UNIVERSITÄT
DUISBURG
ESSEN

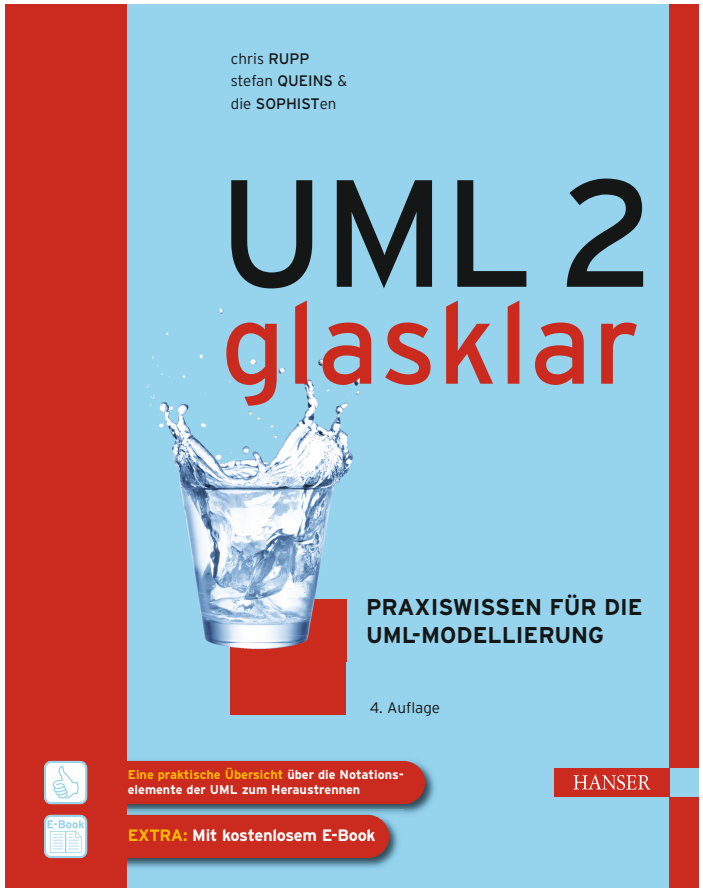
Offen im Denken

Modellierung

Prof. Janis Voigtländer ■ Folienversion: 05.01.2022, 16:46:12 +00:00

Literatur

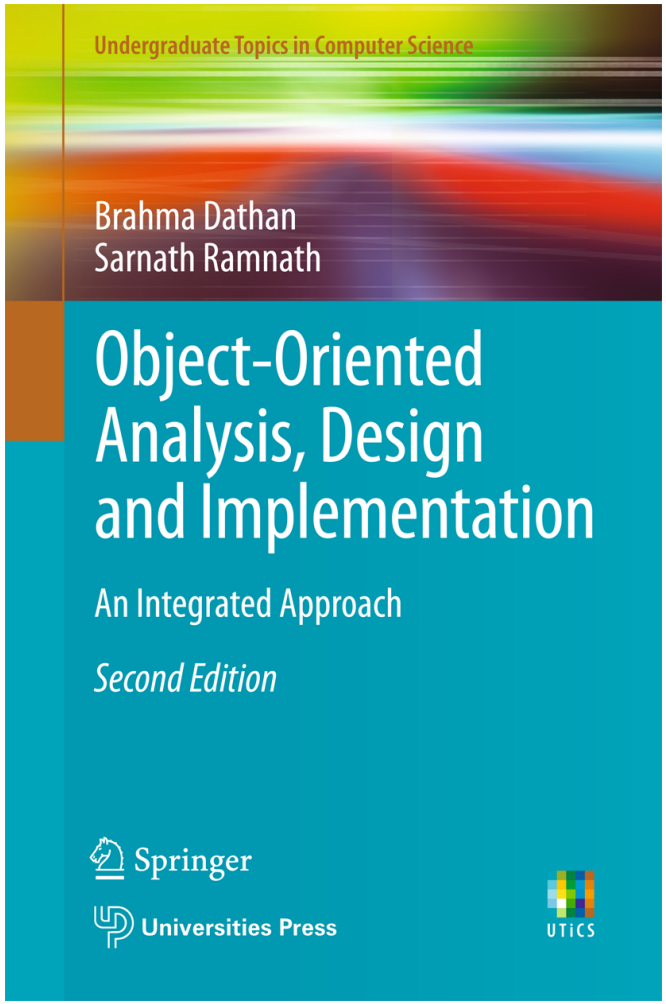
Chris Rupp, Stefan Queins.
UML 2 glasklar.
Hanser Fachbuch, 2012



Buch ist in der Bibliothek verfügbar.

Literatur

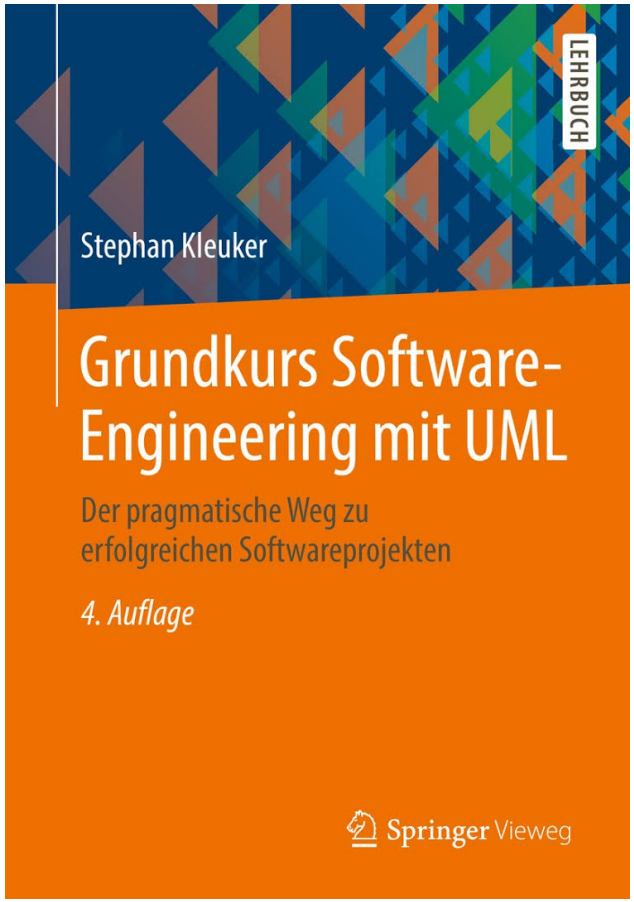
Brahma Dathan,
Sarnath Ramnath.
Object-Oriented Analysis, Design and Implementation – An Integrated Approach.
Springer, 2015



Buch ist in der Bibliothek verfügbar.

Literatur

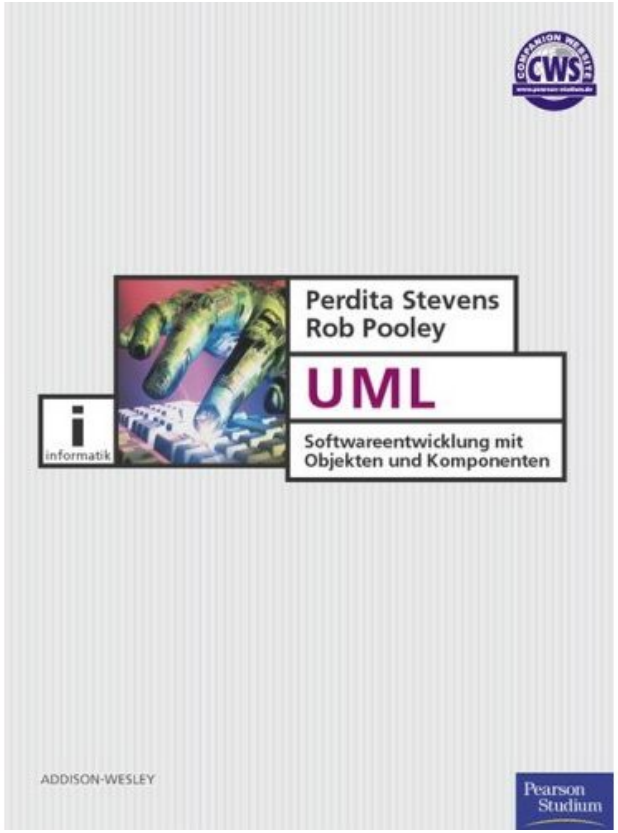
Stephan Kleuker.
Grundkurs Software-Engineering mit UML.
Springer, 2018



https://dx.doi.org/10.1007/978-3-658-19969-2
(elektronische Version über den Uni-Account)

Literatur

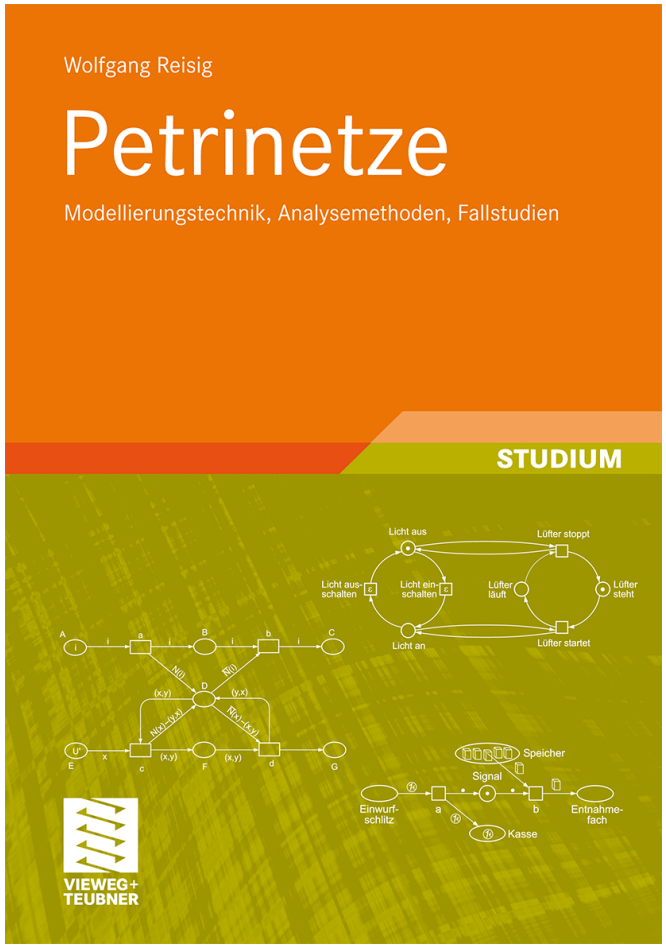
Perdita Stevens, Rob Pooley.
UML – Softwareentwicklung mit Objekten und Komponenten.
Pearson, 2001



Das englische Original ist in der Bibliothek verfügbar.

Literatur

Wolfgang Reisig.
Petrinetze –
Modellierungstechnik,
Analysemethoden, Fallstudien.
Springer, 2010



https://dx.doi.org/10.1007/978-3-8348-9708-4
(elektronische Version über den Uni-Account)

Literatur

Lutz Priese, Harro Wimmel.
Petri-Netze.
Springer, 2008

https://dx.doi.org/10.1007/978-3-540-76971-2
(elektronische Version über den Uni-Account)

Literatur

Tadao Murata.
Petri Nets: Properties, Analysis and Applications.
Proc. of the IEEE, 77(4), pages 541–580, 1989

Petri Nets: Properties, Analysis and Applications

TADAO MURATA, FELLOW, IEEE

Invited Paper

This is an invited tutorial-review paper on Petri nets—a graphical and mathematical modeling tool. Petri nets are a promising tool for describing and studying information processing systems that are characterized as being concurrent, asynchronous, distributed, parallel, nondeterministic, and/or stochastic.

The paper starts with a brief review of the history and the application areas considered in the literature. It then proceeds with introductory modeling examples, behavioral and structural properties, three methods of analysis, subclasses of Petri nets and their analysis. In particular, one section is devoted to marked graphs—the concurrent system model most amenable to analysis. In addition, the paper presents introductory discussions on stochastic nets with their application to performance modeling, and on high-level nets with their application to logic programming. Also included are recent results on reachability criteria. Suggestions are provided for further reading on many subject areas of Petri nets.

I. INTRODUCTION

Petri nets are a graphical and mathematical modeling tool applicable to many systems. They are a promising tool for describing and studying information processing systems that are characterized as being concurrent, asynchronous, distributed, parallel, nondeterministic, and/or stochastic. As a graphical tool, Petri nets can be used as a visual communication aid similar to flow charts, block diagrams, and networks. In addition, tokens are used in these nets to simulate the dynamic and concurrent activities of systems. As a mathematical tool, it is possible to set up state equations, algebraic equations, and other mathematical models governing the behavior of systems. Petri nets can be used by both practitioners and theoreticians. Thus, they provide a powerful medium of communication between them: practitioners can learn from theoreticians how to make their models more methodical, and theoreticians can learn from practitioners how to make their models more realistic.

Historically speaking, the concept of the Petri net has its origin in Carl Adam Petri's dissertation [1], submitted in 1962 to the faculty of Mathematics and Physics at the Technical University of Darmstadt, West Germany. The dissertation was prepared while C. A. Petri worked as a scientist at the University of Bonn. Petri's work [1], [2] came to the attention of A. W. Holt, who later led the Information System Theory Project of Applied Data Research, Inc., in the United States. The early developments and applications of Petri nets (or their predecessors) are found in the reports [3]–[8] associated with this project, and in the Record [9] of the 1970 Project MAC Conference on Concurrent Systems and Parallel Computation. From 1970 to 1973, the Computation Structure Group at MIT was most active in conducting Petri-net related research, and produced many reports and theses on Petri nets. In July 1975, there was a conference on Petri Nets and Related Methods at MIT, but no conference proceedings were published. Most of the Petri-net related papers written in English before 1980 are listed in the annotated bibliography of the first book [10] on Petri nets. More recent papers up until 1984 and those works done in Germany and other European countries are annotated in the appendix of another book [11]. Three tutorial articles [12]–[14] provide a complimental, easy-to-read introduction to Petri nets.

Since the late 1970's, the Europeans have been very active in organizing workshops and publishing conference proceedings on Petri nets. In October 1979, about 135 researchers mostly from European countries assembled in Hamburg, West Germany, for a two-week advanced course on General Net Theory of Processes and Systems. The 17 lectures given in this course were published in its proceedings [15], which is currently out of print. The second advanced course was held in Bad Honnef, West Germany, in September 1986. The proceedings [16], [17] of this course contain 34 articles, including two recent articles by C. A. Petri; one [18] is concerned with his axioms of concurrency theory and the other [19] with his suggestions for further research. The first European Workshop on Applications and Theory of Petri Nets was held in 1980 at Strasbourg, France. Since then, this series of workshops has been held every year at different locations in Europe: 1981, Bad Honnef, West Germany; 1982, Varenna, Italy; 1983, Toulouse, France; 1984, Aarhus, Denmark; 1985, Espoo, Finland; 1986, Oxford, England;

Manuscript received May 28, 1988; revised November 4, 1988. This work was supported by the National Science Foundation under Grant DMC-8518588.

The author is with the Department of Electrical Engineering and Computer Science, University of Illinois, Chicago, IL 60680, USA.

IEEE Log Number 8826798.

0018-9219/89/0400-0541$01.00 © 1989 IEEE

https://dx.doi.org/10.1109/5.24143
(elektronische Version über den Uni-Account)

Literatur

David Harel.
Statecharts: A visual formalism for complex systems.
Science of Computer Programming, 8,
pages 231–274, 1987

Science of Computer Programming 8 (1987) 231-274
North-Holland

STATECHARTS: A VISUAL FORMALISM FOR COMPLEX SYSTEMS*

David HAREL
Department of Applied Mathematics, The Weizmann Institute of Science, Rehovot, Israel

Communicated by A. Pnueli
Received December 1984
Revised July 1986

Abstract. We present a broad extension of the conventional formalism of state machines and state diagrams, that is relevant to the specification and design of complex discrete-event systems, such as multi-computer real-time systems, communication protocols and digital control units. Our diagrams, which we call statecharts, extend conventional state-transition diagrams with essentially three elements, dealing, respectively, with the notions of hierarchy, concurrency and communication. These transform the language of state diagrams into a highly structured and economical description language. Statecharts are thus compact and expressive—small diagrams can express complex behavior—as well as compositional and modular. When coupled with the capabilities of computerized graphics, statecharts enable viewing the description at different levels of detail, and make even very large specifications manageable and comprehensible. In fact, we intend to demonstrate here that statecharts counter many of the objections raised against conventional state diagrams, and thus appear to render specification by diagrams an attractive and plausible approach. Statecharts can be used either as a stand-alone behavioral description or as part of a more general design methodology that deals also with the system's other aspects, such as functional decomposition and data-flow specification. We also discuss some practical experience that was gained over the last three years in applying the statechart formalism to the specification of a particularly complex system.

1. Introduction

The literature on software and systems engineering is almost unanimous in recognizing the existence of a major problem in the specification and design of large and complex *reactive systems*. A reactive system (see [14]), in contrast with a *transformational system*, is characterized by being, to a large extent, event-driven, continuously having to react to external and internal stimuli. Examples include telephones, automobiles, communication networks, computer operating systems, missile and avionics systems, and the man-machine interface of many kinds of ordinary software. The problem is rooted in the difficulty of describing reactive behavior in ways that are clear and realistic, and at the same time formal and

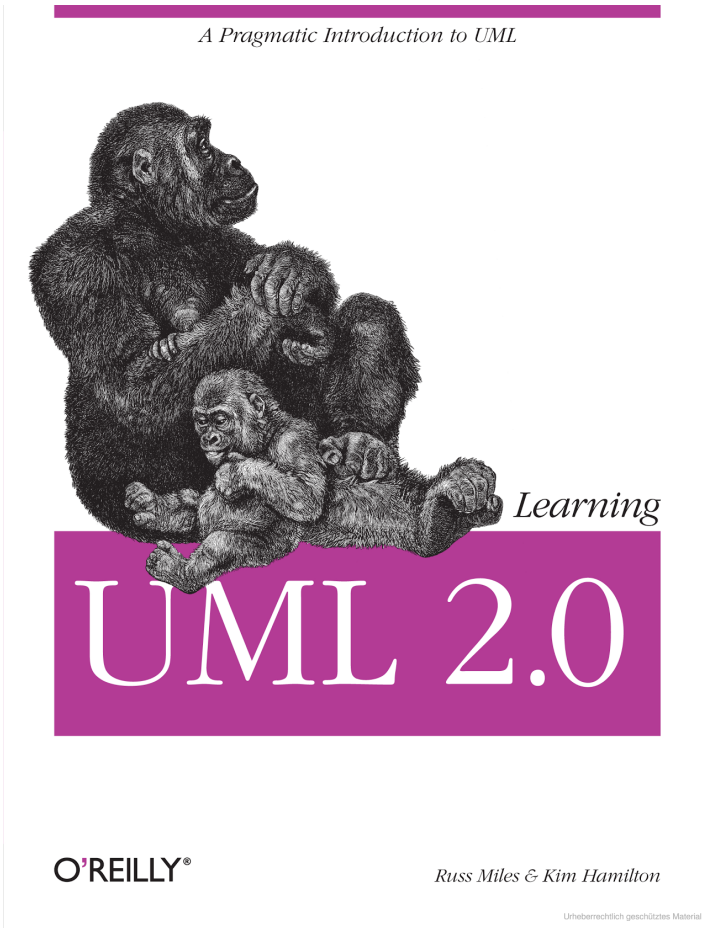
* The initial part of this research was carried out while the author was consulting for the Research and Development Division of the Israel Aircraft Industries (IAI), Lod, Israel. Later stages were supported in part by grants from IAI and AD CAD, Ltd.

0167-6423/87/$3.50 © 1987, Elsevier Science Publishers B.V. (North-Holland)

https://dx.doi.org/10.1016/0167-6423(87)90035-9

Literatur

Russ Miles, Kim Hamilton.
Learning UML 2.0.
O'Reilly, 2006



Buch ist in der Bibliothek verfügbar.

Einführung in die Modellierung

Was ist Modellierung?

UNIVERSITÄT DUISBURG ESSEN
Offen im Denken

Modell

Ein Modell ist eine Repräsentation eines Systems von Objekten, Beziehungen und/oder Abläufen. Ein Modell vereinfacht und abstrahiert dabei im Allgemeinen das repräsentierte System.

System

Der Begriff System wird hier sehr allgemein verwendet. Er kann

- einen Teil der Realität oder ein (noch) nicht bestehendes Gebilde;
- etwas Gegenständliches oder Virtuelles

bezeichnen.

Was ist Modellierung?

UNIVERSITÄT DUISBURG ESSEN
Offen im Denken

Modellierung

Modellierung ist der Prozess, bei dem ein Modell eines Systems erstellt wird.

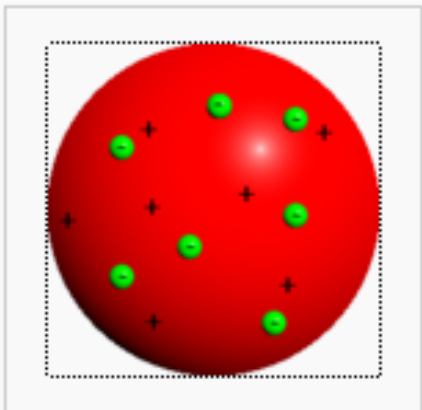
Warum wird überhaupt modelliert?

⇝ Um ein System zu entwerfen, besser zu verstehen, zu visualisieren, zu simulieren, . . .

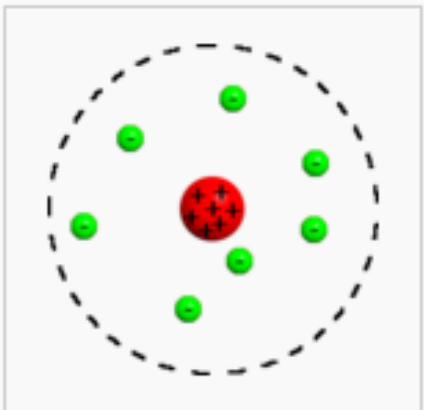
Um etwas konkreter zu werden, betrachten wir kurz klassische Beispiele aus verschiedenen Disziplinen (Physik, Biologie, Klimaforschung).

Atommodelle

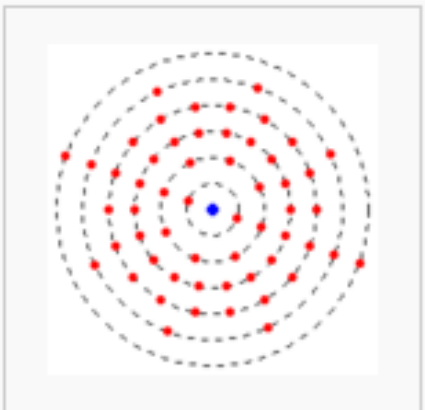
Atome bestehen aus Protonen, Neutronen und Elektronen.
Wie diese Teilchen zusammenwirken, wird in Atommodellen beschrieben, die im Laufe der Zeit immer wieder verändert wurden (obwohl sich die Natur von Atomen ja nicht verändert hat).



Thomsonsches Atommodell



Rutherfordsches Atommodell

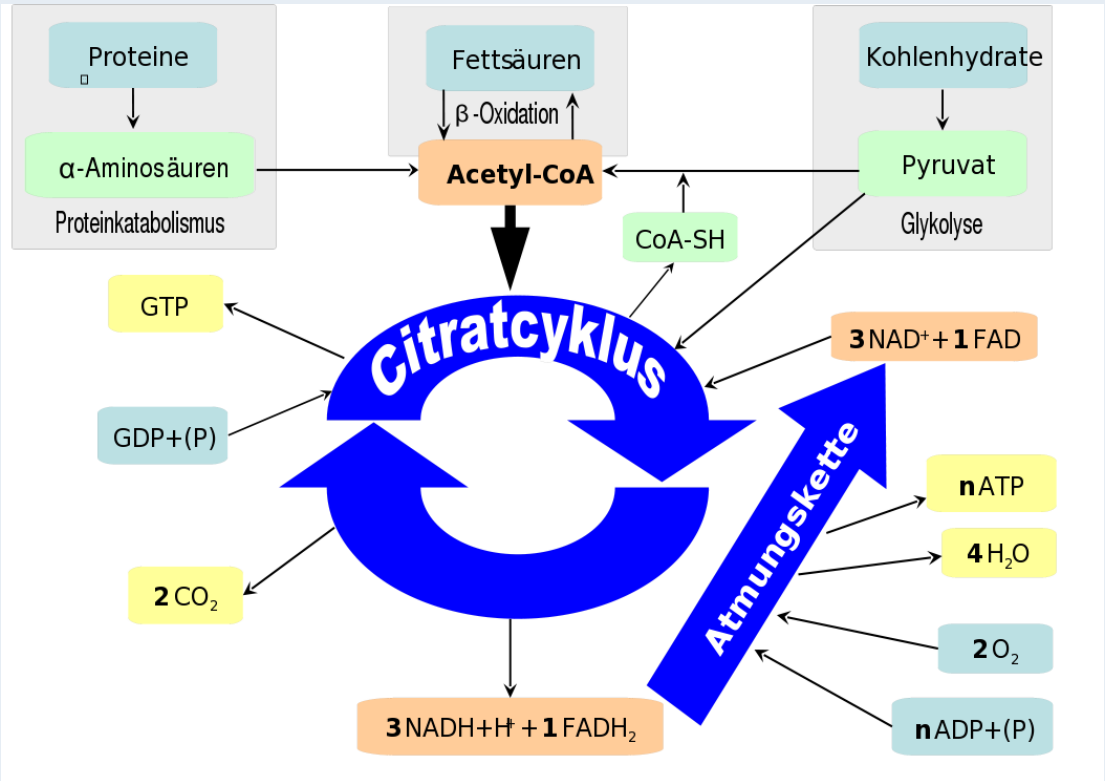


Bohrsches Atommodell

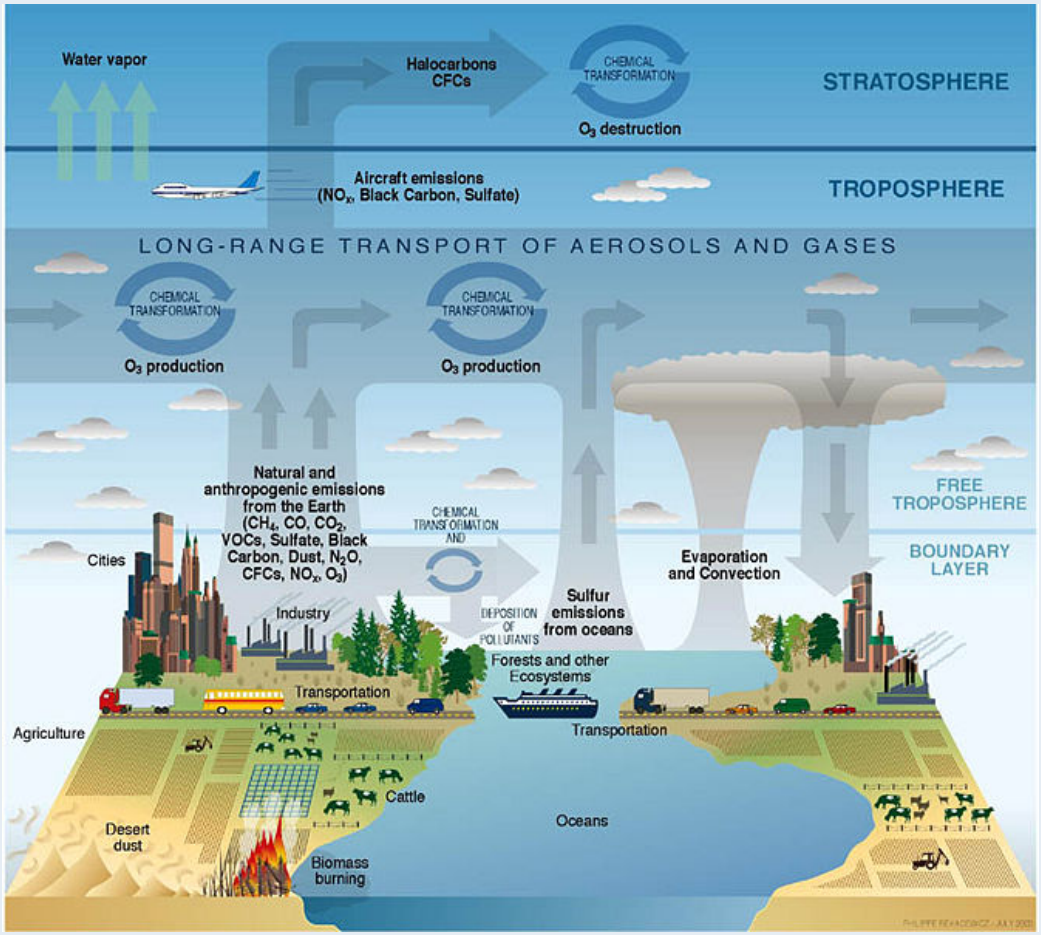
Orbitalmodell

Zitronensäurezyklus

Der Zitronensäurezyklus oder Citratzyklus modelliert den Abbau organischer Stoffe im Körper.



