriebenen Abläufe möglich sind.
- Sequenzdiagramme in der klassischen Form beschreiben damit **Beispielabläufe**.

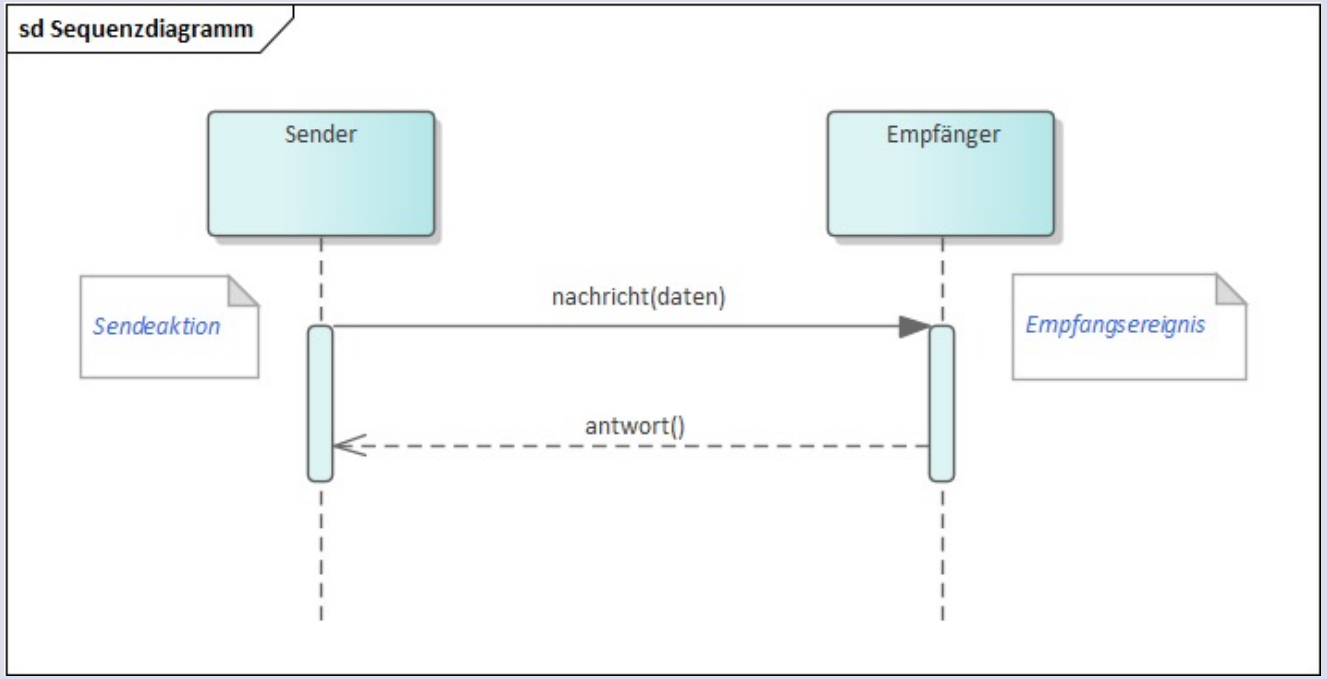
Objektorientierte Analyse und Design
Beschreibung von Interaktionen

- **Zusammenspiel** zwischen mehreren (i. allg. zwei) **Kommunikationspartnern**:
  - Nachrichten- und Datenaustausch
  - Unterschiedliche Granularitäten möglich: System, Komponente, Klasse, …
- Kommunikation ist Folge von Interaktionen (Trace).

- Grundelemente:
  - **Lebenslinien**
  - **Nachrichten** können sein:
    - Aufruf einer Operation/Methode
    - Antwort auf Aufruf
    - Signal (z.B. Zeitereignis)
    - …
- Erweiterung durch **kombinierte Fragmente**.
- Hier: Darstellung der **Interaktion von Objekten** (nicht Klassen)!

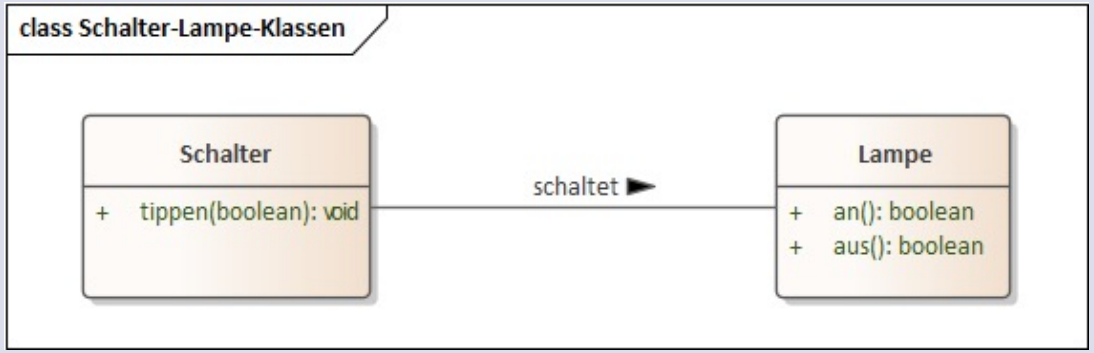
Objektorientierte Analyse und Design
Ereignismodell

- Nachrichtenübermittlung
  - **Sender**
    - Sendeaktion
  - **Empfänger**
    - Empfangsereignis
- Zeitliche **Entkopplung** zwischen Senden und Empfangen
  - Realistische Modellierung von verteilter Kommunikation möglich.

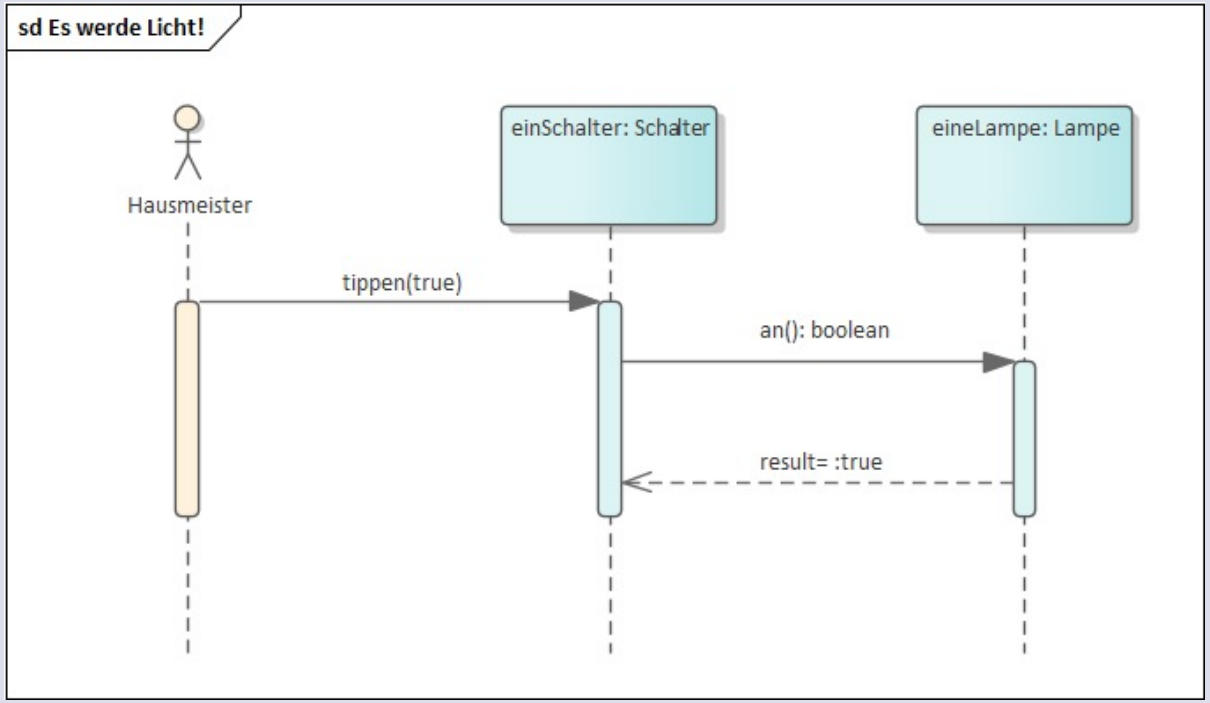


Objektorientierte Analyse und Design
Elemente Sequenzdiagramm

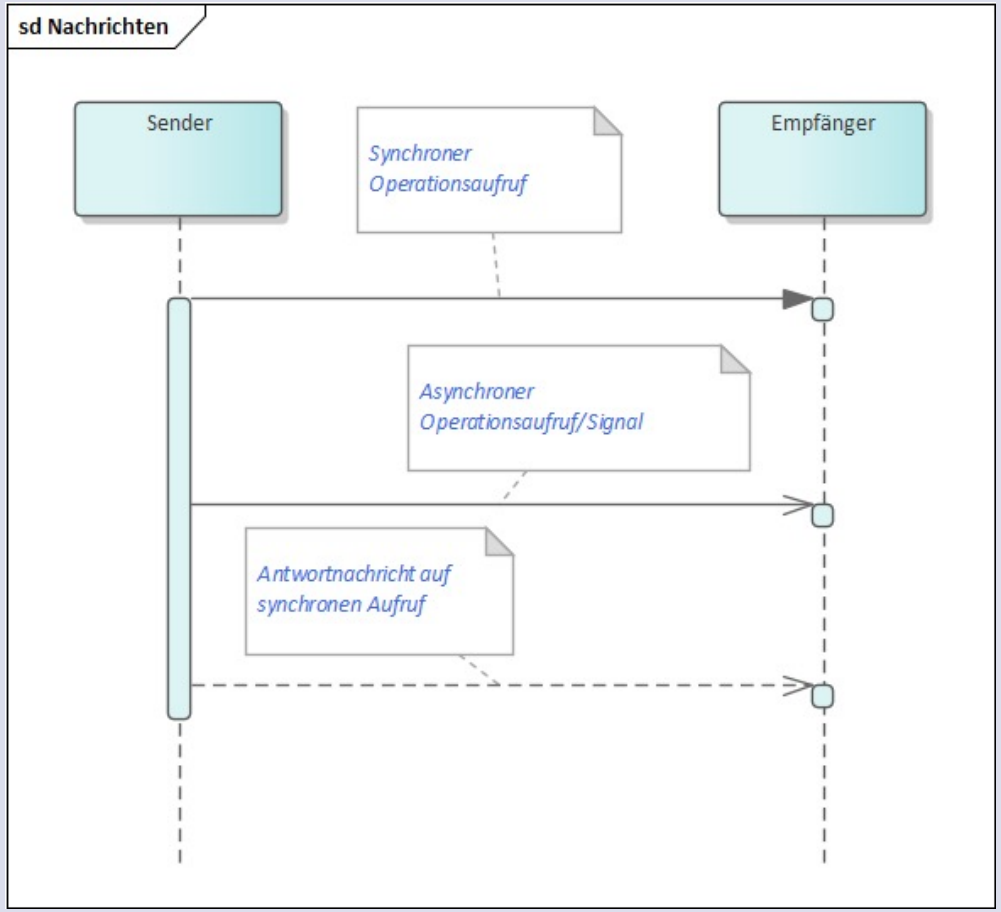
Klassendiagramm



Sequenzdiagramm

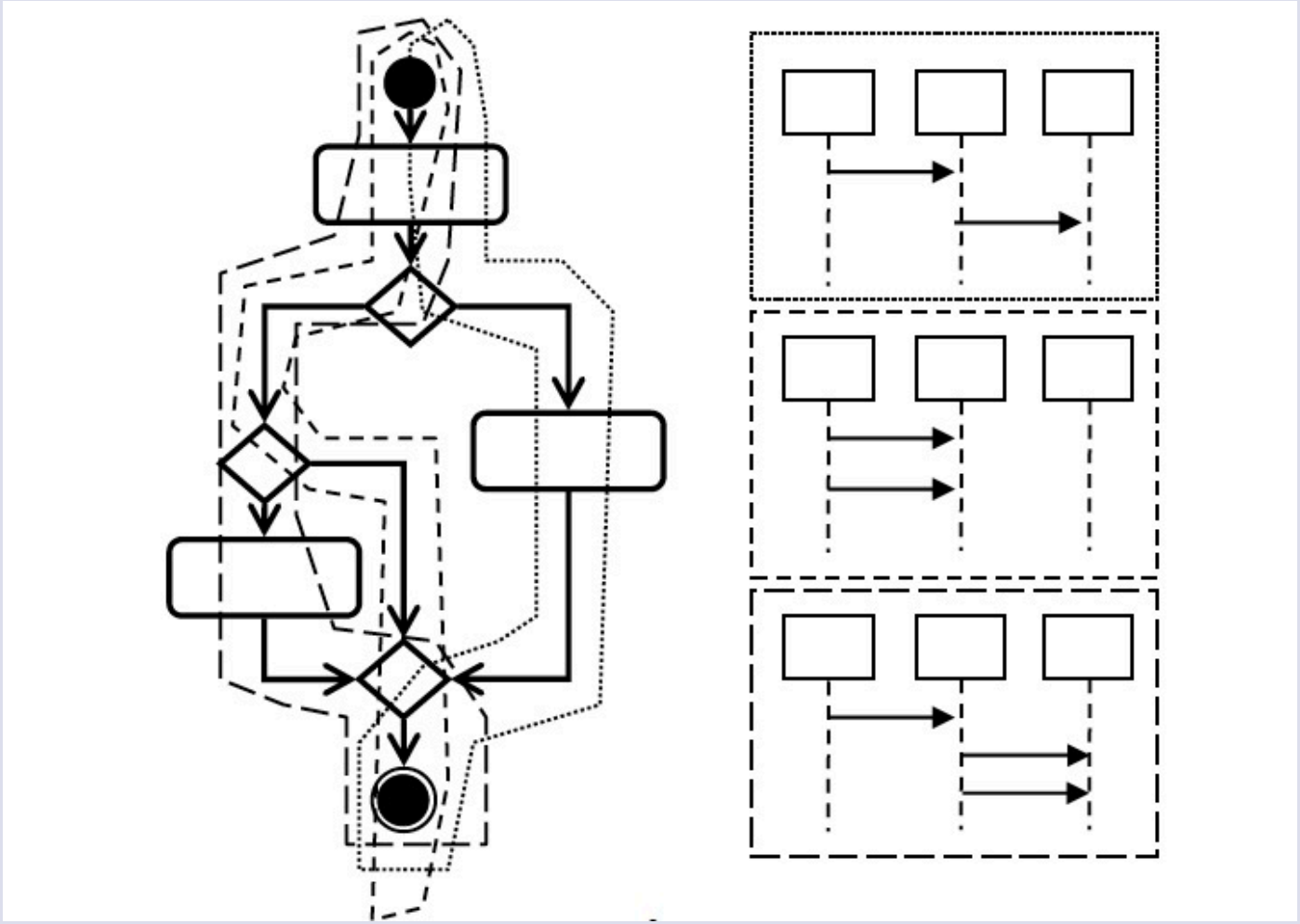


Objektorientierte Analyse und Design
Sequenzdiagramm: Nachrichten



- Modellierungsvarianten
  1. Definiertes **Szenario** (z. B. Anwendungsfall, Testfall) **mit konkreten Werten**.
  2. Menge von mehreren Szenarien (Abstraktion).
- Mischform möglich.

