Softwaretechnik und Programmierparadigmen

VL04: Analyse und Design

Prof. Dr. Sabine Glesner
FG Programmierung eingebetteter Systeme
Technische Universität Berlin

Zentrale Frage: Wie kommt man von den Anforderungen zum Modell?

- Analyse

  „Was macht das System und was macht es nicht?“

  - Gegenstandsbereich
  - Benutzerdynamik
  - Systemgrenzen

- Design

  „Wie erfüllt das System seine Aufgaben intern“

  - Kommunikation zwischen Objekten
  - Zugriff auf Objekte
  - (interne) Klassenschnittstellen

Aufgabe der Analyse-Phase

- Überführung der funktionalen Eigenschaften in eine für die Entwickler verständliche Form
- Vorbereitung der Design-Phase
- Im Vergleich: Das Design berücksichtigt zusätzlich nicht-funktionale Eigenschaften und die Bedingungen der Entwicklungsumgebung.

Aufgabe der Design-Phase

- Überführung der Modelle aus der Analyse zu implementierungsnahen Modellen
- Planung der tatsächlichen Realisierung
- Beschreibung der internen Dynamik des Systems

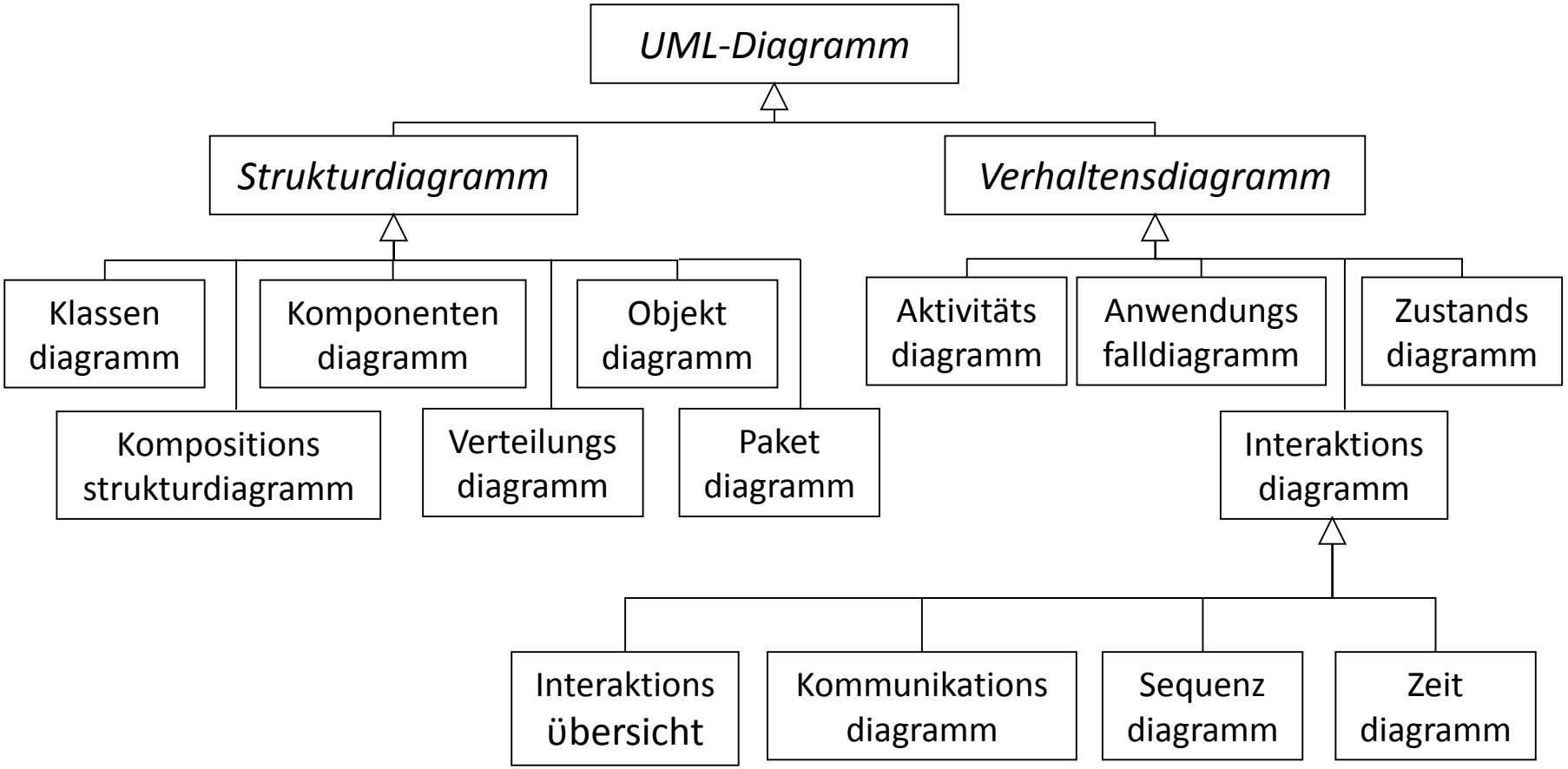
Übersicht

- Analyse-Phase
  - Use Cases (siehe letzte VL)
  - Sequenzdiagramme (siehe letzte VL)
  - Klassenmodell des Gegenstandsbereichs
  - Systemklassendiagramm
- Design-Phase
  - Erweiterte Sequenzdiagramme
  - Aktivitätsdiagramme
  - Statecharts
  - Design-By-Contract (OCL)

Übersicht

- Analyse-Phase
  - Use Cases (siehe letzte VL)
  - Sequenzdiagramme (siehe letzte VL)
  - Klassenmodell des Gegenstandsbereichs
  - Systemklassendiagramm
- Design-Phase
  - Erweiterte Sequenzdiagramme
  - Aktivitätsdiagramme
  - Statecharts
  - Design-By-Contract (OCL)

UML: Diagrammübersicht



Strukturelle Modellierung mit UML

- Strukturmodellierung: Modellierung des statischen Aufbaus des Systems (System Architecture)
- Kern der Strukturmodellierung ist das Klassendiagramm
- Weitere Strukturdiagramme zur Darstellung verschiedener Aspekte des Systems
- Es müssen nicht alle Diagramme verwendet werden, um ein vollständiges UML-Modell zu entwickeln
- Durchgängiges Paradigma: Objektorientierung

Strukturdiagramme (1/3)

- Klassendiagramme (Class Diagram)
  - „Kern“ eines UML-Modells
  - Stellen die statische Struktur dar
    - Beschreibung der Klassen und ihrer Beziehungen
    - Vererbungshierarchien (Gen/Spec-Beziehungen)
    - Aggregation und Komposition
- Objektdiagramme (Object Diagram)
  - Mögliche Instanziierungen des Klassendiagramms
  - Zu einem bestimmten Ausführungszeitpunkt vorhandene Instanzen und deren Beziehungen

Strukturdiagramme (2/3)

- Paketdiagramme (Package Diagram)
  - Partitionierung des Modells in Pakete
  - Aggregations- und Gen/Spec-Beziehungen zwischen Paketen
  - Importbeziehungen:
    - Import einzelner Elemente mit <<access>>
    - Import ganzer Pakete mit <<import>>
  - Darstellung der hierarchischen Struktur des Systems
- Komponentendiagramme (Component Diagram)
  - Komponenten: ausführbare Klassen, kapseln internen Aufbau, stellen Verhalten über Schnittstellen und Ports zur Verfügung
  - Komponentendiagramme stellen die Komponenten und deren Interaktion dar (Ports und Schnittstellen)
  - Darstellung der funktionalen Struktur (Software Architecture)

Strukturdiagramme (3/3)

- Kompositionsstrukturdiagramme (Composite Structure)
  - Konfiguration von miteinander verbundenen Laufzeitelementen
  - z.B. Zusammenarbeit von Klassen oder Objekten zur Erfüllung einer bestimmten Aufgabe
- Verteilungsdiagramme (Deployment Diagram)
  - Beschreibung der physikalischen Struktur (Topologie) von verteilten Systemen
  - Modellierung aller im realen System tatsächlich vorhandenen Hardware- und Software-Knoten und deren Verbindungen (Kommunikationspfade)
  - Physikalische Struktur setzt sich zusammen aus
    - Artefakten (physikalische Informationseinheiten, z.B. Dateien)
    - HW- und SW-Knoten (Ausführungseinheiten, z.B. Geräte)
    - Kommunikationspfaden (physikalische Verbindungen)

Einführung Klassendiagramme

- Objektorientierung kennt Ihr von objektorientierten Programmiersprachen wie Java
- Klassendiagramme erlauben die Modellierung abstrakter objektorientierter Konzepte, die unabhängig von der tatsächlichen Implementierung sind

Erinnerung: wesentliche Prinzipien der Objektorientierung

- OO-System ist dynamische Sammlung kommunizierender Objekte
- Objekte kapseln zusammengehörige Daten und Methoden

- Komplexitätsreduktion durch
  - Zerlegen in kleine Einheiten
  - Definition von Schnittstellen
  - Wiederverwendung
  - Ausnutzen von Abstraktionsebenen (Vererbung)

Einführung Klassendiagramme

- Wesentliche Modellierungselemente
  - Klassen
  - Attribute
  - Methoden
  - Assoziationen
  - Multiplizitäten (in Java z.B. nicht direkt vorhanden)
- Beschreiben mögliche Konfigurationen von Objektdiagrammen

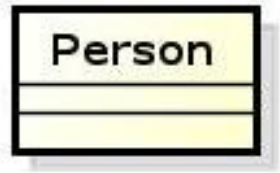
➢Typisieren somit Objektdiagramme

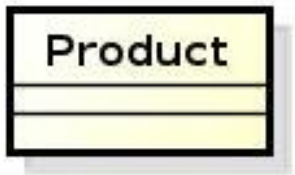
Erinnerung: Online-Shop

- Onlineshop

  Sie werden gebeten für einen kleines Unternehmen, das Schuhe und Kleidung verkauft, die Verwaltungssoftware eines Online-Shops zu entwickeln. Der Online-Shop soll es dem Personen ermöglichen, Produkte in einen Warenkorb zu legen und diesen zu bezahlen.