Modell des Transports von Gasen in der Atmosphäre



Arten von Modellen

visuell vs. textuell

Nicht alle Modelle sind visuell bzw. grafisch. Auch mit textuellen Beschreibungen und Formeln kann man modellieren (siehe beispielsweise mathematische Modelle, Logik, Algebra).

Dennoch werden häufig grafische Darstellungen benutzt, auch aus didaktischen Gründen und um sich besser über die Modelle verständigen zu können.

Arten von Modellen

UNIVERSITÄT DUISBURG ESSEN
Offen im Denken

qualitativ vs. quantitativ

- qualitative Modelle: Welche Objekte gibt es? Welche Merkmale und Beziehungen zueinander haben sie? Was passiert? Warum passiert es? In welcher Reihenfolge geschehen die Ereignisse? Was sind die kausalen Zusammenhänge? Welche Phänomene treten auf?
- quantitative Modelle: Wieviele Objekte gibt es? In welchem Mengenverhältnis zueinander treten verschiedene Arten von Objekten auf? Wie lange dauert ein Vorgang? Wie wahrscheinlich ist ein bestimmtes Ereignis?

Arten von Modellen

black box vs. white box (oder glass box)

- black box: Nur das von außen beobachtbare Verhalten wird beschrieben.
- white box: Es wird auch beschrieben, wie das von außen beobachtbare Verhalten im „Inneren“ des Systems erzeugt wird.

Arten von Modellen

statisch vs. dynamisch

- Ein statisches Modell beschreibt den Aufbau oder einen bestimmten Zustand des Systems.
- Ein dynamisches Modell beschreibt hingegen auch, wie das System sich entwickelt (ein oder mehrere mögliche Abläufe oder sogar das gesamte Systemverhalten).

Arten von Modellen

formal vs. semi-formal vs. nicht-formal

Je nach Exaktheit der Modelle erhält man:

- formale Modelle, die vollkommen exakt in ihren Aussagen sind (vor allem mathematische Modelle)
- semi-formale Modelle, die teilweise exakt sind, jedoch nicht alles vollständig spezifizieren
- nicht-formale Modelle, die als grobe Richtlinie dienen können, jedoch eher vage Aussagen machen

Wozu ist Modellierung gut?

UNIVERSITÄT DUISBURG ESSEN
Offen im Denken

Wozu benötigt man in der Informatik Modelle?

Je komplexer ein Informatik-System sein wird, desto wichtiger ist es, einen Plan zu haben, während man es konstruiert.

Dies führt idealerweise zu:

- Vermeidung von Fehlern
- besserer Qualität
- niedrigeren Kosten
- besserer Dokumentation und Wiederverwendbarkeit

Modellierung ist in der Informatik weniger verbreitet als in den (anderen) Ingenieurwissenschaften, aber ebenso relevant.

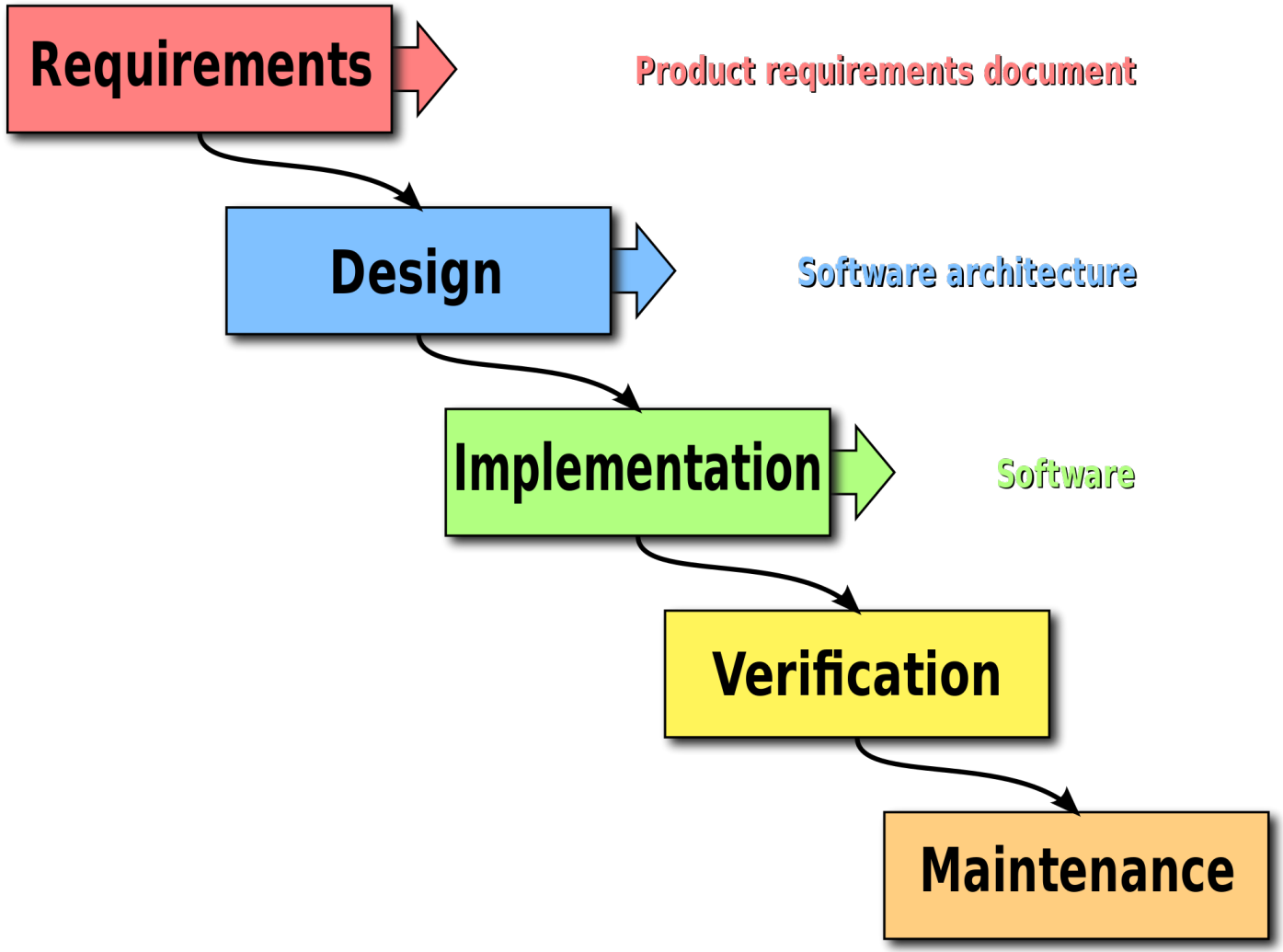
Wozu ist Modellierung gut?

Besonderheiten von Software beeinflussen die Rolle von Modellen:

- Immaterialität; daher gar nicht so leicht, etwa festzustellen, „wie viel“ eines Modells schon umgesetzt wurde
- Einzigartigkeit jedes Softwareprojekts, daher Modelle nicht in der Rolle von Vorlagen zum Zweck mehrfacher Realisierung

Wer modelliert, und wann?

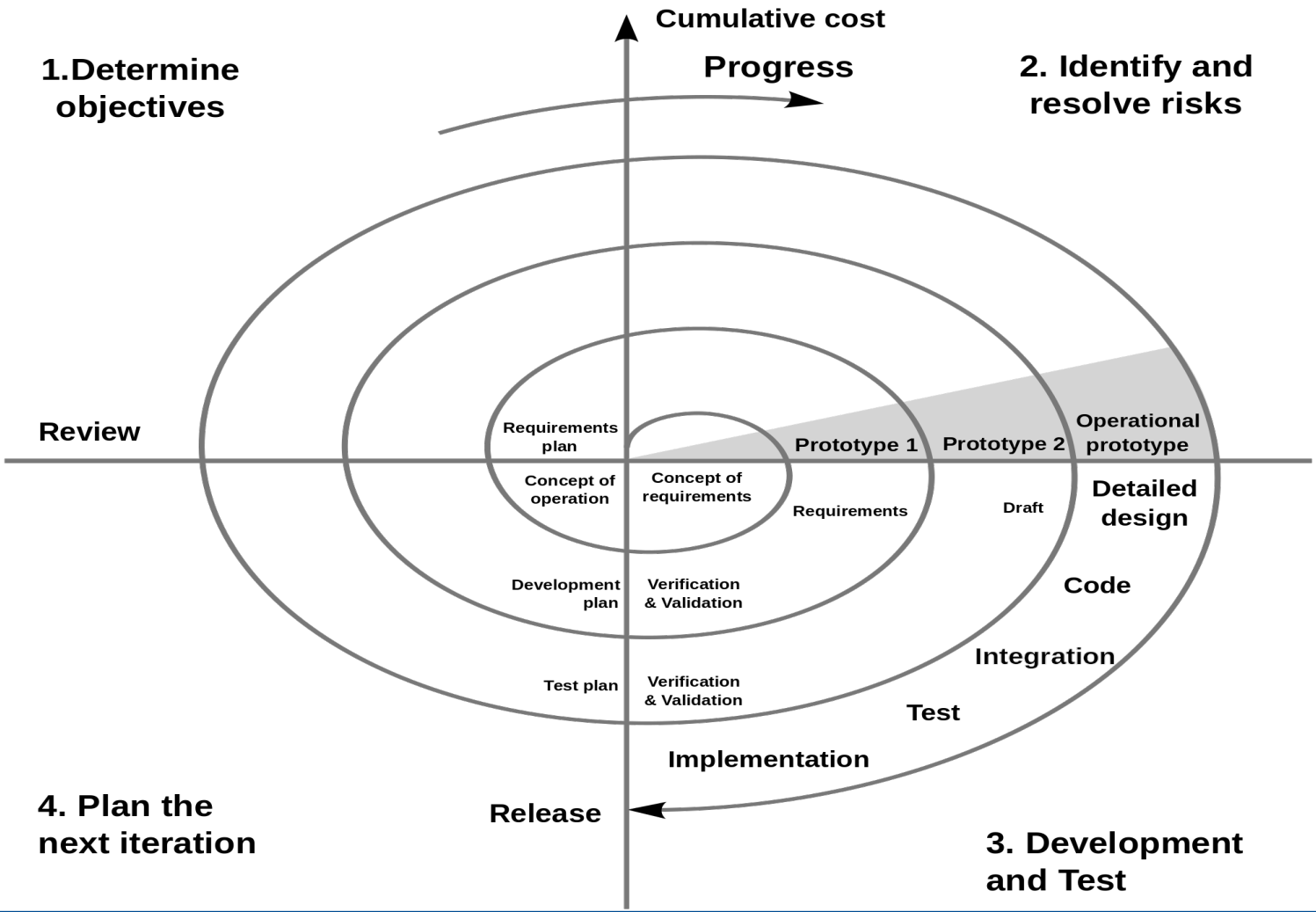
Ein klassischer Softwareentwicklungs-Prozess, „Wasserfall“:



Wer modelliert, und wann?

UNIVERSITÄT DUISBURG ESSEN
Offen im Denken

Alternativer iterativer Ansatz, „Spiral“:



Wer modelliert, und wann?

„Sonderfälle“:

- Agile Software Development
- Open Source Development
- Model Driven Engineering
- . . .

Weitere Aspekte

Weitere wichtige Gesichtspunkte sind:

- Analyse:
  - Ist ein Modell korrekt? Ist es in sich konsistent?
  - Stimmt das Modell mit der späteren Implementierung überein? (Hier werden Verfahren zum Testen und zur Verifikation benötigt.)
- Werkzeuge, Software-Tools:
  ... werden benötigt zum Zeichnen, zum Darstellen (und Wechseln zwischen verschiedenen Darstellungen), zum Archivieren, zur Code-Generierung, zur Analyse, ...

Inhalt der Lehrveranstaltung

UNIVERSITÄT DUISBURG ESSEN
Offen im Denken

Inhalt (nicht exakt in dieser Reihenfolge)

- Mathematische Grundlagen
- Graphen für statische und dynamische Systembeschreibungen
- Petrinetze
- UML (Unified Modeling Language)

Aus Gründen der Übersichtlichkeit und Didaktik werden wir uns hauptsächlich mit kleinen Modellen befassen.

Die Verwendung von Modellen zur Verifikation von Eigenschaften wird nicht im Zentrum stehen.

Inhalt der Lehrveranstaltung

UNIVERSITÄT DUISBURG ESSEN
Offen im Denken

Die in der Lehrveranstaltung vorgestellten Modellierungsmethoden sind nicht die einzigen Modellierungsmethoden in der Informatik.

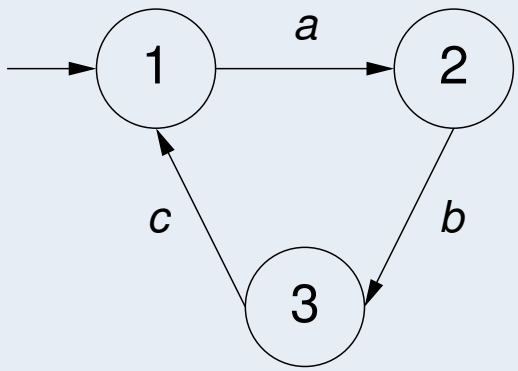
Denn hier: Fokus auf visuelle Modellierung mit Hilfe von Diagrammen

Weniger: algebraische Modellierungsmethoden, die sich stärker an der Mathematik orientieren

Inhalt der Lehrveranstaltung

Graphen

- Graphen sind netzartige Strukturen, bestehend aus Knoten und Kanten.
- Sie können eingesetzt werden für
  - statische Modellierung: z.B. Komponenten und Beziehungen zwischen den Komponenten
  - dynamische Modellierung: z.B. Zustände und Übergänge zwischen den Zuständen



Beispiel: Wolf, Ziege, Kohlkopf

Wir modellieren folgendes (Nicht-Informatik-)System, um eine mögliche Lösung zu finden.

Wolf-Ziege-Kohlkopf-Problem

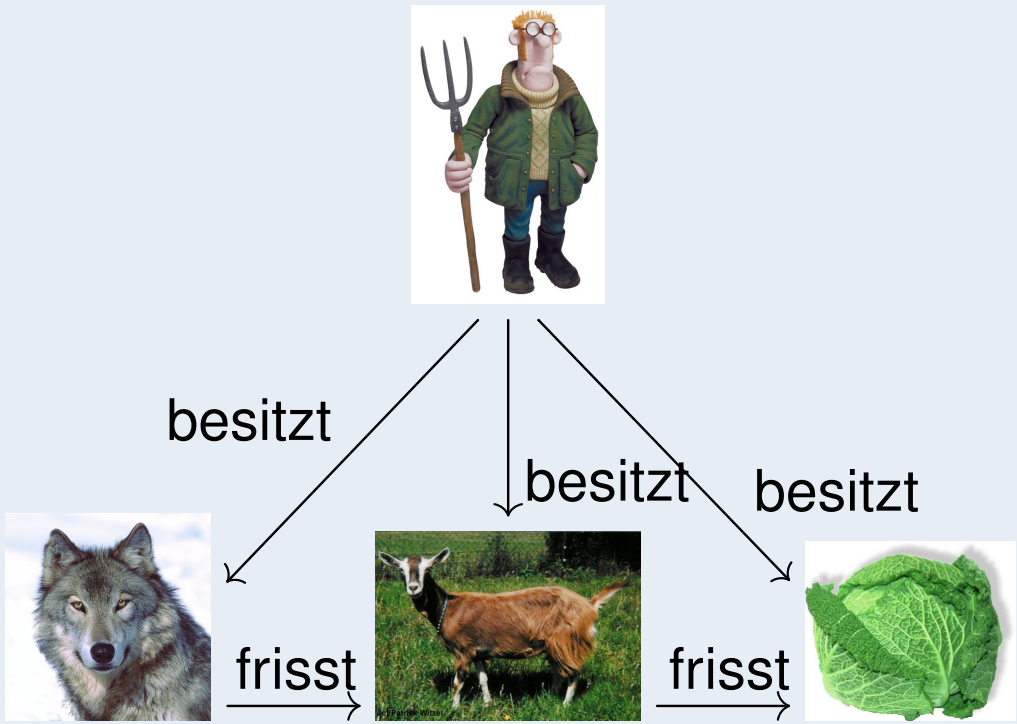
Ein Farmer will einen Fluss überqueren. Er hat einen Wolf, eine Ziege und einen Kohlkopf bei sich. Wenn sie allein gelassen werden, so frisst die Ziege den Kohlkopf und der Wolf die Ziege. Zur Überquerung des Flusses steht ein Boot mit zwei Plätzen zur Verfügung. Nur der Farmer kann rudern und er kann das Boot entweder allein benutzen oder ein Tier oder den Kohlkopf mitnehmen.

Beispiel: Wolf, Ziege, Kohlkopf

Statisches Modell I: Beteiligte Akteure/Objekte

Beispiel: Wolf, Ziege, Kohlkopf

Statisches Modell II: Fress- und Eigentumsbeziehungen zwischen den Akteuren

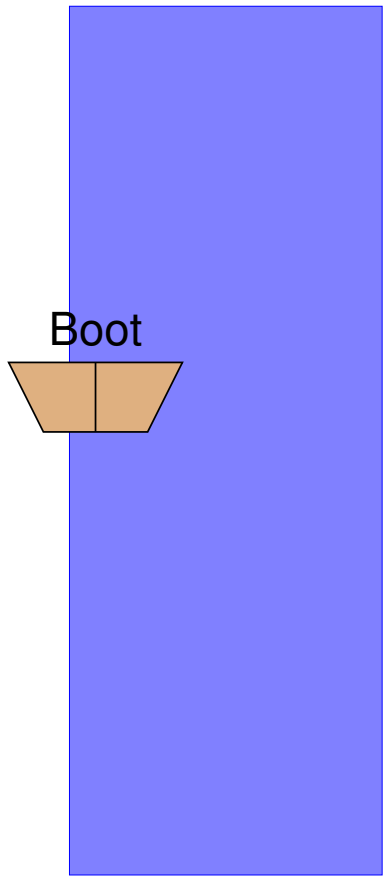


Beispiel: Wolf, Ziege, Kohlkopf

UNIVERSITÄT DUISBURG ESSEN

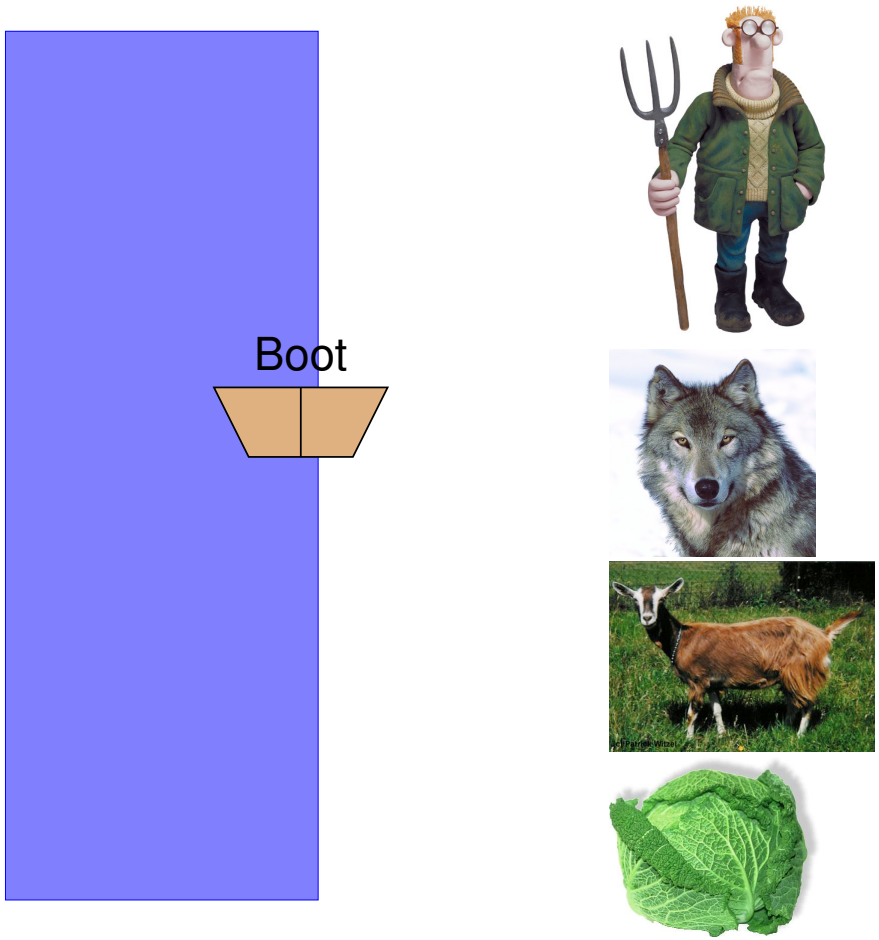
Offen im Denken

Ausgangssituation: vor Überquerung des Flusses



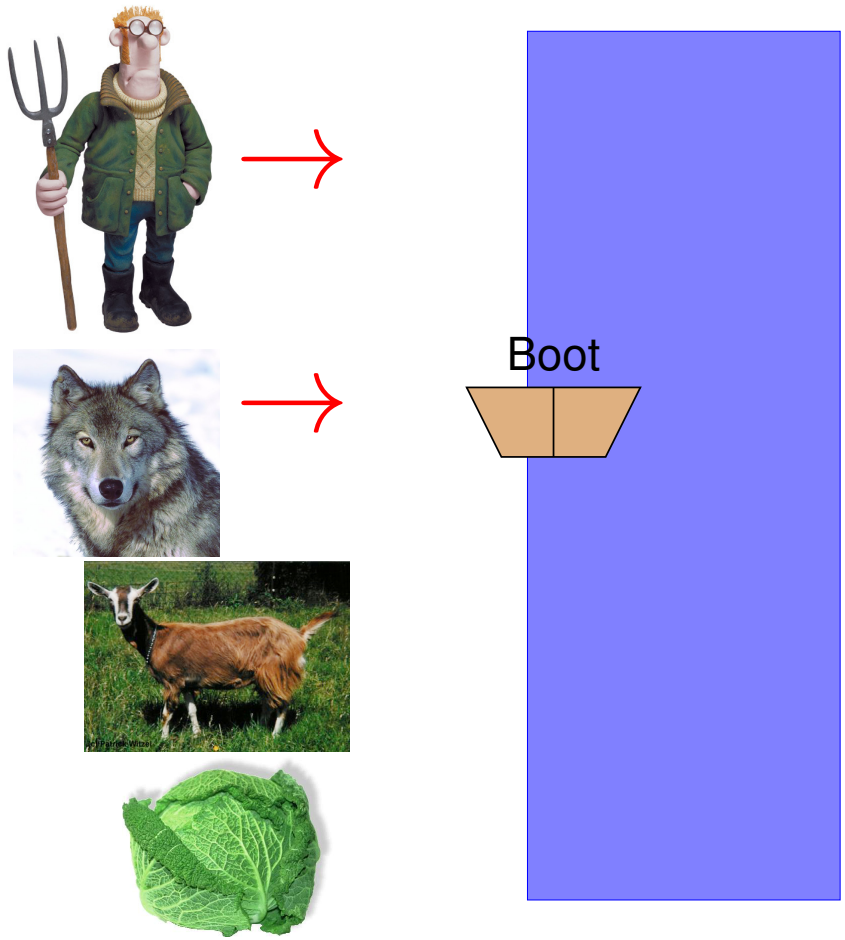
Beispiel: Wolf, Ziege, Kohlkopf

Zielsituation: nach Überquerung des Flusses



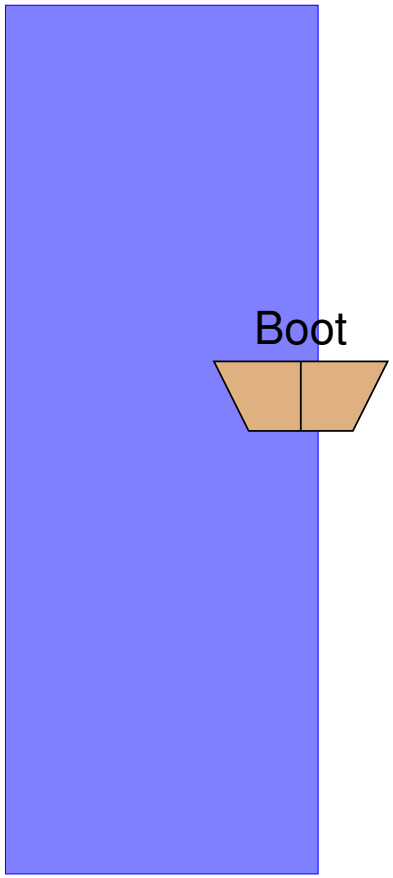
Beispiel: Wolf, Ziege, Kohlkopf

Dynamisches Modell: Beispielablauf, erster Schritt;
Farmer und Wolf setzen gemeinsam über



Beispiel: Wolf, Ziege, Kohlkopf

Dynamisches Modell: Beispielablauf, zweiter Schritt;
Ziege frisst Kohlkopf



Syntax und Semantik

Man unterscheidet bei der Modellierung zwischen:

- Syntax: Symbole und Formen/Diagramme, die für die Darstellung des Modells genutzt werden dürfen

  Im Beispiel: Bild der Ziege, blaue Fläche, etc.

- Semantik: Bedeutung, die sich jeweils dahinter verbirgt

  Im Beispiel: Die blaue Fläche symbolisiert den Fluss.
  Die Pfeile bedeuten: „Fluss wird überquert“. Etc.

Syntax und Semantik

Zu einer Syntax gibt es nicht immer eine dazugehörige Semantik.
(Im Wolf-Ziege-Kohlkopf-Beispiel ist die Semantik noch sehr vage.)

Wünschenswert ist jedoch im Allgemeinen, dass die Bedeutung aller Symbole und Formen möglichst präzise festgelegt wird.

$\rightsquigarrow$ Einigung auf eine gemeinsame Sprache/Notation,
oder auf gemeinsame visuelle Beschreibungsformen;
zur Vermeidung von Missverständnissen

Graphen bilden eine solche Sprache/Notation als formale Grundlage vieler diagrammatischer Modellierungstechniken.

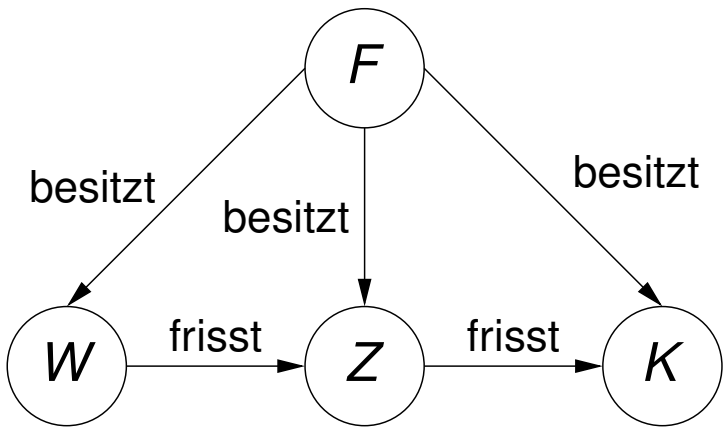
Graphen

Graphen können in vielfältiger Weise zur Modellierung eingesetzt werden. Wir betrachten kurz zwei typische Weisen.