Objektorientierte Analyse und Design
Aktivitäts- und Sequenzdiagramm

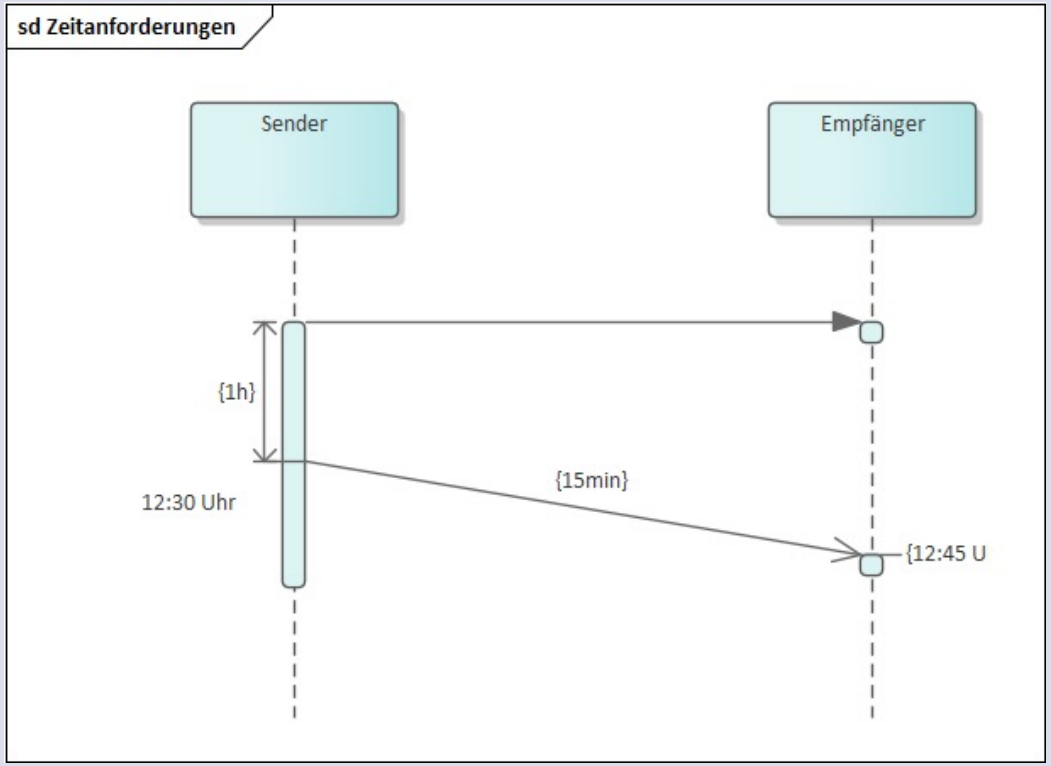


Modellierungsvariante 1

Quelle: Kleuker

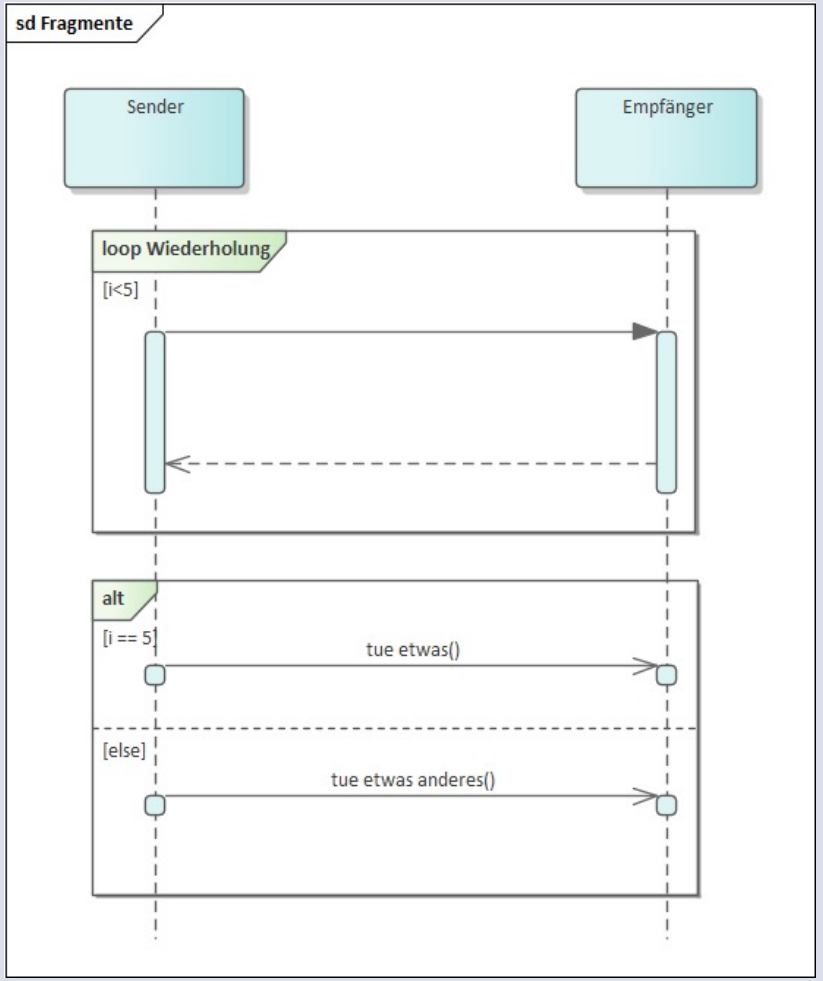
Objektorientierte Analyse und Design
Sequenzdiagramm: Zeitanforderungen

- Bisher: keine Aussage über Ausführungszeiten, Zeitpunkte etc.
- Erweiterung: **Modellierung von Zeitanforderungen**
  - Zeitpunkt/-intervall
    - 12.45 Uhr, [12.30…12.45] Uhr
  - Zeitdauer/-intervall
    - 5 Minuten, [5…10] Minuten



Objektorientierte Analyse und Design
Sequenzdiagramm: Fragmente

- Bisher: Beschreibung sequentieller Abläufe (“Geradeausfall“).
- Erweiterung: **Zusätzliche Operatoren** zur Beschreibung von Wiederholungen, Alternativen, Parallelitäten, …
- Erhöhung der Komplexität und Verringerung der Lesbarkeit.



Objektorientierte Analyse und Design
Iterative Entwicklung und Validierung

1. Ableitung von Methodennamen.
2. Zeichnen eines Sequenz–diagramms mit dieser Methode; feststellen, ob weitere Methoden benötigt werden.
3. Ergänzung von Methoden–parametern (Name, Typ).
4. Ergänzung des Sequenz–diagramms um Parameter; feststellen, ob weitere Methoden benötigt werden.

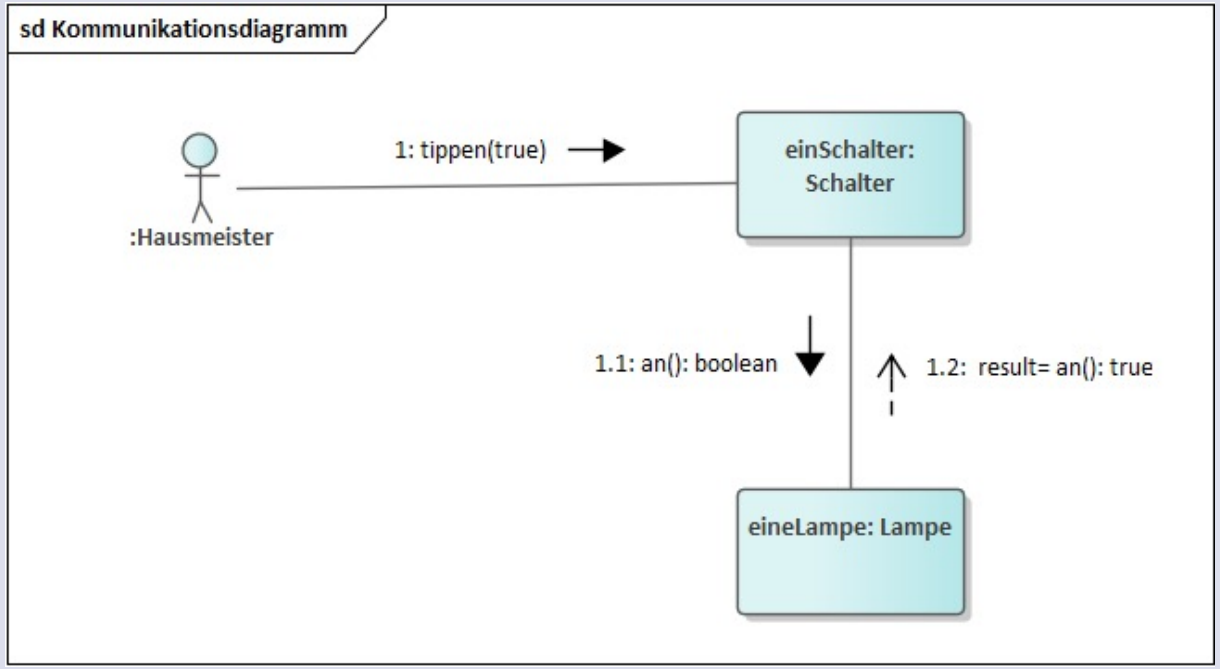
- Falls kein Sequenzdiagramm herleitbar, auf Ursachen–forschung gehen (Modellfehler?)
- Optimales Ziel: **Mögliche Durchläufe durch Aktivitätsdiagramme werden abgedeckt**.

Objektorientierte Analyse und Design
Message Sequence Charts (MSC)

- **Message Sequence Charts** werden seit langem in der Telekommunikation (Echtzeitsystemen) eingesetzt.
- Grafische und textuelle Notation (Grammatik) möglich.
- Standardisierung als Teil der Specification and Description Language (**SDL**).
- Mit UML 2 sind die Möglichkeiten der MSC übernommen worden.

Objektorientierte Analyse und Design
Kommunikationsdiagramm

- **Kommunikationsdiagramme** zeigen Interaktionen innerhalb einer (komplexen) Struktur an.
- Zeitliche **Reihenfolge** durch **Nummerierung der Nachrichten**.
- Eingeschränkte Ausdrucks–möglichkeiten gegenüber dem Sequenzdiagramm.
- UML 1.x: Kollaborationsdiagramm.



Objektorientierte Analyse und Design
Vorteil Sequenzdiagramm

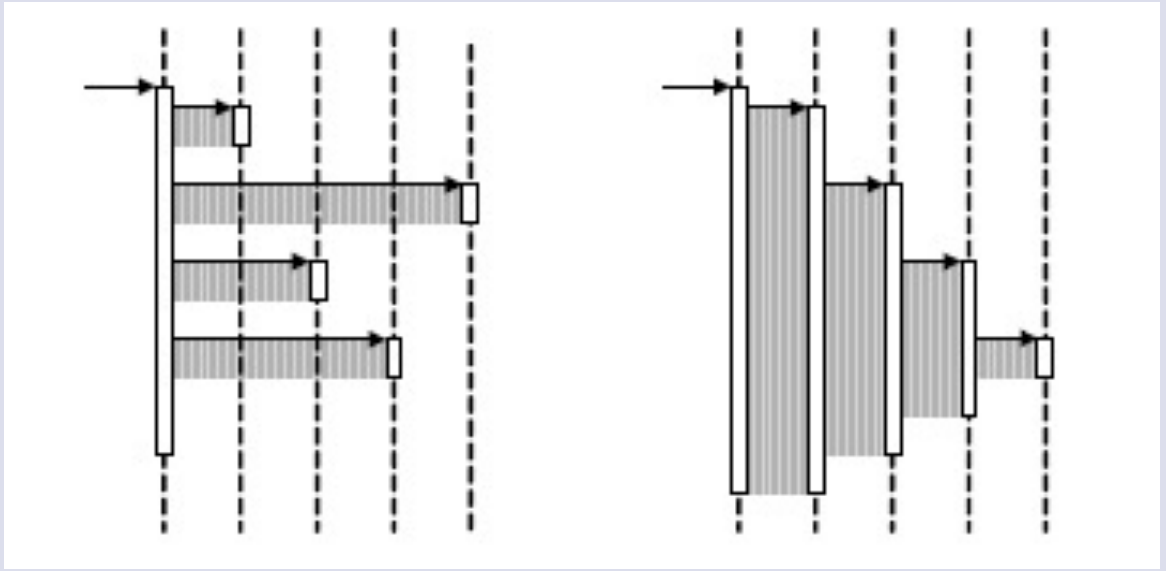
- Die Interaktionsdiagramme dienen also dazu, die **Verantwortlichkeiten der Aktivitäten** festzulegen. Pro Aktivität legt man ein Diagramm an.
- Im Rahmen der Erstellung der Sequenzdiagramme werden **weitere Methoden für Klassen** deklariert.
- Bei kleineren und mittleren Interaktionsdiagrammen sollte man die Form der Sequenz–diagramme wählen, diese sind
  - übersichtlicher,
  - bei der Erzeugung leichter zu behandeln als Kommunikations–diagramme und
  - betonen die Dynamik der Objekte.

Objektorientierte Analyse und Design
Vorteil Kommunikationsdiagramm

- Die Form der **Kommunikations–diagramm**e sollte man verwenden, wenn man eine **kompakte Darstellung** benötigt. Das Kommunikationsdiagramm betont das Zusammenspiel der Objekte.
- Notfalls wechsle man zwischen beiden Darstellungsarten hin und her. Sie sind was die Aufrufstruktur angeht nahezu semantisch äquivalent.
- Bei sehr großen Diagrammen nehme man eine Aufteilung vor, indem man ab einer Nachricht den gesamten weiteren Ablauf in ein weiteres Interaktionsdiagramm auslagert oder Referenzblöcke im Diagramm verwendet.
- Bis zu einem gewissen Grad kann aus den Sequenzdiagrammen Code als Grundlage zur weiteren Implementierung generiert werden. Sie können auch aus vorhandenem Code generiert werden.

Objektorientierte Analyse und Design
Delegation 1

- Am Sequenzdiagramm erkennt man sehr schön die Verteilung und die Aufrufstruktur einer Methode.
- Zwei extreme Fälle, die auftreten können, sind:
  - Zentral
  - Dezentral



Zentral **Dezentral**

Objektorientierte Analyse und Design
Delegation 2

Dezentrale Struktur

- Jedes Objekt kennt nur einige wenige andere Objekte und weiß, welches ihm zu dem Zweck helfen kann, seine Aufgabe erledigen zu können.
- Wie im Detail die Funktion gelöst wird, ist dem Objekt egal. Die Delegation erfolgt dezentral.