Klassen



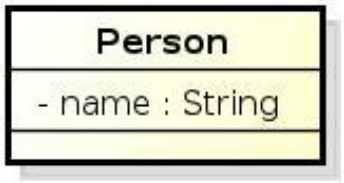


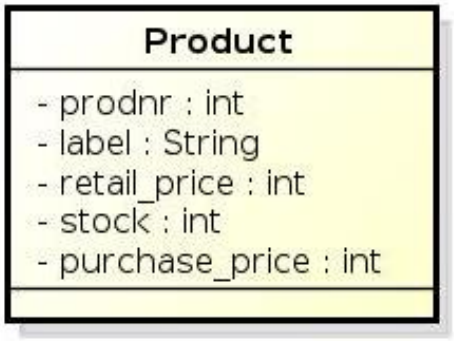
Klassen

- Beschreiben eine Menge von „Dingen“ auf einer abstrakten Ebene
- Objekte sind Instanzen von Klassen, also Elemente aus der entsprechenden Menge von Dingen
- Zu jeder Klasse können beliebig viele Instanzen existieren

Attribute


Shop
- name : String
- website : String

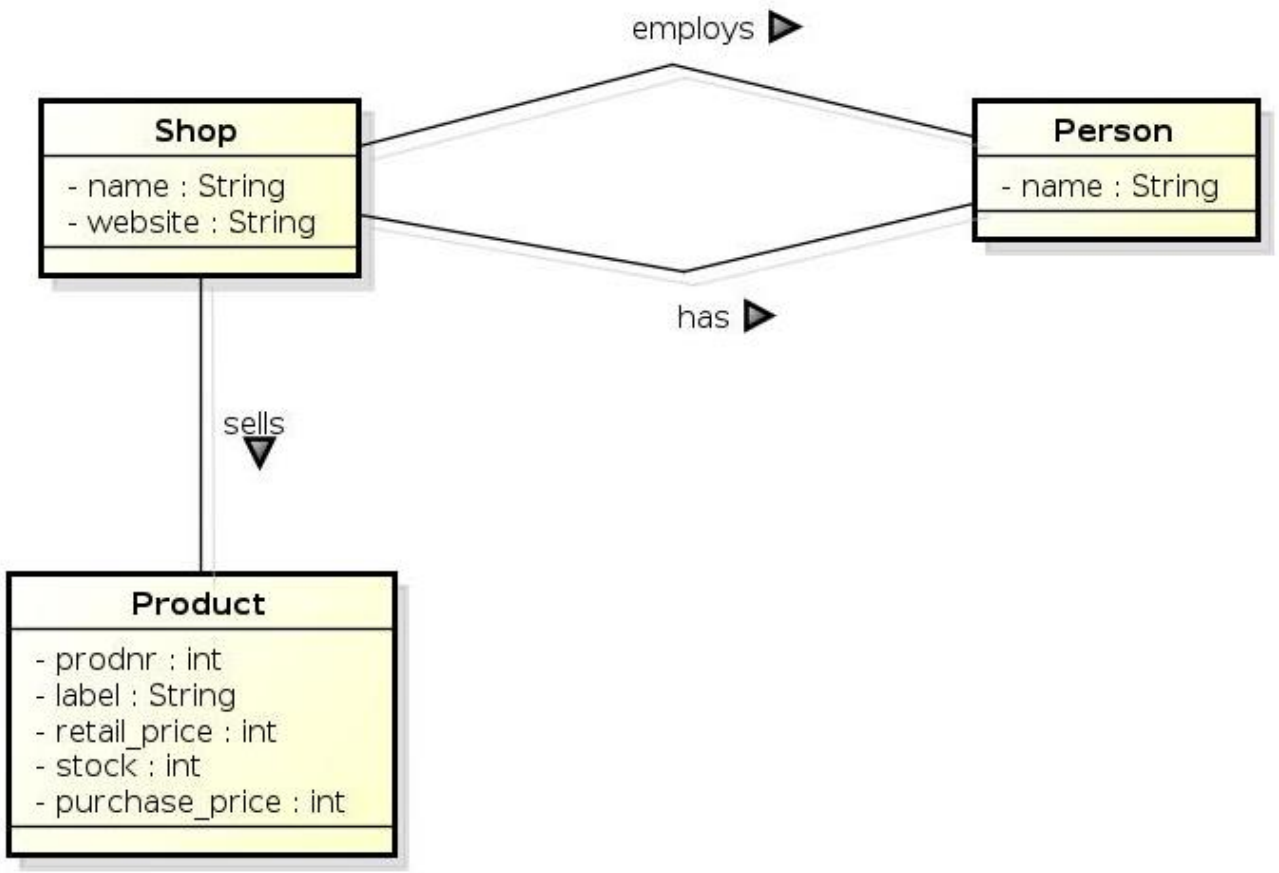

Person
- name : String


Product
- prodnr : int
- label : String
- retail_price : int
- stock : int
- purchase_price : int

Attribute

- Klassen können getypte Attribute enthalten
- Attribute können mit einer Sichtbarkeit versehen werden
  - public: jede andere Klasse kann auf eigenes Attribut zugreifen (lesen + ändern)
  - private: Zugriff nur von eigener Klasse
  - (protected): Zugriff nur von eigener Klasse und Unterklassen
- Methoden sind spezielle Attribute

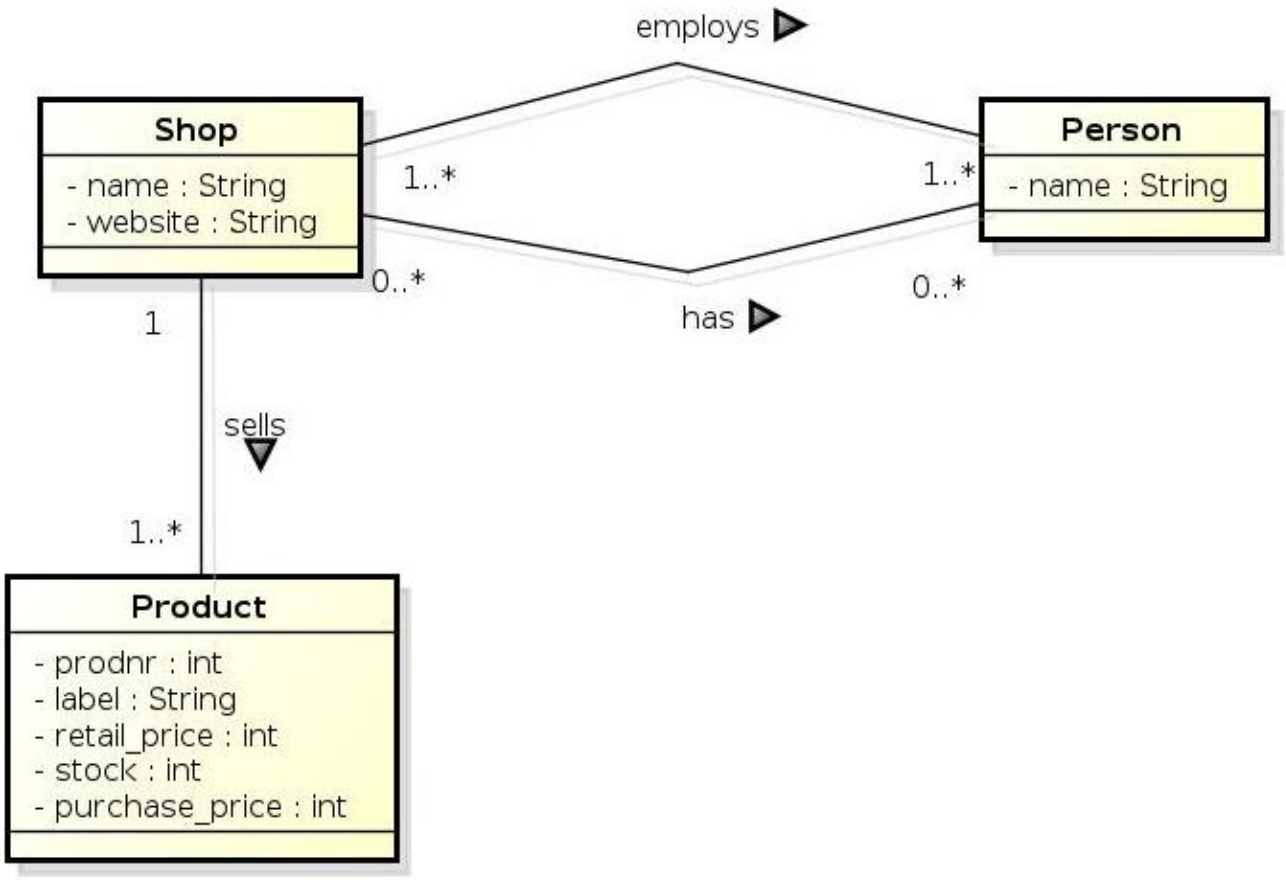
Einfache Assoziationen



Einfache Assoziationen

- Erlauben das Verbinden von Klassen
- Assoziationsbezeichnung hat eine Leserichtung
- Im Objektdiagramm existieren dann entsprechend Links zwischen Objekten