Graphen zur statischen Modellierung

Die Knoten sind Komponenten oder Objekte, die untereinander über Kanten verbunden werden, wenn sie in Beziehung stehen.

Beispiel: Beziehungen zwischen Farmer, Wolf, Ziege, Kohlkopf



Flache Zustandsdiagramme

Graphen zur dynamischen Modellierung

Die Knoten sind Zustände und Kanten beschreiben Zustandsübergänge.

Klassischer Vertreter: flache Zustandsdiagramme
(auch Transitionssysteme genannt)

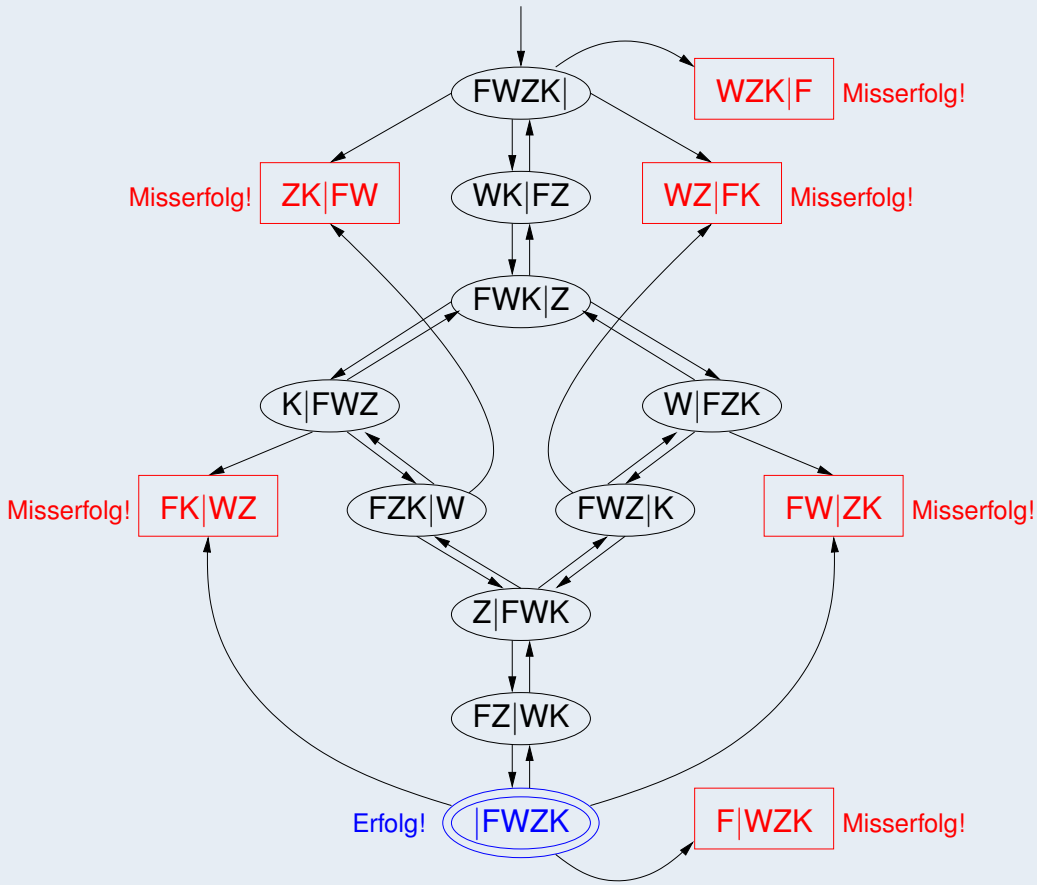
Flaches Zustandsdiagramm

Ein flaches Zustandsdiagramm besteht aus einem gerichteten und eventuell kantenbeschrifteten Graphen, wobei Zustände die Knotenmenge und Übergänge die Kantenmenge bilden, und außerdem der Hervorhebung eines Startzustands (meist durch eine eingehende Pfeilspitze).

Flache Zustandsdiagramme

UNIVERSITÄT DUISBURG ESSEN
Offen im Denken

Beispiel: flaches Zustandsdiagramm für das Wolf-Ziege-Kohlkopf-Problem



Flache Zustandsdiagramme

UNIVERSITÄT DUISBURG ESSEN
Offen im Denken

Bemerkungen:

- Der senkrechte Strich | steht hier für den Fluss. Links und rechts davon befinden sich die Akteure/Objekte (F = Farmer, W = Wolf, Z = Ziege, K = Kohlkopf). Deren Reihenfolge ist egal, und das Boot braucht man nicht anzugeben, denn es ist immer beim Farmer.
- Übergänge sind hier aus Gründen der Übersichtlichkeit nicht beschriftet. Sinnvolle Beschriftungen wären die ausgeführten Aktionen (z.B. „Farmer bringt Ziege über den Fluss“).
- Eckige (rote) Zustände symbolisieren hier Misserfolg (z.B. „Ziege frisst Kohlkopf“; solche Aktionen haben Priorität vor Überfahrten). Kanten, die aus solchen Zuständen herausführen, wurden hier weggelassen.
- Die doppelte (blaue) Ellipse symbolisiert hier Erfolg (erwünschter Zielzustand ist erreicht).

Flache Zustandsdiagramme

Weitere Bemerkungen:

- Es gibt mehrere (sogar unendlich viele und beliebig lange) Wege zum Zielzustand. Die zwei kürzesten enthalten jeweils sieben Übergänge.
- Flache Zustandsdiagramme für selbst relativ einfache Systeme werden oft erstaunlich groß (sogenannte Zustandsexplosion).

Wichtig:
Flache Zustandsdiagramme stellen im Allgemeinen alle Zustände und alle Übergänge eines Systems explizit dar.

Statische Modellierung von Operationen (als Vorbereitung für UML)

Statische Modellierung von Operationen

- Beim Entwurf eines Systems in der Informatik, eines Programms, oder vielleicht auch einer Datenbank, stellt sich oft zunächst die Frage, welche Operationen angeboten und umgesetzt werden sollen, und auf was für Daten diese arbeiten müssen.
- Dabei geht es noch nicht darum, was und wie die Operationen etwas genau „tun“ werden, sondern welche Aufrufe/Verwendungen syntaktisch erlaubt sind und wie Operationen kombiniert werden können.
- Mindestens kann das später der Dokumentation dienen; im Idealfall hilft es bereits bei der Software-Implementierung, etwa durch präzises Erfassen, welche Fälle von Eingaben überhaupt behandelt werden müssen, oder durch die Möglichkeit der Konsistenzprüfung und damit Vermeidung von Fehlern (etwa bei versuchter Verwendung von Operationen in nicht sinnvoller Kombination).

Beispiel: Arithmetische Operationen

Zum Beispiel könnte man (etwa beim Entwurf einer Taschenrechner-App) übliche arithmetische Operationen auf der Menge $\mathbb{N}$ der natürlichen Zahlen, inklusive Null, wie folgt statisch zu modellieren beginnen:

$$+ \ : \ \mathbb{N} \times \mathbb{N} \to \mathbb{N}$$

$$* \ : \ \mathbb{N} \times \mathbb{N} \to \mathbb{N}$$

$$/ \ : \ \mathbb{N} \times \mathbb{N} \to ?$$

Beispiel: Arithmetische Operationen

Um verbotene Division durch Null auszuschließen, Einschränkung des zweiten Arguments auf die Menge $\mathbb{N}_+ \subset \mathbb{N}$ der positiven natürlichen Zahlen:

$$/ \ : \ \mathbb{N} \times \mathbb{N}_+ \to ?$$

Entscheidung darüber, ob man nur ganzzahlige Division umsetzen, oder auch rationale Zahlen unterstützen möchte:

$$/ \ : \ \mathbb{N} \times \mathbb{N}_+ \to \mathbb{N} \quad \text{vs.} \quad / \ : \ \mathbb{N} \times \mathbb{N}_+ \to \mathbb{Q}$$

Dabei gibt es keine „richtige“ oder „falsche“ Entscheidung, sondern es kommt auf den Anwendungszweck und -kontext an. Dies sinnvoll herauszuarbeiten wäre hier gerade Teil des Modellierens.

Beispiel: Arithmetische Operationen

Mit mehreren vorliegenden „Datentypen“ ($\mathbb{N}$, $\mathbb{N}_+$, eventuell $\mathbb{Q}$) kann man einige statische Informationen zu Operationen noch präzisieren, zum Beispiel durch Festhalten von:

$$+ : \mathbb{N}_+ \times \mathbb{N} \to \mathbb{N}_+$$

$$+ : \mathbb{N} \times \mathbb{N}_+ \to \mathbb{N}_+$$

$$* : \mathbb{N}_+ \times \mathbb{N}_+ \to \mathbb{N}_+$$

aber nicht etwa:

$$* : \mathbb{N}_+ \times \mathbb{N} \to \mathbb{N}_+$$

Beispiel: Arithmetische Operationen

Dann können wir rein anhand der statischen Informationen, also ohne wirklich etwas auszurechnen, feststellen, dass etwa $3/((5+0)*4)$ okay ist, $3/((5*0)*4)$ aber nicht sicher.

Umgang mit solchen arithmetischen Ausdrücken oder „Termen“ kennen Sie natürlich aus der Schule, insbesondere neben dem Aufstellen auch das Umformen von Termen.

Interessant aus Modellierungssicht ist nun, dass so ein algebraischer Zugang aber nicht nur für Zahlen und Operationen auf diesen möglich ist, sondern auch allgemein für Operationen in praktisch beliebigen anderen Anwendungsdomänen.

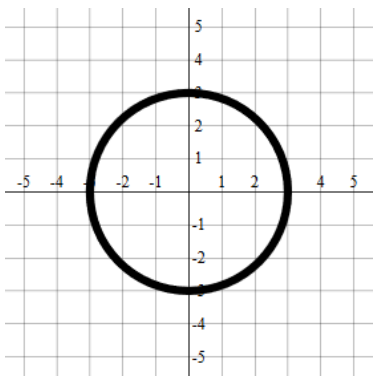
Beispieldomäne: Vektorgrafik

Angenommen, wir möchten für ein Grafikprogramm diverse mögliche Operationen zum Zeichnen und Manipulieren von Bildern modellieren.

Statt mathematischen Zahlenbereichen wie eben, sind hier neben grundlegenden Datentypen, die Sie etwa aus Python kennen oder in GPT bald kennenlernen werden (Int, Float, String), domänenspezifische Typen wie Color, Points, Picture relevant.

Eine Operation könnte dann wie folgt „getypt“ sein:

$$\text{circle} \;:\; \text{Float} \rightarrow \text{Picture}$$

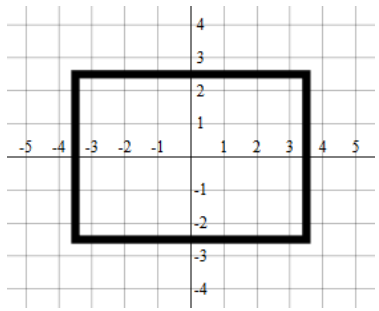


circle(3)

Beispieldomäne: Vektorgrafik

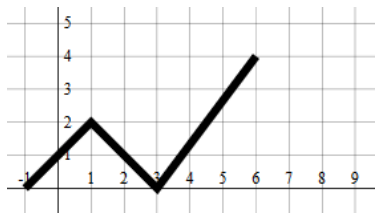
Weitere Operationen für Grundfiguren:

$$\text{rectangle} \;:\; \text{Float} \times \text{Float} \to \text{Picture}$$



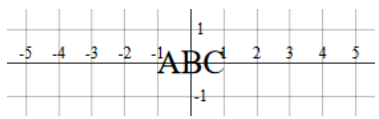
rectangle$(7,5)$

$$\text{path} \;:\; \text{Points} \to \text{Picture}$$



path$([(-1,0),\ldots])$

$$\text{print} \;:\; \text{String} \to \text{Picture}$$

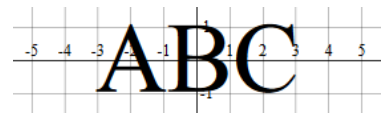


print("ABC")

Beispieldomäne: Vektorgrafik

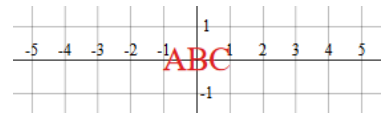
Operationen zur Manipulation bestehender Figuren:

$$\text{scale} : \text{Picture} \times \text{Float} \times \text{Float} \rightarrow \text{Picture}$$



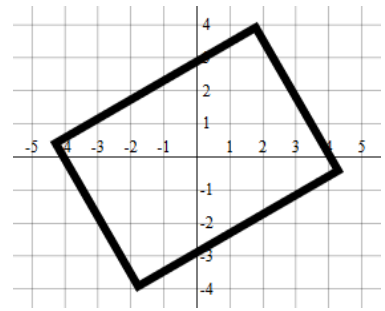
scale(print("ABC"), 3, 3)

$$\text{color} : \text{Picture} \times \text{Color} \rightarrow \text{Picture}$$



color(print("ABC"), red)

$$\text{rotate} : \text{Picture} \times \text{Float} \rightarrow \text{Picture}$$

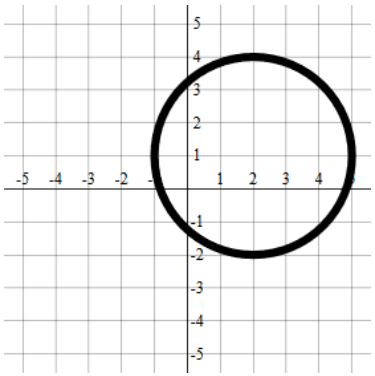


rotate(rectangle(7, 5), 30)

Beispieldomäne: Vektorgrafik

Operationen zur Positionierung:

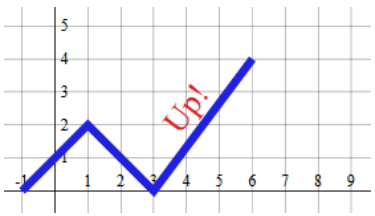
$$\text{move} \;:\; \text{Picture} \times \text{Float} \times \text{Float} \to \text{Picture}$$



$\text{move}(\text{circle}(3), 2, 1)$

... und zur Komposition aus Teilbildern:

$$\& \;:\; \text{Picture} \times \text{Picture} \to \text{Picture}$$



$\text{color}(\text{path}([\ldots]), \text{blue}) \;\&\; \text{move}(\text{rotate}(\text{color}(\text{print}(\text{"Up!"}), \text{red}), 53), 4, 2.5)$

Beispieldomäne: Vektorgrafik

Anhand der statischen Informationen können wir nun wieder bestimmte Aufrufe/Kombinationen von Operationen als nicht sinnvoll erkennen.

Zum Beispiel:

- circle(circle(3)) – circle(3) ist Picture, nicht Float
- print(circle(3)) – ein Picture ist kein druckbarer Text
- print(3) – korrekt wäre: print("3")
- scale(print("ABC"),3) – falsche Anzahl der Argumente
- color(red,print("ABC")) – falsche Reihenfolge der Argumente
- rotate(30,30) – Zahl ist kein Bild
- "ABC" & "DEF" – Komposition nur für Bilder vorgesehen

Andererseits sind natürlich nicht alle Programmierfehler automatisch so erkennbar (z.B. rectangle(7,5) vs. rectangle(5,7)).

Bemerkung:

Falls Sie zum Beispiel C oder Java kennen, eine Angabe wie

$$\text{rectangle} \quad : \quad \text{Float} \times \text{Float} \rightarrow \text{Picture}$$

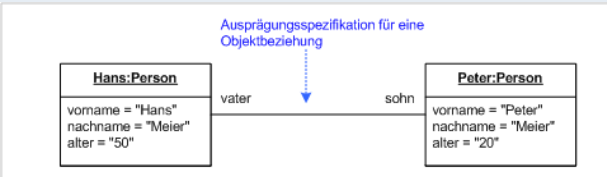
entspräche da der folgenden Deklaration:

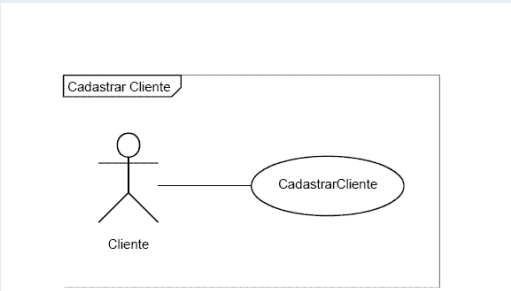
```
Picture  rectangle(Float x, Float y)
```

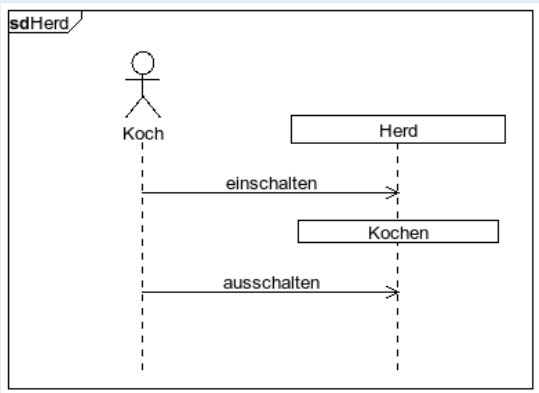
UML: Einführung und Objekt-Orientierung

UML = Unified Modeling Language

- Standard-Modellierungssprache für Software Engineering
- basiert auf objekt-orientierten Konzepten
- sehr umfangreich, enthält viele verschiedene Arten von Modellen
- entwickelt von Grady Booch, James Rumbaugh, Ivar Jacobson (1997)







UML = Unified Modeling Language

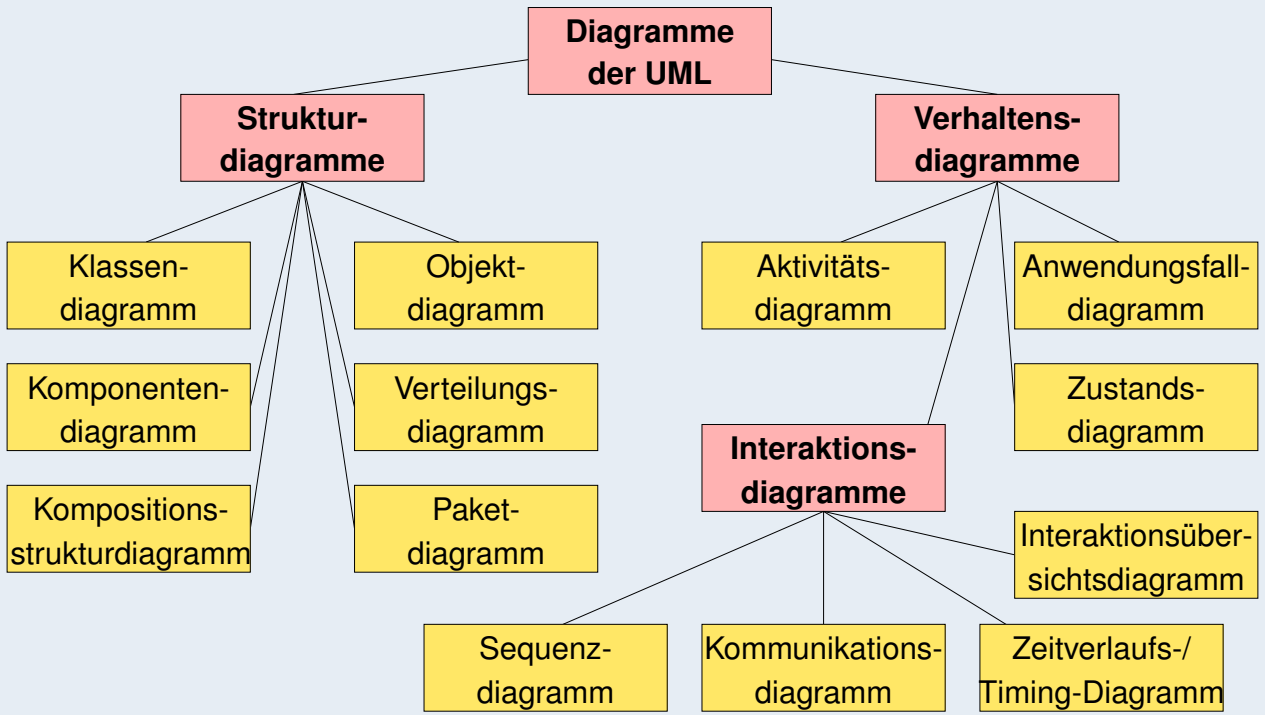
Bei UML handelt es sich um eine ganze Familie von Modellierungsmitteln.

UML beinhaltet Mittel zur

- vor allem visuell-grafischen statt textuell-mathematischen Modellierung von
- sowohl statischen als auch dynamischen Aspekten,
- in qualitativer und quantitativer Hinsicht,
- von vor allem (objekt-orientierten) Software-Systemen,
- unter „white box“- oder „black box“-Sicht,
- mit allgemein bestenfalls semi-formaler (Syntax und) Semantik,
- zur Verwendung bei Softwareentwicklung „im Großen“ mit strukturierten Entwicklungsprozessen.

UML = Unified Modeling Language

Übersicht UML-Diagramme



Wir werden in der Vorlesung einige dieser Begriffe mit Leben füllen.

UML = Unified Modeling Language

Zunächst betrachten wir nur Klassen- und Objektdiagramme. Dabei geht es um statische Modellierung in folgender Hinsicht:

- Dinge, ihre Eigenschaften, und Beziehungen zwischen ihnen;
- wie sich der Zustand eines Systems jeweils zusammensetzt, nicht wie/wohin er sich entwickeln kann;
- welche Operationen gegebenenfalls angeboten werden.

Das klingt weniger spannend als Modellierung des Verhaltens eines Systems, aber:

- präzise statische Modellierung ist wichtige Hilfe für Implementierung größerer Software-Systeme,
- und erlaubt die Anwendung von anerkannten (aus Erfahrung gewonnenen) Design-Prinzipien, etwa zum angemessenen Einsatz der Programmiertechnik Vererbung.

Grundidee der Objekt-Orientierung

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

Etwas genauer . . .

Daten (= Attribute) werden zusammen mit der Funktionalität (= Methoden) in Objekten organisiert bzw. gekapselt.

Jedes Objekt ist in der Lage, Nachrichten (= Methodenaufrufe) zu empfangen, Daten zu verarbeiten und Nachrichten zu senden.

Diese Objekte, bzw. die Objekttypen, können – einmal realisiert – in verschiedenen Kontexten wiederverwendet werden.

Naiv illustriert aus Programmierersicht, an Beispieldomäne Vektorgrafik:

Mit unseren bisher spezifizierten Operationen könnte ein kleines Programm mit schrittweisem Aufbau eines bestimmten Bildes wie folgt aussehen:

$$\begin{aligned} &\mathsf{p1} = \mathsf{rectangle}(7,5);\\ &\mathsf{p2} = \mathsf{color}(\mathsf{p1},\mathsf{red});\\ &\mathsf{p3} = \mathsf{rotate}(\mathsf{p2},30);\\ &\mathsf{p4} = \mathsf{move}(\mathsf{p3},4,2.5); \end{aligned}$$

Stattdessen objekt-orientiert, zum Beispiel (kommt natürlich auf genaue Syntax der Programmiersprache an):

$$\begin{aligned} &\mathsf{p} = \textbf{new}\ \mathsf{Rectangle}(7,5);\\ &\mathsf{p.color}(\mathsf{red});\\ &\mathsf{p.rotate}(30);\\ &\mathsf{p.move}(4,2.5); \end{aligned}$$

Hintergrund ist, dass es statt Operationen mit explizitem Vorkommen von Picture als Ein- und Ausgabe, also etwa:

$$\begin{aligned} \text{color} &: \text{Picture} \times \text{Color} \rightarrow \text{Picture} \\ \text{rotate} &: \text{Picture} \times \text{Float} \rightarrow \text{Picture} \\ \text{move} &: \text{Picture} \times \text{Float} \times \text{Float} \rightarrow \text{Picture} \end{aligned}$$

nun in einer „Klasse“ Picture, die eine „Unterklasse“ Rectangle hat, analoge Operationen gibt, die aber implizit auf jeweils einem Picture-„Objekt“ arbeiten:

```
class  Picture {
    void  color(Color c);
    void  rotate(Float a);
    void  move(Float x, Float y);
}
```

Operationen, die einen anderen Wert als das (veränderte) Objekt selbst zurückliefern, sind natürlich weiterhin auch möglich.

Also wenn wir etwa als Operation vorher noch gehabt hätten:

$$\text{extent} \ : \ \text{Picture} \ \rightarrow \ \text{Float}$$

dann entspräche dem jetzt:

```
class  Picture {
    Float  extent();
}
```

Tatsächliche Syntax in UML ist anders als hier gerade in Anlehnung an Java gezeigt, nämlich: „color(c : Color)“ statt „void color(Color c)“,
„extent() : Float“ statt „Float extent()“.

Behauptete Vorteile der objekt-orientierten Programmierung:

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

Ein Beispiel für die Modellierung von Objekten der realen Welt:

Fahrkartenautomat

- Daten: Fahrziele, Zoneneinteilung, Fahrtkosten
- Funktionalität: Tasten drücken, Preise anzeigen, Münzen einwerfen, Fahrkarten auswerfen

Konzepte

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

  Beispiel: die Klasse der VRR-Fahrkartenautomaten

- Objekt: eine Instanz einer Klasse

  Beispiel: der Fahrkartenautomat am Duisburger Hauptbahnhof, Osteingang

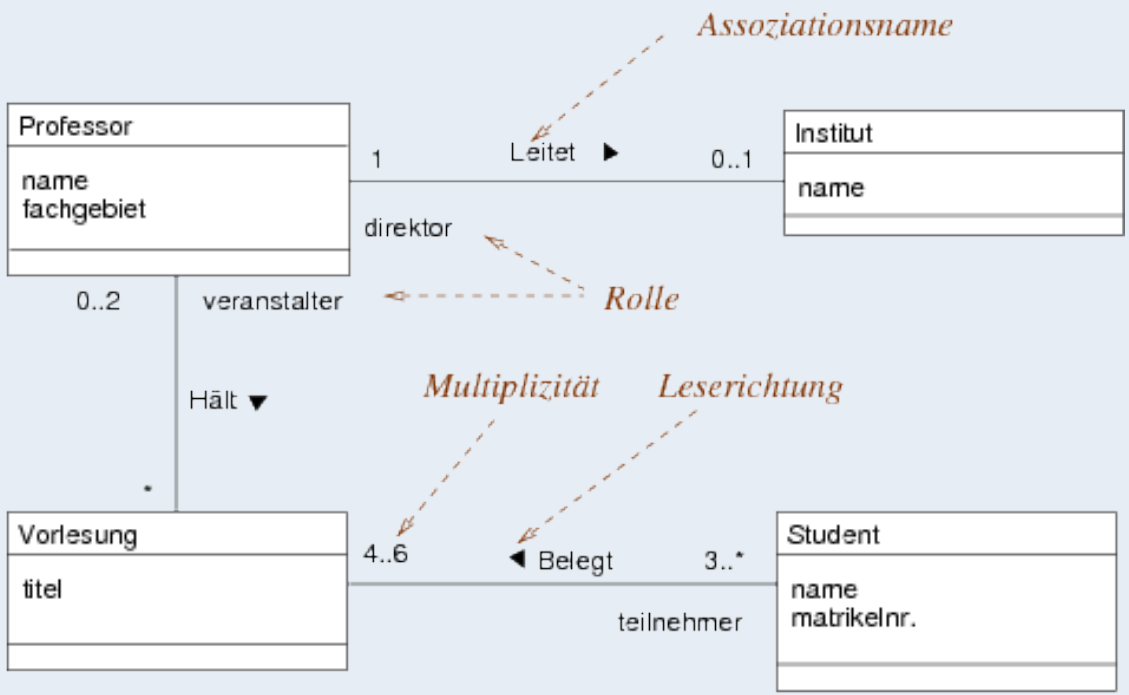
Klassen- und Objektdiagramme

Klassendiagramme zur statischen Modellierung

UNIVERSITÄT DUISBURG ESSEN
Offen im Denken

Klassendiagramme stellen verschiedene Klassen eines Systems/Programms/Moduls mit ihren diversen Beziehungen untereinander dar.