Zentrale Struktur

- Ein Objekt kennt viele andere Objekte und weiß um deren Möglichkeiten.
- Ein Objekt hat das Wissen über den Ablauf der Funktion und kontrolliert die Teilaufgaben. Die Delegation der Verantwortlichkeiten erfolgt zentral.

Objektorientierte Analyse und Design
Delegation 3

- Welche der beiden Strukturen ist besser?
- Die Stufenstruktur (also: dezentrale Delegation) ist „objektorientierter“ und berücksichtigt besser die Beziehungen des vorhanden–en Klassendiagramms. Das muss aber nicht immer die bessere Lösung sein.
- Muss man z.B. in Zukunft die Reihenfolge von Aktionen ändern, dann ist dafür eine zentralisierte Struktur besser. Änderungen sind dann nur lokal in der zentralen Klasse notwendig.
- Auch eine Mischform kann gut sein. Die Verantwortlichkeits-muster (später) helfen, die richtige Form zu bekommen.

Objektorientierte Analyse und Design

ZUSTANDSDIAGRAMM

Objektorientierte Analyse und Design
Motivation

- Beobachtung: Das **Verhalten** von Personen, Tieren und Maschinen ist häufig abhängig von
  - der momentanen Situation (z.B. Umwelt)
  - Ereignissen in der Vergangenheit
- Modellierung: Für Objekte kann das Vorhandensein unterschiedlicher **Zustände** von Bedeutung sein.
- Das zustandsabhängige Reagieren auf Ereignisse wird in Zustandsautomaten beschrieben.
- Sie eignen sich für die Darstellung des dynamischen Verhaltens.

- Ein Mobiltelefon besitzt die Zustände
  - eingeschaltet und frei
  - ausgeschaltet
  - eingeschaltet und in Betrieb
  - gesperrt (nicht angemeldet oder Konto abgelaufen)
- Ein Auftrag besitzt z.B. die Zustände
  - angeboten
  - erteilt
  - abgeschlossen
  - storniert
  - abgerechnet
  - reklamiert

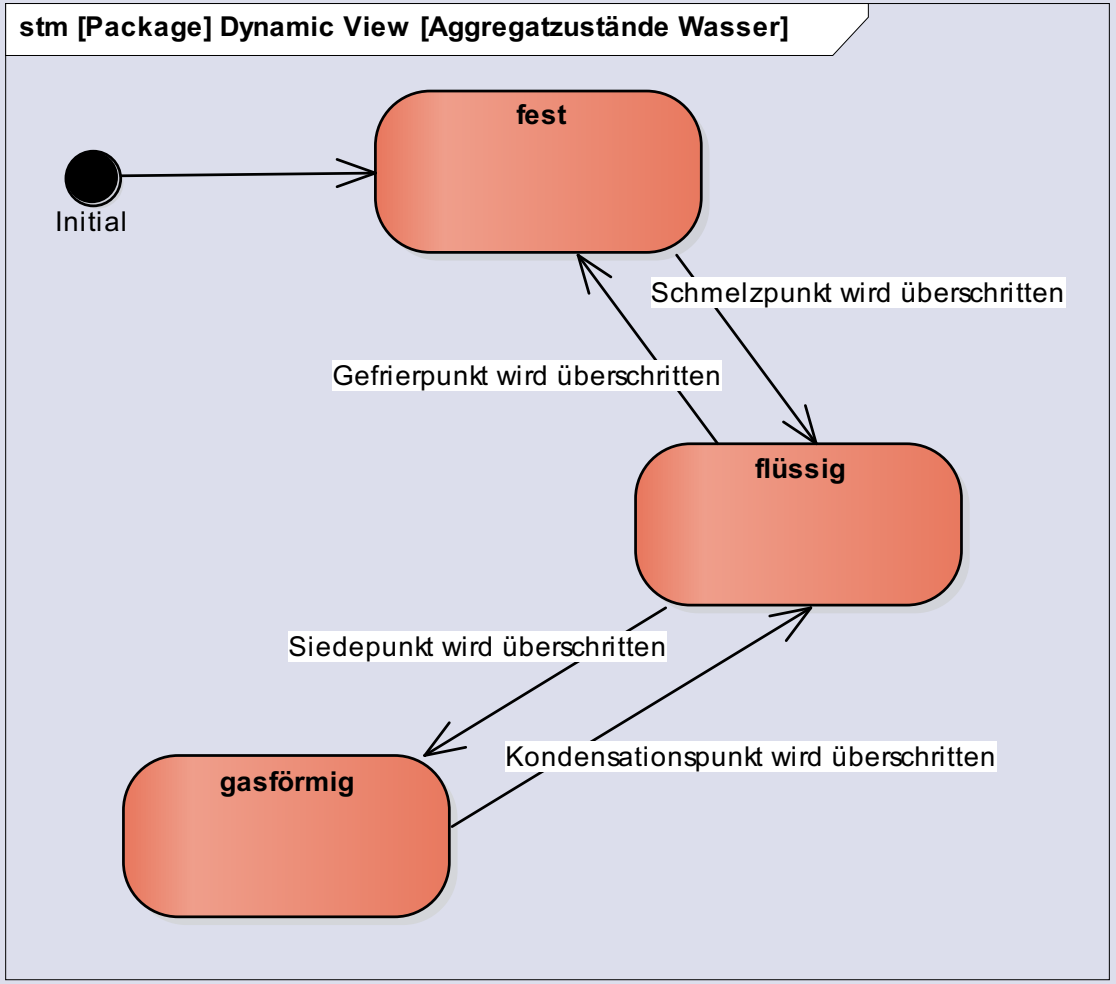
Objektorientierte Analyse und Design
Zustandsautomaten

- Für welche Objekte ist eine Zustandsmodellierung sinnvoll?
  - Objekte, die **bedeutsam für das Gesamtsystem** oder zumindest Teile davon sind.
  - Objekte, deren **Dienste** stark vom Zustand **abhängig** sind
  - Hinweis: Nicht jede **Änderung eines Attributwertes** des Objektes wird als Zustands–änderung angesehen.
- Ein **Zustandsdiagramm** zeigt das **diskrete** Verhalten einer Instanz einer Klasse („Lebensgeschichte“) und besteht im Wesentlichen aus
  - Zuständen und deren
  - Übergängen (Transitionen)
- Für wichtige Objekte mit mehreren bedeutsamen Zuständen werden in der Analyse oder im Software-Enwturf in einem UML-Zustandsdiagramm dargestellt.

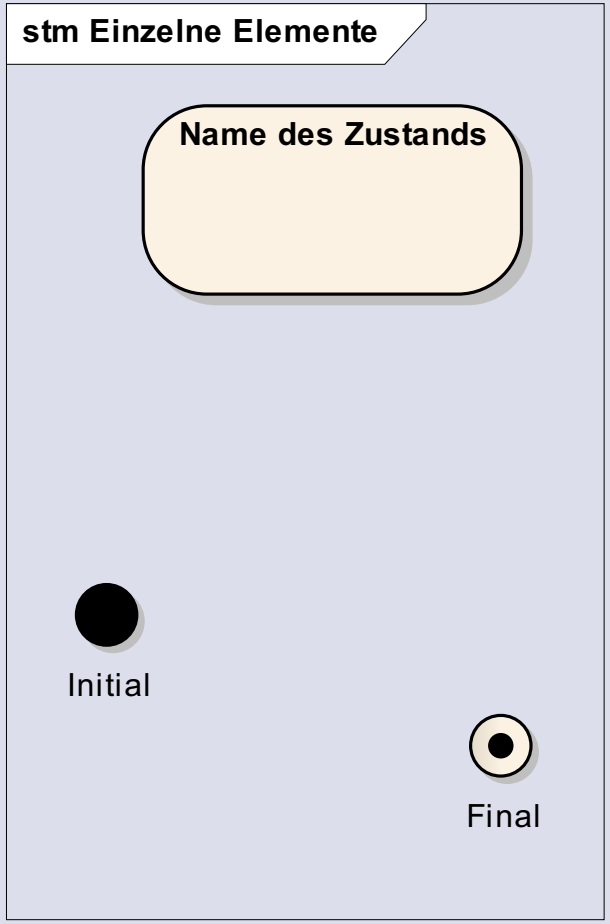
Objektorientierte Analyse und Design
Was ist ein Zustand?

- Definition:
  - Ein Zustand ist eine unterscheidbare Eigenschaft eines Objekts, zu dem auch ein spezifisches Verhalten gehört.
  - Objekte verharren in einem Zustand, können aber durch Ereignisse in einen anderen wechseln.
- Zustandsdiagramme veranschaulichen das dynamische Verhalten von (Teil-)Systemen bzw. Objekten.
- Das dynamische Verhalten beschreibt Änderungen über die Zeit.

Objektorientierte Analyse und Design
Beispiel Aggregatzustände Wasser

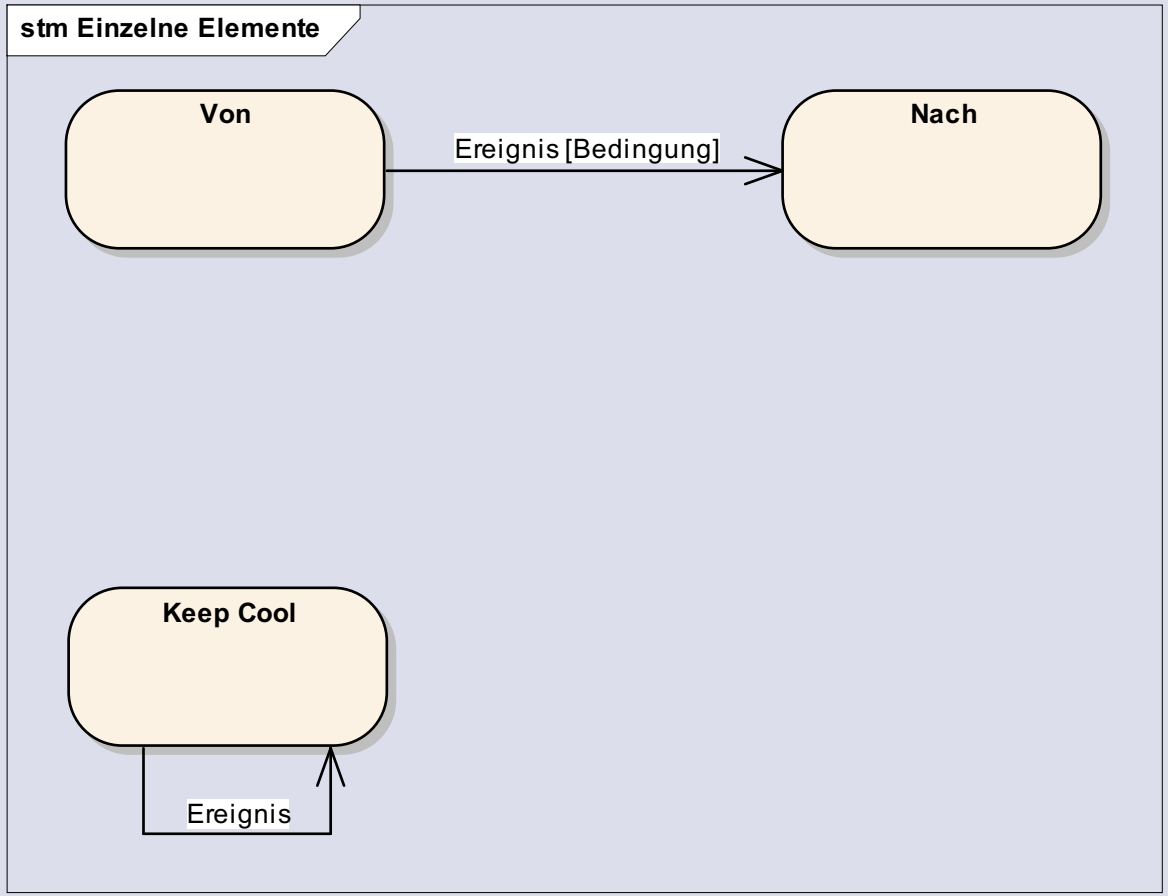


Objektorientierte Analyse und Design
Darstellungselemente in UML



- Zustände beschreiben die momentane Situation in einem System. Je nach Situation (Zustand) reagiert das System unterschiedlich auf interne oder externe Ereignisse. Es geht in einen anderen Zustand über.
- Start (Initial)-/Endpunkte (Final) zeigen, in welchen Zustand ein System beim Starten gelangt bzw. aus welchem es beendet wird.

Objektorientierte Analyse und Design
Darstellungselemente in UML

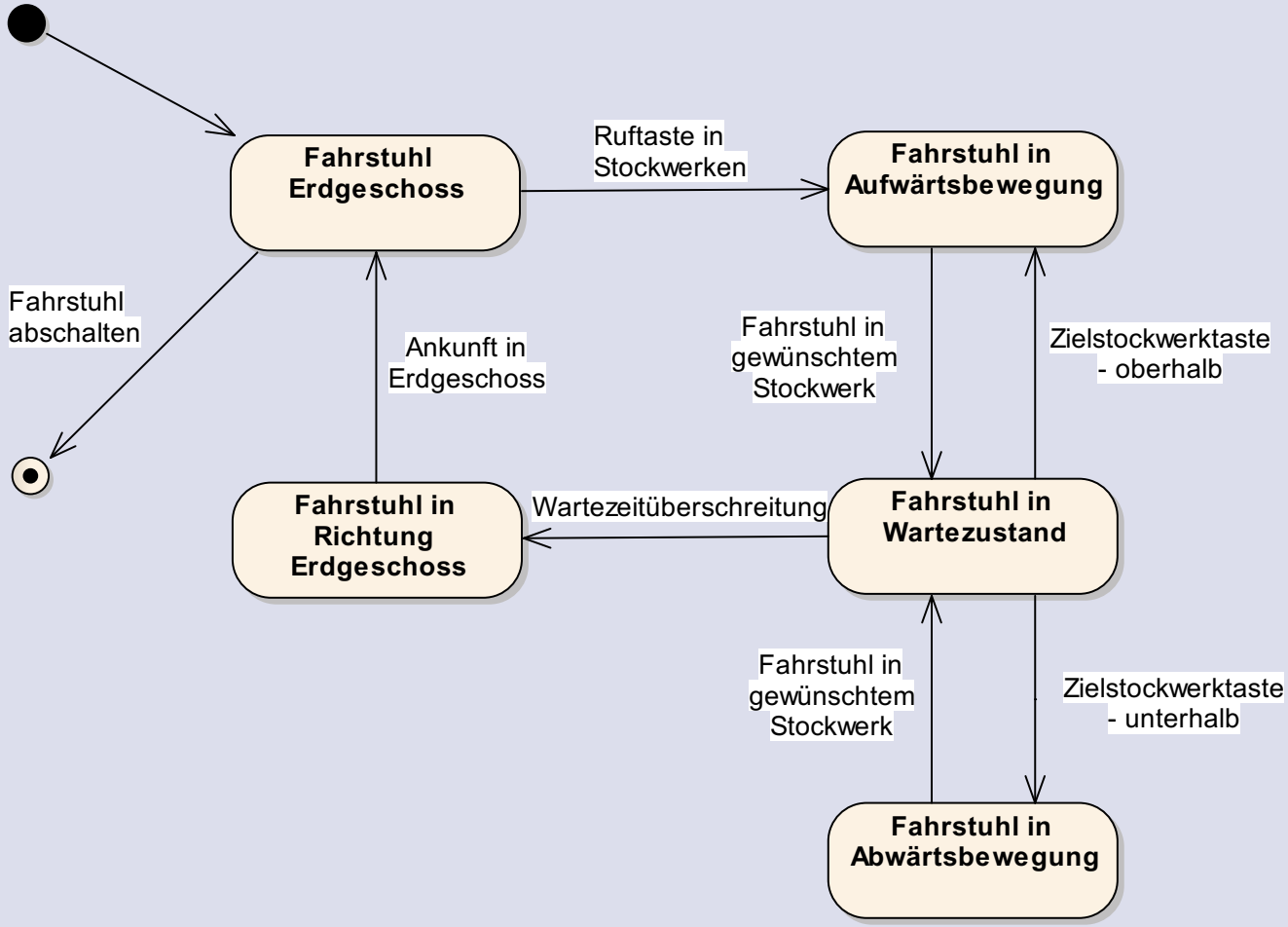


- Transitionen sind Übergänge, die durch interne oder externe Ereignisse ausgelöst werden. Transitionen sind von kurzer Dauer, nicht unterbrechbar und können von Bedingungen abhängig sein.
- Transitionen können zum Verbleiben im Zustand führen (notwendig, wenn Reaktion auf wichtige Ereignissen explizit auszudrücken ist).