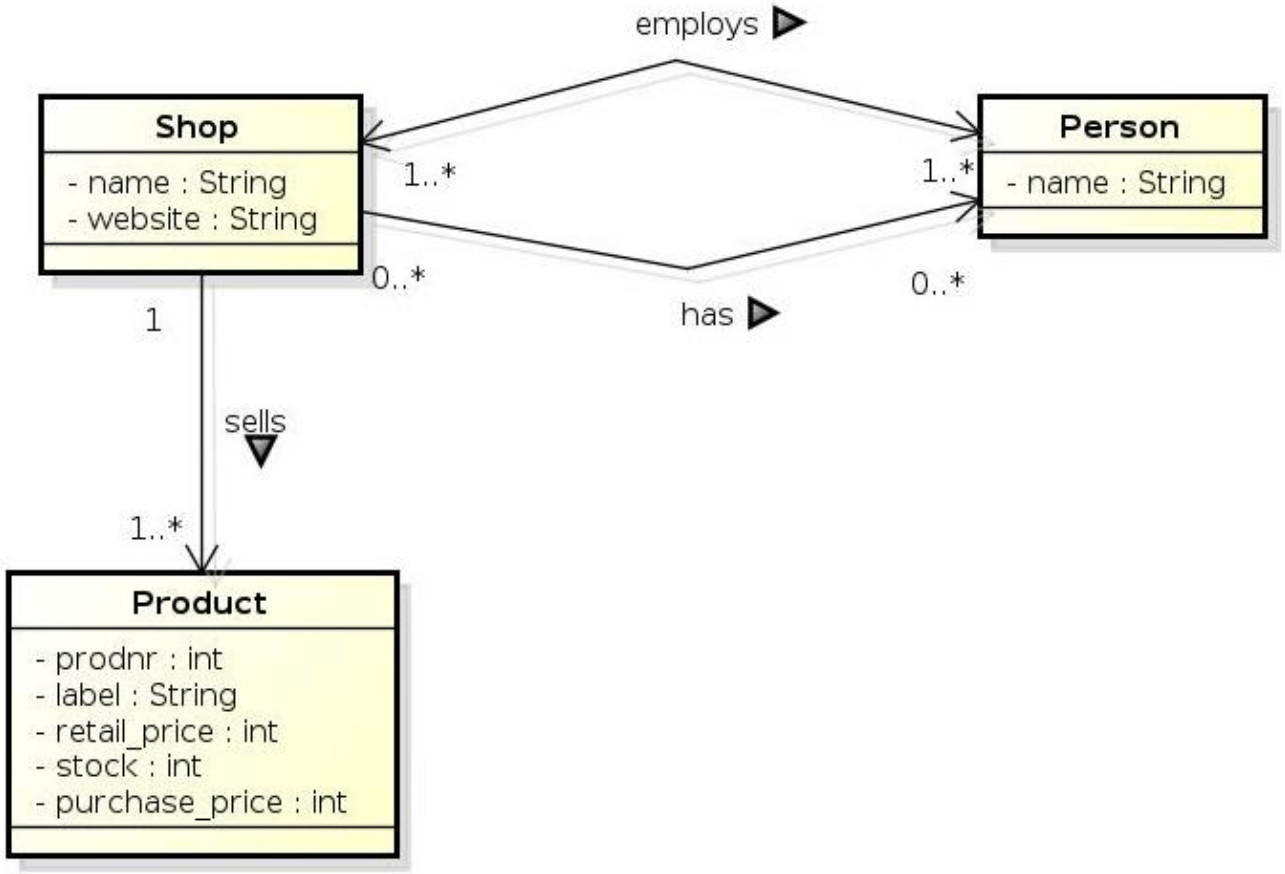
Multiplizitäten



Multiplizitäten

- Erlauben die Anzahl der verbundenen Objekte im Objektdiagramm zu beschränken
- Typische Multiplizitäten
  - 0..*: Keine Beschränkung (äquivalent zum Weglassen der Multiplizität)
  - 1..*: Mindestens eine Verbindung
  - 1: Genau eine Verbindung
  - m..n: Mindestens m, höchstens n Verbindungen

Navigierbarkeit

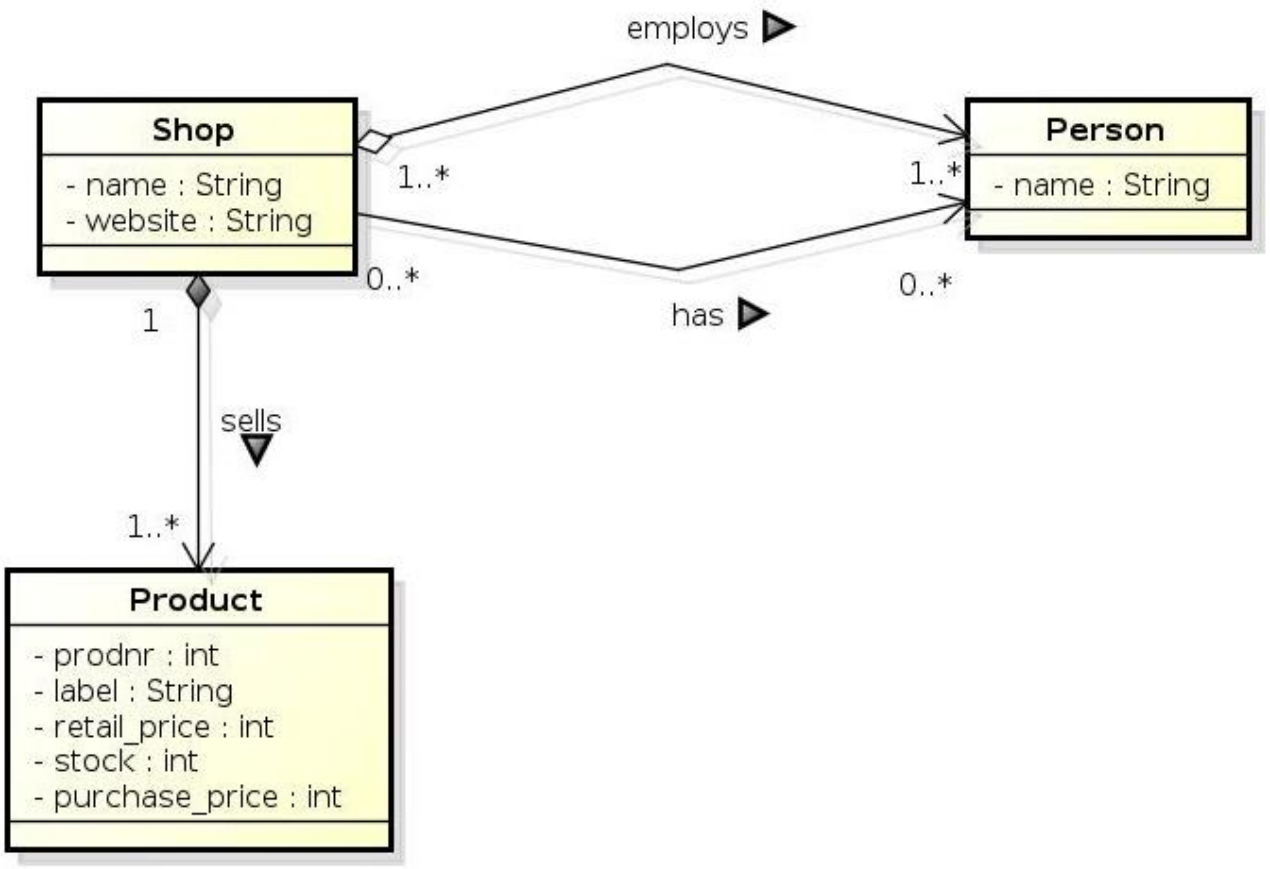


Navigierbarkeit

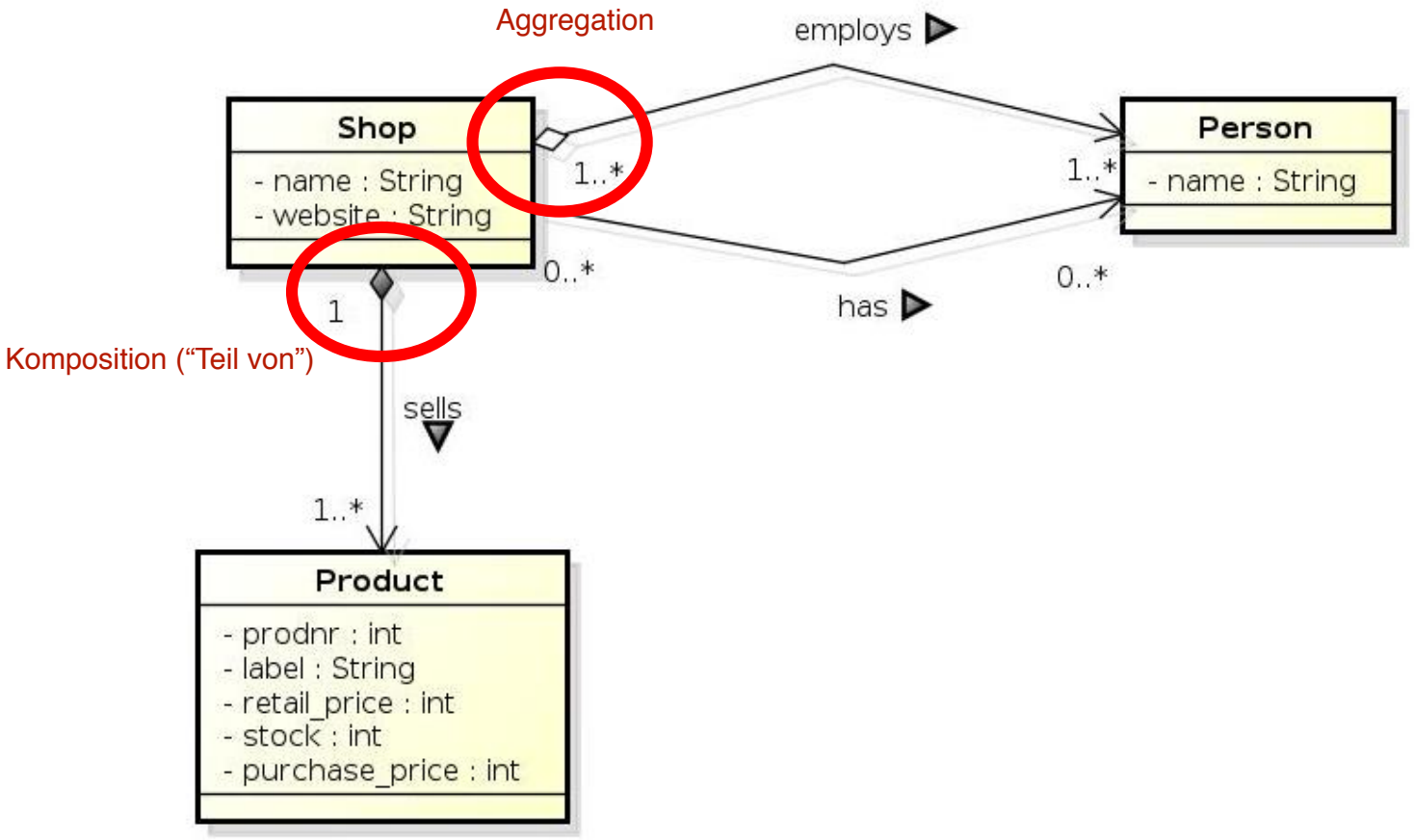
- Beschreibt, ob Objekte über den Link zum anderen Objekt navigieren dürfen oder nicht
- Das Weglassen von Navigationspfeilen entspricht einer Unterspezifikation (kann gewünscht sein)