Beispiel:



Klassendiagramme zur statischen Modellierung

UNIVERSITÄT DUISBURG ESSEN
Offen im Denken

Entsprechend stellen sich vor Implementierung eines objekt-orientierten Systems bei der (statischen) Modellierung insbesondere folgende Fragen:

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

Zur Beantwortung kennt die Literatur bestimmte mehr oder weniger systematische Vorgehensweisen.

Beispiel: Modellierung einer Bibliothek

An einem Beispiel, Klassen finden: Use Case “Register New Member”

Actions performed	System responses
1. The customer fills out an application form containing the customer’s name, address and phone number, and gives this to the clerk.	
2. The clerk issues a request to add a new member.	
	3. The system asks for data about the new member.
4. The clerk enters the data into the system.	
	5. Reads in the data, and if the member can be added, generates an identification number and remembers information about the member. Informs the clerk whether the member was added and outputs the member’s name, address, phone and id number.
6. The clerk gives the user their identification number.	

Beispiel: Modellierung einer Bibliothek

An Naivität kaum zu übertreffen: einfach mal alle Nomen heraussuchen

Actions performed	System responses
1. The **customer** fills out an **application form** containing the **customer's name**, **address** and **phone number**, and gives this to the **clerk**.	
2. The **clerk** issues a **request** to add a new **member**.	
	3. The **system** asks for **data** about the new **member**.
4. The **clerk** enters the **data** into the **system**.	
	5. Reads in the **data**, and if the **member** can be added, generates an **identification number** and remembers **information** about the **member**. Informs the **clerk** whether the **member** was added and outputs the **member's name**, **address**, **phone** and **id number**.
6. The **clerk** gives the **user** their **identification number**.	

Beispiel: Modellierung einer Bibliothek

UNIVERSITÄT DUISBURG ESSEN
Offen im Denken

Bereinigung dieser Wortsammlung:

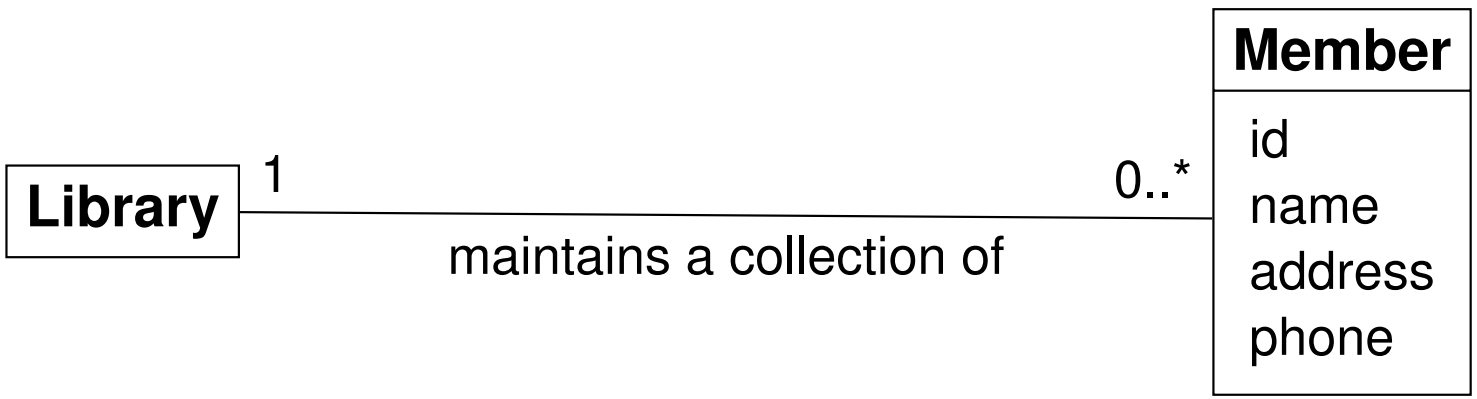
- Als (zusammengesetzte) Einheiten stechen hervor:
  member, **system**
- Bezüglich der anderen vorkommenden Nomen:
  - **customer** – wird ein **member**, ist also ein Synonym
  - **user** – ein weiteres Synonym
  - **application form** – ist ein externes Konstrukt zur Informationsabfrage
  - **request** – nur ein Menüeintrag, wird wie **application form** nicht Teil einer Datenstruktur sein
  - **customer's name**, **address**, **phone number** – Attribute von **member**
  - **clerk** – lediglich ein Akteur, hat keine Repräsentation in Software
  - **identification number** – wird Teil von **member**
  - **data**, **information** – werden als **member** gespeichert

Beispiel: Modellierung einer Bibliothek

UNIVERSITÄT DUISBURG ESSEN
Offen im Denken

Ermittlung von Beziehungen:

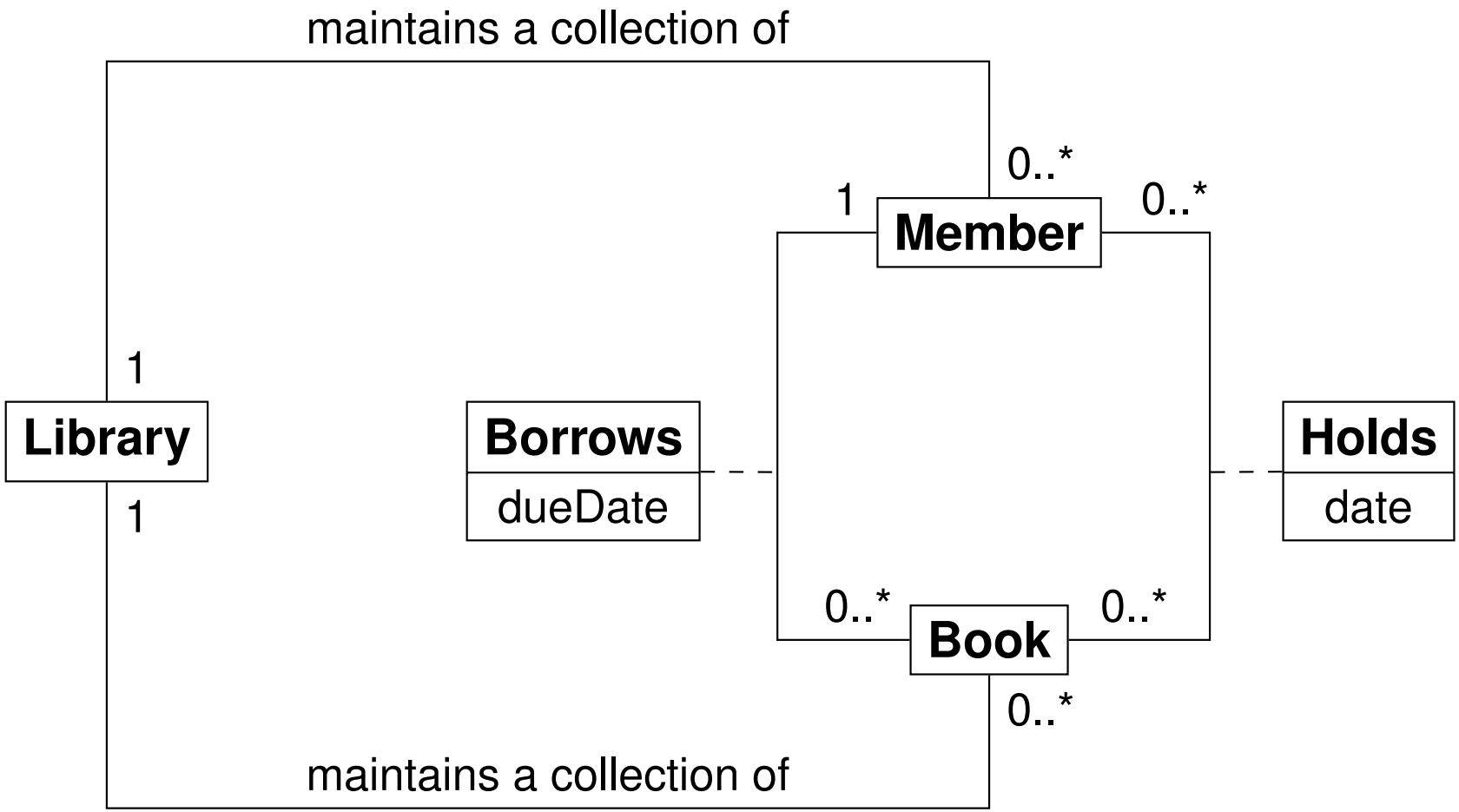
"… [the **system**] remembers **information** about the **member**"



Beispiel: Modellierung einer Bibliothek

UNIVERSITÄT DUISBURG ESSEN
Offen im Denken

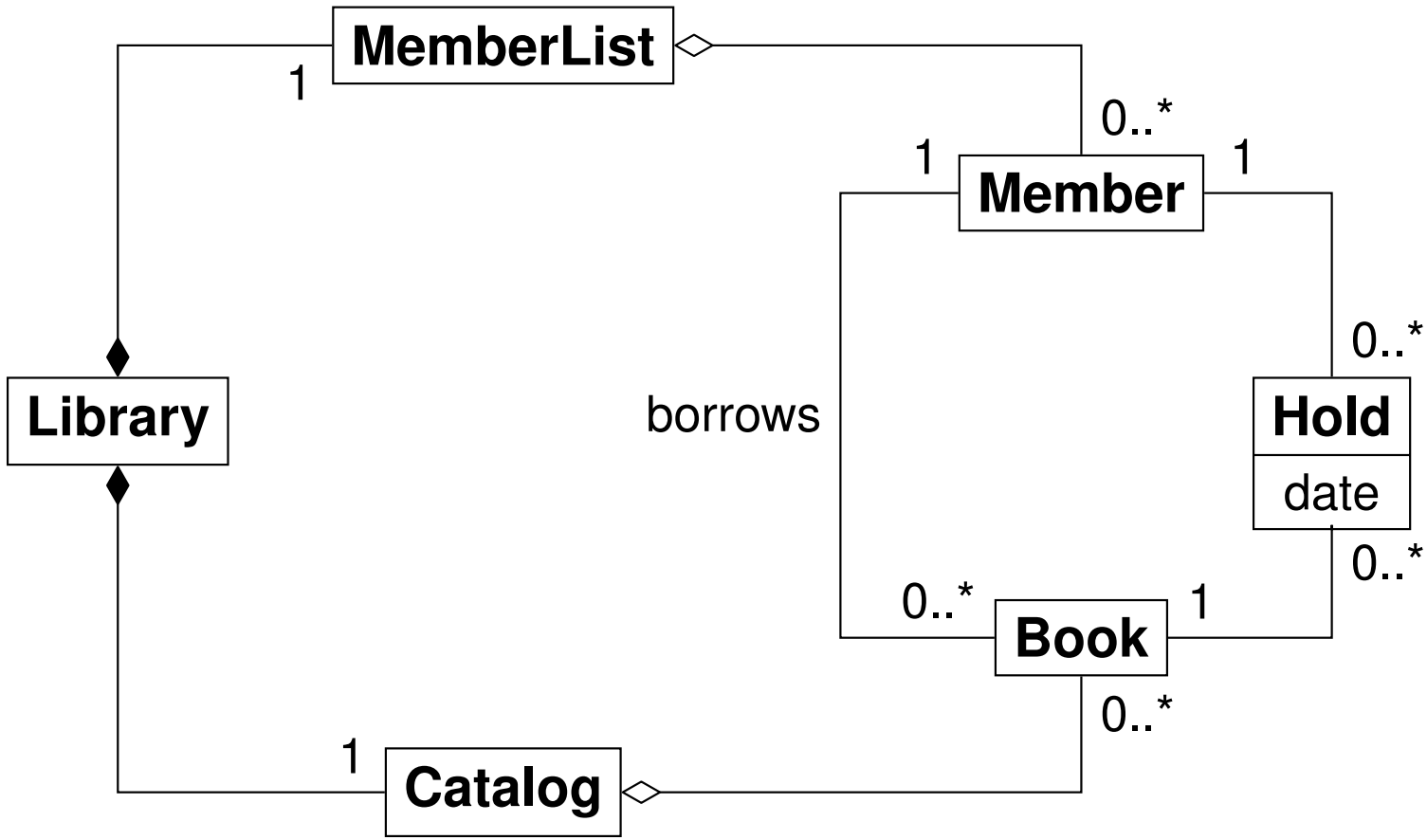
Informiert durch weitere Use Cases:



Beispiel: Modellierung einer Bibliothek

UNIVERSITÄT DUISBURG ESSEN
Offen im Denken

Verfeinert für die Implementierung:

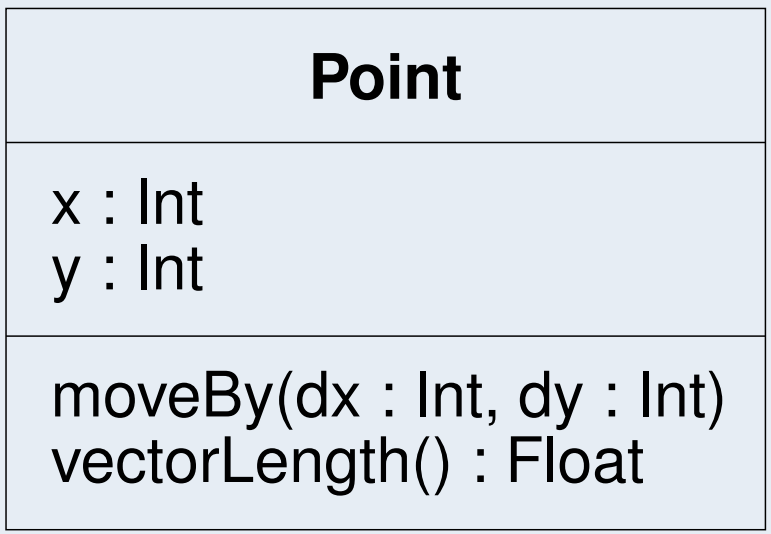


Syntax für statische Modellierung einer Klasse

UNIVERSITÄT DUISBURG ESSEN
Offen im Denken

Beispiel: Klasse von Punkten mit *x*-, *y*-Koordinaten (also zweidimensional) und diversen Operationen

Grafische Darstellung einer Klasse



Klassenname

Attribute (evtl. mit Typ)

Operationen / Methoden

Bemerkung: Obwohl auch Aktivitäten aufgeführt werden (etwa moveBy), handelt es sich dennoch um statische Modellierung. (Warum?)

Attribute & Operationen / Methoden

Weitere Bemerkungen:

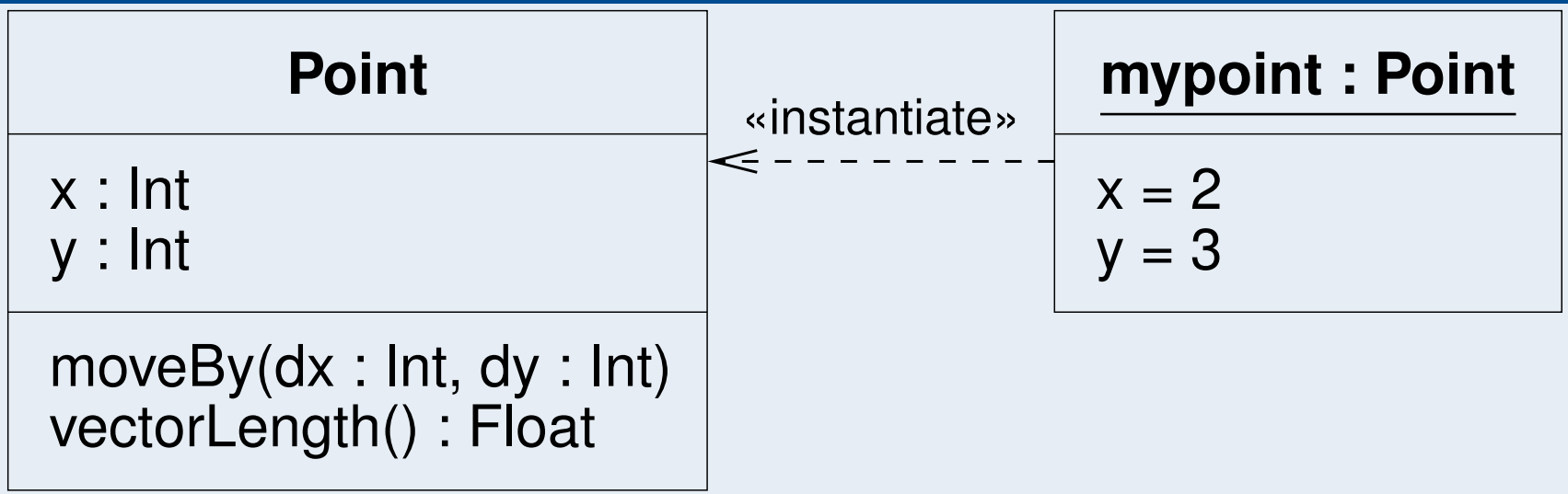
- Bei den Attributen handelt es sich um sogenannte Instanzattribute, das heißt, sie gehören letztlich zu den Instanzen einer Klasse (zu den Objekten, nicht zur Klasse selbst).
- Man kann die Sichtbarkeit eines Attributes bzw. einer Methode spezifizieren, indem man bestimmte Modifikatoren ($+$, $-$, $\#$, $\sim$) vor den Attribut-/Methodennamen schreibt.
- Attribute haben im Allgemeinen Typen, manchmal auch Vorgabewerte (= initiale Werte). Dies wird dann folgendermaßen notiert: x : Int = 0

Instanziierung

Eine Klasse beschreibt den allgemeinen Aufbau bestimmter Objekte.

Eine Instanz einer Klasse stellt ein konkretes Objekt mit den entsprechenden Werten für die Attribute dar.

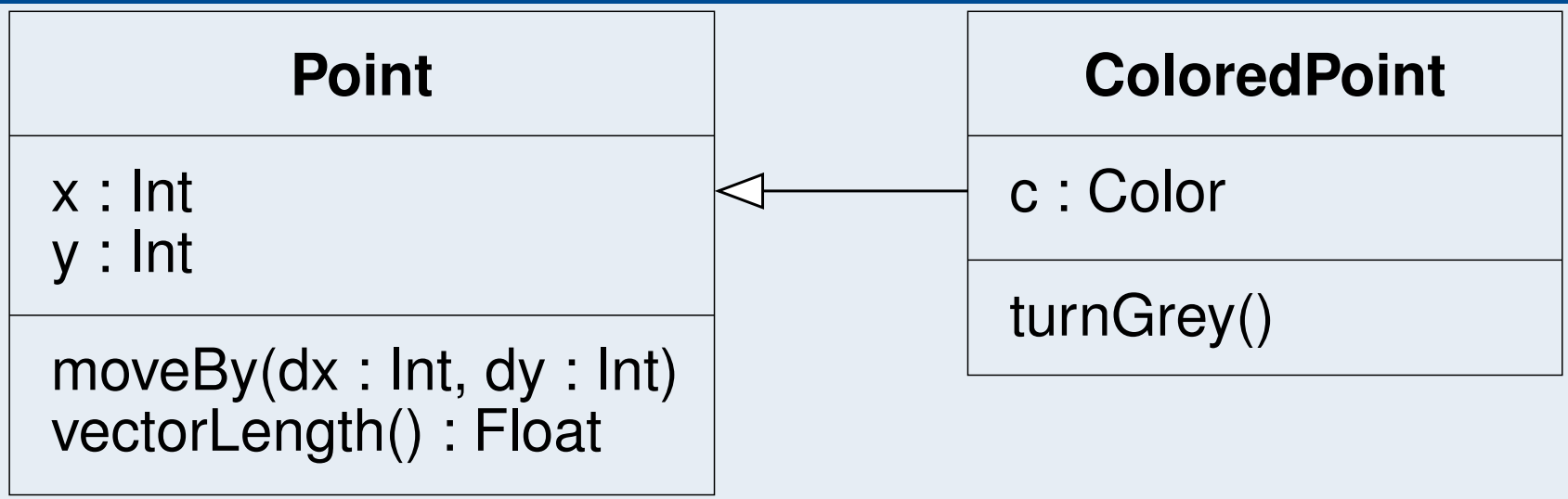
Grafische Darstellung einer Instanz (Objekt) einer Klasse



Generalisierung/Spezialisierung & Polymorphie

Es ist möglich, neue Klassen von bestehenden Klassen erben zu lassen. (Generalisierung/Spezialisierung)

Grafische Darstellung von Vererbung

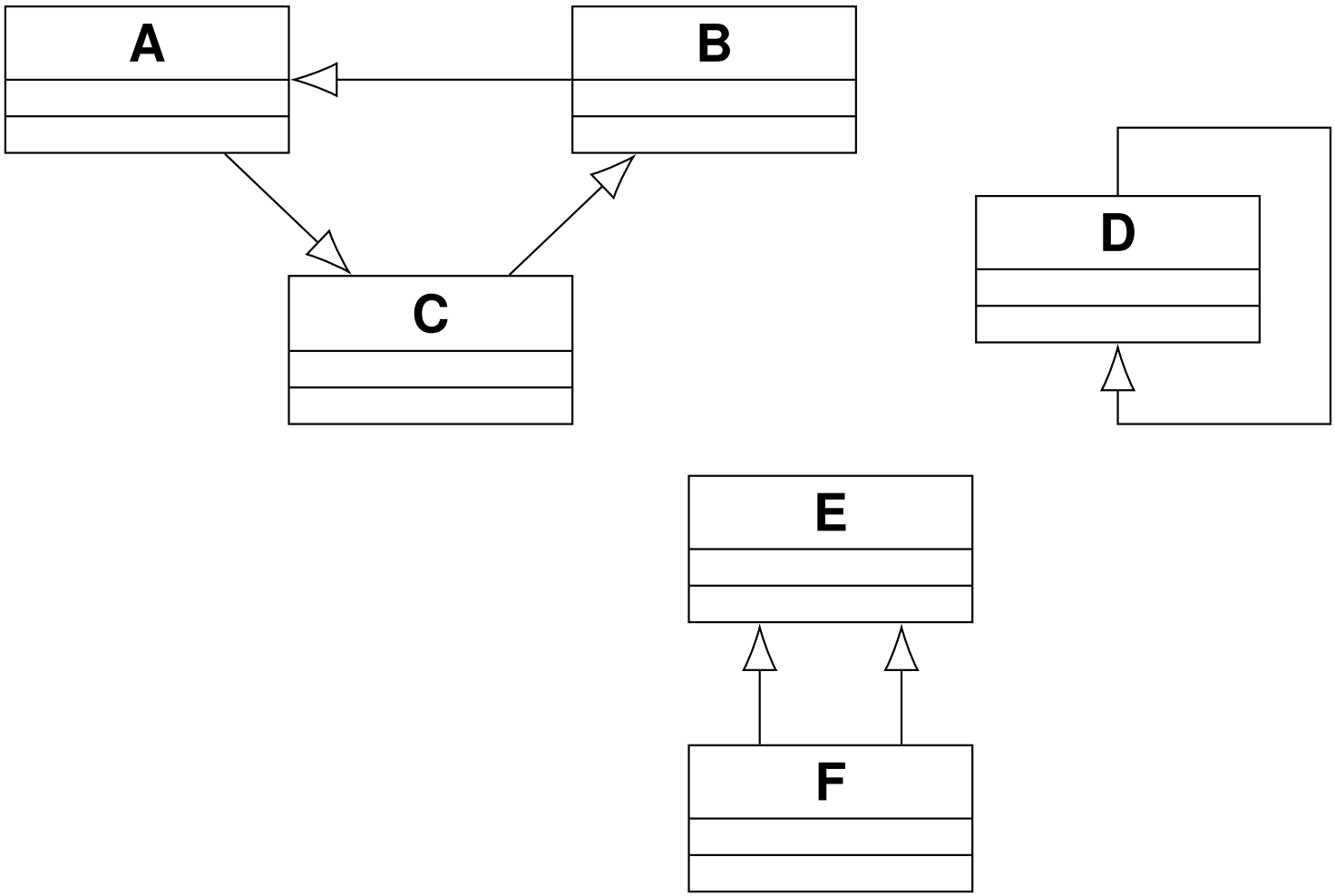


In einer solchen Situation nennt man **Point** eine Superklasse bzw. Oberklasse und **ColoredPoint** eine Subklasse bzw. Unterklasse.

Bemerkungen:

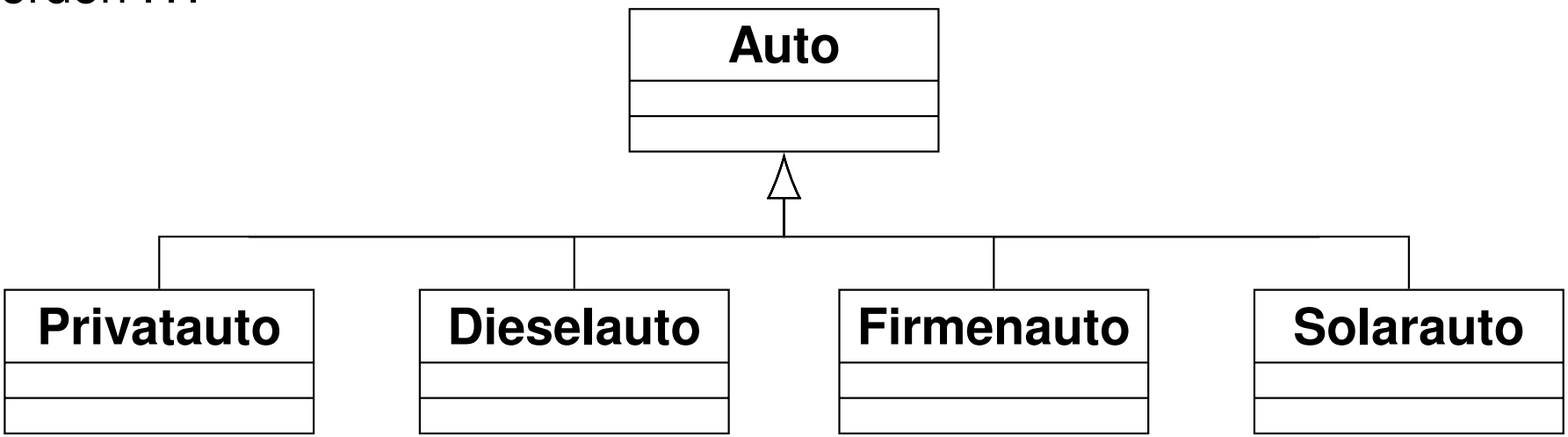
- Die Subklasse erbt die Attribute und Methoden (sowie Assoziationen, Aggregationen und Kompositionen; siehe später) der Superklasse, und kann diesen noch weitere hinzufügen. Außerdem kann man in der Subklasse Methoden der Superklasse überschreiben, also durch neue ersetzen.
- Ein Objekt einer Unterklasse kann auch als ein Objekt jeder seiner Oberklassen angesehen werden. $\Rightarrow$ Polymorphie

Weitere Bemerkung: Konstellationen wie die folgenden sind verboten.



Generalisierung/Spezialisierung & Polymorphie

Mitunter können Klassen in recht unterschiedlicher Weise spezialisiert werden . . .