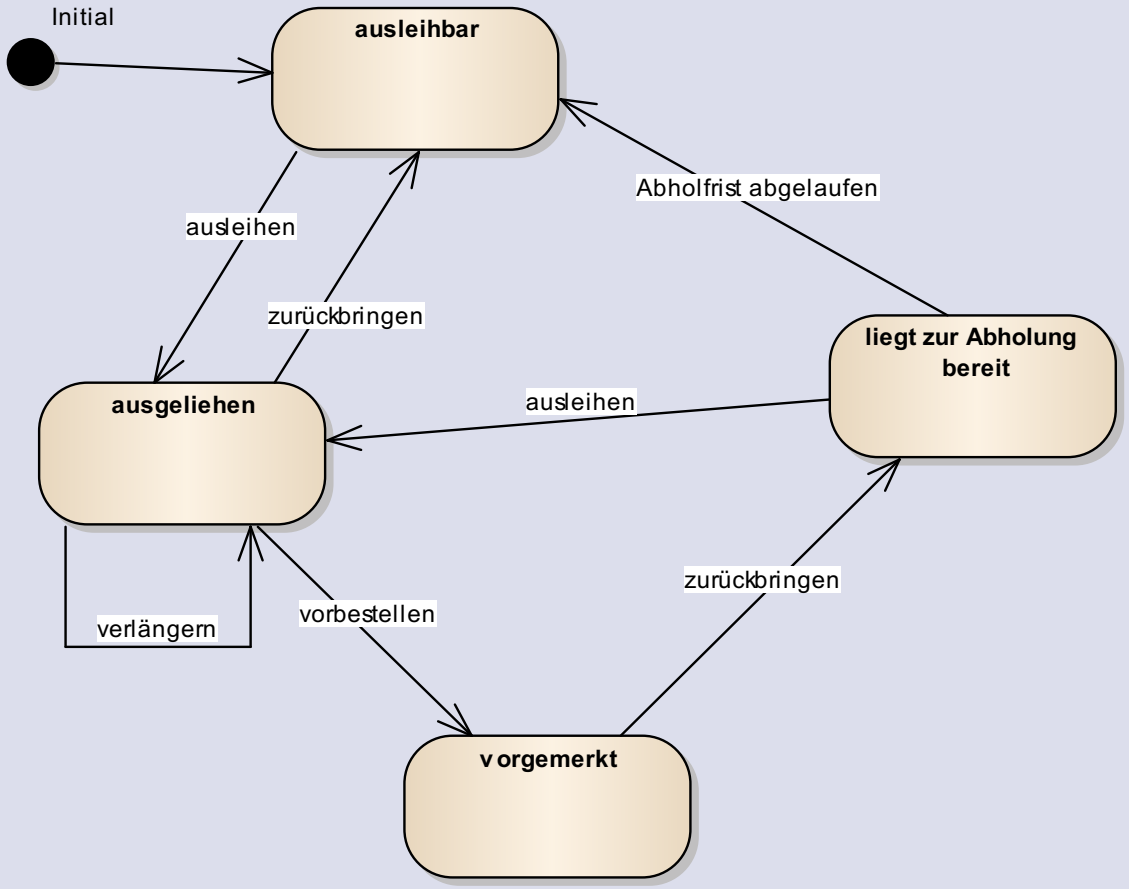
Objektorientierte Analyse und Design
Zustandsübergang (Transition)

- Eine Zustandsänderung beschreibt einen Wechsel von einem Zustand in den nächsten unter Angabe eines **Ereignis**ses (trigger) und einer evtl. vorhandenen **Bedingung** (guard).
- Ereignisse dienen dem **Auslösen** der Zustandsänderung.
- Prinzipiell müssen in jedem Zustand alle möglichen Ereignisse bedacht werden.
- Es gibt folgende Möglichkeiten:
  - Ein Ereignis gehört zum normalen Ablauf und wird explizit angegeben. Auch wichtige Ereignisse ohne Zustandsänderung werden explizit angegeben.
  - Ein Ereignis entspricht einer wichtigen Fehlersituation. Sie wird ebenfalls angegeben und führt zur Fehlerbehandlung in einem neuen Zustand.

Objektorientierte Analyse und Design
Zustandsautomat eines Fahrstuhls



Objektorientierte Analyse und Design
Beispiel Hochschulbibliothek



Objektorientierte Analyse und Design
Darstellung als Tabelle

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- Zustandsmaschinen lassen sich neben einer grafischen UML-Darstellung auch in einer Tabelle angeben.

State \ Next State		Initial	ausleihbar	ausgeliehen	vorgemerkt	liegt zur Abholung bereit
		S0	S1	S2	S3	S4
Initial	S0					
ausleihbar	S1			ausleihen		
ausgeliehen	S2		zurückbringen	verlängern	vorbestellen	
vorgemerkt	S3					zurückbringen
liegt zur Abholung bereit	S4		Abholfrist a...	ausleihen		

Objektorientierte Analyse und Design Implementierung in C/C++

- Zustandsmaschinen sind eine häufig eingesetzte Methode, die nicht nur bei der objekt-orientierten Software-Entwicklung zum Einsatz kommt.
- In der Sprache C/C++ lassen sich Zustandsmaschinen mit 2 geschachtelten `switch-case`-Anweisung effizient implementieren.

```
int accept(int event,int source) {
 int target=source;
 switch (source) {
  case AUSGELIEHEN :
   switch (event) {
    case ZURUECKBRINGEN :
     target=AUSLEIHBAR;
     break;
    case VORBESTELLEN :
     target=VORGEMERKT;
     break;
   }
   break;
  case VORGEMERKT :
   switch (event) {
    […]
   }
 return target;
}
```

Objektorientierte Analyse und Design
Tipp Zustandsdiagramm

- Benennen Sie den Zustand als **Adjektiv** (Eigenschaftswort, Wiewort) oder als **Partizip**.
- Beispiele:
  - frisch, sauer, süß, schön, …
  - stehend, fahrend, wartend, …

Objektorientierte Analyse und Design
Checkliste für Zustandsdiagramme

Frage	Hinweise
Sind die Zustände eindeutig benannt?	Jeder Zustand darf nur einmal dargestellt werden.
Sind die Transitionen eines Zustandes deterministisch?	Ein Ereignis darf nicht mehr als eine Transition auslösen.
Beschreiben die Zustände Eigenschaften und keine Aktivitäten?	Objekte verharren in einem Zustand bis ein Ereignis auftritt.
Gibt es genau einen Anfangszustand?	Es muss genau einen initialen Zustand geben.
Hat jede Transition ein auslösendes Ereignis (ggfs. Bedingung)?	Jeder Zustandsübergang muss durch ein Ereignis veranlasst werden.
Sind in jedem Zustand alle möglichen Ereignisse berücksichtigt?	Jedes Ereignis ist auf einen Zustandswechsel zu untersuchen.
Ist ein Endzustand möglich?	Ein definiertes Ereignis beendet die Zustandsmaschine.
Ist jeder Zustand durch eine Transition erreichbar/verlassbar?	Für jeden Zustand muss es eine eintretende/eine austretende Transition geben. Ausnahmen sind der Anfangs- und der Endzustand (ggfs. definierter Fehlerzustand).

Objektorientierte Analyse und Design

MEHR DETAILS

Objektorientierte Analyse und Design
Verhalten (Aktivitäten und Aktionen)

- Die bisherige Notation ist häufig für die erste Analyse und den ersten Entwurf hinreichend.
- Im zweiten Schritt wird das Verhalten in Form von **Aktivitäten** und **Aktionen** beschrieben.
- Das **Zustandsdiagramm** wird „angereichert“ und bietet Zusammenspiel mit dem **Aktivitätsdiagramm**.
- Vorher noch etwas Theorie!

Objektorientierte Analyse und Design
Mealy- und Moore-Automaten

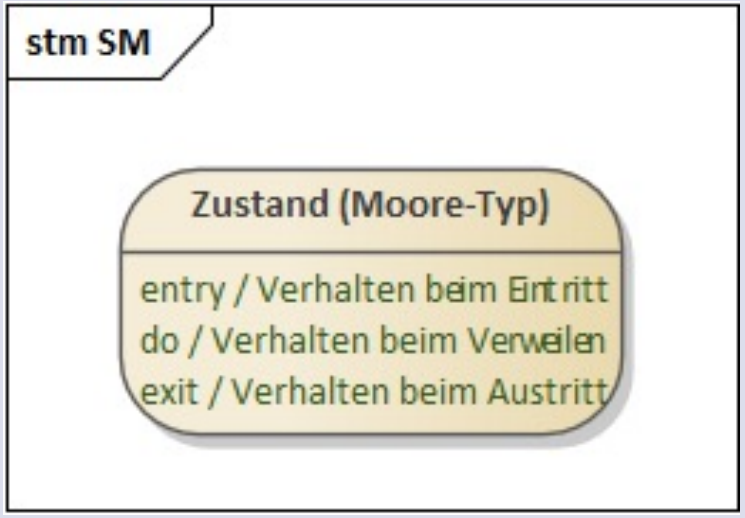
- Ein UML-Zustandsdiagramm ist eine Variante eines **endlichen deterministischen Automaten** (DEA):
  - Ausgabe (Aktion) abhängig vom Zustand und Ereignis, also von der Transition: **Mealy**-Automat.
  - Ausgabe (Aktion) abhängig vom Zustand: **Moore**-Automat.
  - Sind äquivalent (umwandelbar).
- UML erlaubt beide Schreibweisen und die Kombination von beiden.
- Aktivitäten/Aktionen sind möglich:
  - auf der **Transition**,
  - beim **Ein- oder Austritt** und
  - **während** Objekt im Zustand ist.

Objektorientierte Analyse und Design
Details: Transition und Trigger

- Der Übergang zwischen zwei Zuständen dauert **konzeptionell keine Zeit** und ist **nicht unterbrechbar**.

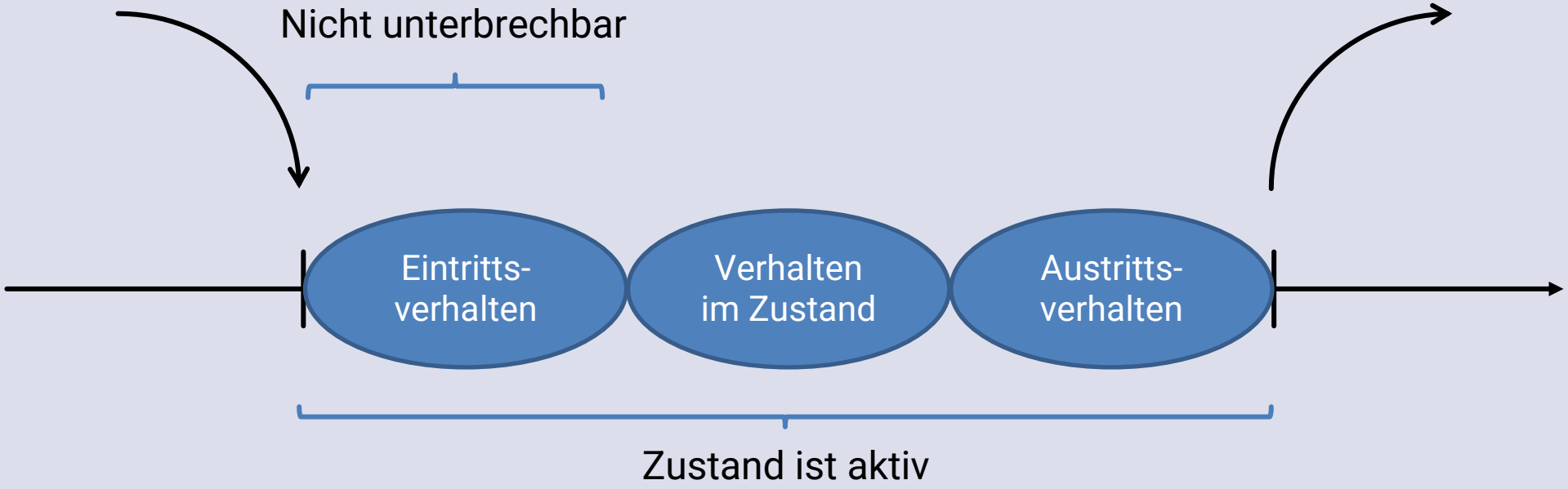
- CallTrigger: **Methoden- oder Funktionsaufruf**
- SignalTrigger: Eintretendes **Ereignis**
- ChangeTrigger: Wert einer **Variablen** (Attribut) **ändert** sich
- TimeTrigger: **Ablauf** eines Intervalls oder Zeitpunkt
- AnyTrigger: **Beliebiger** Auslöser, der nicht von anderen abdeckt wird.