  = alles ist erlaubt

Aggregation und Komposition



Aggregation und Komposition



Aggregation und Komposition

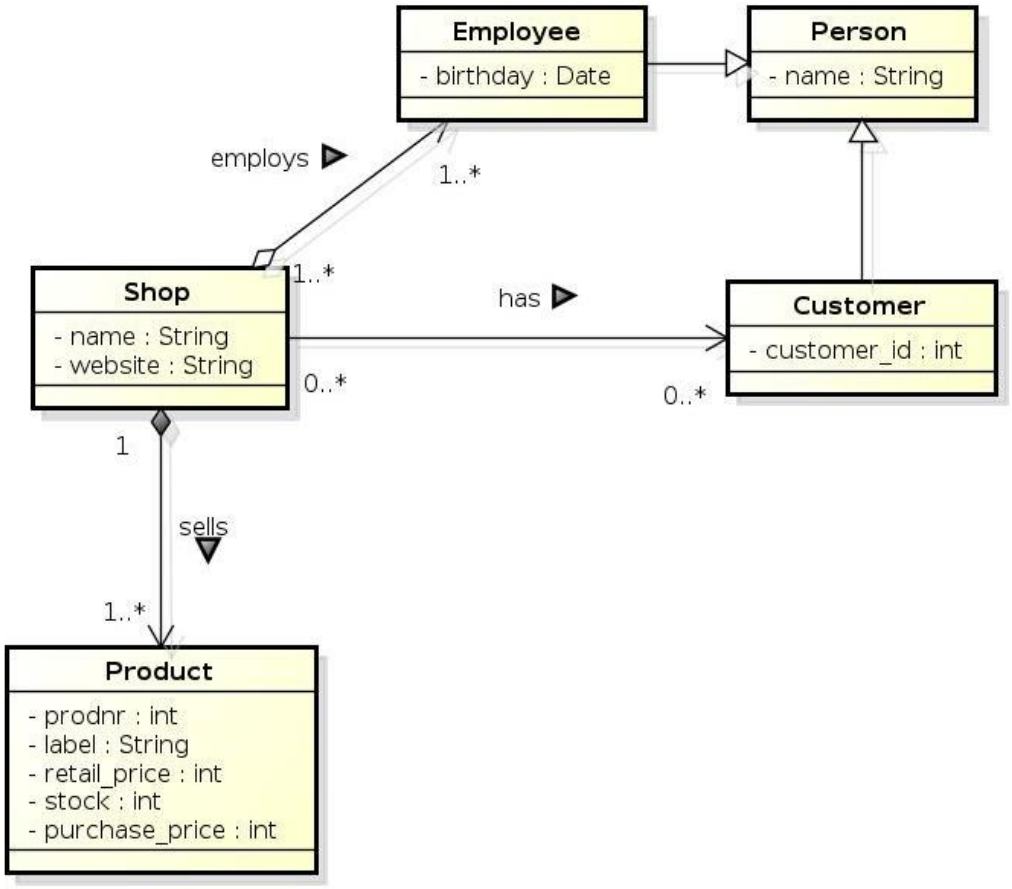
- Keine klare Definition von Aggregation in der UML-Community
- Aggregation (nicht ausgemalte Raute)
  - Klasse beherbergt andere Klassen
  - Logisches Enthaltensein: Spezielle Rolle der Assoziation wird betont
- Komposition (ausgemalte Raute)
  - Spezialfall der Aggregation
  - Physisches Enthaltensein: Die Existenz der beherbergten Klasse hängt von der Existenz der beherbergenden Klasse ab

Erinnerung: Online-Shop

- Onlineshop

  Weiterhin soll das System gleichzeitig auch die Mitarbeiter verwalten. Sowohl Kunden als auch Mitarbeiter sollen registriert werden können.

Generalisierung/Spezialisierung



Generalisierung/Spezialisierung

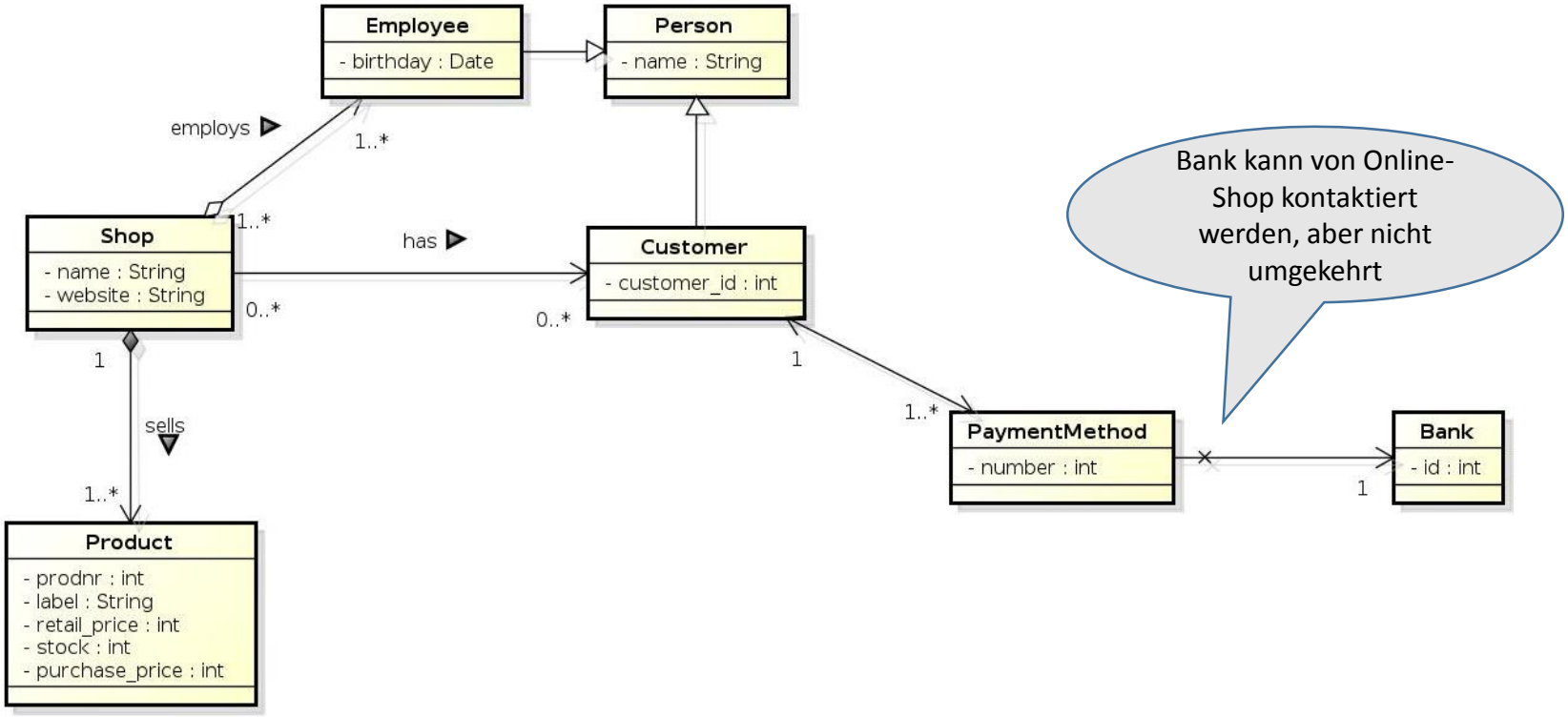
- Übertragung des Vererbungskonzepts objektorientierter Sprachen auf Klassendiagramme

Erinnerung: Online-Shop

- Onlineshop

  Als Bezahlmethoden sind zunächst Bankeinzug und Kreditkartenzahlung vorgesehen. Bevor die Bestellung aufgegeben wird, muss durch die Bank sichergestellt werden, dass die Bezahlung tatsächlich erfolgen kann.