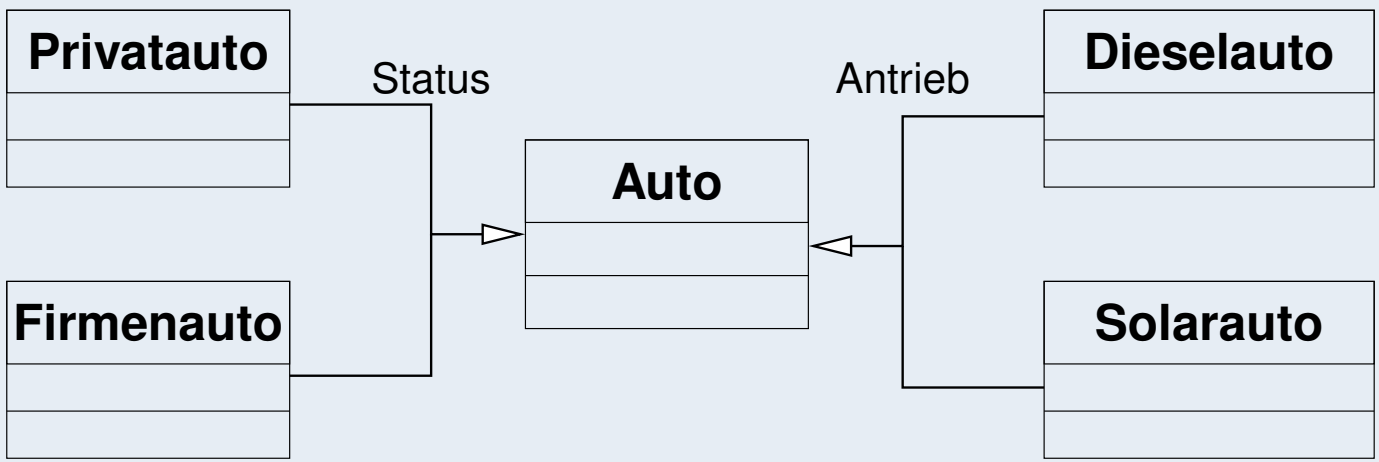
. . . wobei konzeptionell verschiedene Aspekte zur Spezialisierung führen.

Hier etwa sind Privatauto und Dieselauto in ganz verschiedener Hinsicht spezieller als einfach nur ein Auto. Es ergibt auch nicht besonders viel Sinn, Privatauto und Dieselauto in Kontrast zu sehen.
Hingegen sind Privatauto und Firmenauto ein natürliches Kontrastpaar, ebenso Dieselauto und Solarauto.

In solchen Situationen können einzelne Generalisierungsbeziehungen zu Gruppen zusammengefasst werden.

Generalisierungsgruppen

Am Beispiel:



Dabei wird ein sinnvoller Name der jeweiligen Generalisierungsgruppe (hier: Status bzw. Antrieb) im Diagramm annotiert.

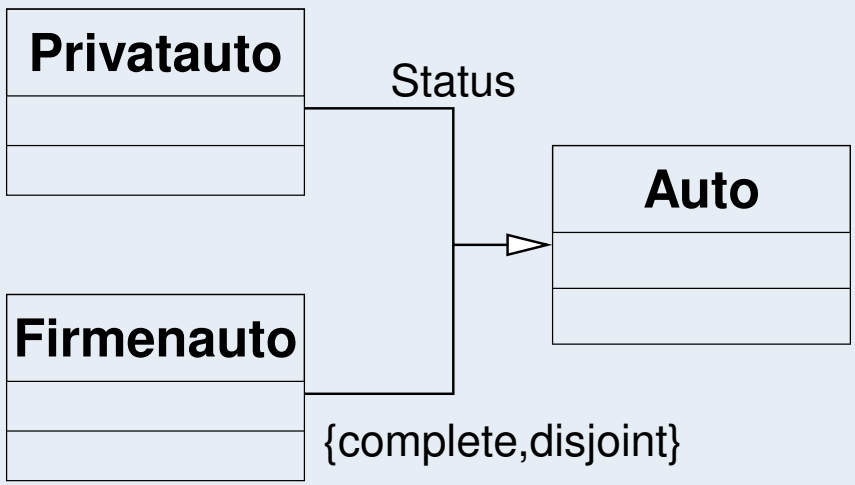
Generalisierung/Spezialisierung & Polymorphie

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

- complete/incomplete:
  - complete: die Generalisierungsgruppe ist vollständig, das heißt, sie überdeckt konzeptionell alle denkbaren Instanzen der Oberklasse.
  - incomplete: die Generalisierungsgruppe ist unvollständig, das heißt, es gibt durch sie nicht erfasste Instanzen.
- overlapping/disjoint:
  - overlapping: es sind Instanzen denkbar, die konzeptionell zu mehr als einer der spezialisierenden Klassen gehören könnten.
  - disjoint: die spezialisierenden Klassen überlappen sich konzeptionell nicht.

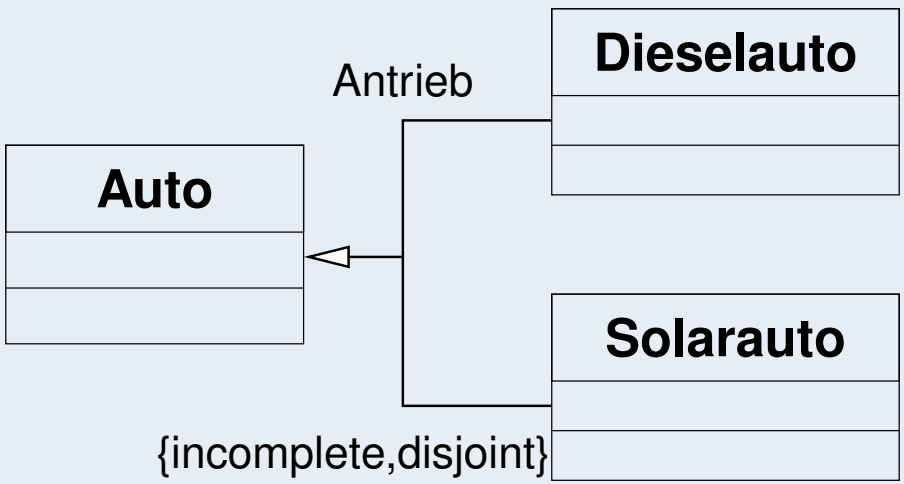
Beispiele für die Eigenschaften complete/incomplete und disjoint/overlapping:

{complete,disjoint}



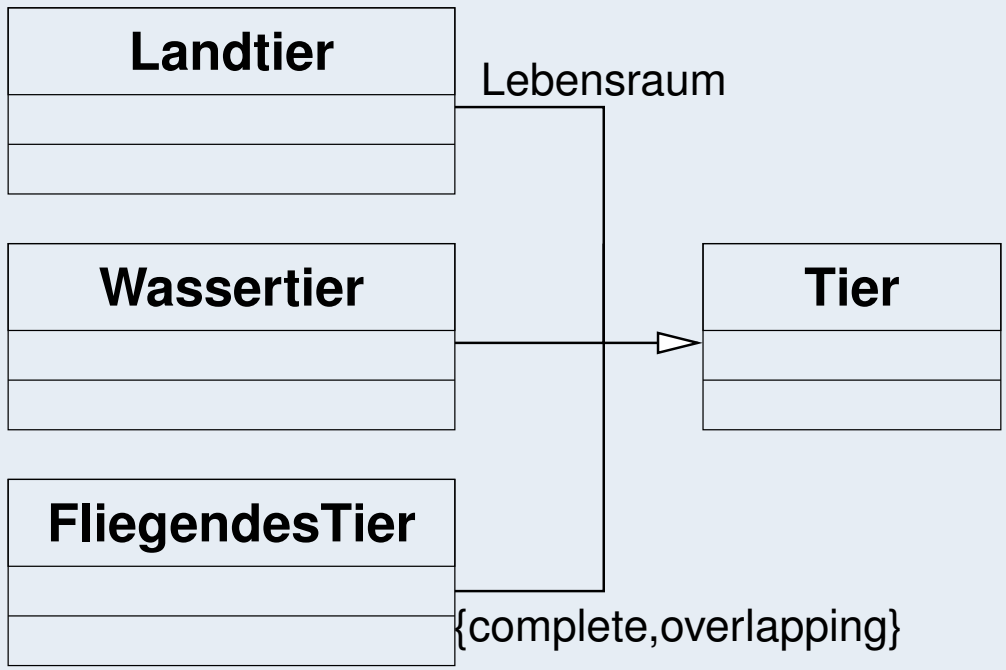
Hier handelt es sich um eine konzeptionelle Partitionierung der Instanzen der Oberklasse.

{incomplete,disjoint}



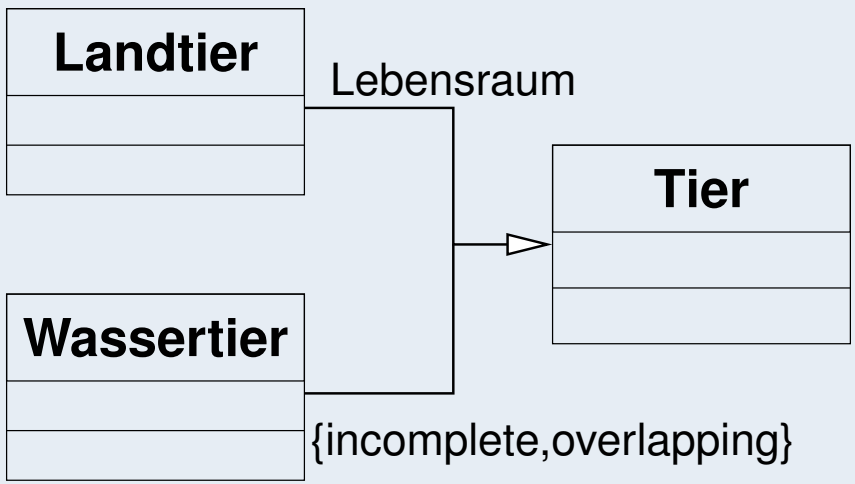
Warum unvollständig? ⇝ Es fehlt z.B. eine Klasse Benzinauto.

{complete,overlapping}



Schildkröten sind sowohl Land- als auch Wassertiere.

{incomplete,overlapping}



Fliegende Tiere fehlen.

Andere Beziehungen zwischen Klassen

Neben Vererbung (also Generalisierung/Spezialisierung) sind andere Beziehungen zwischen Klassen (und letztlich zwischen deren Objekten) in UML darstellbar.

Wir betrachten hier die folgenden Arten von Beziehungen:

- Assoziation
- Aggregation
- Komposition

Dabei drücken diese Beziehungen verschieden starke Verbindungen aus, mit Komposition als stärkster, Assoziation als schwächster Form der Verbindung.

Beziehungen zwischen Klassen: Assoziation

UNIVERSITÄT DUISBURG ESSEN
Offen im Denken

Wir beginnen mit der schwächsten Beziehung: der Assoziation.

Assoziation

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

> es irgendeinen semantischen, nicht-hierarchischen Zusammenhang zwischen den Klassen gibt, der sinnvoll benannt werden kann.

Letztlich drückt eine Assoziation nur aus, dass Objekte der einen Klasse für zumindest einen Teil ihrer Funktionalität eine persistente (abzuspeichernde) Verbindung zu bestimmten Objekten der anderen Klasse brauchen.

Üblicherweise werden Assoziationen durch Referenzen realisiert. Das heißt, eine der Klassen hat ein Attribut vom Typ der anderen Klasse.
Diese Attribute werden jedoch nicht explizit eingetragen.

Beziehungen zwischen Klassen: Assoziation

Beispiel für eine Assoziation

Eine Person kann ein Auto besitzen.

Obige Angabe schließt bewusst nicht aus, dass eine Person auch mehrere, oder gar kein Auto, besitzen könnte. Oder dass ein Auto von mehreren Personen besessen werden könnte.

Mögliche Ausprägungen auf der Ebene von Objekten wären also etwa:

1. $\{(person_1, auto_1), (person_2, auto_2), (person_3, auto_3)\}$
2. $\{(person_1, auto_1), (person_1, auto_2), (person_3, auto_3)\}$
3. $\{(person_1, auto_1), (person_1, auto_2), (person_2, auto_2)\}$

Beziehungen zwischen Klassen: Assoziation

Oft wird eine Leserichtung der Assoziation eingeführt:

Separat kann eine Navigationsrichtung eingeführt werden, die beschreibt, Objekte welcher Klasse ihre Assoziationspartner kennen (und daher deren Methoden aufrufen können):

Beziehungen zwischen Klassen: Assoziation

Hier hätten also **Person**-Objekte Referenzen auf **Auto**-Objekte.

Mengentheoretisch ausgedrückt, zum Beispiel:

2. $person_1 \mapsto \{auto_1, auto_2\}$, $person_2 \mapsto \emptyset$, $person_3 \mapsto \{auto_3\}$

Beziehungen zwischen Klassen: Assoziation

Die Navigationsrichtung kann im Prinzip von der Leserichtung abweichen:

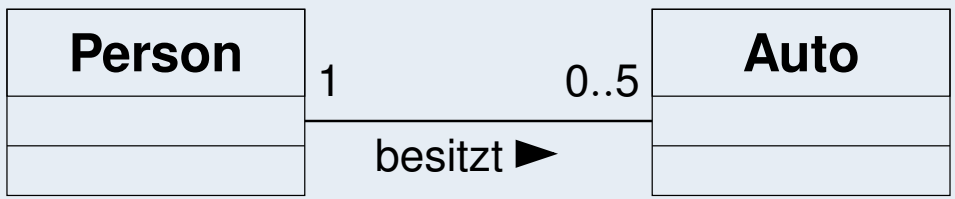
Allerdings ist das ungewöhnlich und deutet (sofern nicht sogar in beide Richtungen Navigierbarkeit vorliegt) auf einen Modellierungsfehler hin.

Hier kennen sich Vertreter beider Klassen gegenseitig:

Beziehungen zwischen Klassen: Assoziation

UNIVERSITÄT DUISBURG ESSEN
Offen im Denken

An beiden Enden einer Assoziation können Multiplizitäten in Form von Intervallen *m..n* (oder einfach nur *m* für *m..m*) angegeben werden.



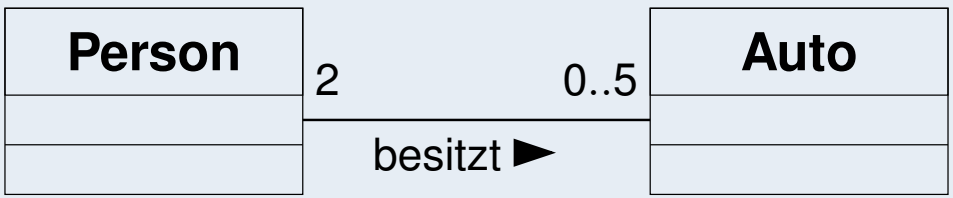
Hier besitzt jede Person bis zu fünf Autos. Und jedes Auto ist im Besitz genau einer Person.

Falls die Multiplizität (am anderen Ende einer navigierbaren Assoziation) Werte größer als Eins umfasst, muss dies in der Implementierung durch eine Kollektion (Liste, Menge, Array) von Referenzen realisiert werden.

Beziehungen zwischen Klassen: Assoziation

UNIVERSITÄT DUISBURG ESSEN
Offen im Denken

Was bedeuten zum Beispiel folgende Multiplizitäten?



Jedenfalls nicht, dass immer zwei Personen zusammen genau die gleichen Autos besitzen müssen.

Möglich wäre etwa folgende Situation:

$$\{(person_1, auto_1), (person_2, auto_1), \\ (person_1, auto_2), (person_3, auto_2)\}$$

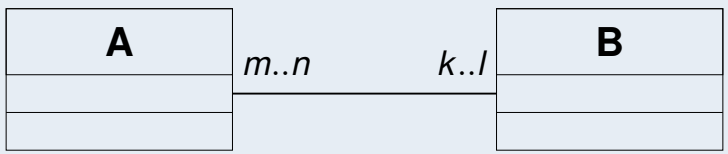
und noch Existenz weiterer Personen ohne Existenz weiterer Autos.

Beziehungen zwischen Klassen: Assoziation

UNIVERSITÄT DUISBURG ESSEN
Offen im Denken

Multiplizitäten allgemein:

In folgendem Diagramm können *i* Instanzen von **A**, mit $m \leq i \leq n$, mit einer Instanz von **B** assoziiert sein. Umgekehrt können *j* Instanzen von **B**, mit $k \leq j \leq l$, mit einer Instanz von **A** assoziiert sein.



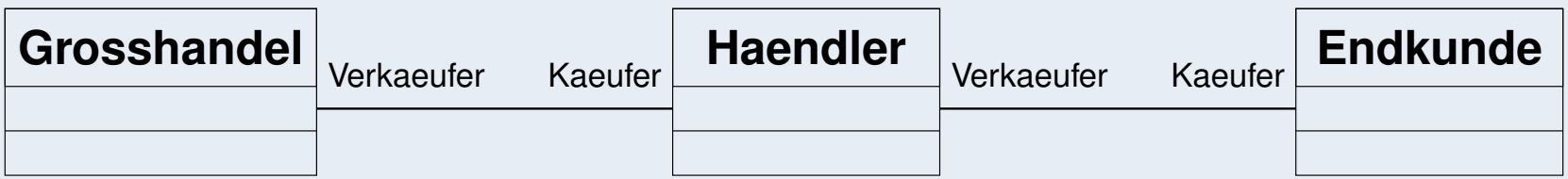
Falls es keine obere Schranke geben soll, wird ein Stern (= unendlich) verwendet. Beispielsweise steht $2..*$ für „mindestens zwei“.

In UML ist keine Standardmultiplizität vorgegeben.
Für die folgenden Diagramme (und in Übung/Klausur) wird, wenn keine Angabe vorhanden ist, die Multiplizität $0..*$ als Standard angenommen.

Beziehungen zwischen Klassen: Assoziation

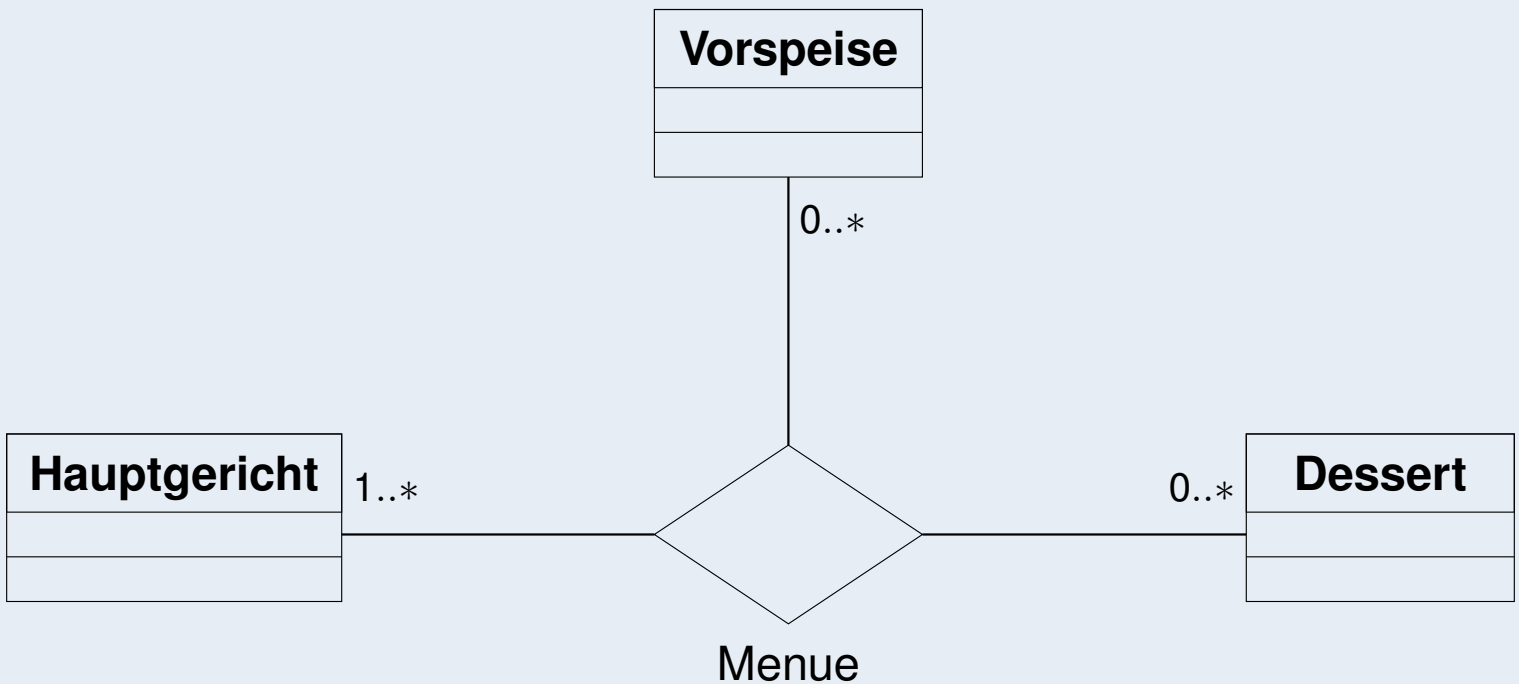
UNIVERSITÄT DUISBURG ESSEN
Offen im Denken

Klassen können in verschiedenen Assoziationen verschiedene Rollen spielen. Rollen werden auch an den Assoziationen notiert (und machen es manchmal überflüssig, die Assoziation selbst zu benennen oder eine Leserichtung anzugeben).



n-äre Assoziation

Neben binären (zweistelligen) Assoziationen gibt es auch *n*-äre Assoziationen, die eine Beziehung zwischen $n > 2$ Klassen beschreiben.



Beziehungen zwischen Klassen: Aggregation

Die nächste Beziehung – Aggregation – ist etwas stärker.

Aggregation

Es gibt eine Aggregation zwischen den Klassen **A** und **B**, wenn Instanzen der Klasse **A** Instanzen der Klasse **B** als Teile enthalten. (Ein „Ganzes“ enthält mehrere „Teile“.)

Dabei ist durchaus denkbar, dass ein Teil zu mehreren (oder keinem) Ganzen gehört.

Eine explizite Benennung ist oft überflüssig, da aus der Angabe als Aggregation bereits die Natur der Beziehung folgt.

Beziehungen zwischen Klassen: Aggregation

Beispiel für eine Aggregation (mit Benennung)

Ein Parkplatz „enthält“ mehrere Autos.

Jedoch existiert ein Auto nicht nur solange es auf einem Parkplatz steht.

Beziehungen zwischen Klassen: Komposition

Die stärkste der betrachteten Beziehungen ist die Komposition.

Komposition

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

Instanzen der Klasse **A** Instanzen der Klasse **B** als Teile enthalten und die Lebenszeit der Teile vom „Ganzen“ kontrolliert wird.

Das heißt, die Teile können (oder müssen sogar) gelöscht werden, sobald die zugehörige Instanz der Klasse **A** gelöscht wird.

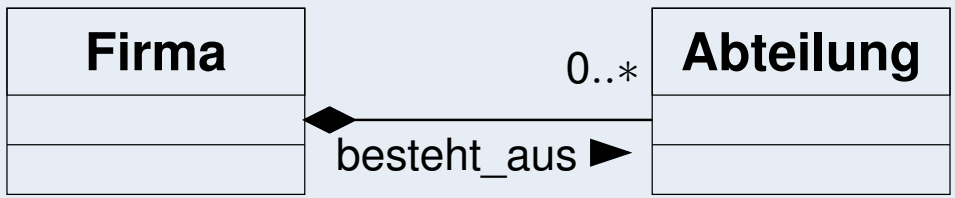
Außerdem (oder eigentlich wegen Obigem) darf ein Teil nicht gleichzeitig zu mehr als einem Ganzen gehören.

Auch hier ist eine explizite Benennung oft überflüssig.

Beziehungen zwischen Klassen: Komposition

Beispiel für eine Komposition (mit Benennung)

Eine Firma besteht aus einer beliebigen Anzahl Abteilungen.



Die Abteilungen existieren nicht mehr, sobald die Firma nicht mehr existiert.

Bemerkung: Bei einer Komposition darf die Multiplizität, die an der schwarzen Raute stünde, nur 0..1 oder 1 sein. Am üblichsten ist 1, dementsprechend wird in dem Fall an diesem Ende gar keine Multiplizität angegeben: jedes Teil gehört zu genau einem Ganzen.

Beziehungen zwischen Klassen

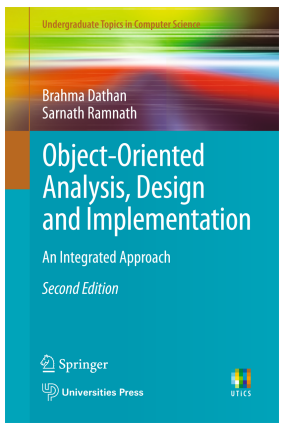
„Merksätze“ (aus: Dathan & Ramnath, Object-Oriented Analysis, Design and Implementation – An Integrated Approach, siehe Literaturfolien zu Beginn des Semesters):

- An association normally represents something that will be stored as part of the data and reflects all links between objects of two classes that may ever exist. It describes a relationship that will exist between instances at run time and has an example.
- . . . , associations should be shown if a class possesses, controls, is connected to, is related to, is a part of, has as parts, is a member of, or has as members some other class in the system.
- . . . association should not be used to denote relationships that: (i) can be drawn as a hierarchy, (ii) stems from a dependency alone, (iii) or relationships whose links will not survive beyond the execution of any particular operation.

Beziehungen zwischen Klassen

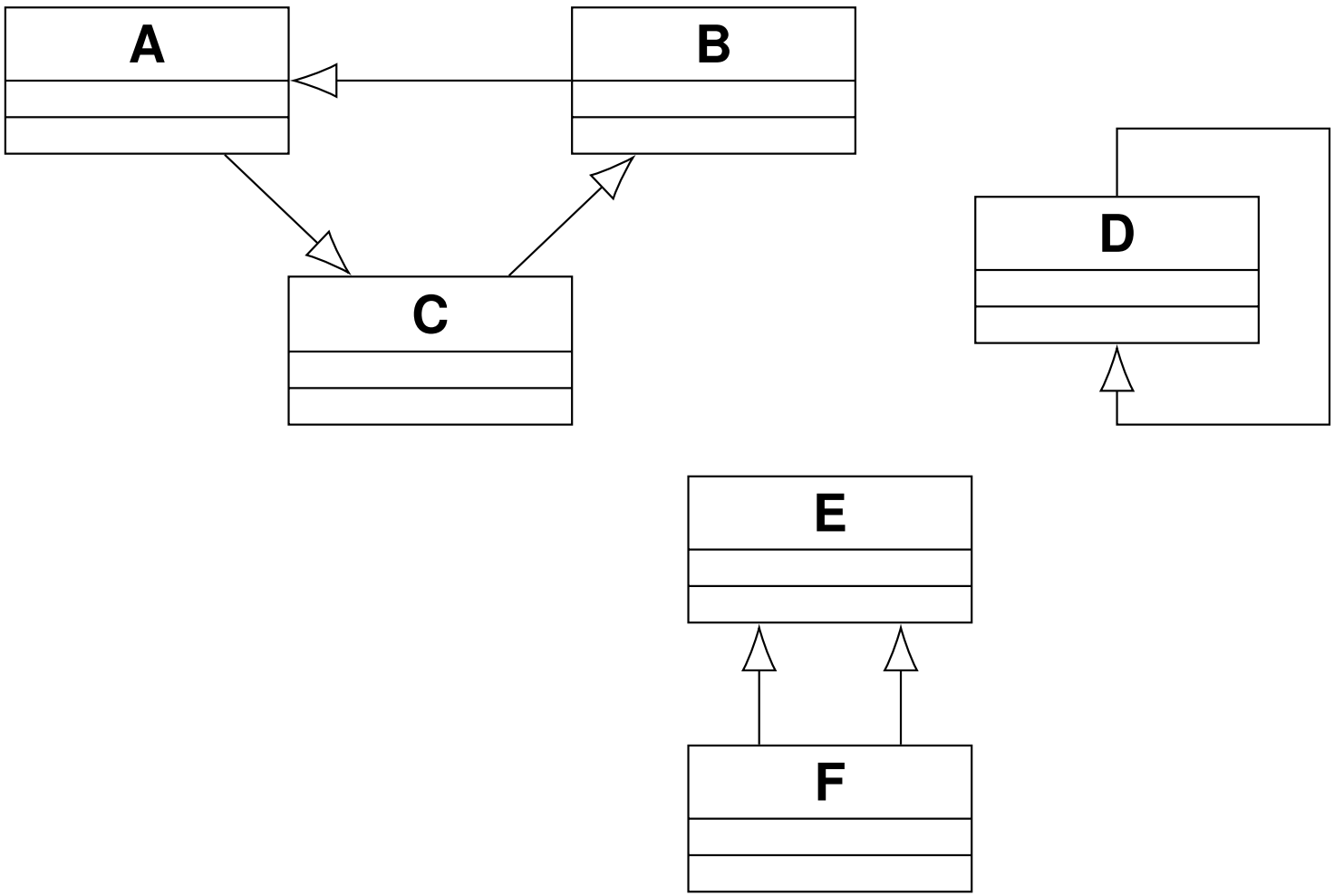
„Merksätze“ (aus: Dathan & Ramnath, Object-Oriented Analysis, Design and Implementation – An Integrated Approach, siehe Literaturfolien zu Beginn des Semesters):

- Aggregation is a kind of association where the object of class A is ‘made up of’ objects of class B. This suggests some kind of whole-part relationship between A and B.
- Composition implies that each instance of the part belongs to only one instance of the whole, and that the part cannot exist except as part of the whole.