Objektorientierte Analyse und Design
Details: Zustand



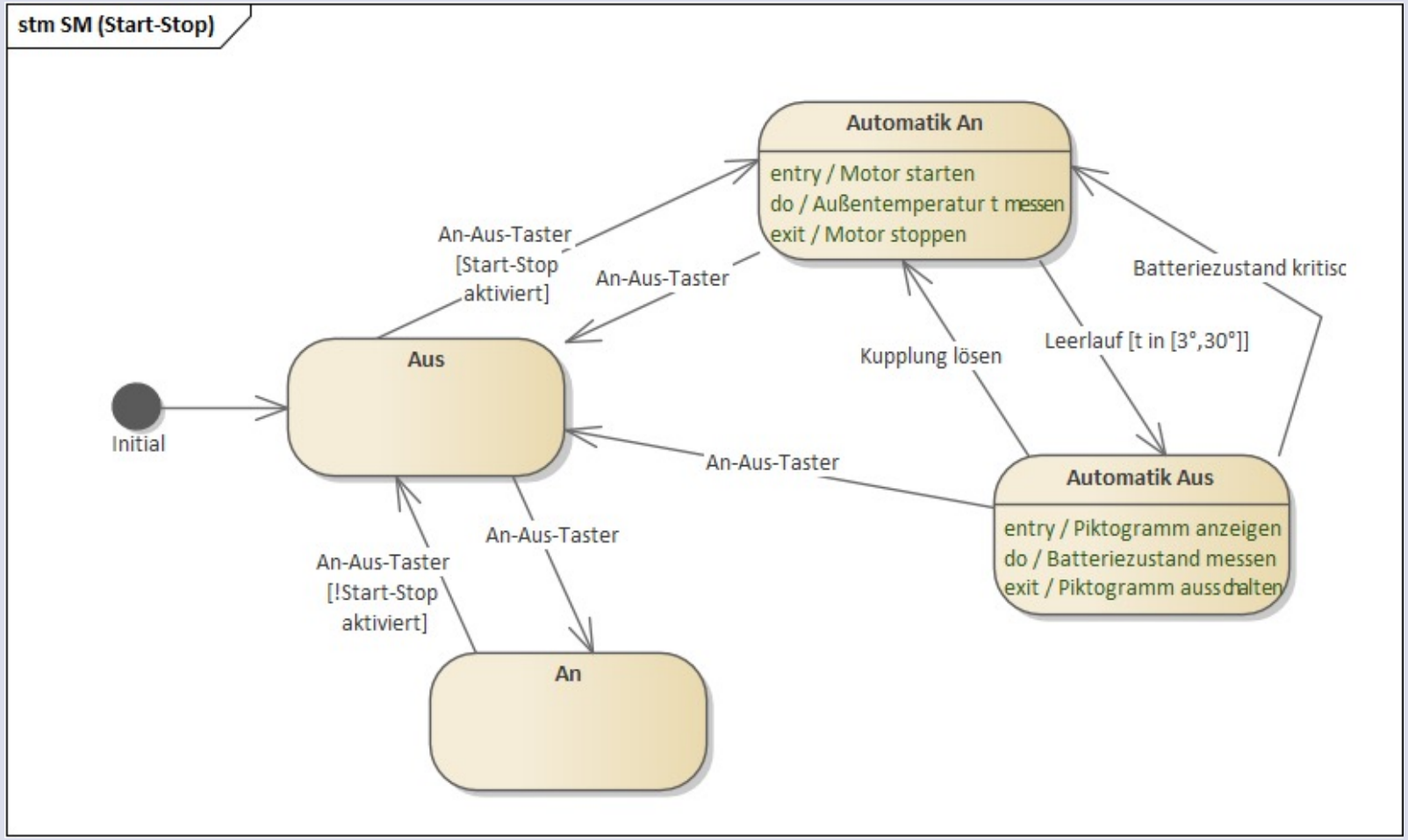
- Aktivitäten/Aktionen werden unterschieden:
  - beim Eintritt (**entry**)
  - beim Austritt (**exit**)
  - beim Verweilen im Zustand (**do**)
- Dieses Verhalten kann mit den **Aktivitätsdiagrammen** dokumentiert werden.

Objektorientierte Analyse und Design
Zeitlicher Ablauf



- Mögliche Probleme:
  - Ereignisse (Trigger) verfallen, wenn keine Transition gültig ist.
  - Ein-/Austrittsverhalten dauert lange; Ereignisse laufen auf.
  - Nichtdeterminismus bei parallelen/nebenläufigen Zustandsmaschinen.

Objektorientierte Analyse und Design
Beispiel: Start-Stop-Automatik (einfach)

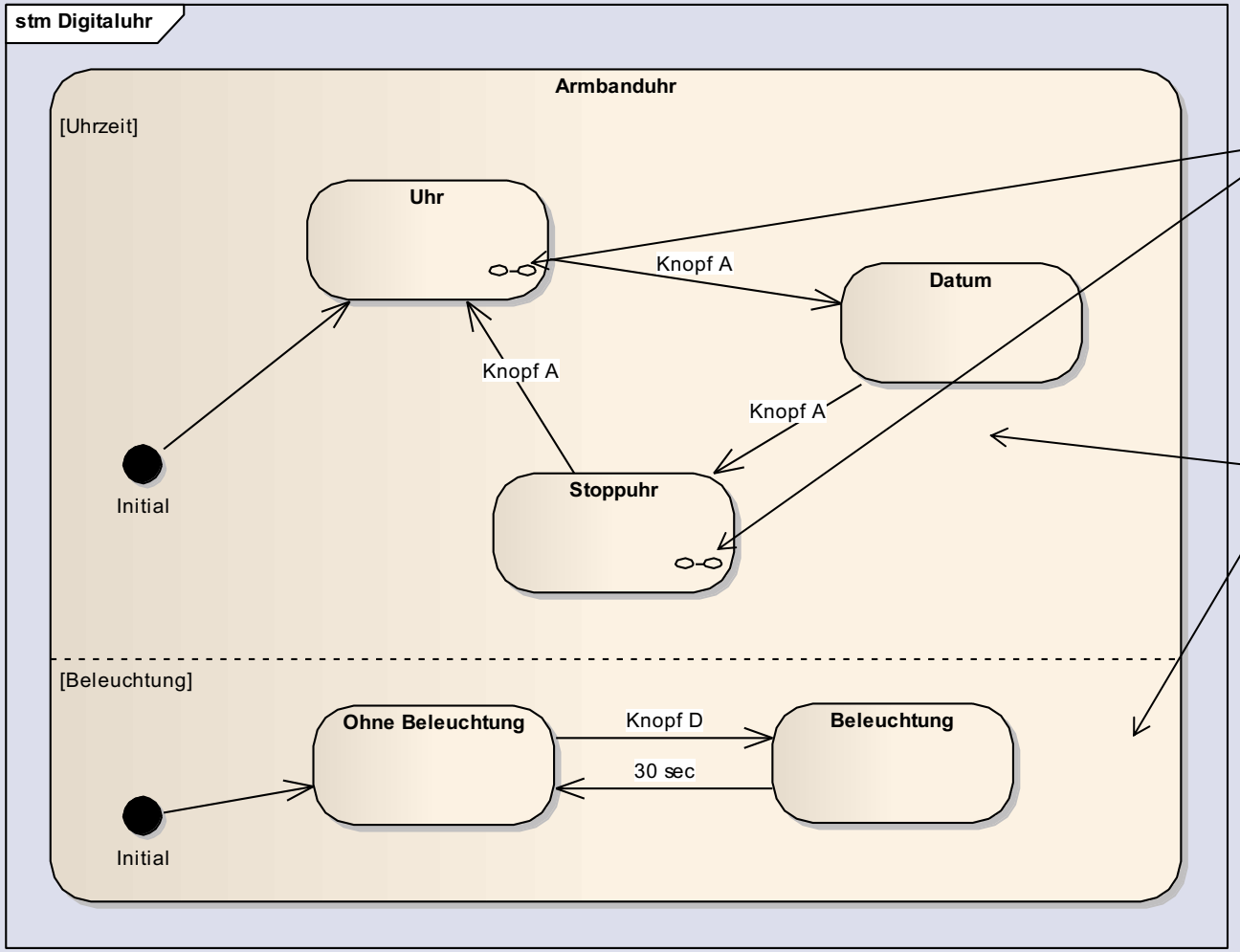


Objektorientierte Analyse und Design
Erweiterte Konzepte: Noch mehr Details

- Zustandsdiagramme können in realen Anwendungen sehr groß werden.

- Einführung weiterer Konzepte/Notationen, um Komplexität in der Darstellung zu verringern:
  1. Parallelität
  2. Hierarchie
     - Gedächtnis (Historie)

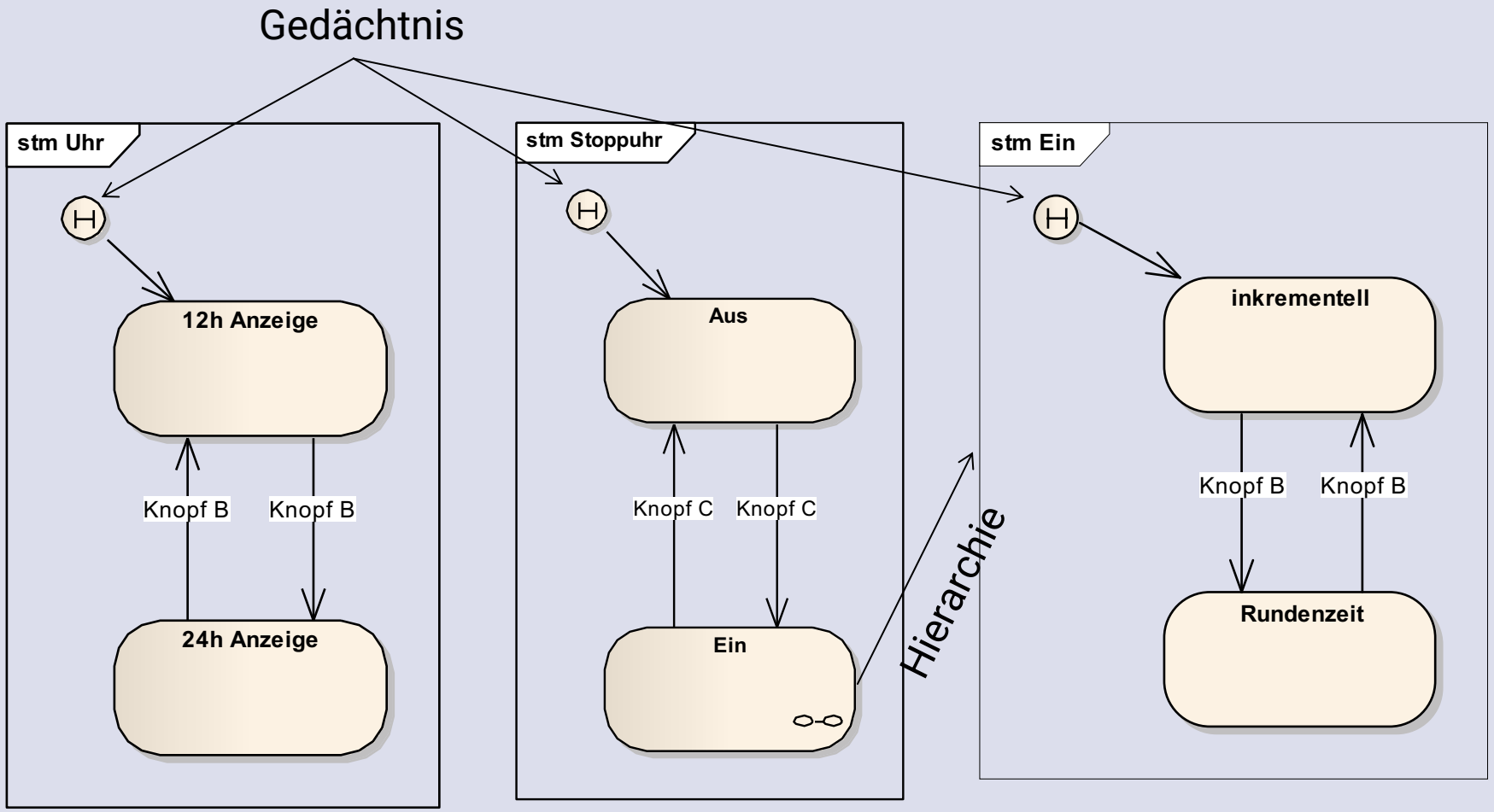
Objektorientierte Analyse und Design
Beispiel Digitaluhr



Hierarchie

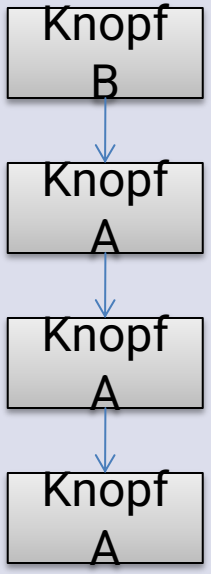
Parallele Zustände

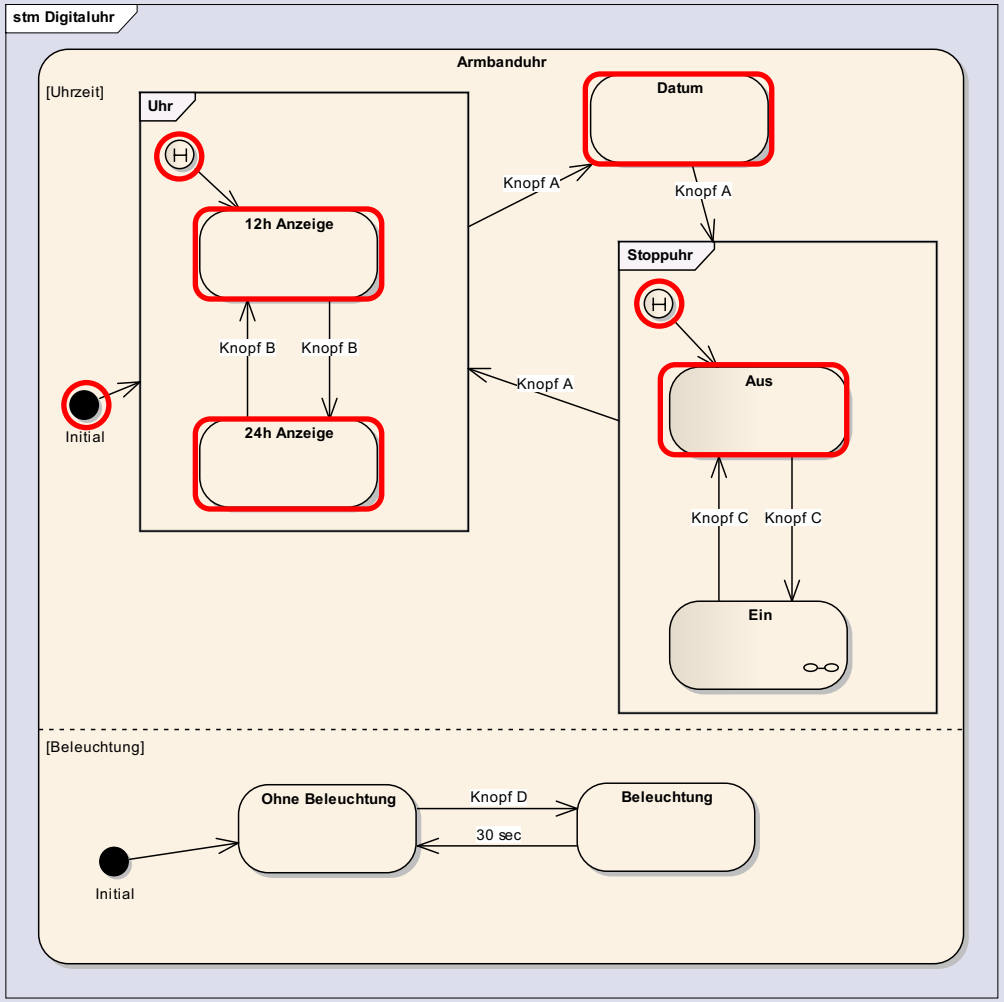
Objektorientierte Analyse und Design
Beispiel: Hierarchie Uhr und Stoppuhr