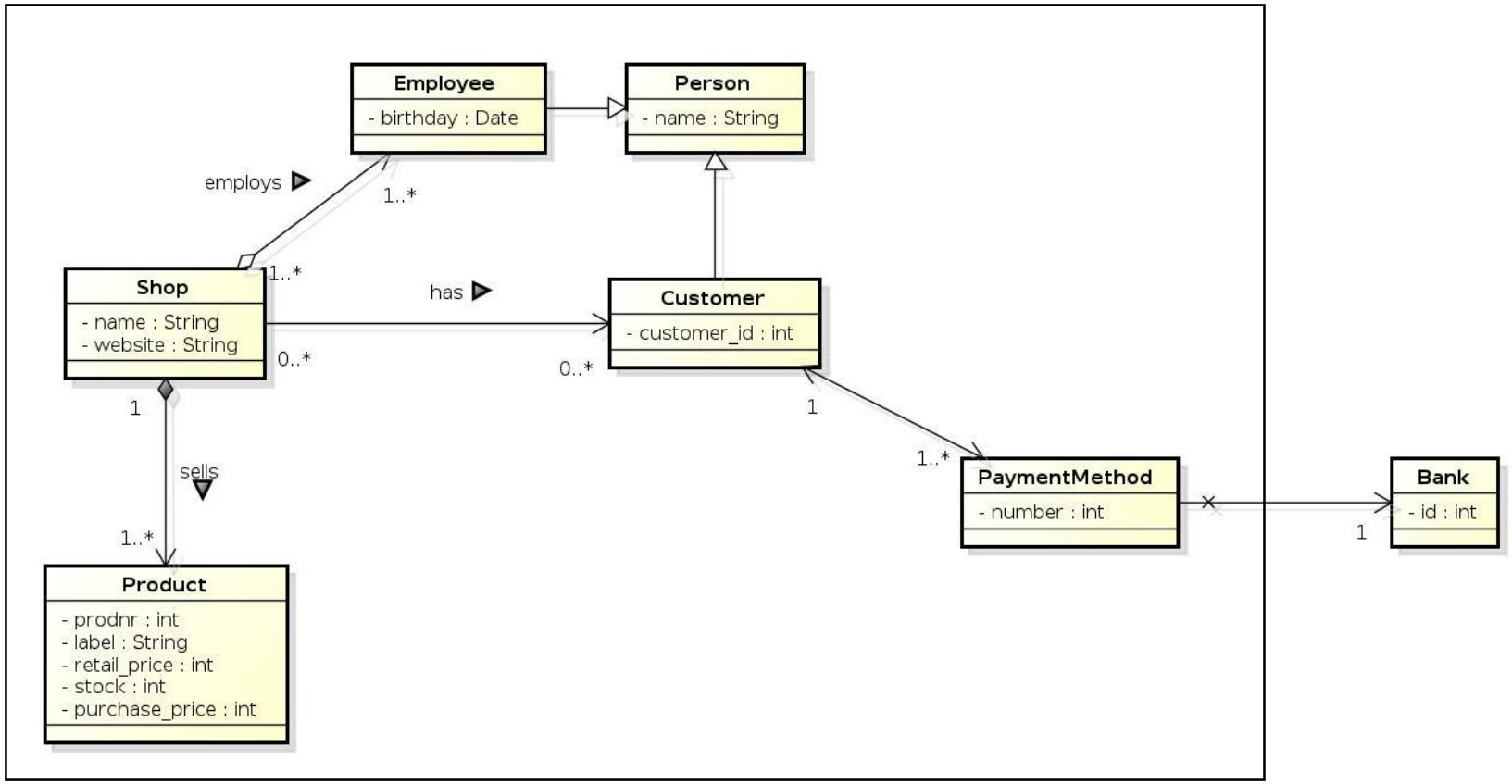
Nochmal Navigierbarkeit



Vom Klassenmodell zum Systemklassenmodell

- Klassenmodell beschreibt alle Klassenmodell, die in dem Kontext eine gewisse Rolle spielen
- Auf dem Weg zum Entwurf muss jedoch entschieden werden, was Teil des Systems sein wird und was nicht
- Systemgrenzen müssen festgelegt werden
- Dies erfordert es ggf. Schnittstellen zu definieren um zu beschreiben, wie über Systemgrenzen hinweg interagiert werden kann

Systemklassenmodell



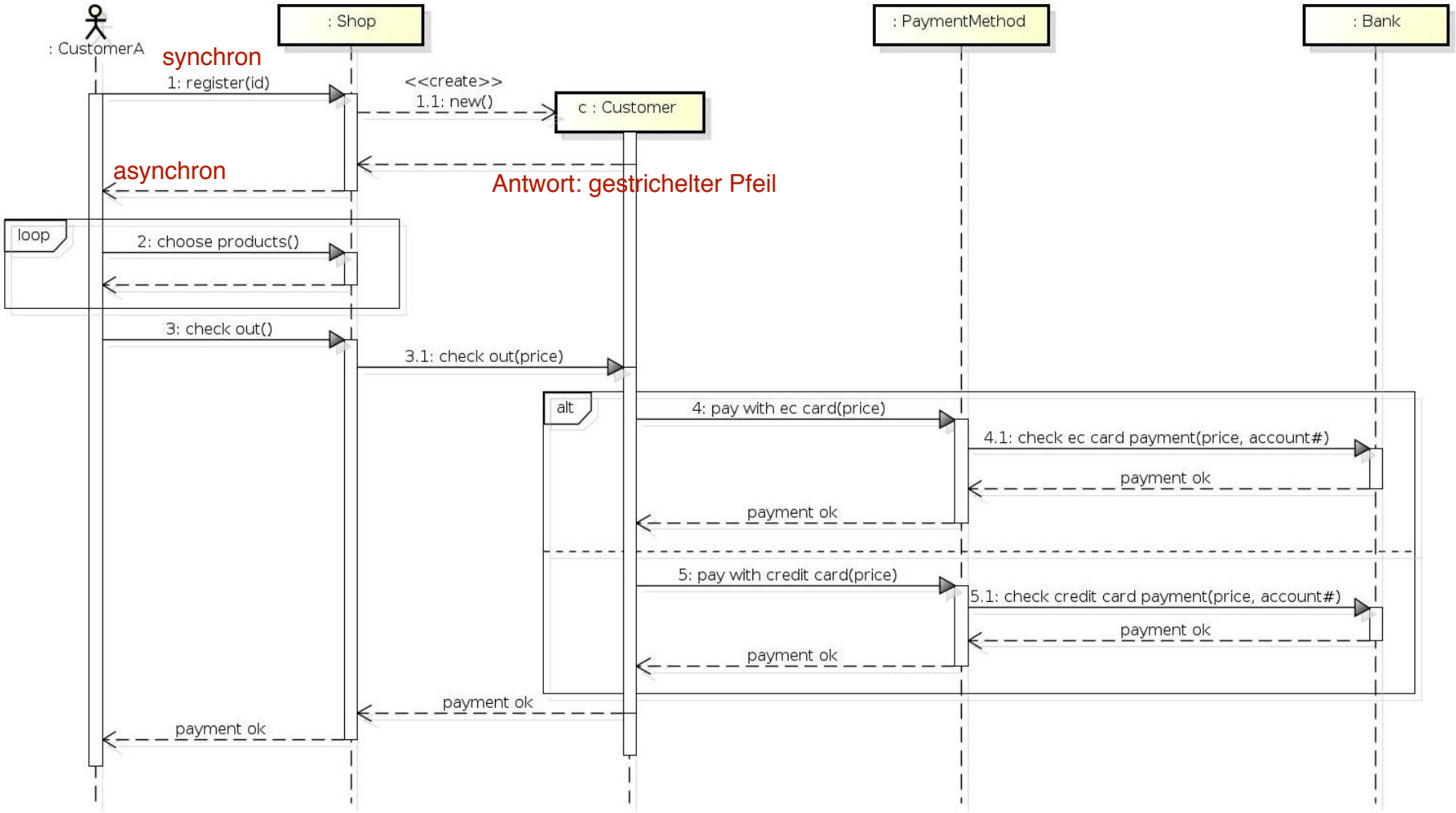
Übersicht

- Analyse-Phase
  - Use Cases (siehe letzte VL)
  - Sequenzdiagramme (siehe letzte VL)
  - Klassenmodell des Gegenstandsbereichs
  - Systemklassendiagramm
- Design-Phase
  - Erweiterte Sequenzdiagramme
  - Aktivitätsdiagramme
  - Statecharts
  - Design-By-Contract (OCL)

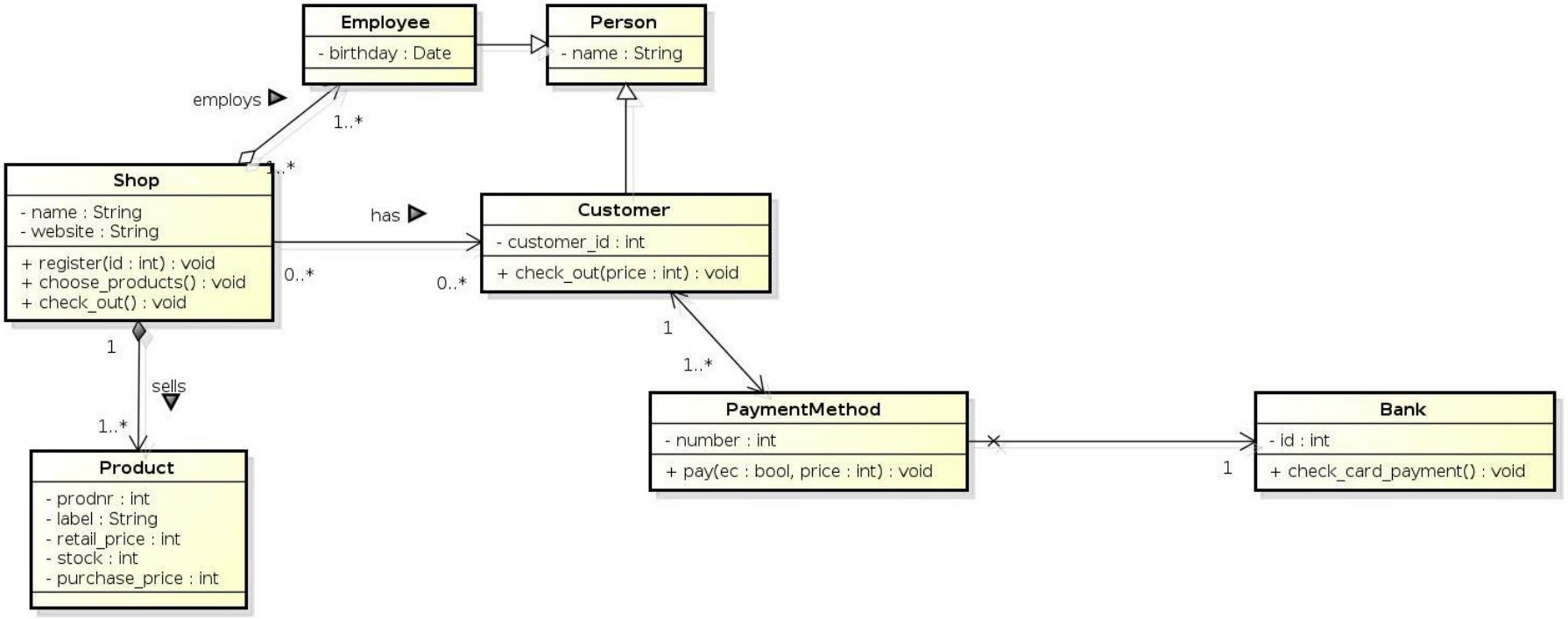
Analyse von Use Case Diagrammen und Sequenzdiagrammen

- Bisher Use Cases und Sequenzdiagramme als abstrakte Requirements, die das System als Ganzes betrachten
- Aufgabe: Verfeinern der vorhandenen Sequenzdiagrammen, die die Interaktionen innerhalb des Systems mit einbeziehen
- Weiterhin: Hinzufügen von Szenarien, die in den Requirements noch nicht berücksichtigt werden konnten