Beziehungen zwischen Klassen

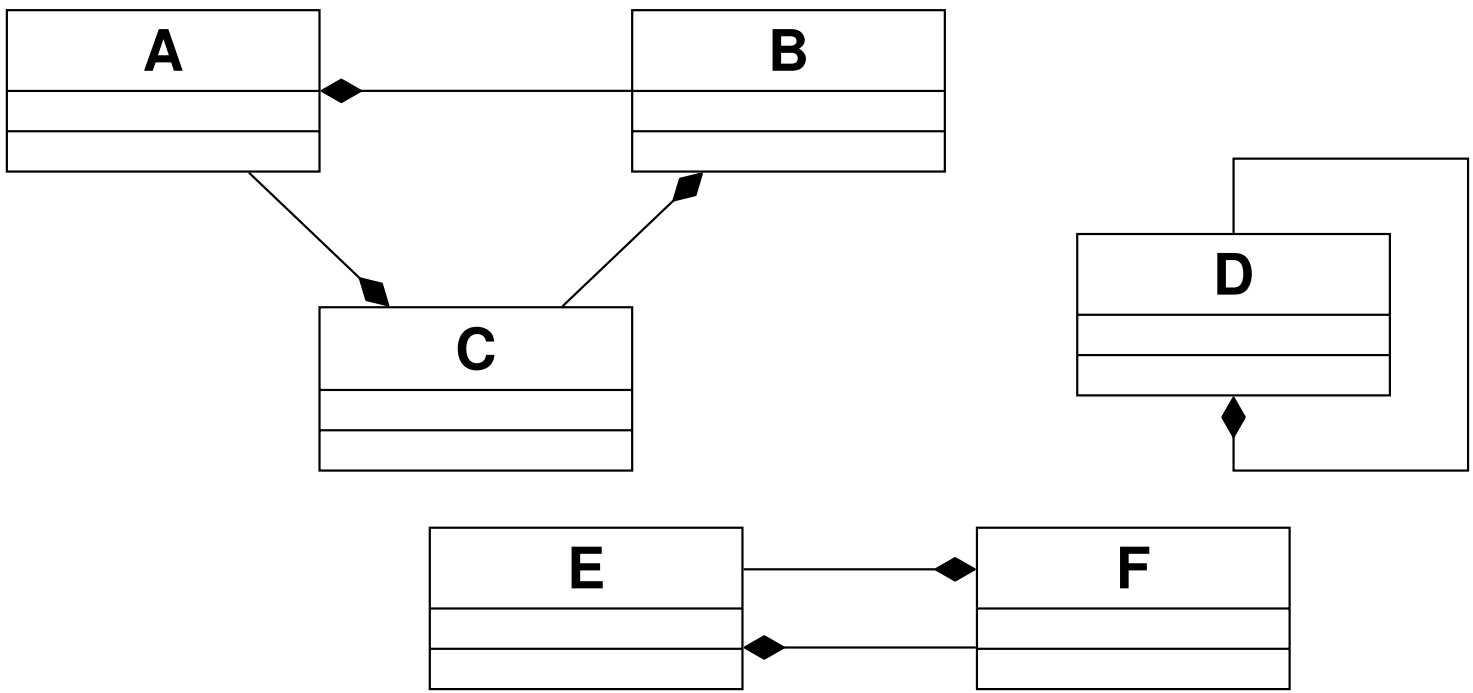
Zur Erinnerung, solche Konstellationen sind bei Vererbung verboten:



Beziehungen zwischen Klassen

Für Assoziation und Aggregation gibt es keine solchen Einschränkungen bezüglich Zyklen, Selbstreferenz oder Vorliegen mehrerer Beziehungen zwischen den gleichen Klassen.

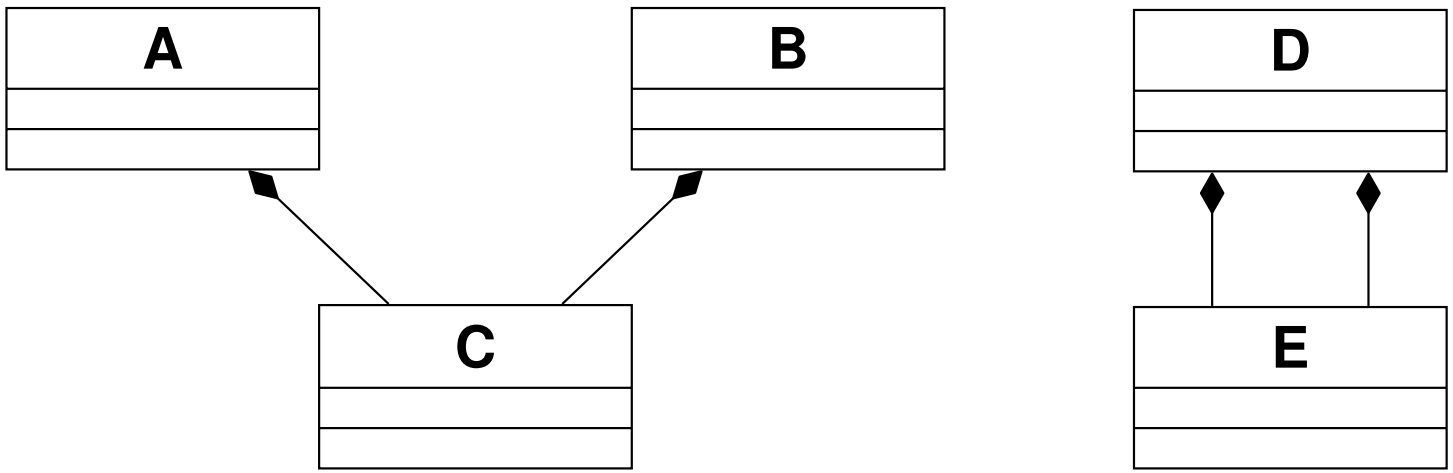
Für Komposition hingegen sind Zyklen jeglicher Art wieder verboten:



Beziehungen zwischen Klassen

. . . und selbst für nichtzyklische Situationen ergeben sich gewisse Einschränkungen, nämlich aus der Forderung, dass ein „Teil“ nicht gleichzeitig zu mehr als einem „Ganzen“ gehören darf.

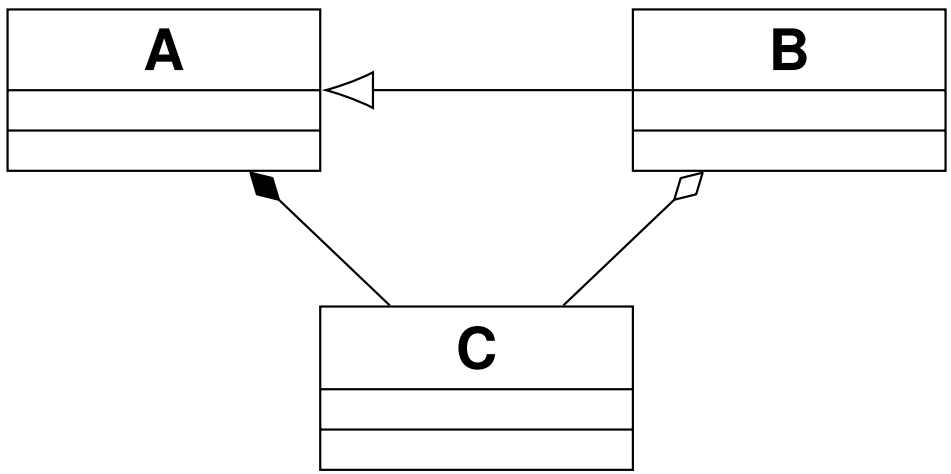
Zum Beispiel sind in folgenden beiden Situationen nicht alle denkbaren Multiplizitäten (0..1 oder 1) an den schwarzen Rauten sinnvoll:



Beziehungen zwischen Klassen

Zu beachten ist auch das Zusammenspiel von Assoziationen etc. mit Vererbung, denn eine Subklasse erbt neben den Attributen und Methoden immer auch die Assoziationen, Aggregationen und Kompositionen der Superklasse.

Zum Beispiel gibt es hier:

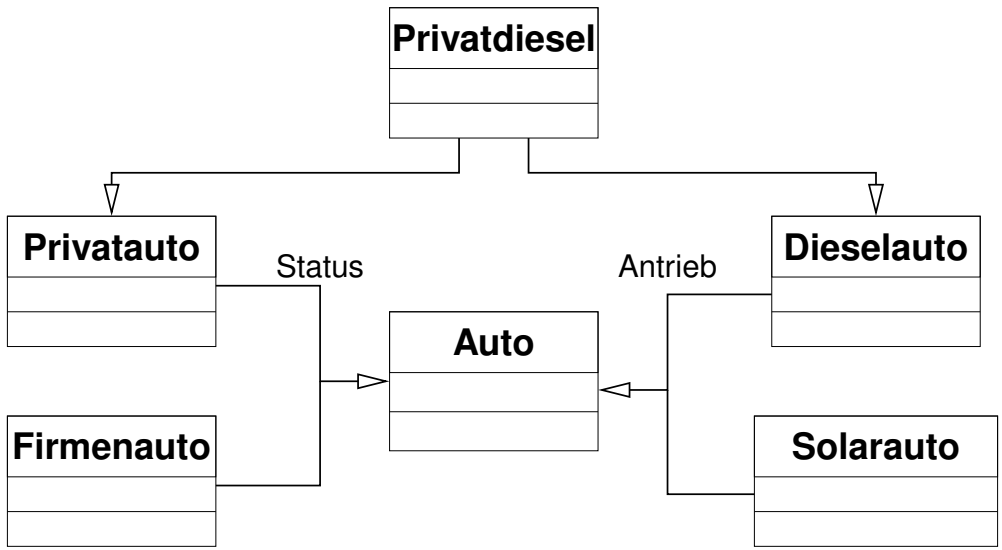


zwischen **B** und **C** sowohl eine Komposition als auch eine Aggregation.

Beziehungen zwischen Klassen

Andererseits kann geeigneter Einsatz von Assoziation / Aggregation / Komposition einige Verwendungen von Vererbung überflüssig machen.

Zum Beispiel ist konzeptionell denkbare Mehrfachvererbung:



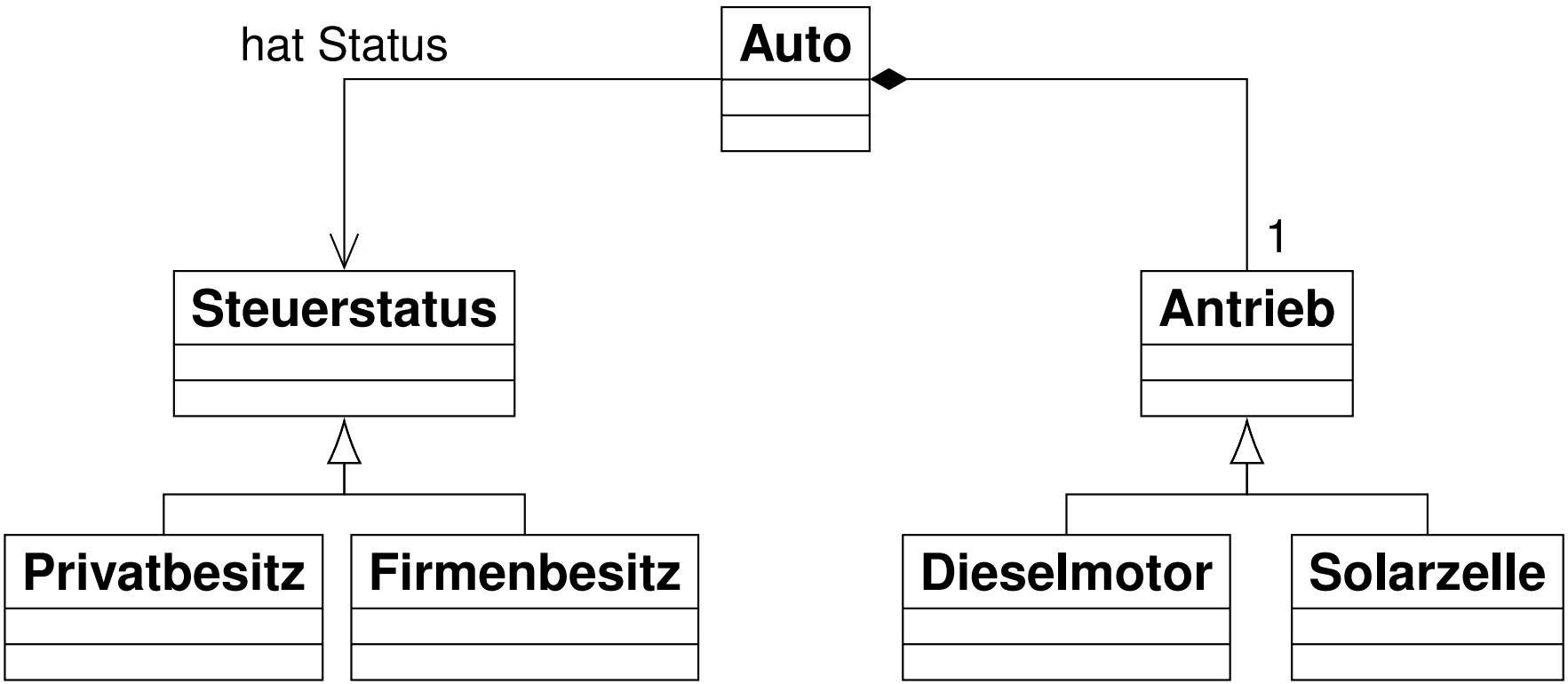
in der Praxis eher problematisch.

Eine alternative Modellierung ist hier möglich . . .

Beziehungen zwischen Klassen

UNIVERSITÄT DUISBURG ESSEN
Offen im Denken

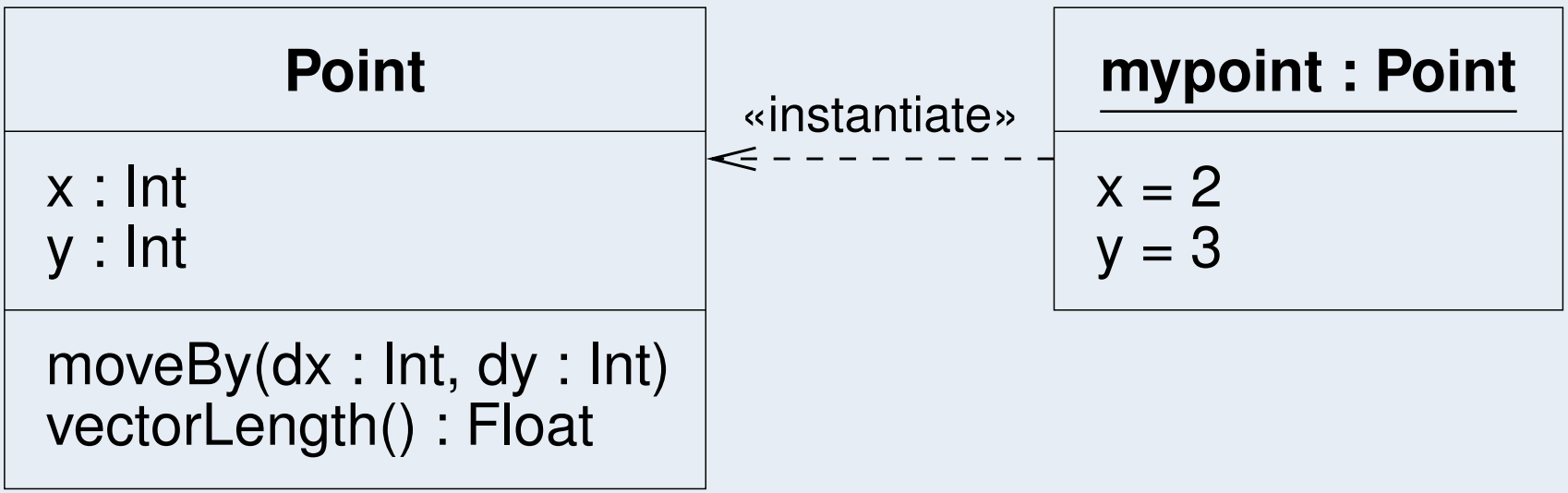
. . . durch Repräsentation der Aspekte „steuerlicher Status“ und „Antrieb“ in eigenen Klassen, und deren Verwendung über Assoziation / Aggregation / Komposition, etwa wie folgt:



Objektdiagramme

Wir betrachten nun Objektdiagramme. Ein Objekt ist, wie bekannt, eine Ausprägung einer Klasse.

Klassen und Objekte



Ein Objektdiagramm beschreibt eine Art Momentaufnahme des Systems: eine Menge von Objekten, wie sie zu einem bestimmten Zeitpunkt vorhanden sind, samt ihren Beziehungen zueinander.

Objektdiagramme

Anmerkungen:

- Ein Objekt kann, muss aber nicht (sondern kann anonym bleiben), durch einen Namen bezeichnet werden. Immer jedoch muss eine Klasse angegeben werden, von der dieses Objekt eine Ausprägung ist (in der Form „: Point“).
- Die Klassen müssen im Objektdiagramm nicht mit abgebildet werden.
- Nicht unbedingt alle Attributbelegungen eines Objekts müssen angegeben werden (es sei denn wir fordern dies in Übung oder Klausur). Daher können durchaus auch Ausprägungen abstrakter Klassen auftauchen.
- Vererbungspfeile zwischen Objekten gibt es nicht!

Links

Ein Link beschreibt eine Beziehung zwischen Objekten. Er ist eine Ausprägung einer auf Klassenebene bestehenden Assoziation (auch Aggregation oder Komposition).

Bemerkungen:

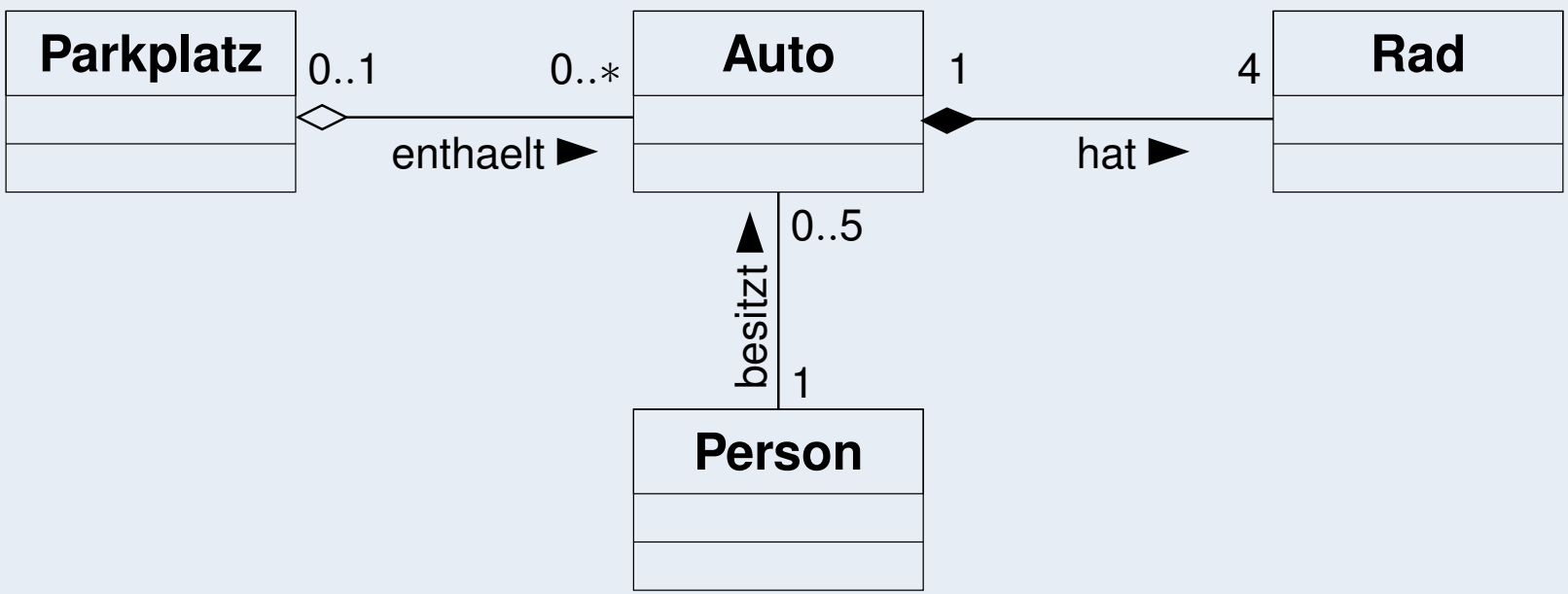
- Links sind nicht mit Multiplizitäten beschriftet, denn ein Link repräsentiert genau eine Beziehung zwischen konkreten Partnern.
- Es ist jedoch darauf zu achten, dass insgesamt die Multiplizitätsbedingungen des Klassendiagramms eingehalten werden. Das heißt, die Anzahlen der Objekte, die miteinander in Beziehung stehen, müssen innerhalb der jeweiligen Schranken sein.

Objektdiagramme

Zurück zu altem Beispiel:

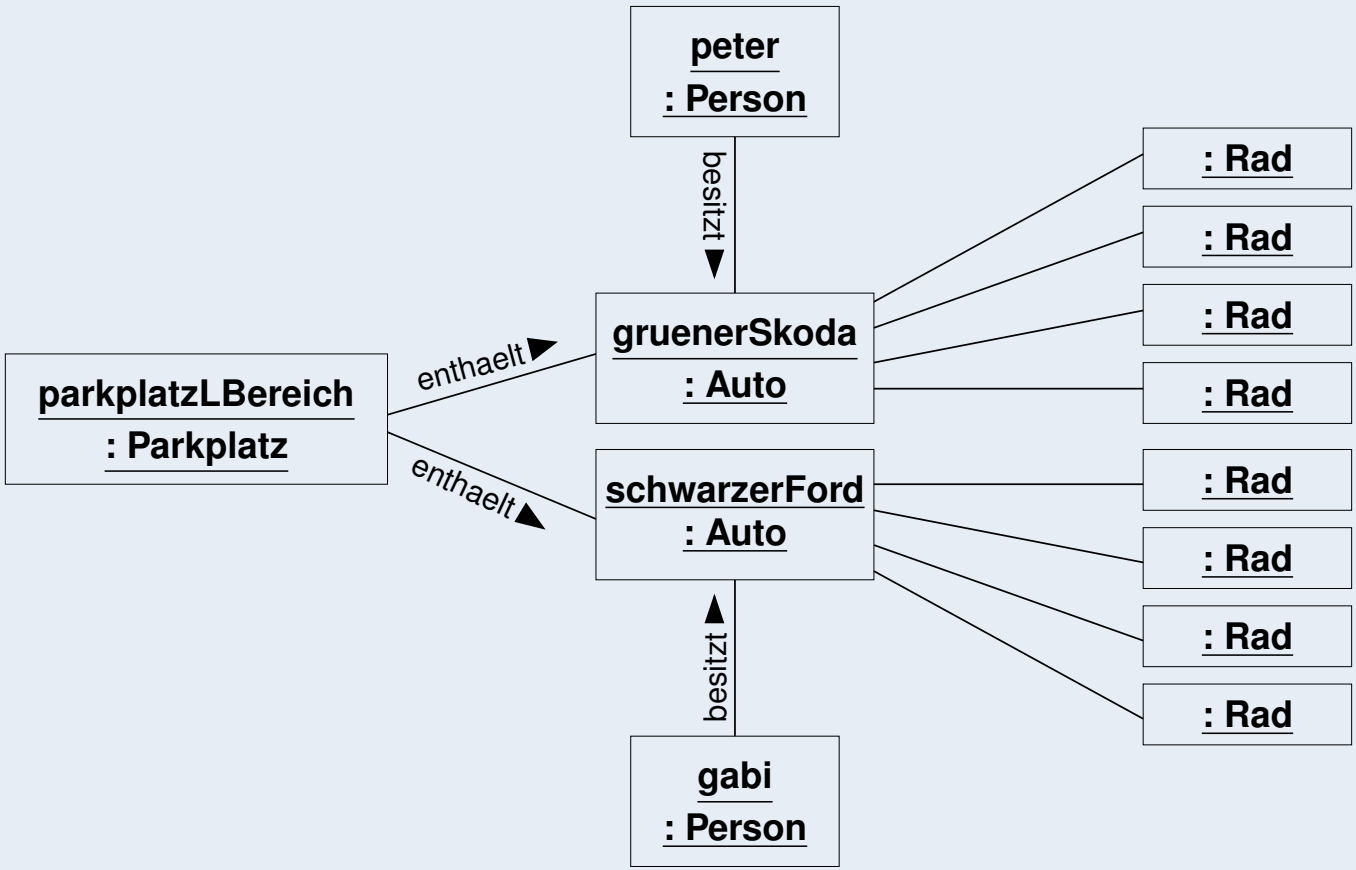
Parkplatz, Autos mit Besitzern, und nun auch noch Räder

Wir betrachten zunächst das (nun erweiterte) Klassendiagramm:



Objektdiagramme

Ein Objektdiagramm passend zu diesem Klassendiagramm:



Objektdiagramme

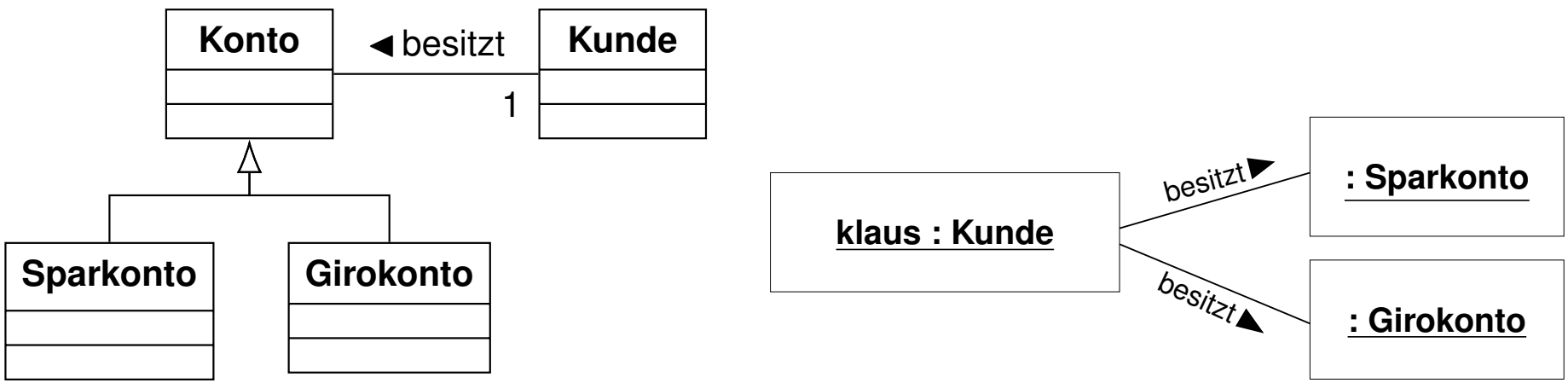
Anmerkungen:

- Auch Aggregations- und Kompositionssymbole dürfen in Objektdiagrammen auftauchen.
- Andererseits können Assoziationsnamen, Navigations- und Leserichtungen auch weggelassen werden.