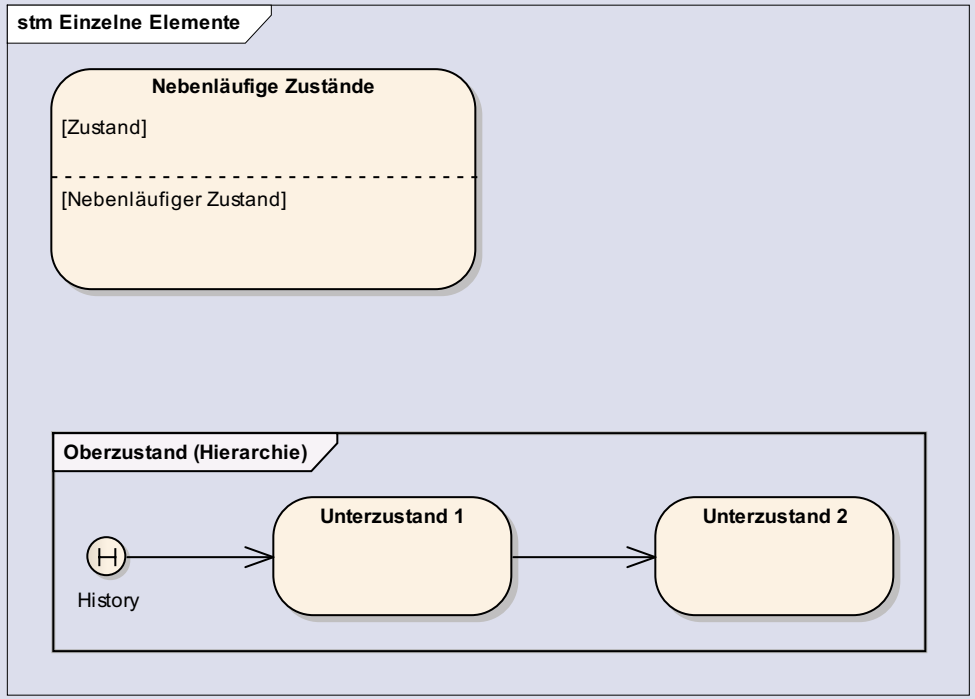
Objektorientierte Analyse und Design
Beispiel: Vollständige Hierarchie

Ereignisse





Objektorientierte Analyse und Design
Erweiterte Darstellungselemente in UML



- **Nebenläufige** Zustände sind voneinander unabhängig.
- Durch die Einführung einer **Hierarchie** lassen sich Zuständen gruppieren.
- Hierarchische Zustände mit **Gedächtnis** (H) starten mit dem zuletzt gültigen Unterzustand.

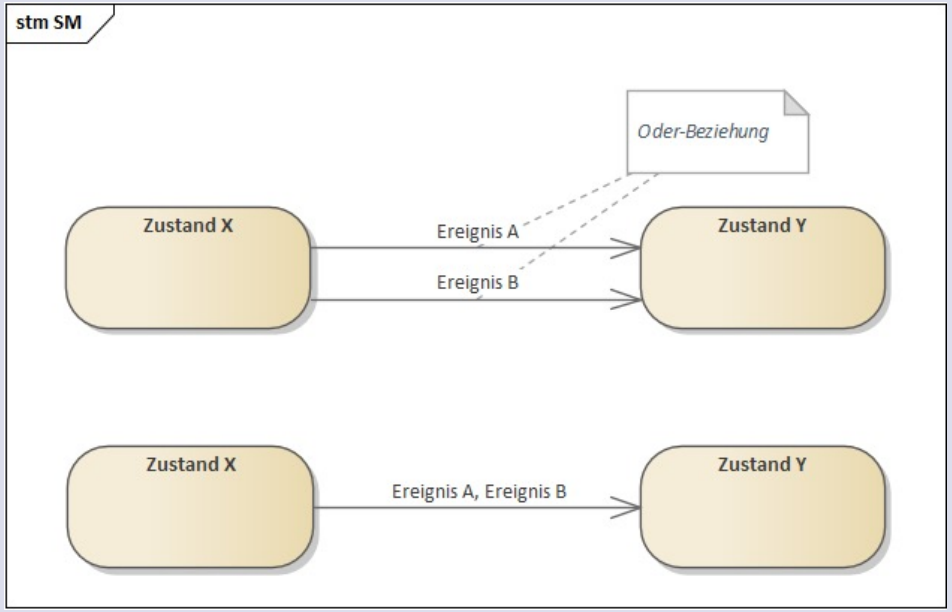
Objektorientierte Analyse und Design
Pseudozustände in UML

- UML bietet weitere **Pseudozustände**, die die Schreibweise komplexer Zustandsmaschinen vereinfachen können.
- Diese Zustände sind **transient**, d. h. in diesen Zuständen wird nicht verweilt sondern in dann folgenden gewechselt.
- Es wird hier empfohlen, diese zusätzlichen Notationselemente **nicht zu nutzen**!
- Begründung:
  - die korrekte Benutzung (Schreiben/Lesen) ist sehr **fehleranfällig**,
    - z. B. 3 Möglichkeiten der Verzweigungen.
  - Nicht notwendig, da durch alles durch DEA ausgedrückt werden kann (Theoretische Informatik!).

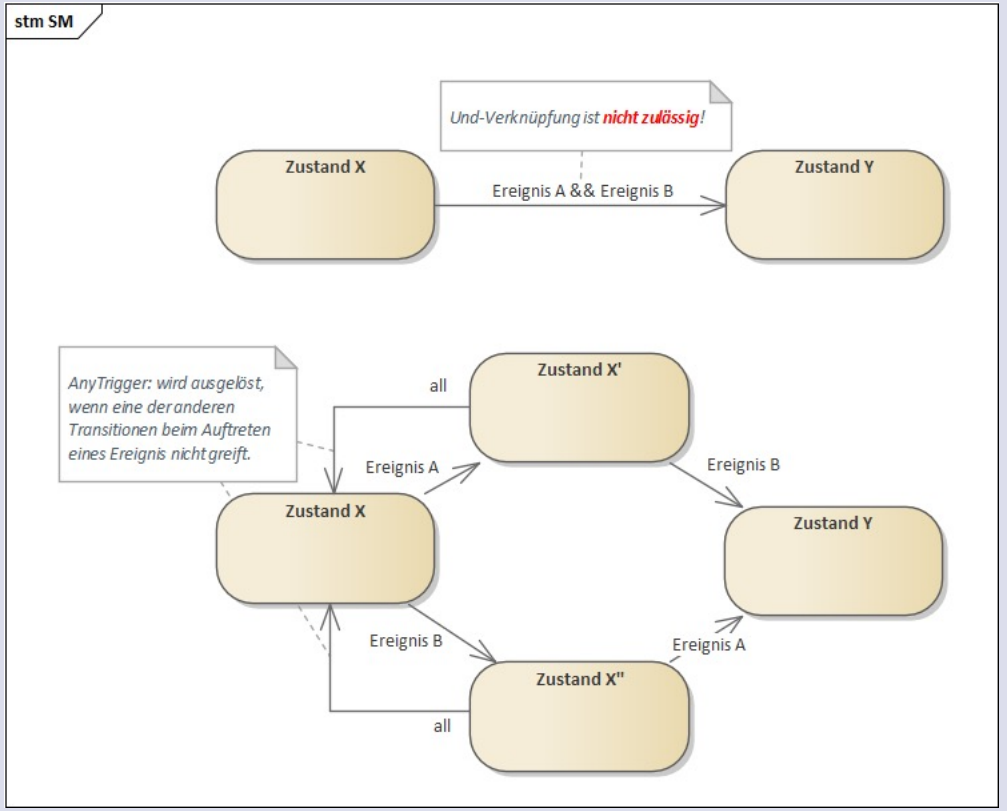
Objektorientierte Analyse und Design
Verknüpfung von Ereignissen

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

Oder-Verknüpfung



Und-Verknüpfung



Objektorientierte Analyse und Design
Aktivitäts- vs. Zustandsdiagramm

- Zustandsdiagramme eignen sich gut, um das Verhalten eines Objekts über mehrere Anwendungsfälle zu beschreiben.
- Zustandsdiagramme eignen sich nicht sehr gut, bei der Beschreibung von mehreren Objekten (Andere Diagramme sind häufig besser geeignet).
- Aktivitäts- und Zustands–diagramme werden teilweise in fälschlicher Weise durch–mischt (auch im Internet!).
- Offensichtliche Unterschiede:
  - an den Pfeilen (Transitionen) stehen im Zustandsdiagramm Ereignisse
  - im Aktivitätsdiagramm folgt jeder Aktion genau eine Folgeaktion.

Objektorientierte Analyse und Design
Ausblick Zustandsdiagramm

- Zustandsdiagramme sind ein **Basiswerkzeug** für die Beschreibung von dynamischen Verhalten (Änderungen über die Zeit).
- Zustandsdiagramme für Klassen einsetzen, die ein „interessantes“ Verhalten aufweisen (bspw. durch Ereignisse gesteuert).
- Diese werden in Ingenieurwissenschaften und in der Informatik an **vielen Stellen eingesetzt** (UML ist nur ein Beispiel).
- Zahlreiche **Ergänzungen** und **Erweiterungen** (z.B. Petri-Netze), hier nur Einführung.

Objektorientierte Analyse und Design

TIMING-DIAGRAMM

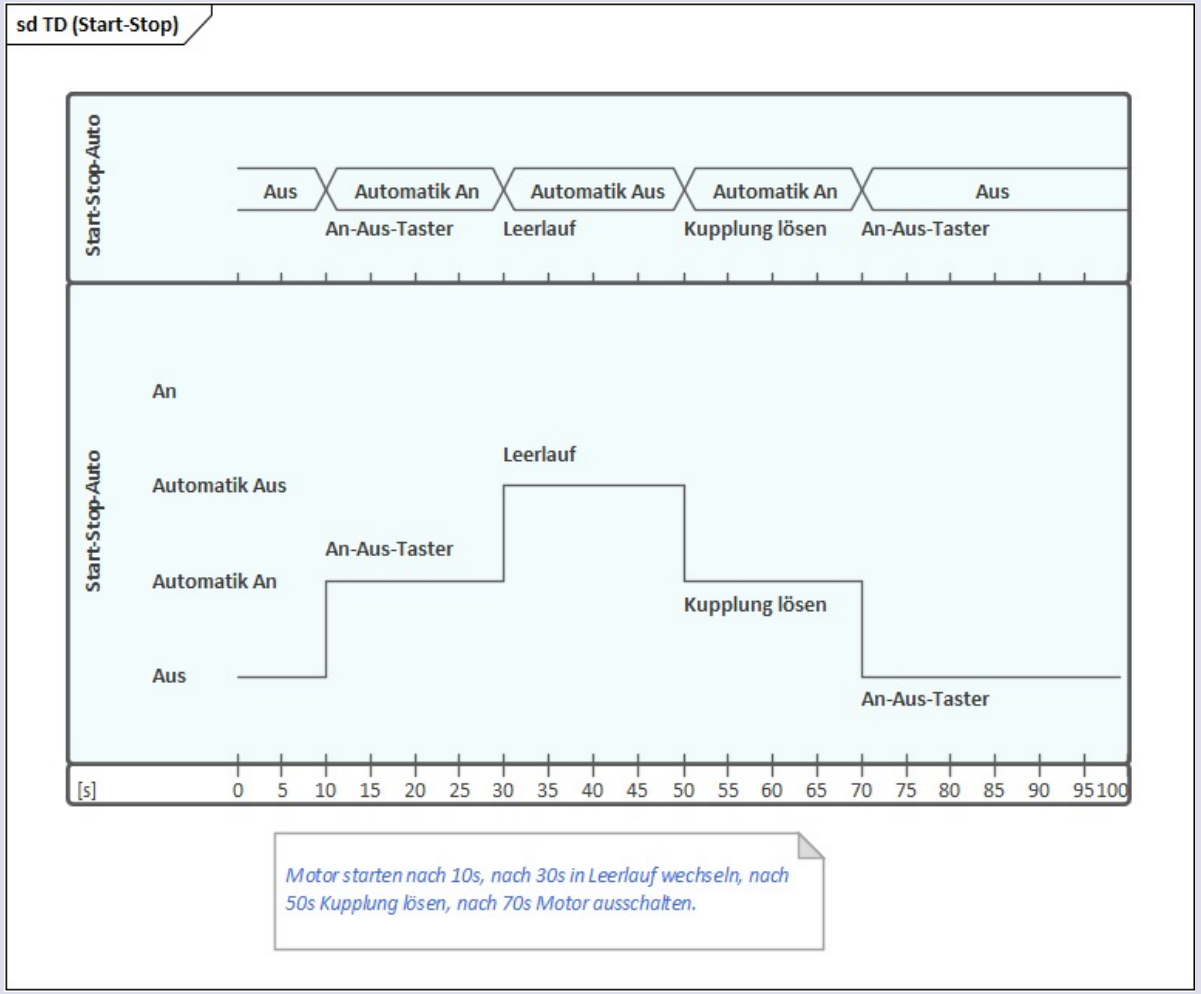
Objektorientierte Analyse und Design
Timing-Diagramm

- Timing-Diagramm zeigt **zeitliches Verhalten** von Objekten.
- Kann als Verlaufsdarstellung von Zuständen verwendet werden.
- Ursprünglich aus der Elektrotechnik bekannt als Beschreibung digitaler Schaltungen.

- 2 Darstellungsmöglichkeiten:
  1. **Zeitverlaufslinie**: erlaubt Darstellung komplexer Interaktionen zwischen mehreren Objekten und Zustandsmaschinen
  2. **Werteverlaufslinie**: kompakte Darstellung, wenn viele Objekte zu visualisieren sind.

Objektorientierte Analyse und Design
Timing-Diagramm: Start-Stop-Automatik

- Beispiel der Start-Stop-Automatik.
- **Konkretes Scenario** wird beschrieben.



Objektorientierte Analyse und Design

VOM DIAGRAMM ZUM PROGRAMM

Objektorientierte Analyse und Design
Analyse des Ist-Standes

- Bekannter Weg: Kundenwünsche, Anforderungsformulierung, Analyse-Modell
- Analysemodell kann realisiert werden, aber:
  - Klassen kaum für Wiederverwendung geeignet
  - Programme meist nur aufwändig erweiterbar
- Viele unterschiedliche Lösungen zu gleichartigen Problemen
  - deshalb: fortgeschrittene Designtechniken studieren.
  - aber: um fortgeschrittenes Design zu verstehen, muss man die Umsetzung von Klassendiagrammen in Programme kennen (dieses Kapitel).
  - aber: um fortgeschrittenes Design zu verstehen, muss man einige OO-Programme geschrieben haben.