Beispiel


: CustomerA
: Shop
: PaymentMethod
: Bank
synchron
1: register(id)
<<create>>
1.1: new()
c : Customer
asynchron
Antwort: gestrichelter Pfeil
loop
2: choose products()
3: check out()
3.1: check out(price)
alt
4: pay with ec card(price)
4.1: check ec card payment(price, account#)
payment ok
payment ok
5: pay with credit card(price)
5.1: check credit card payment(price, account#)
payment ok
payment ok
payment ok
payment ok

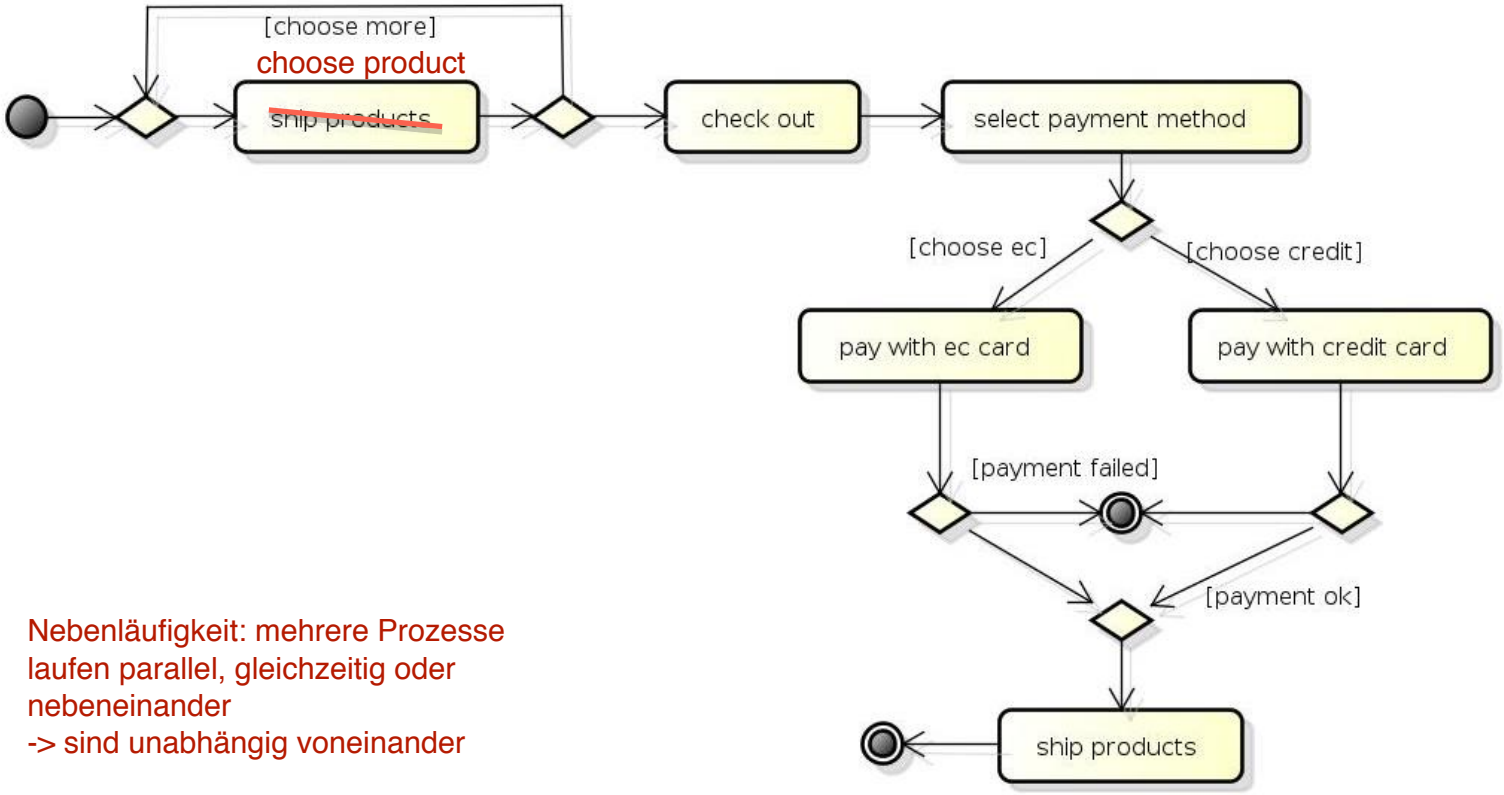
Aufnehmen der entsprechenden Methoden in das Klassendiagramm



Von Sequenzdiagrammen zu Aktivitätsdiagrammen

- Beschreibung der Systemoperation als Reihenfolge von Aktionen
- Pro Use Case ein Aktivitätsdiagramm, das eine vollständige Beschreibung der Szenarien ermöglicht
- Vollständige Erfassung der Szenarien aus den zugehörigen Sequenzdiagrammen innerhalb eines Aktivitätsdiagramms

Beispiel

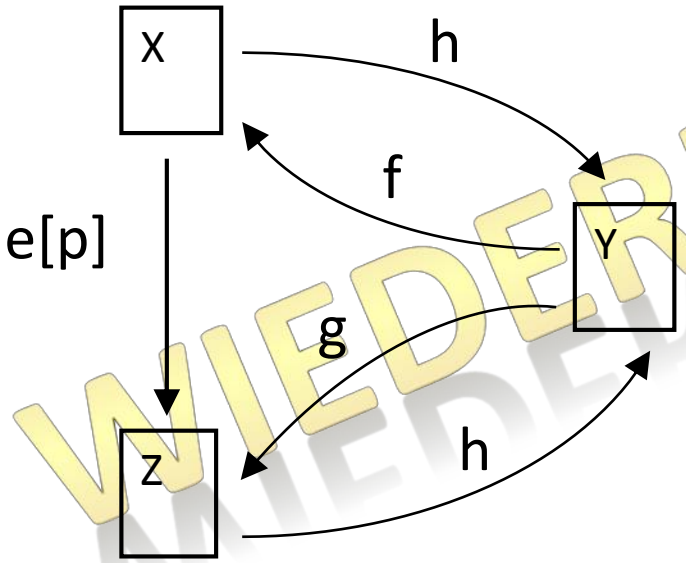


Nebenläufigkeit: mehrere Prozesse laufen parallel, gleichzeitig oder nebeneinander
-> sind unabhängig voneinander

Übersicht

- Analyse-Phase
  - Use Cases (siehe letzte VL)
  - Sequenzdiagramme (siehe letzte VL)
  - Klassenmodell des Gegenstandsbereichs
  - Systemklassendiagramm
- Design-Phase
  - Erweiterte Sequenzdiagramme
  - Aktivitätsdiagramme
  - Statecharts
  - Design-By-Contract (OCL)

Endliche Automaten



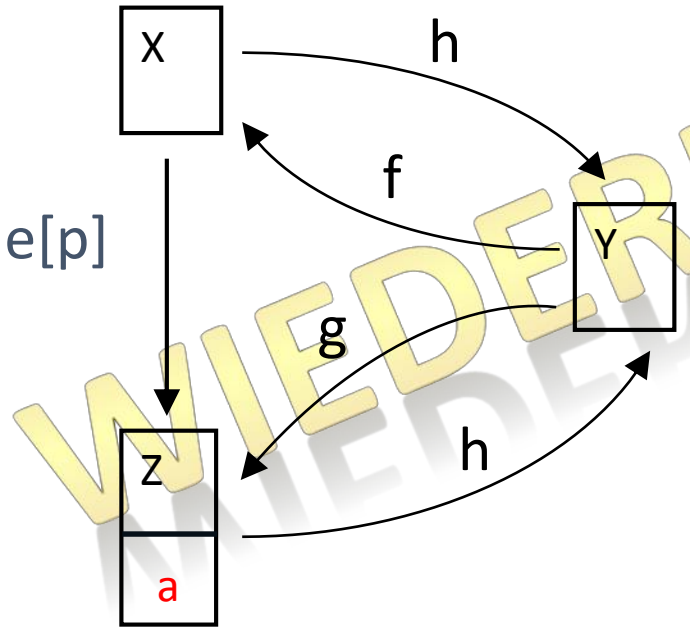
- *finite state machines* (FSM)
- **Zustandsdiagramm**
- **Übergänge** markiert mit
  - Externe Ereignissen (f,g,h,e)
  - Bedingungen (p)
- Kann Ereignisse **aussenden**
- Beispiel:
  - Ereignis e bewirkt Übergang von Zustand X nach Zustand Z, falls Bedingung p erfüllt ist.

Endliche Automaten

Zwei Arten von FSMs, *Moore* und *Mealy*

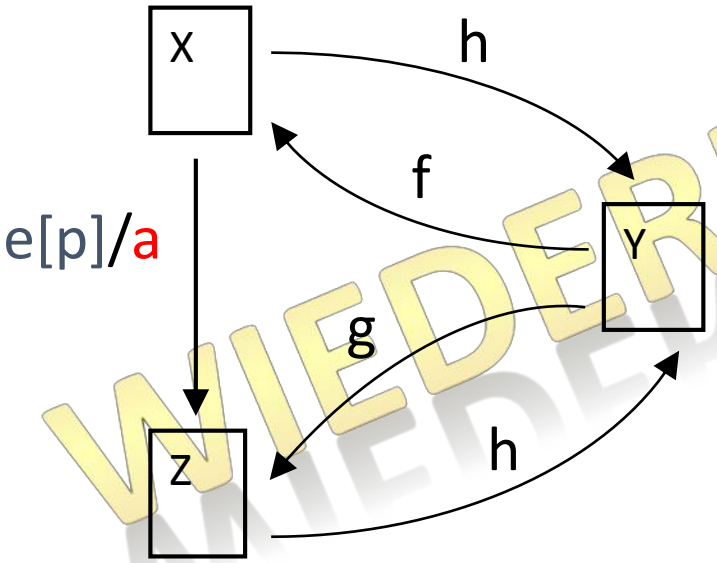
- Für **beide** gilt:
  - Kreise/Kästchen = **Zustände** (*Vertices*)
  - Pfeile = **Übergänge/Transitionen** (*Edges*)
  - Transitionen werden durch **Ereignisse** ausgelöst und durch **Bedingungen** (*Guards*) abgesichert
- **Unterschiedliches** Verhalten nur beim **Aussenden** von Ereignissen

Moore Automaten



- Ausgabeereignisse **in den Zuständen**
- Wenn der Automat im Zustand X ist und Ereignis e auftritt und Bedingung p erfüllt ist, gelangt der Automat in den Zustand Z.
- **Solange** Automat in Zustand Z wird das Ereignis a gesendet (beliebig viele Takte)

Mealy Automaten



- Ausgabeereignisse **an den Transitionen**
- Wenn der Automat im Zustand X ist und Ereignis e auftritt und Bedingung p erfüllt ist, gelangt der Automat in den Zustand Z.
- Dabei wird **einmalig** Ereignis a gesendet (genau einen Takt).