Objektdiagramme

Achtung: Beziehungen (Assoziationen, Aggregationen, Kompositionen) zwischen Klassen werden vererbt und müssen dann auch entsprechend im Objektdiagramm bei den Ausprägungen von Unterklassen auftauchen.

Beispiel:



nächstes Thema: Petrinetze

Wiederholung einiger mathematischer Hilfsmittel

Menge

Menge M von Elementen, oft beschrieben als Aufzählung

$$M = \{0, 2, 4, 6, 8, \dots\}$$

oder als Menge von Elementen mit einer bestimmten Eigenschaft

$$M = \{n \mid n \in \mathbb{N} \text{ und } n \text{ gerade}\} = \{n \in \mathbb{N} \mid n \text{ gerade}\}.$$

Allgemeines Format: $M = \{x \mid E(x)\}$

M ist Menge aller Elemente, die die Eigenschaft E erfüllen.

$$M = \{x \in X \mid E(x)\}$$

M ist Menge aller entsprechenden Elemente aus Grundmenge X.

Mengenlehre

Bemerkungen:

- Die Elemente einer Menge sind ungeordnet, das heißt, ihre Ordnung spielt keine Rolle. Beispielsweise gilt:

$$\{1,2,3\} = \{1,3,2\} = \{2,1,3\} = \{2,3,1\} = \{3,1,2\} = \{3,2,1\}$$

- Ein Element kann nicht mehrfach in einer Menge auftreten. Es ist entweder in der Menge, oder es ist nicht in der Menge. Beispielsweise gilt:

$$\{1,2,3,4,4\} = \{1,2,3,4\} \neq \{1,2,3\}$$

Mengenlehre

Element einer Menge

Wir schreiben $a \in M$, falls ein Element a in der Menge M enthalten ist.

Anzahl der Elemente einer Menge

Für eine endliche Menge M gibt $|M|$ die Anzahl ihrer Elemente an.

Teilmengenbeziehung

Wir schreiben $A \subseteq B$, falls jedes Element von A auch in B enthalten ist.

Leere Menge

Mit $\emptyset$ oder $\{\}$ bezeichnen wir die leere Menge. Sie enthält keine Elemente und ist (echte) Teilmenge jeder anderen Menge.

Mengenvereinigung

Die Vereinigung zweier Mengen A und B ist diejenige Menge, welche die Elemente enthält, die in A oder B (oder in beiden) vorkommen. Man schreibt dafür $A \cup B$.

$$A \cup B = \{x \mid x \in A \text{ oder } x \in B\}$$

Mengenschnitt

Der Schnitt zweier Mengen A und B ist diejenige Menge, welche die Element enthält, die sowohl in A als auch in B vorkommen. Man schreibt dafür $A \cap B$.

$$A \cap B = \{x \mid x \in A \text{ und } x \in B\}$$

Kreuzprodukt (Kartesisches Produkt)

Das Kreuzprodukt zweier Mengen A und B ist diejenige Menge, welche alle Paare (a, b) enthält, wobei die erste Komponente des Paars aus A, die zweite aus B kommt. Man schreibt dafür $A \times B$.

$$A \times B = \{(a, b) \mid a \in A \text{ und } b \in B\}$$

Kreuzprodukt (Kartesisches Produkt)

$$A \times B = \{(a,b) \mid a \in A \text{ und } b \in B\}$$

Beispiele:

- $\{1,2\} \times \{3,4,5\} = \{(1,3),(1,4),(1,5),(2,3),(2,4),(2,5)\}$
- $\{1,2\} \times \emptyset = \emptyset$

Anmerkungen:

- Für endliche Mengen A und B gilt $|A \times B| = |A| \cdot |B|$.
- Wenn $A \subseteq B$, dann $A \times C \subseteq B \times C$ und $C \times A \subseteq C \times B$.

Mengenlehre

Weitere Bemerkungen:

- Wir betrachten nicht nur Paare, sondern auch Tupel aus mehr als zwei Komponenten (Tripel, Quadrupel, Quintupel, ...).
  Ein Tupel $(a_1, \ldots, a_n)$ bestehend aus n Komponenten heißt auch n-Tupel.
- In einem Tupel sind die Komponenten geordnet! Es gilt z.B.:

$$(1,2,3) \neq (1,3,2) \in \mathbb{N} \times \mathbb{N} \times \mathbb{N}$$

- Ein Element kann mehrfach in einem Tupel auftreten. Tupel unterschiedlicher Länge sind immer verschieden. Beispielsweise:

$$(1,2,3,4) \neq (1,2,3,4,4)$$

Potenzmenge

Die Potenzmenge einer Menge M ist diejenige Menge, welche alle Teilmengen von M enthält. Man schreibt dafür $\mathscr{P}(M)$.

$$\mathscr{P}(M) = \{A \mid A \subseteq M\}$$

Beispiele:

- $\mathscr{P}(\{1,2,3\}) = \{\emptyset, \{1\}, \{2\}, \{3\}, \{1,2\}, \{1,3\}, \{2,3\}, \{1,2,3\}\}$
- $\mathscr{P}(\emptyset) = \{\emptyset\}$
- $\mathscr{P}(\mathscr{P}(\emptyset)) = \{\emptyset, \{\emptyset\}\}$

Anmerkung: Für endliche Mengen M gilt $|\mathscr{P}(M)| = 2^{|M|}$.

Mengenlehre: Anwendung

UNIVERSITÄT DUISBURG ESSEN
Offen im Denken

Beispiel: Zustandsmodellierung

Angenommen, wir betrachten einen einfachen Snackautomaten für Riegel und Chips. Von jedem dieser beiden Snacks hat er maximal 30 Stück auf Vorrat. Der Automat hat eine gelbe und eine rote Warnleuchte („kein Wechselgeld mehr“ bzw. „keine Scheine mehr akzeptiert“), die unabhängig voneinander leuchten können. Die Menge der möglichen Zustände dieses Automaten können wir als

$$\mathscr{P}(\{\text{gelb}, \text{rot}\}) \times \{0, 1, \ldots, 30\} \times \{0, 1, \ldots, 30\}$$

beschreiben. Das Element $(\emptyset,\ 20, 10)$ dieser Menge zum Beispiel entspricht dem Zustand, in dem beide Warnleuchten ausgeschaltet sind und noch 20 Riegel und 10 Packungen Chips vorrätig. Wären bei diesem Vorrat die Warnleuchten beide eingeschaltet, so befände sich der Automat stattdessen im Zustand $(\{\text{gelb}, \text{rot}\},\ 20, 10)$.

Funktion (Abbildung)

Funktionen bilden Elemente eines Definitionsbereiches auf Elemente eines Wertebereiches ab. Man schreibt: „$f : A \to B$“.
Paare aus einem Element a des Definitionsbereiches A und dem (eindeutig gegebenen) Element $b = f(a)$ des Wertebereiches B, auf welches die Funktion es abbildet, notiert man in der Form „$a \mapsto b$“.
Die gleiche Notation verwendet man, um eine allgemeine Zuordnungsvorschrift anzugeben: „$a \mapsto f(a)$“.

Beispiel: Quadratfunktion auf der Menge der ganzen Zahlen

$$f : \mathbb{Z} \to \mathbb{N}, \quad f(z) = z^2, \quad \text{bzw. Angabe als: } z \mapsto z^2$$

Angabe konkreter Paare:

$$\ldots, -3 \mapsto 9, -2 \mapsto 4, -1 \mapsto 1, 0 \mapsto 0, 1 \mapsto 1, 2 \mapsto 4, 3 \mapsto 9, \ldots$$

Gerichteter und kantenbeschrifteter Graph

Sei L eine Menge von Beschriftungen (oder Labels).

Ein gerichteter und kantenbeschrifteter Graph $G = (V, E)$ besteht aus

- einer Knotenmenge V und
- einer Kantenmenge $E \subseteq V \times L \times V$.

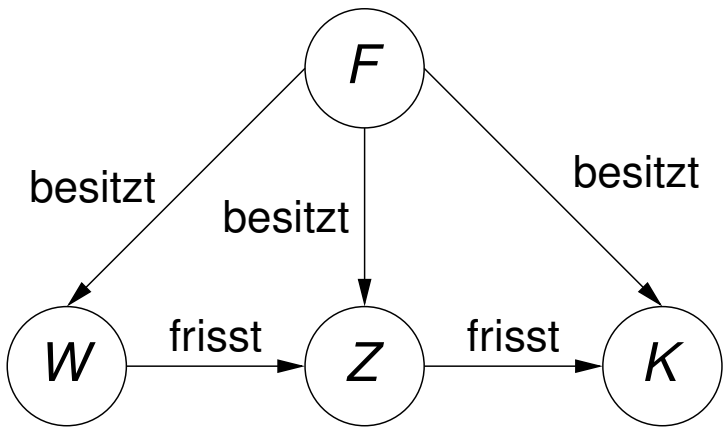
Bemerkung: V steht für vertices und E für edges.

Beispiel: (Farmer, Wolf, Ziege, Kohlkopf)

$$V = \{F, W, Z, K\} \qquad L = \{\text{besitzt}, \text{frisst}\}$$

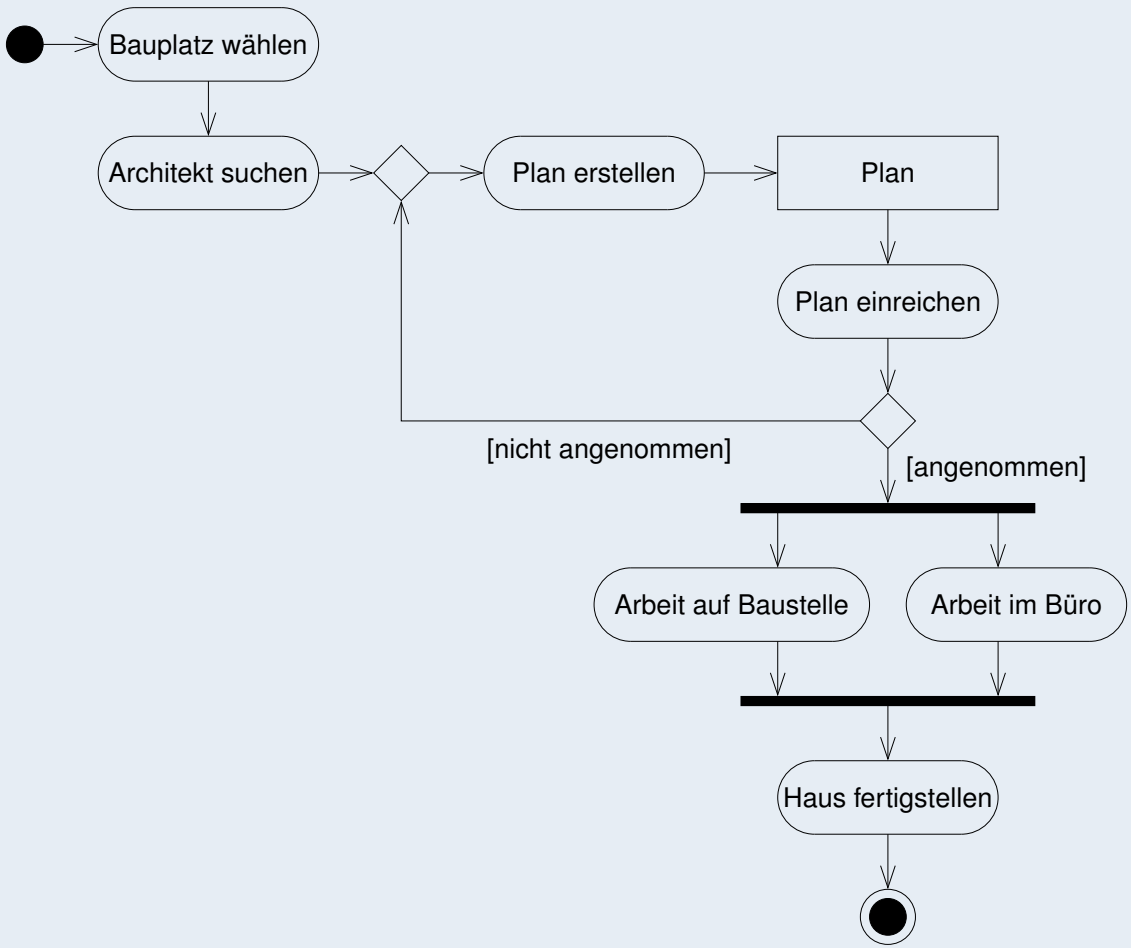
$$E = \{(F, \text{besitzt}, W), (F, \text{besitzt}, Z), (F, \text{besitzt}, K), (W, \text{frisst}, Z), (Z, \text{frisst}, K)\}$$

Bildhaft:



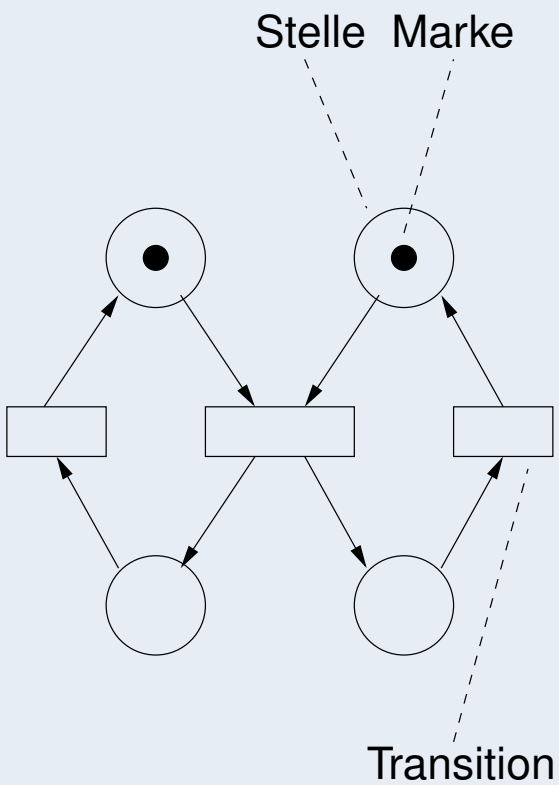
Beispiel für ein UML-Aktivitätsdiagramm

Hausbau



Petrinetze

- Modellierung nebenläufiger und verteilter Systeme, zur Beschreibung der gemeinsamen Nutzung von Ressourcen
- Schwerpunkt auf der Modellierung des dynamischen Verhaltens
- etablierter Ansatz, der vielfältig eingesetzt wird
- besitzt formale Semantik
- entwickelt von Carl Adam Petri (1962)



Petrinetze: Grundlagen und Erreichbarkeitsgraphen

Motivation: Petrinetze

UNIVERSITÄT DUISBURG ESSEN
Offen im Denken

Petrinetze sind ein Formalismus zur Modellierung mit folgenden Eigenschaften:

- Vorstellung von Systemübergängen, bei denen (gemeinsame) Ressourcen konsumiert und neu erzeugt werden können.
- Einfache Modellierung von Kapazitäten, räumlicher Verteilung der Ressourcen, von Nebenläufigkeit, Parallelität und (Zugriffs-)Konflikten.
- Intuitive grafische Darstellung.

Petrinetze werden in der Praxis vielfach benutzt.

In UML sind sie abgewandelt als sogenannte Aktivitätsdiagramme (englisch: activity diagrams) eingegangen.

Motivation: Petrinetze

Parallelität versus Nebenläufigkeit:

Parallelität

Zwei Ereignisse finden parallel statt, wenn sie gleichzeitig ausgeführt werden.

Nebenläufigkeit

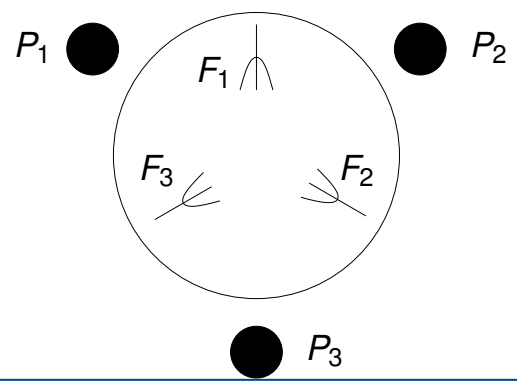
Zwei Ereignisse sind nebenläufig, wenn sie parallel ausgeführt werden können (jedoch nicht müssen), das heißt, wenn zwischen ihnen keine kausale Abhängigkeit besteht.

Das bedeutet: Nebenläufigkeit ist der allgemeinere Begriff.

Motivation: Petrinetze

Ein verbreitetes Beispiel: Dining Philosophers

- Es sitzen drei (oder vier, oder fünf, . . .) Philosophen P_i um einen runden Tisch, zwischen je zwei Philosophen liegt eine Gabel (fork) F_i.
- Philosophen werden von Zeit zu Zeit hungrig und benötigen dann zum Essen beide benachbarte Gabeln.
- Jeder Philosoph nimmt zu einem beliebigen Zeitpunkt beide Gabeln nacheinander auf (die rechte zuerst), isst und legt anschließend beide Gabeln wieder zurück (gleichzeitig).



Motivation: Petrinetze

Fragen, die man (zum Beispiel) mit Petrinetzen untersuchen kann:

- Kann das modellierte System kontinuierlich immer weiteren Fortschritt machen, oder lässt es sich in Sackgassen manövrieren?
- Bekommt jede modellierte Aktion die Chance, auch tatsächlich ausgeführt zu werden?
  Nur einmalig oder sogar beliebig oft wiederholt?
- Besteht Fairness für verschiedene Akteure?
- Bedingen bestimmte Aktionen einander, oder schließen sich gegenseitig aus?
- Gibt es Beschränkungen für eventuellen Ressourcenverbrauch und -erzeugung?

Dabei sind bestimmte Analysen sogar für Systeme möglich, deren (flaches) Zustandsdiagramm unendlich wäre.

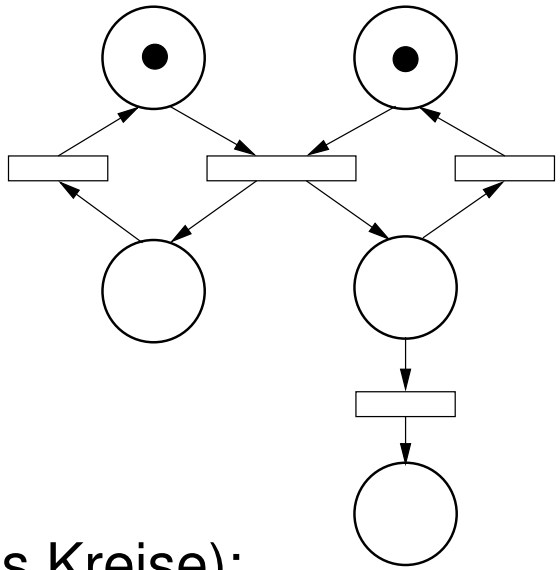
Motivation: Petrinetze

Praktische Anwendungen für Petrinetze:

- Modellierung von Arbeitsabläufen
  (work flow, business processes)
- Modellierung und Analyse von Web Services
- Beschreibung von grafischen Benutzungsoberflächen
- Prozessmodellierung bei Betriebssystemen
- Ablaufbeschreibungen in ingenieurwissenschaftlichen Anwendungen
- ...

Petrinetze: Informelle Einführung

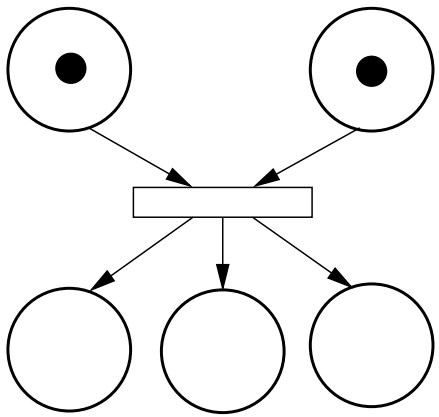
Beispiel für ein Petrinetz:



Grafische Darstellung:

- Stellen (dargestellt als Kreise):
  Mögliche Plätze für Ressourcen
- Marken (dargestellt als kleine ausgefüllte Kreise):
  Ressourcen
- Transitionen (dargestellt durch Rechtecke):
  Systemübergänge

Mehr zur Darstellung und Bedeutung einer Transition:



Vorbedingung (Marken, die konsumiert werden)

Nachbedingung (Marken, die erzeugt werden)

Das Entfernen der Marken der Vorbedingung und Erzeugen der Marken der Nachbedingung nennt man Schalten bzw. Feuern der Transition.

Allgemeiner als im Beispiel oben muss nicht unbedingt genau eine Marke pro Pfeil konsumiert oder erzeugt werden.

Und weder müssen die Stellen der Nachbedingung vor dem Schalten leer sein, noch die Stellen der Vorbedingung danach.

Petrinetze: Definitionen

Petrinetz (Definition / Syntax)

Ein Petrinetz ist ein Tupel $N = (S, T, {}^\bullet(), ()^\bullet, m_0)$, wobei

- S eine endliche, nichtleere Menge von Stellen und
- T eine endliche, nichtleere Menge von Transitionen ist.
- Außerdem gibt es für jede Transition t zwei Funktionen ${}^\bullet t : S \to \mathbb{N}$ und $t^\bullet : S \to \mathbb{N}$, die angeben, wie viele Marken durch t aus einer Stelle entnommen bzw. in eine Stelle gelegt werden.
- Und $m_0 : S \to \mathbb{N}$ ist die Anfangsmarkierung (oder initiale Markierung).

Der Wert ${}^\bullet t(s)$ bzw. $t^\bullet(s)$ wird jeweils als Gewicht bezeichnet.

Markierung

Eine Markierung ist eine Funktion $m : S \to \mathbb{N}$.

Sie kann festhalten:

- wie viele Marken aktuell in einzelnen Stellen liegen,
- wie viele Marken einzelnen Stellen zu entnehmen sind, oder
- wie viele Marken einzelnen Stellen hinzuzufügen sind.

Falls eine Reihenfolge $s_1, \ldots, s_n$ der Stellen fixiert wurde, kann eine Markierung m auch durch ein Tupel $(m(s_1), \ldots, m(s_n))$ ausgedrückt werden.

Petrinetze: Darstellung

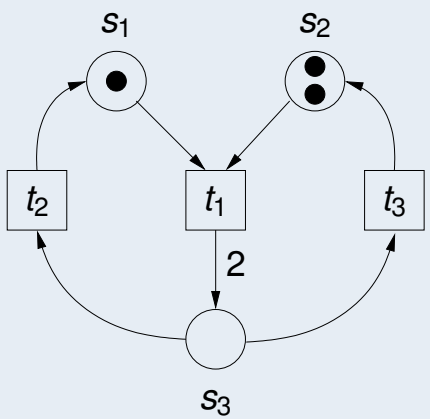
UNIVERSITÄT DUISBURG ESSEN
Offen im Denken

Genaueres/Verbindung zur grafischen Darstellung:

- Stellen werden als Kreise, Transitionen als Quadrate oder Rechtecke, Marken als kleine ausgefüllte Kreise dargestellt.
- Kanten zwischen Stellen und Transitionen werden als Pfeile dargestellt.
- Die Kanten sind eigentlich mit dem jeweiligen Gewicht beschriftet. Dieses kann jedoch weggelassen werden, falls es den Wert 1 hat. Nur falls ein Gewicht den Wert 0 hat, wird die ganze Kante weggelassen.

Petrinetze: Darstellung

Betrachten wir den Zusammenhang zwischen der mathematischen Notation und der grafischen Darstellung an einem Beispiel:



Stellenreihenfolge:
s_1, s_2, s_3

$$S = \{s_1, s_2, s_3\}$$
$$T = \{t_1, t_2, t_3\}$$

$${}^\bullet t_1 \colon s_1 \mapsto 1,\ s_2 \mapsto 1,\ s_3 \mapsto 0$$
$$t_1^\bullet \colon s_1 \mapsto 0,\ s_2 \mapsto 0,\ s_3 \mapsto 2$$
$$\underline{\text{oder:}}\ {}^\bullet t_1 = (1,1,0) \quad t_1^\bullet = (0,0,2)$$

$$\ldots$$

$$m_0 \colon s_1 \mapsto 1,\ s_2 \mapsto 2,\ s_3 \mapsto 0$$
$$\underline{\text{oder:}}\ m_0 = (1,2,0)$$

Petrinetze: Wiederholung

Wichtige Konzepte:

- Markierungen: Funktionen $m : S \to \mathbb{N}$
- Vor- und Nachgewichte: ${}^\bullet t$ und $t^\bullet$, selbst Markierungen

Zusammenhang mit grafischer Darstellung:

- Anfangsmarkierung zur Belegung der Stellen
- kein Pfeil von s zu t, falls ${}^\bullet t(s) = 0$
- kein Pfeil von t zu s, falls $t^\bullet(s) = 0$
- Pfeil von s zu t, falls ${}^\bullet t(s) = 1$
- Pfeil von t zu s, falls $t^\bullet(s) = 1$
- Pfeil mit Beschriftung n von s zu t, falls ${}^\bullet t(s) = n > 1$
- Pfeil mit Beschriftung n von t zu s, falls $t^\bullet(s) = n > 1$

Ordnung und Operationen auf Markierungen:

Seien $m, m' : S \to \mathbb{N}$ zwei Markierungen, also Abbildungen von Stellen auf natürliche Zahlen.

Ordnung (Definition)

Es gilt $m' \leq m$ falls für alle $s \in S$ gilt: $m'(s) \leq m(s)$.

In diesem Fall sagt man, dass m' durch m überdeckt wird.

Beispiele: Sei $|S| = 3$, $m = (0,1,2)$, $m' = (0,0,1)$.
Dann gilt $m' \leq m$, aber nicht $m \leq m'$.

Es gilt auch $(0,1,0) \leq (0,1,0)$.

Aber nicht $(3,1,2) \leq (5,1000,1)$.

Und weder $(0,1,2) \leq (0,2,1)$, noch $(0,2,1) \leq (0,1,2)$.

Ordnung und Operationen auf Markierungen:

Seien $m, m' : S \to \mathbb{N}$ zwei Markierungen, also Abbildungen von Stellen auf natürliche Zahlen.

Addition (Definition)

Wir definieren $m'' = m \oplus m'$, wobei $m'' : S \to \mathbb{N}$ mit $m''(s) = m(s) + m'(s)$ für alle $s \in S$.

Beispiel: $(0,1,2) \oplus (0,0,1) = (0,1,3) = (0,0,1) \oplus (0,1,2)$

Subtraktion (Definition)

Falls $m' \leq m$, definieren wir $m'' = m \ominus m'$, wobei $m'' : S \to \mathbb{N}$ mit $m''(s) = m(s) - m'(s)$ für alle $s \in S$.

Beispiel: $(0,1,2) \ominus (0,0,1) = (0,1,1)$

Weitere Konzepte:

Aktivierung (Definition)

Eine Transition t ist für eine Markierung m aktiviert falls ${}^\bullet t \leq m$ gilt.
(Das heißt, falls in m je Stelle genug Marken vorhanden sind, um die Vorbedingung von t zu erfüllen.)

Schalten (Definition)

Sei t eine Transition und m eine Markierung, für die t aktiviert ist.
Dann kann t schalten, was zu der Nachfolgemarkierung $m' = m \ominus {}^\bullet t \oplus t^\bullet$ führt; zu lesen als $(m \ominus {}^\bullet t) \oplus t^\bullet$.

Symbolisch dargestellt: $m\,[t\rangle\, m'$.