Objektorientierte Analyse und Design
UML-Toolsuiten / CASE-Werkzeuge

Theorie:

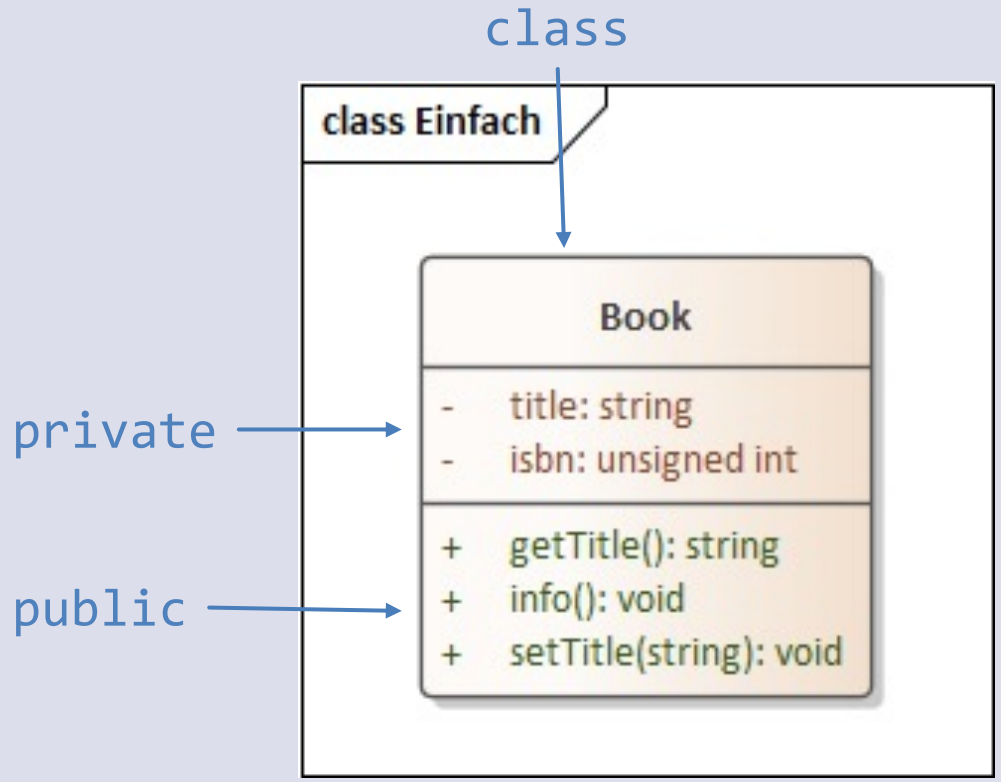
- Werkzeuge unterstützen die **auto–matische Umsetzung** von Klassen–diagrammen in Programmgerüste.
- Entwickler/-innen müssen die Gerüste mit Code (Funktionalität) füllen.
- viele Werkzeuge unterstützen **Roundtrip-Engineering**, d.h. Änderungen im Code werden auch zurück in das Designmodell übernommen.
- Roundtrip beinhaltet auch **Reverse-Engineering**.

Praxis:

- sehr gute kommerzielle Werkzeuge; allerdings muss man für Effizienz eine Vielzahl von Werkzeugen nutzen; d. h. auf deren Entwicklungsweg einlassen.
- ordentliche nicht kommerzielle Ansätze für Teilgebiete; allerdings Verknüpfung von Werkzeugen wird aufwändig.

Objektorientierte Analyse und Design
Übersetzung einfacher Diagramme

- Struktur einfacher **Klassendiagramme** (ohne Beziehungen) direkt in (Java,…) **überführbar**.
  - Betrachtung von Klassenattributen /-methoden (C++: `static`).
  - Vorsicht bei Vererbung und virtuellen Methoden (C++: `virtual`).



Objektorientierte Analyse und Design
Code-Ergänzung durch Entwickler/-in

- Bei der Realisierung kann vereinbart werden, dass **get**- und **set**-Methoden in Übersichten weggelassen (und damit als gegeben angenommen) werden.
- Funktionalität muss meistens **händisch durch Entwickler/-in** ergänzt werden.

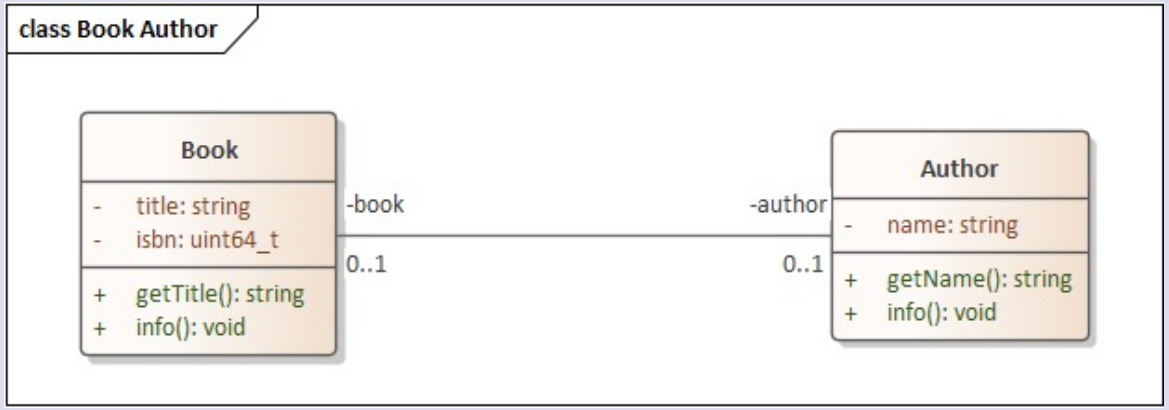
```
void Book::info() const {
    std::cout << "Book: " << title
        << ", ISBN: " << isbn << std::endl;
}

const std::string& Book::getTitle() const {
    return title;
}
```

Objektorientierte Analyse und Design
Umgang mit Assoziationen im Design

- Assoziationen bestehen zunächst nur aus **Namen**, deren **Sichtbarkeit** (üblicherweise private durch – symbolisiert) und **Multiplizitäten**.
- Für die Implementierung ist jede Assoziation zu konkretisieren (Richtung der **Navigierbarkeit**).
- In **C++** lassen sich häufig **Assoziationen** als (smarte) **Zeiger** und/oder **Referenzen** realisieren.

Objektorientierte Analyse und Design
Multiplizität 1

- Einfaches Beispiel:



- Bei der Verwendung von Zeigern und/oder Referenzen ist in C++ zu betrachten:
  - **Wer erzeugt das Objekt?**
  - **Wer zerstört das Objekt?**
- Die Lebensdauer von Objekten ist zu betrachten.
- Bei Komposition ist es (relativ) einfach.

Objektorientierte Analyse und Design
Multiplizität 1 – Code 1/3

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

```cpp
class Book {
public:

    Book(const std::string& t, uint64_t i);

    void info() const;
    void associate(Author* a);
    const std::string& getTitle() const;

private:
    std::string title;
    uint64_t isbn;
    Author* author = nullptr;
};
```

```cpp
class Author {
public:

    Author(const std::string& n);

    void associate(Book* b);
    void info() const;
    const std::string& getName() const;

private:
    std::string name;
    Book* book = nullptr;
};
```

Objektorientierte Analyse und Design
Multiplizität 1 – Code 2/3


```
Book::Book(const std::string& t, uint64_t i):
    title(t),isbn(i) {}

void Book::associate(Author* a) {
    author=a;
}

const std::string& Book::getTitle() const {
    return title;
}
```

```
Author::Author(const std::string& n):
    name(n) {}

void Author::associate(Book* b) {
    book=b;
}

const std::string& Author::getName() const {
    return name;
}

void associate(Book* b, Author* a) {
    b->associate(a);
    a->associate(b);
}
```

Objektorientierte Analyse und Design
Multiplizität 1 – Code 3/3

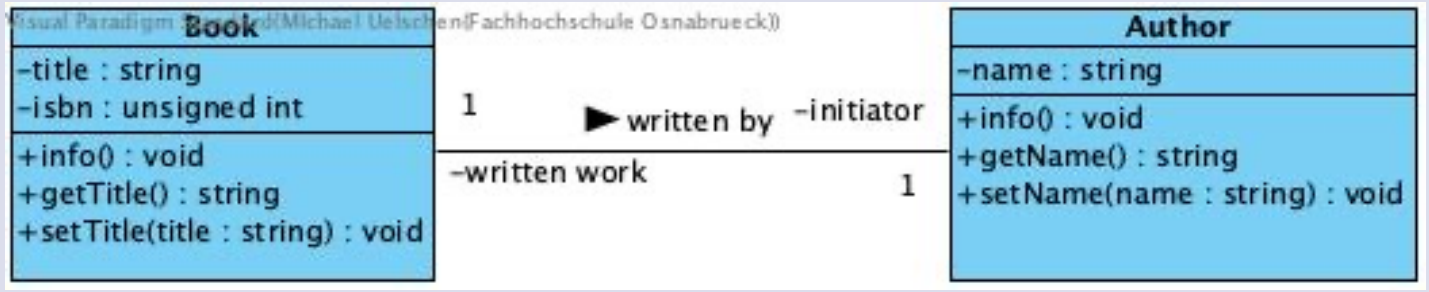

```
auto book=std::make_unique<Book>("Verachtung",
                          9783423280020);

auto author=std::make_unique<Author>("Adler
                                     Olsen");

associate(book.get(),author.get());

book->info();
author->info();
```

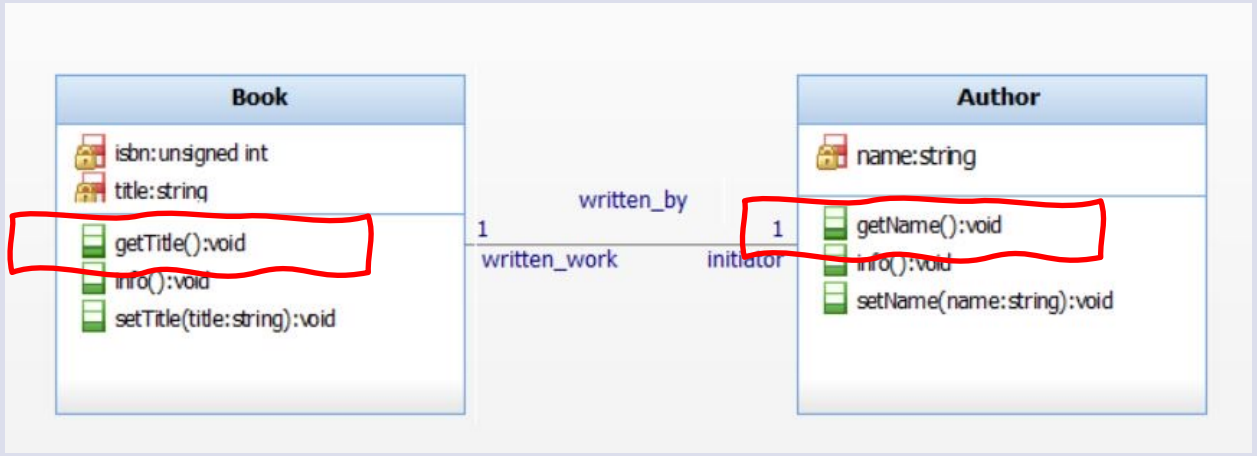
Objektorientierte Analyse und Design
Code-Erzeugung mit Werkzeugen

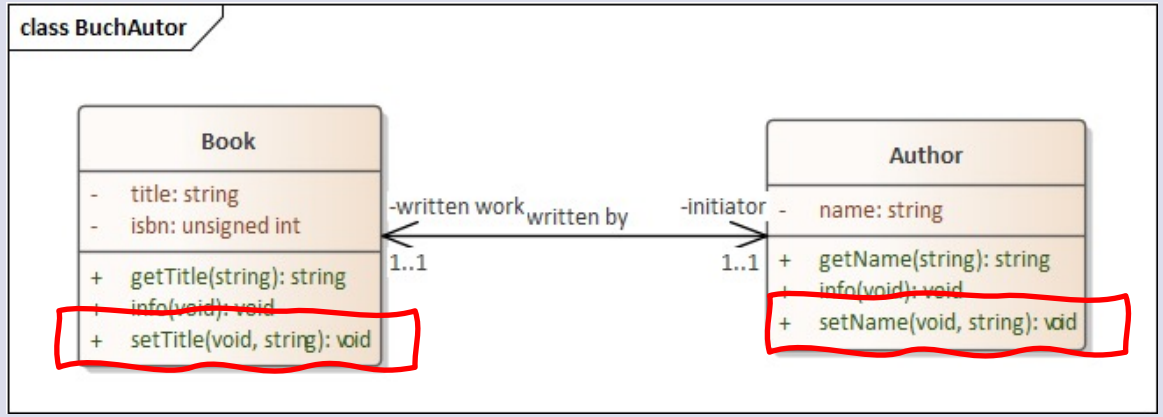
- Idee: Vergleiche **Code-Erzeugung** an einfachem Beispiel durch 3 Werkzeuge
  - Visual Paradigm (export)
  - Enterprise Architect (import)
  - IBM Rhapsody (import)
- Betrachte nur **Klassendiagramm** und verwende Zielsprache **C++**.
- Vorgehen:
  - Beispiel wird mit VP entworfen, C++ Code erzeugt und als **XMI** (XML Metadata Interchange) exportiert.
  - Modell wird mit EA und Rhapsody importiert und C++ Code erzeugt.
- Hinweis: Code-Erzeugung lässt sich umfangreich durch die Werkzeuge konfigurieren (hier nur Vorgabe).