Mathematisches Modell

Unterschied zu TheGi2 Definition!!

EA = (*S*, *s0*, *Σ*, *Γ*, *δ*, *ω*)

S = endliche Menge an **Zuständen**

s0 = eindeutiger **Startzustand** (aus S)

Σ = endliche Menge an **Eingabeereignissen**

Γ = endliche Menge an **Ausgabeereignissen**

δ = **Zustandsüberführungsfunktion** (S x Σ -> S)

-> bei nicht-determ. Automaten: Überführungsrelation

ω = **Ausgabefunktion** (S x Σ -> Γ oder S -> Γ)

Mealy Moore

Moore vs. Mealy

Welches Automatenmodell ist mächtiger?

Keins. Beide Modelle lassen sich **ineinander überführen**, sind also gleich mächtig!

Welches Automatenmodell hat mehr Zustände?

Moore Automaten besitzen **mindestens** so viele Zustände wie äquivalente **Mealy** Automaten!

Zustandsvariable für Zustände (enum, Zahl)
-> Case-Switch + Ausgabe

Vorteile

- **Formale** Spezifikation
  - Ermöglicht **Simulation** und **Verifikation** Funktioniert ein System, wie geplant?
- **Grafische Darstellung**
- **Geeignet für** HW und SW Hardware & Software

WIEDERHOLUNG

Nachteile

- Nur atomare Zustände
  - Hierarchischer Entwurf mit Verfeinerungen nicht möglich
- **Zustandsexplosion** bei Nebenläufigkeit

WIEDERHOLUNG

Lösung: Statecharts mit Nebenläufigkeit, z.B. warten parallel auf verschiedene Bedingungen statt alle möglichen Zustände, die sich aus verschiedenen Reihenfolgen ergeben einzeln darzustellen

Zustandsexplosion

Endliche Automaten haben **keine** **Möglichkeit** Nebenläufigkeit zu modellieren

- **Ausmultiplizieren** des gesamten möglichen Verhaltens nötig
- **Exponentieller** **Anstieg** der Zustände

Zustandsexplosion

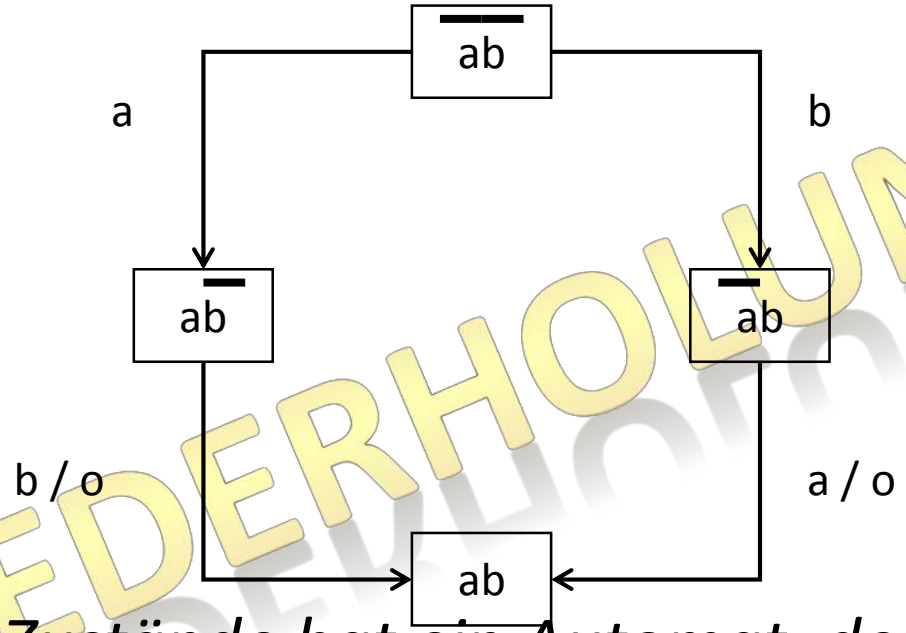
Beispiel:

- Endlicher Automat wartet **gleichzeitig** auf zwei Eingabesignale (a, b)
- Eingabesignale können in **beliebiger Reihenfolge** kommen
- Automat sendet Ausgabesignal (o), wenn **beide** Eingabesignale angekommen sind

Wie viele Zustände hat der Automat?

2^2 = 4

Zustandsexplosion



Wie viele Zustände hat ein Automat, der auf 3 (4, 5, 100) Eingabesignale wartet?

8 16 32 2^{100}

Statecharts

erweitern endliche Automaten, um Nebenläufigkeit besser darstellen zu können

- Eignen sich besonders um das Kommunikationsverhalten von nebenläufigen Systemen grafisch zu modellieren (und zu analysieren)
- Übergangsdiagramm mit Und- / Oder-Konstrukten, Hierarchie und Verfeinerung und Broadcast-Kommunikation

Modellierung größerer Systeme

Betrachte folgende Aussagen:

- In allen Fahrzuständen führt Bremsen zum Stillstand.
- Das Radio funktioniert gleich, egal ob man fährt oder steht
- Wenn die Kupplung betätigt wurde, wähle den Gang.
- Abfahren besteht aus Angurten, Zündung einschalten, Gang einlegen und Gas geben.

Konsequenz für Behandlung der Zustände:

- Zusammenfassen von Zuständen zu einem Oberzustand
- Erklären der Unabhängigkeit (Orthogonalität) von Zuständen
- Verlangt nach Ausgabe, falls Vorbedingung erfüllt
- Verfeinerung eines Zustands

Statechart-Konstrukte

Erweiterungen in Zustandsdiagrammen:

- Oder-Zustand (XOR-Komposition)
- Und-Zustand (AND-Komposition)
- Hierarchie und Verfeinerung
- Broadcast Ereignis wird in allen Teilen des Systems sichtbar, auch in den nebenläufigen Prozessen
- Zusammenführung
- Historie

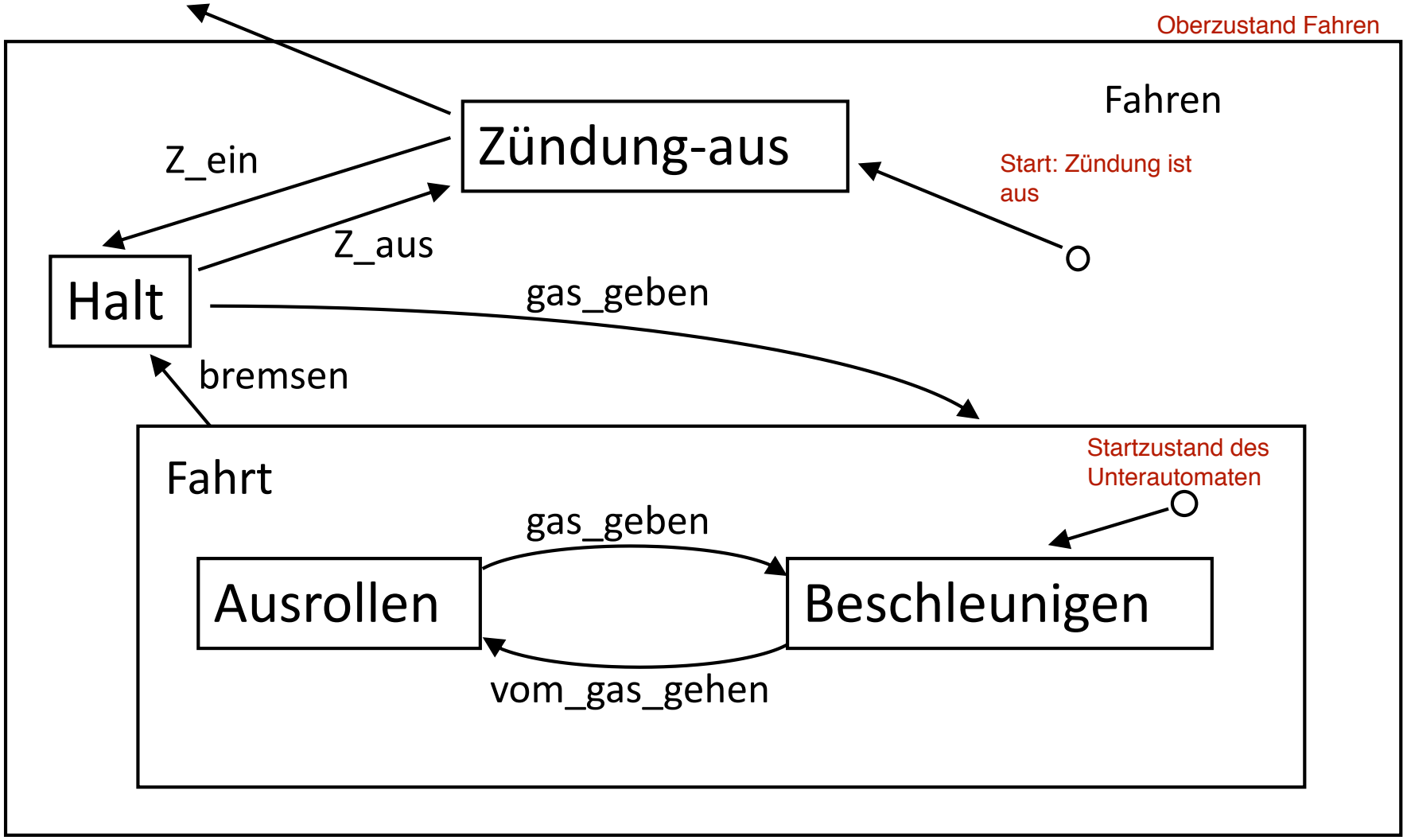
Oder-Zustand

- Zusammenfassung eines Teils des Automaten zu einem Oberzustand
- Innerhalb Oberzustand:
  - interner Anfangszustand (Default-Zustand)
  - wird bei Übergang in Oberzustand eingenommen
  - auch Übergänge in bzw. aus beliebigen internen Zuständen heraus möglich
  - Oberzustand selbst ist kein Zustand, d.h. Automat ist immer in einem der internen Zustände
- Beispiel Auto: Alle Zustände, in denen PKW fährt, bilden Oberzustand. Ereignis „Bremsen“ führt heraus.