Objektorientierte Analyse und Design
Klassendiagramm: 3 Werkzeuge…

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES







Objektorientierte Analyse und Design
Code-Erzeugung mit Visual Paradigm


```
class Author;
class Book;

class Book
{
   private: string _title;
   private: unsigned int _isbn;
   private: Author* _initiator;

   public: void info();

   public: string getTitle();

   public: void setTitle(string aTitle);
};
```

```
void Book::info() {
   throw "Not yet implemented";
}

string Book::getTitle() {
   return this->_title;
}

void Book::setTitle(string aTitle) {
   this->_title = aTitle;
}
```

Objektorientierte Analyse und Design
Code-Erzeugung mit Enterprise Architect


```
class Book
{

public:
    Book();
    virtual ~Book();

    string getTitle(string DuplicateParam_1);
    void info(void);
    void setTitle(void, string title);

private:
    string title;
    unsigned int isbn;
    Author *initiator;

};
```

```
Book::Book(){
}

Book::~Book(){
}

string Book::getTitle(string DuplicateParam_1){

    return  NULL;
}

void Book::info(void){

}

void Book::setTitle(void, string title){

}
```

Objektorientierte Analyse und Design
Code-Erzeugung mit IBM Rhapsody 1

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

```
class Book {
public :
    Book();
    ~Book();

    void getTitle();
    void info();
    void setTitle(const string& title);

    Author* getInitiator() const;
    void setInitiator(Author* p_Author);

protected :

    //## auto_generated
    void cleanUpRelations();
```

```
private :

    unsigned int getIsbn() const;
    void setIsbn(unsigned int p_isbn);

protected :

    unsigned int isbn;
    string title;
    Author* initiator;


public :

    void __setInitiator(Author* p_Author);
    void _setInitiator(Author* p_Author);
    void _clearInitiator();
};
```

Objektorientierte Analyse und Design
Code-Erzeugung mit IBM Rhapsody 2


```
Book::Book() {
    initiator = NULL;
}

Book::~Book() {
    cleanUpRelations();
}

void Book::getTitle() {
}

void Book::info() {
}

void Book::setTitle(const string& title) {
}
```

```
Author* Book::getInitiator() const {
    return initiator;
}

void Book::setInitiator(Author* p_Author) {
    if(p_Author != NULL)
        {
            p_Author->_setWritten_work(this);
        }
    _setInitiator(p_Author);
}
```

Objektorientierte Analyse und Design
Code-Erzeugung mit IBM Rhapsody 3


```
void Book::cleanUpRelations() {
  if(initiator != NULL) {
  Book* p_Book = initiator->getWritten_work();
    if(p_Book != NULL) {
      initiator->__setWritten_work(NULL);
    }
    initiator = NULL;
  }
}

unsigned int Book::getIsbn() const {
    return isbn;
}
```

```
void Book::setIsbn(unsigned int p_isbn) {
    isbn = p_isbn;
}

void Book::__setInitiator(Author* p_Author) {
    initiator = p_Author;
}

void Book::_setInitiator(Author* p_Author) {
    if(initiator != NULL) {
            initiator->__setWritten_work(NULL);
        }
    __setInitiator(p_Author);
}

void Book::_clearInitiator() {
    initiator = NULL;
}
```

Objektorientierte Analyse und Design
Multiplizität n

- **Umsetzung mit Container** (Sammlung, Collection) hängt von Art der Collection ab,
  - ob Daten geordnet sein sollen,
  - ob doppelte Daten erlaubt sind,
  - ob es eine spezielle Zuordnung (Schlüssel-Wert-Paar) gibt.
- Implementierung in C++, z. B.
  - N ist unbekannt/variabel
    - `std::vector` oder `std::list`
  - N ist bekannte, feste Größ
    - `std::array`
- Standardhilfsklassen, z. B. aus der Java-Klassenbibliothek oder der C++-STL werden typischerweise in Klassendiagrammen nicht aufgeführt.
- Anmerkung: man sieht die UML-Notation für generische (oder parametrisierte) Klassen.
- UML-Werkzeuge unterscheiden sich bei der Generierung und beim Reverse-Engineering beim Umgang mit Collections.

Objektorientierte Analyse und Design
Multiplizität n – Code

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

```
class Book {
public:

    Book(const std::string& t, uint64_t i);

    void info() const;
    void associate(Author* a);
    const std::string& getTitle() const;

private:
    std::string title;
    uint64_t isbn;
    std::list<Author*> authors;
};

void Book::associate(Author* a) {
    authors.push_back(a);
}
```

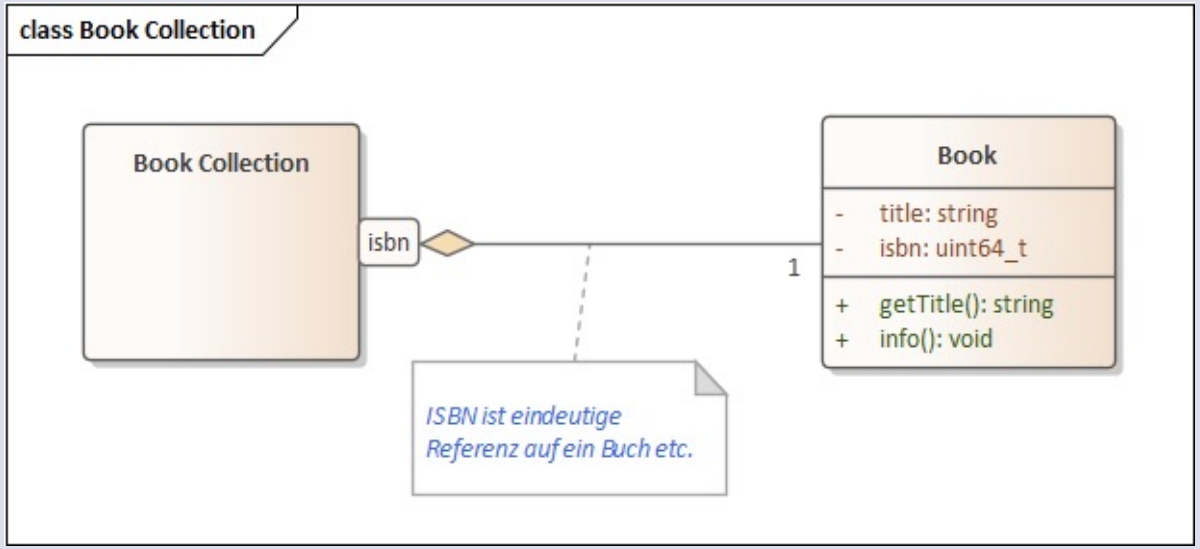
```
auto book=std::make_unique<Book>(
        "Programmieren in C",9783446154971);
auto author1=std::make_unique<Author>(
        "Brian Kernighan");
auto author2=std::make_unique<Author>(
        "Dennis Ritchie");

associate(book.get(),author1.get());
associate(book.get(),author2.get());
```

Objektorientierte Analyse und Design
Containerklassen in UML

- **Randbedingungen** (Constraints) stehen in geschweiften Klammern:
  - **unique**: eindeutig, nur einmal
    - Datentyp: Menge (set)
  - **ordered**: geordnet, sortiert oder Reihenfolge beibehaltend
    - Datentyp: Liste (list), Vektor (vector)
  - **notunique, unordered**: MultiSet
  - Default ohne Angabe ist: {unique, unordered}
- Weitere Möglichkeiten durch Sprache **Object Constraint Language** (OCL).
- Objektorientierte Programmier–sprachen haben verschiedene Umsetzungen von Containern
  - UML lässt meist trotz Randbedingungen verschiedene Umsetzungen zu
- C++: Beispielumsetzungen für Menge
  - `std::set`
  - `std::multiset`
  - `std::unordered_set`
  - `std::unordered_multiset`

Objektorientierte Analyse und Design
Qualifizierte Assoziationen

- **Qualifizierendes Attribut** als Teil der Assoziation angeben
  - Lässt sich mit Wörterbuch (Map, Dictionary) realisieren.
- Beispiel
  - Zu jeder der Vorlesung bekannten Matrikelnummer gehört genau ein Student.
  - Andere Multiplizitäten (0..1, *) möglich.



Objektorientierte Analyse und Design
Qualifizierte Assoziationen – Code 1/2


```
class Book {
public:

    Book(const std::string& t, uint64_t i);

    void info() const;
    const std::string& getTitle() const;
    uint64_t getISBN() const;

private:
    std::string title;
    uint64_t isbn;
};
```

```
class BookCollection {
public:

    BookCollection();

    void add(Book* b);
    void info() const;

private:
    /* --Mit qualifiziertem Attribut (ISBN).*/
    std::map<uint64_t,Book*> books;
};
```

Objektorientierte Analyse und Design
Qualifizierte Assoziationen – Code 2/2


```
void BookCollection::add(Book* b) {
    books[b->getISBN()]=b;
}

void BookCollection::info() const {
    for(auto b: books)
        b.second->info();
}
```

Objektorientierte Analyse und Design
Arten der Zugehörigkeit

Aggregation

- Objekte haben unabhängige, ggfs. unterschiedliche Lebensdauern.
- Das Aggregat („Ganzes“) hat Verweise (C++: (smarte) Zeiger) auf die einzelnen Elemente.
- Das Erzeugen/Zerstören der Objekte („Teile“) ist i. allg. nicht Aufgabe des Aggregats.

Komposition

- Die Lebensdauer des „Teils“ ist abhängig vom „Ganzen“.
- Die Komposition verwaltet das Teil
  a. als Klassenvariable oder
  b. als dynamisches Objekt mit (smarten) Zeigern.
- Die Komposition ist verantwortlich für das Erzeugen und insbesondere für das Zerstören der einzelnen Teile (Speicherfreigabe in C++!).

Objektorientierte Analyse und Design
Beispiel für Komposition


```
class RFID {
public:

    using TagType = uint64_t;

    RFID(TagType tt):tag(tt) {}
    TagType getTag() const { return tag; }

private:
    TagType tag;
};
```

Objektorientierte Analyse und Design
Arten der Zugehörigkeit (Komposition)

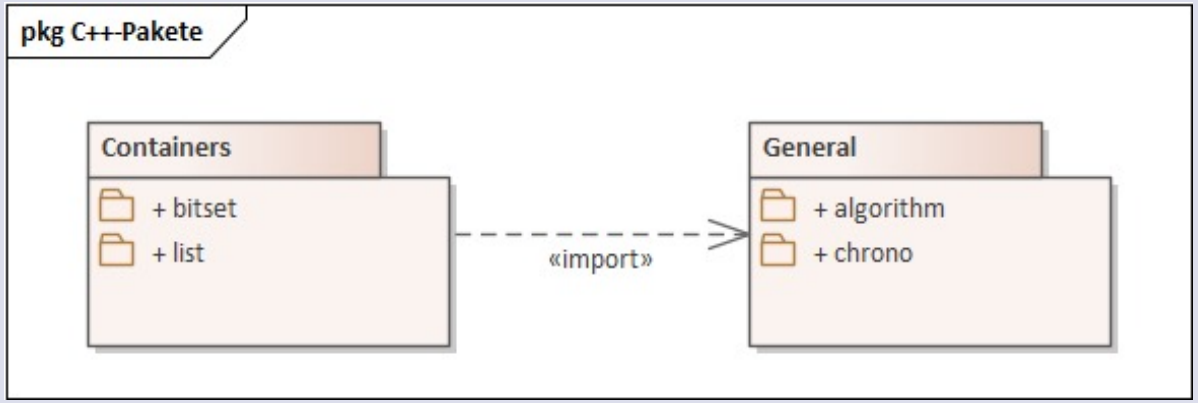

```
class Book {
public:

    Book(const std::string& t, uint64_t i);

    void info() const;
    void initializeTag(RFID::TagType tag);
    const std::string& getTitle() const;

private:
    std::string title;
    uint64_t isbn;
    std::unique_ptr<RFID> rfid;
};
```

```
void Book::initializeTag(RFID::TagType tag) {
    rfid=std::make_unique<RFID>(tag);
}
```

Objektorientierte Analyse und Design

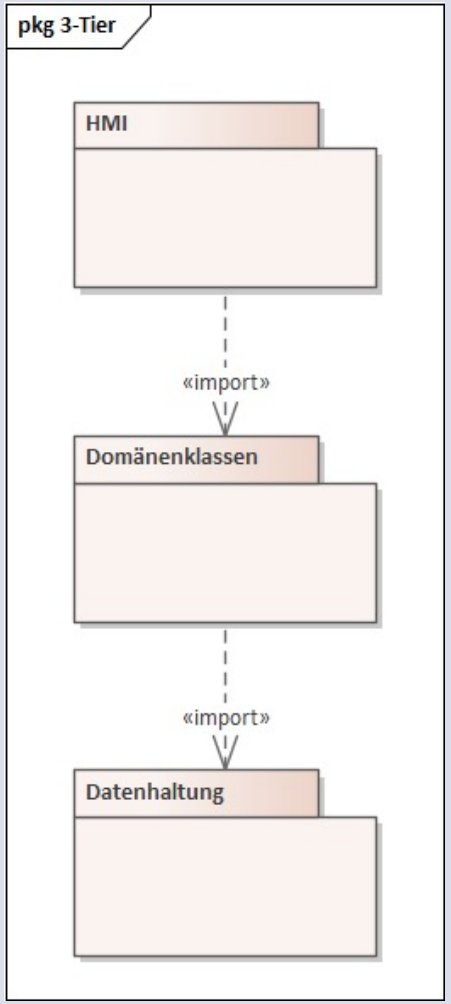
PAKETDIAGRAMM

- Für große Systeme entstehen viele Klassen; bei guten Entwurf:
  - Klassen die eng zusammenhängen (gemeinsame Aufgabengebiete).
  - Klassen, die nicht oder nur schwach zusammenhängen (Verknüpfung von Aufgabengebieten).
  - Strukturierung durch SW-Pakete; Pakete können wieder Pakete enthalten.



Objektorientierte Analyse und Design
Typische 3-Schichten-SW-Architektur

- Ziel: Klassen eines oberen Pakets **greifen** nur auf Klassen eines unteren Paketes **zu**.
- **Änderungen** der oberen Schicht beeinflussen untere Schichten nicht.
- Domänenmodell (Analyse) liefert typischerweise nur Fachklassen.
- Datenhaltung steht für **Persistenz**.
- Klassen in Schicht sollten **gleichen Abstraktionsgrad** haben.



Objektorientierte Analyse und Design
Umsetzung von Paketen in C++

- Java hat **package-Konzept** zur Strukturierung von Dateien und Namensräumen.
- C++ erlaubt (bisher) nur ein eingeschränkte **Aufteilung in Komponenten**:
  - Dateisystem: keine Semantik sondern Textersetzung durch Präprozessor (á la 1970).
  - Namensräume
- Ab **C++20** (yeah!) bietet C++ eigenständiges **Modulkonzept** an.
  - Import und Export von Schnittstellen.
  - Orthogonal zu Namensräumen
  - Vermeidung/Reduzierung von Präprozessor-Direktiven
- https://www.youtube.com/watch?v=szHV6RdQdg8

Objektorientierte Analyse und Design
Paketabhängigkeiten optimieren

- Ziel ist es, dass Klassen sehr eng zusammenhängen; es weniger Klassen gibt, die eng zusammenhängen und viele Klassen und Pakete, die nur lose gekoppelt sind.
- Möglichst bidirektionale oder **zyklische Abhängigkeiten** vermeiden.
- Bei Paketen können Zyklen teilweise durch die Verschiebung von Klassen aufgelöst werden.
- Wenig globale Pakete (sinnvoll für projektspezifische Typen (z. B. Aufzählungen und Ausnahmen).
- Es gibt viele Designansätze, Abhängigkeiten zu verringern bzw. ihre Richtung zu ändern.

Objektorientierte Analyse und Design

ZUSAMMENFASSUNG

Objektorientierte Analyse und Design Zusammenfassung

- Einführung in Design mit UML: Erweiterung **Klassendiagramm**, **Sequenz**- und **Zustandsdiagramm**.
- **Umsetzung** Klassendiagramm in **C++**-Quelltexte
- Strukturierung von Komponenten in Pakete
- Offen: Objekt-, Kompositionsstruktur-, Verteilungsdiagramm

- Demnächst (jetzt wird es interessant):
  - Entwurfsmuster: Lösungen zu wiederkehrenden Aufgabenstellungen