Und-Zustand

- Teilen eines Oberzustands in Teilautomaten
- Parallele Ausführung der Teilautomaten
- Darstellung im Diagramm: Trennung durch gestrichelte Linien
- Bei Übergang in Und-Zustand:
  - Jeder Teilautomat wird in seinem Anfangszustand gestartet

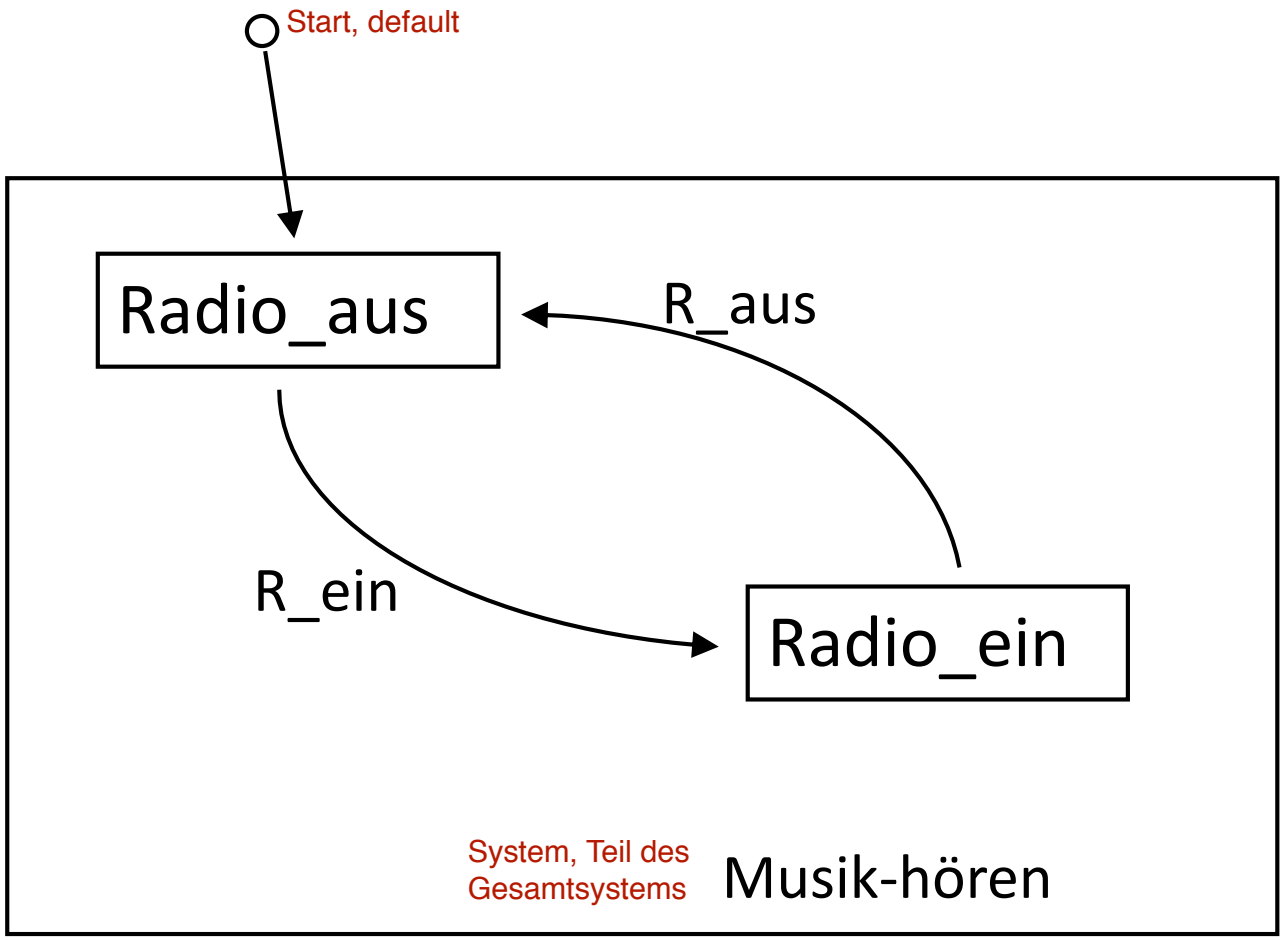
Beispiel Automat für Auto



Hierarchie und Verfeinerung

- Oberzustand aktiv gdw. einer der Unterzustände aktiv
- höchstens ein Unterzustand aktiv
- schematisch dargestellt durch Schachtelung
- Vorteile
  - Reduktion: Zusammenfassen ähnlicher Zustände, insgesamt weniger Kanten
  - entspricht Entwurf mittels Verfeinerung

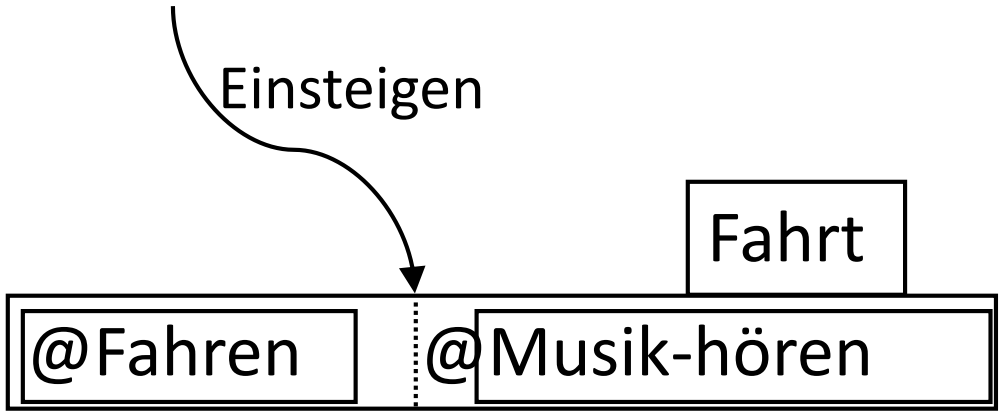
Weiteres Beispiel: Musik im Auto



Broadcast-Kommunikation

- Ereignisse in Statecharts sind global sichtbar
- das Auslösen eines Ereignisses stellt *Broadcast* an alle Teilautomaten dar
- Ereignisse können daher zur Synchronisation nebenläufiger Prozesse verwendet werden
- Eintritt eines Ereignisses löst Übergänge in allen Teilautomaten aus

Beispiel Und-Komposition



@... bezeichnet Zustandsnamen eines Oberzustands, dessen genaue Definition an anderer Stelle angegeben ist.

Join (Zusammenführung)

- Mehrere Übergänge, ausgehend von verschiedenen Teilautomaten eines Und-Zustands, können auf gemeinsamen Nachfolgezustand führen, der nicht zum Und-Zustand gehört.
- Diese Übergänge werden gemeinsam durchgeführt.
- Voraussetzung dafür: alle Ereignisse und Bedingungen, die zu diesen Übergängen führen, müssen eingetreten sein.

History (Gedächtnis)

- Problem: Bei Eintritt in Oberzustand soll vorangegangener Zustand wieder eingenommen werden.
- Gelöst durch History (H)
  - erinnert letzten Zustand auf oberster Ebene
- Rekursive History (H*)
  - auch rekursiv für alle internen Zustände

Charakterisierung von Statecharts

Statecharts =

Übergangsdiagramm
+ Hierarchie
+ Orthogonalität
+ Broadcast-Kommunikation

zwei Typen: synchrone und asynchrone Semantik

Interpretation der Statecharts teilweise frei, z.B. bei Nebenläufigkeit

Hier fehlen Folien zu Syntax und Semantik von Statecharts, siehe aktuellere Folienversion

Übersicht

- Analyse-Phase
  - Use Cases (siehe letzte VL)
  - Sequenzdiagramme (siehe letzte VL)
  - Klassenmodell des Gegenstandsbereichs
  - Systemklassendiagramm
- Design-Phase
  - Erweiterte Sequenzdiagramme
  - Aktivitätsdiagramme
  - Statecharts
  - Design-By-Contract (OCL)

Spezifikation von Methoden mit Contracts

- Beschreibung von Methoden mit Vor- und Nachbedingungen
- Aufrufer und Implementierer einer Methode müssen sich an Vertrag halten

- Aufnehmen eines neuen Mitarbeiters

  context Shop::hireEmployee(p) : void
  pre: not employee->contains(p)
  post: employee = employee@pre -> including(p)

Spezifikation von Methoden mit Contracts

- Object Constraint Language zur Beschreibung von Bedingungen auf Objektdiagrammen
- Bietet auch die Möglichkeit Contracts zu beschreiben

Mehr dazu in der nächsten Vorlesung

Zusammenfassung

- Analyse-Phase: „Was macht das System und was macht es nicht?“
- Design-Phase: „Wie erfüllt das System seine Aufgaben intern?“

- Strukturdiagramme insbes. Klassendiagramm
- Klassendiagramm: Objektorientierung
  - Attribute, Assoziationen, Multiplizitäten, Navigierbarkeit
  - Aggregation/Komposition, Generalisierung/Spezialisierung

- Verfeinerte Sequenzdiagramme
- Aktivitätsdiagramme
- Statecharts
- Contracts zur Spezifikation von Methoden