Weitere Konzepte:

Erreichbarkeit (Definition)

Man nennt eine Markierung m erreichbar (von m_0 aus), wenn es eine endliche Folge von Transitionen $t_1, \ldots, t_n$ gibt mit

$$m_0 \,[t_1\rangle\, m_1 \,[t_2\rangle\, \cdots\, m_{n-1} \,[t_n\rangle\, m\,,$$

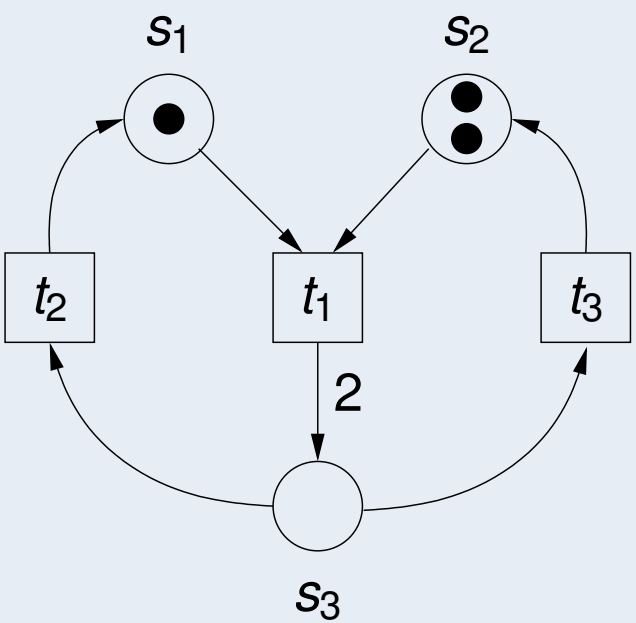
wobei m_0 die Anfangsmarkierung des Petrinetzes ist.

Man schreibt auch $m_0 \,[t_1 \ldots t_n\rangle\, m$, oder $m_0 \,[\tilde{t}\,\rangle\, m$ mit $\tilde{t} = t_1 \ldots t_n$.

Die Sequenz $\tilde{t}$ nennt man Schaltfolge.

Auch die leere Schaltfolge $\tilde{t} = \varepsilon$ ist möglich. In diesem Fall ändert sich die Markierung nicht: $m \,[\varepsilon\rangle\, m$, für jede Markierung m.

Petrinetze: Dynamik / Semantik



Stellenreihenfolge: s_1, s_2, s_3

Die Markierung $m_2 = (1,1,1)$ ist in zwei Schritten erreichbar:

1. $^\bullet t_1 = (1,1,0) \leq (1,2,0) = m_0$

   $m_1 = m_0 \ominus {}^\bullet t_1 \oplus t_1^\bullet$
   $= (1,2,0) \ominus (1,1,0) \oplus (0,0,2)$
   $= (0,1,2)$

2. $^\bullet t_2 = (0,0,1) \leq (0,1,2) = m_1$

   $m_2 = m_1 \ominus {}^\bullet t_2 \oplus t_2^\bullet$
   $= (0,1,2) \ominus (0,0,1) \oplus (1,0,0)$
   $= (1,1,1)$

Es gilt also: $m_0 \,[t_1\rangle\, m_1 \,[t_2\rangle\, m_2$, oder: $m_0 \,[t_1\, t_2\rangle\, m_2$.

Es gilt auch: $m_0 \,[t_1\rangle\, m_1 \,[t_3\rangle\, (0,2,1)$.

Nicht-Determinismus

Petrinetze sind ein nicht-deterministischer Mechanismus.

Das heißt, zu einer Markierung kann es unter Anwendung verschiedener Transitionen mehrere direkte Nachfolgemarkierungen geben.

Die Frage, wer die Transition bzw. Nachfolgemarkierung auswählt, stellt sich für die Modellierung nicht. Das Modell beschreibt alle Möglichkeiten und trifft keine Auswahl.

Flaches Zustandsdiagramm eines Petrinetzes (Definition)

Sei $N = (S, T, {}^\bullet(), ()^\bullet, m_0)$ ein Petrinetz.

Dann besteht das zu N gehörende flache Zustandsdiagramm aus folgenden Komponenten:

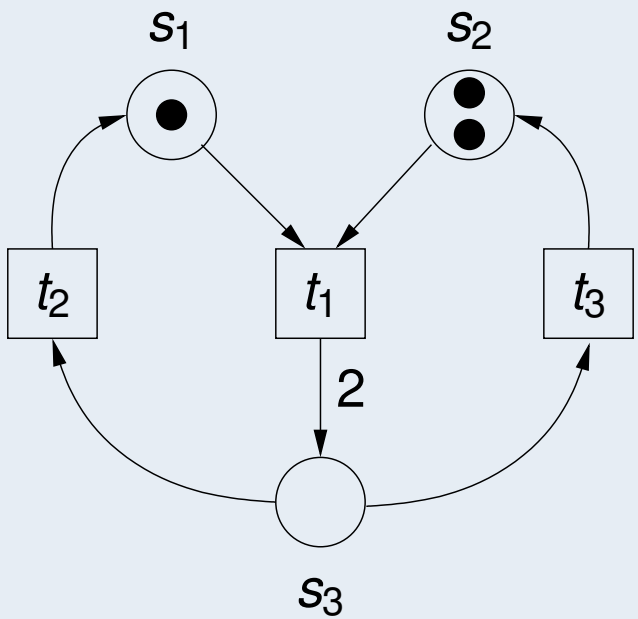
- Knotenmenge V bzw. Z (Zustände):
  Menge aller von m_0 aus erreichbaren Markierungen
- Kantenbeschriftungsmenge L:
  Menge aller Transitionen
- Kantenmenge E bzw. U (Übergänge):
  $(m, t, m') \in U \iff m\,[t\rangle\, m'$
- Startzustand $z_0 \in Z$:
  die Anfangsmarkierung m_0

Petrinetze: Dynamik / Semantik

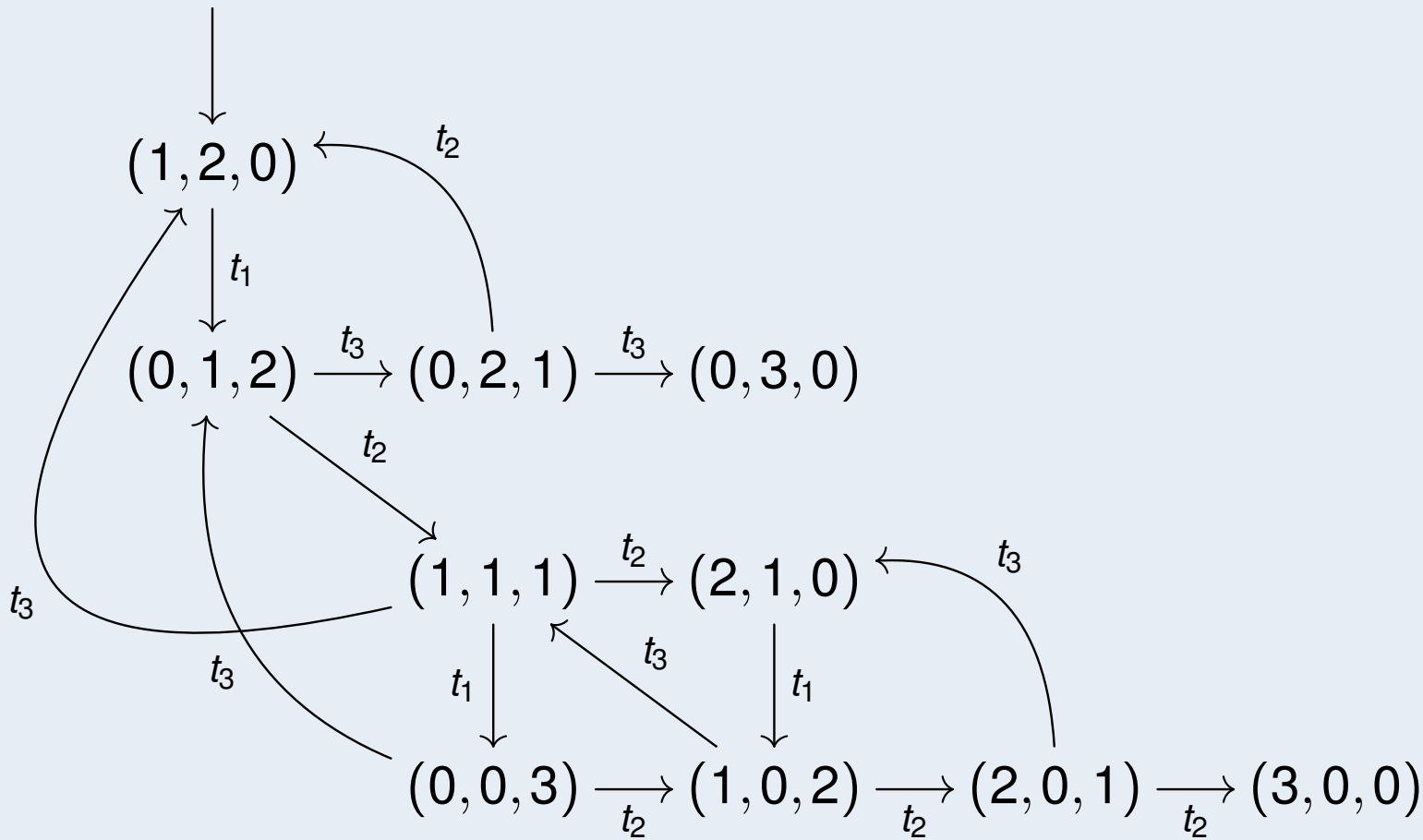
Anmerkungen:

- Das flache Zustandsdiagramm eines Petrinetzes nennt man auch dessen Erreichbarkeitsgraph.
- Trotz Endlichkeit des Petrinetzes kann der Erreichbarkeitsgraph unendlich werden!

Beispiel: Bestimme den Erreichbarkeitsgraph für das folgende Petrinetz



Erreichbarkeitsgraph für das Beispielnetz (mit Stellenreihenfolge s_1, s_2, s_3)



Eigenschaften von Petrinetzen

Wir betrachten nun auf allgemeine Weise Begriffe wie Lebendigkeit und Deadlock (= Verklemmung), später noch weitere.

Starke Lebendigkeit (Definition)

Man nennt ein Petrinetz stark lebendig, wenn es für jede Transition t und jede von m_0 aus erreichbare Markierung m eine Markierung m' gibt, die von m aus erreichbar ist und für die t aktiviert ist.

Bezüglich des Erreichbarkeitsgraphen bedeutet dies, für jede Transition t: von jedem Knoten des Graphen aus ist ein Übergang erreichbar, der mit t beschriftet ist.

Schwache Lebendigkeit (Definition)

Man nennt ein Petrinetz schwach lebendig, wenn es für jede Transition t eine von m_0 aus erreichbare Markierung gibt, für die t aktiviert ist.

Bezüglich des Erreichbarkeitsgraphen bedeutet dies, dass es für jede Transition t mindestens einen Übergang gibt, der mit t beschriftet ist.

Verklemmung (Definition)

Man sagt, dass ein Petrinetz eine Verklemmung (oder einen Deadlock) enthält, wenn es eine von m_0 aus erreichbare Markierung gibt, für die keine Transition aktiviert ist.

Bezüglich des Erreichbarkeitsgraphen bedeutet dies, dass es einen Knoten gibt, von dem aus es keinen Übergang gibt.

Ein Petrinetz, das keine Verklemmung enthält, nennt man verklemmungsfrei.

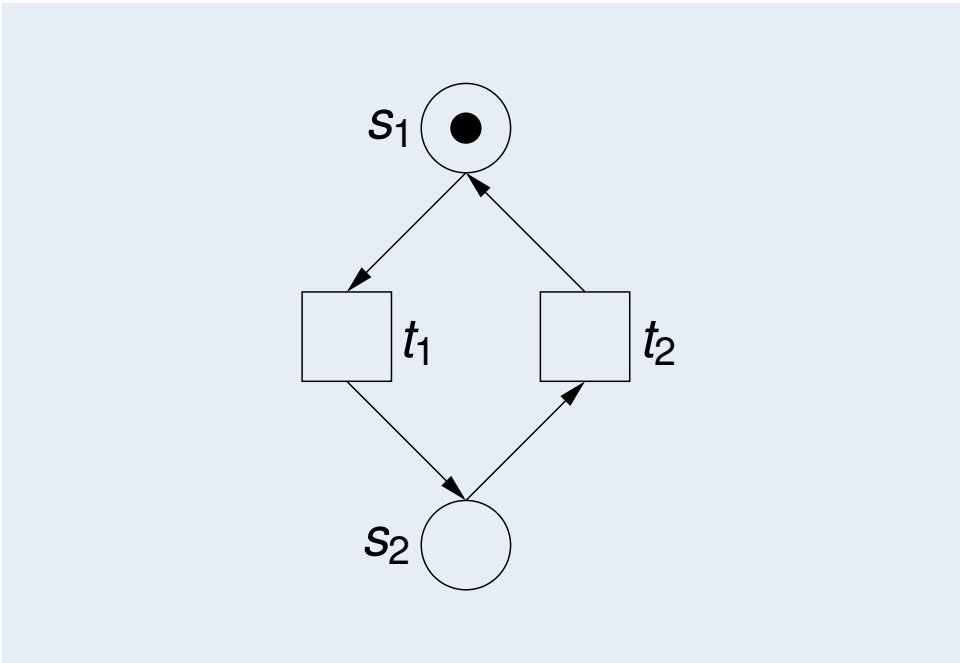
Stärke der starken Lebendigkeit

Da wir nur Petrinetze betrachten, deren Transitionsmenge nicht leer ist, gilt:

Jedes stark lebendige Petrinetz ist sowohl schwach lebendig als auch verklemmungsfrei.

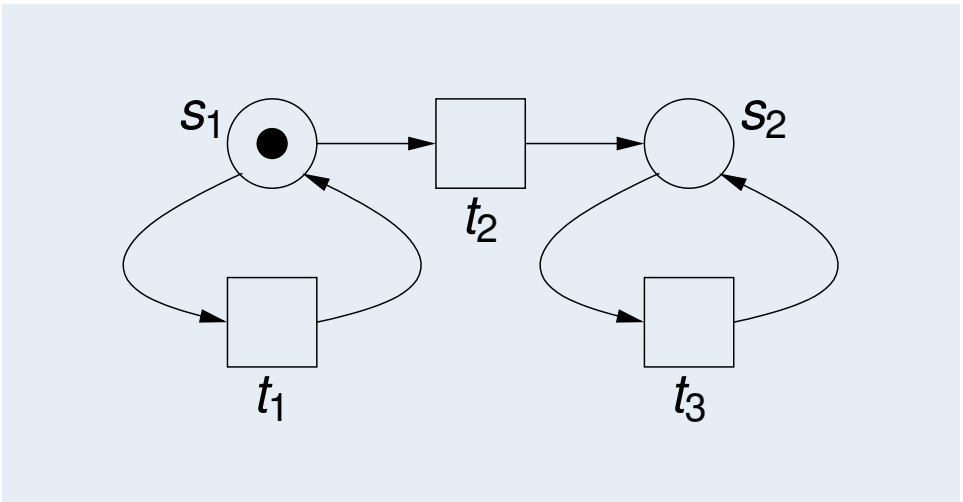
Beispiele für Lebendigkeit und Verklemmungen:

Ein Beispiel für ein stark lebendiges Petrinetz . . .



Beispiele für Lebendigkeit und Verklemmungen:

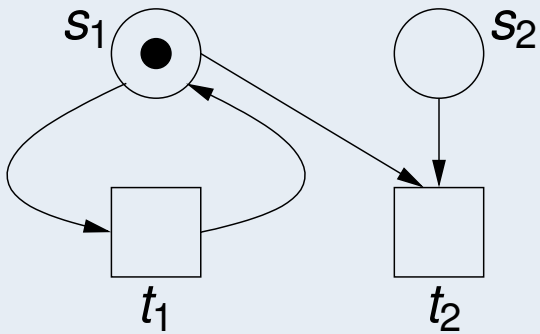
Ein Beispiel für ein schwach lebendiges und verklemmungsfreies Petrinetz, das jedoch nicht stark lebendig ist . . .



Petrinetze: Lebendigkeit und Verklemmungen

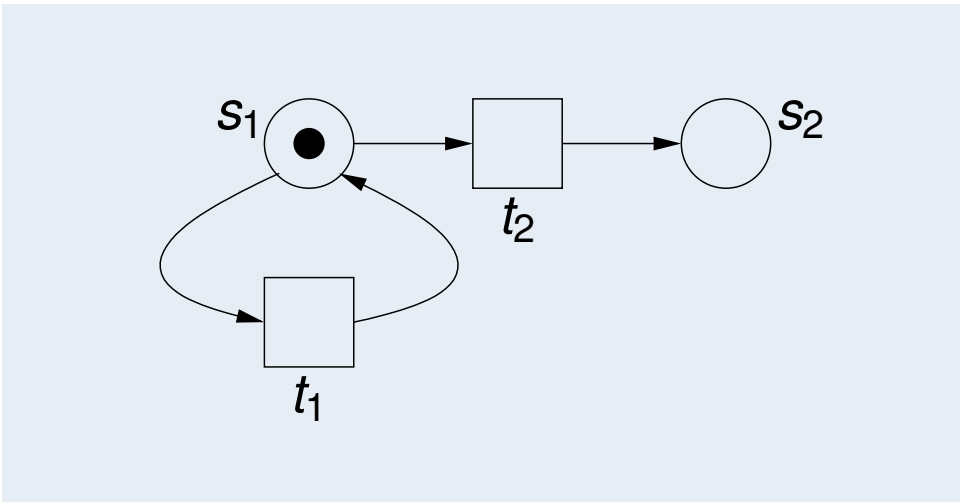
Beispiele für Lebendigkeit und Verklemmungen:

Ein Beispiel für ein verklemmungsfreies Petrinetz, das jedoch nicht schwach lebendig ist . . .



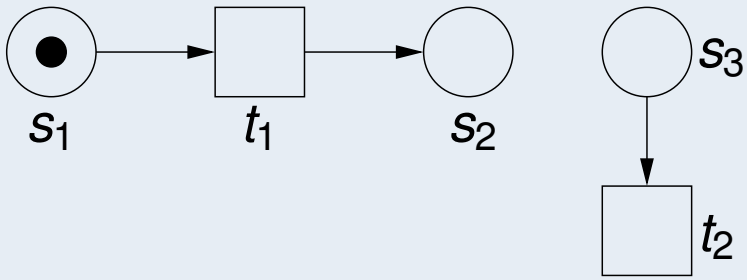
Beispiele für Lebendigkeit und Verklemmungen:

Ein Beispiel für ein schwach lebendiges Petrinetz, das jedoch eine Verklemmung enthält . . .

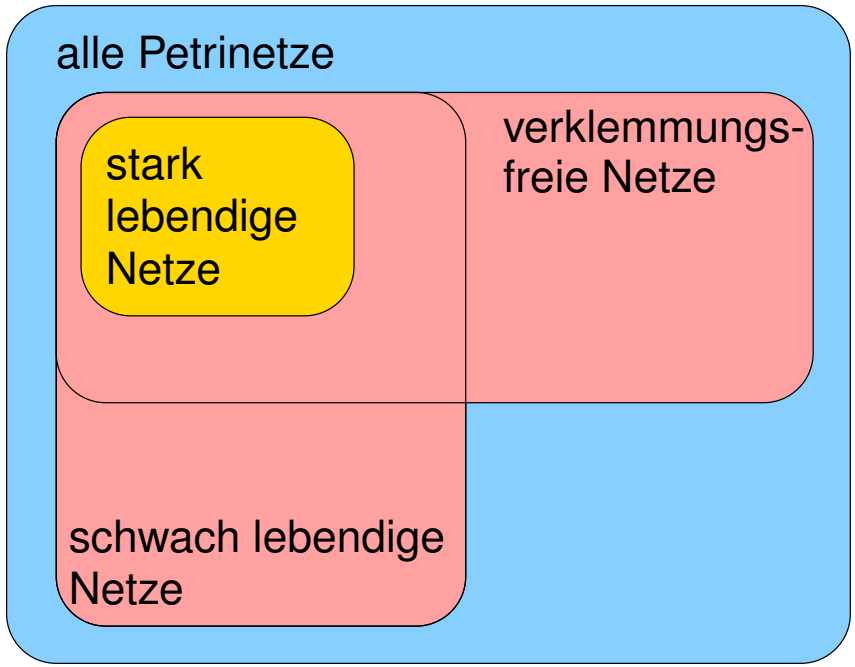


Beispiele für Lebendigkeit und Verklemmungen:

Ein Beispiel für ein Petrinetz, das eine Verklemmung enthält und das auch nicht schwach lebendig ist . . .



Überblick über die verschiedenen Petrinetzklassen:



Sichere, beschränkte und unbeschränkte Petrinetze (Definition)

Man nennt ein Petrinetz ...

- 1-sicher, wenn
  - Für jede Transition t und für jede Stelle s gilt: ${}^\bullet t(s) \leq 1$ und $t^\bullet(s) \leq 1$, also alle Gewichte sind höchstens 1, und
  - für jede erreichbare Markierung m und jede Stelle s gilt, dass $m(s) \leq 1$.
- beschränkt, wenn es eine Konstante $c \in \mathbb{N}$ gibt, so dass für jede erreichbare Markierung m und jede Stelle s gilt, dass $m(s) \leq c$.
- unbeschränkt, wenn es für jede Konstante $c \in \mathbb{N}$ eine erreichbare Markierung m und eine Stelle s gibt mit $m(s) > c$.

Beobachtung: Ein Petrinetz ist unbeschränkt genau dann, wenn sein Erreichbarkeitsgraph unendlich groß ist.

Petrinetze: Kausalität, Nebenläufigkeit, Konflikt

Weitere wichtige Begriffe bei Petrinetzen sind Kausalität, Nebenläufigkeit und Konflikt.

Wir beschäftigen uns auch damit etwas genauer.

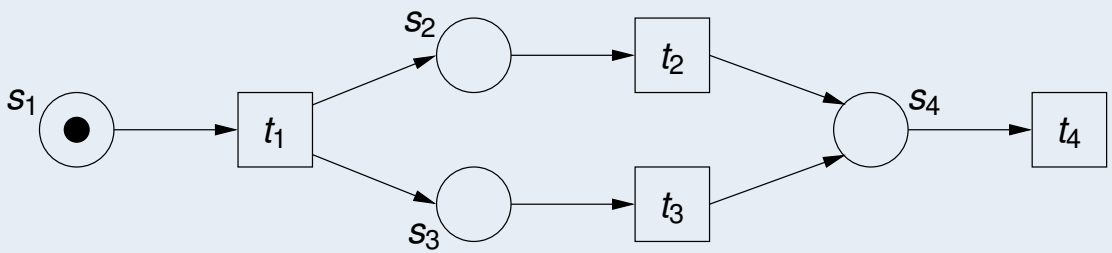
Kausalität (Definition)

In einem Petrinetz nennt man die Transition t_1 eine notwendige Bedingung für das Schalten der Transition t_2 genau dann, wenn für alle Schaltfolgen $\tilde{t}$ gilt:

$$\text{falls } m_0 \,[\tilde{t}\, t_2\rangle\, m \text{ für eine Markierung } m, \text{ dann enthält } \tilde{t} \text{ definitiv die Transition } t_1.$$

Bezüglich des Erreichbarkeitsgraphen bedeutet dies, dass jeder Knoten, von dem aus es einen mit t_2 beschrifteten Übergang gibt, nur über Wege erreichbar ist, in denen t_1 vorkommt.

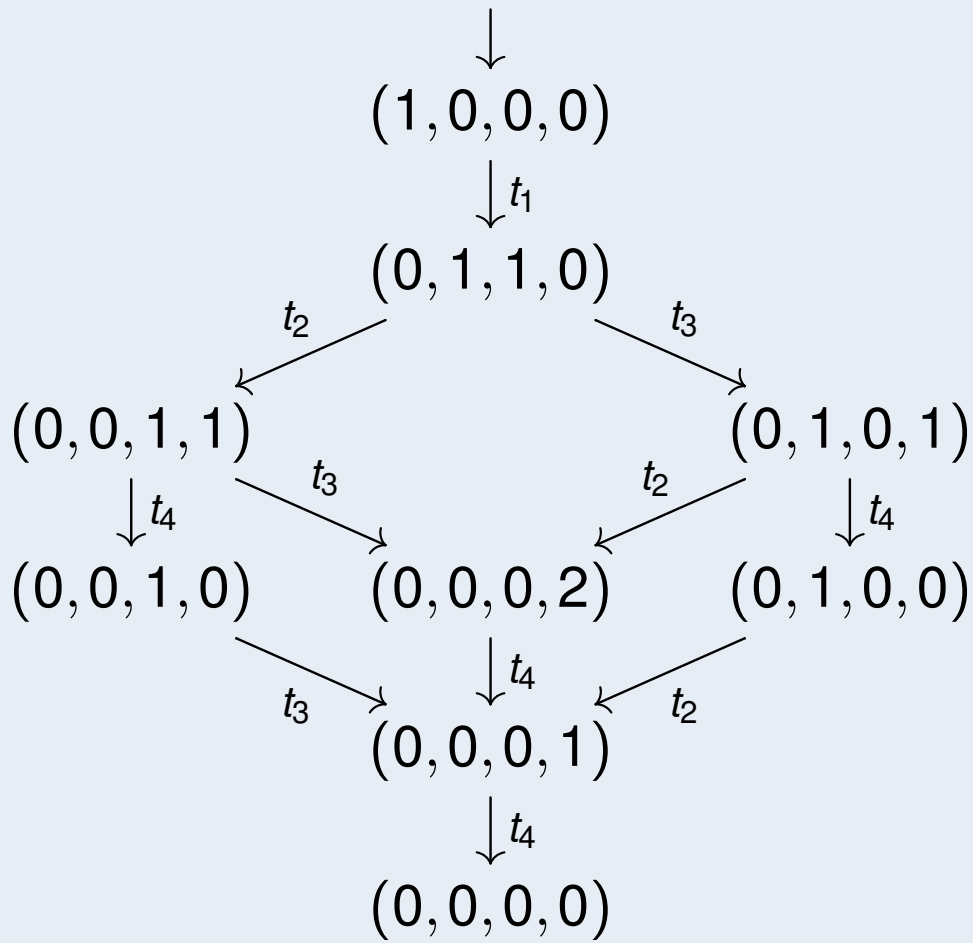
Beispiel für Kausalität:



- Hier ist t_1 eine notwendige Bedingung für t_4.
- Aber t_2 ist hier keine notwendige Bedingung für t_4. Denn nicht jede Schaltfolge, die zu t_4 führt, enthält t_2 (z.B. $\tilde{t} = t_1\, t_3$).

  Analoges gilt für t_3.

Erreichbarkeitsgraph dazu: (mit Stellenreihenfolge s_1, s_2, s_3, s_4)



Transitivität der Kausalität

Wenn t_1 eine notwendige Bedingung für t_2 ist, und t_2 eine notwendige Bedingung für t_3 ist, dann ist t_1 eine notwendige Bedingung für t_3.

Nebenläufigkeit (Definition)

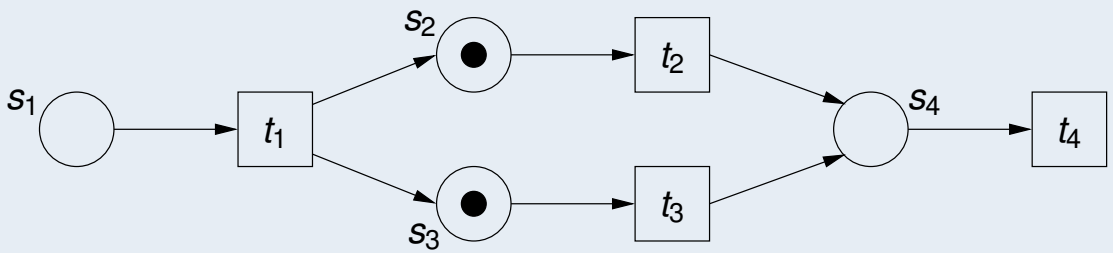
Die Transitionen $t_1, \ldots, t_n$ einer Menge $T' \subseteq T$ nennt man für die Markierung m nebenläufig aktiviert, wenn

$$\bullet t_1 \oplus \cdots \oplus \bullet t_n \leq m.$$

Das heißt, wenn die Markierung m genug Marken enthält, um alle Transitionen aus T' „gleichzeitig“ zu feuern.

Beobachtung: Wenn die Transitionen einer Menge T' für die Markierung m nebenläufig aktiviert sind, so ist dies auch für jede Teilmenge von T' der Fall.

Beispiele für Nebenläufigkeit:



Die Transitionen t_2 und t_3 sind für die hier gezeigte Markierung nebenläufig aktiviert.

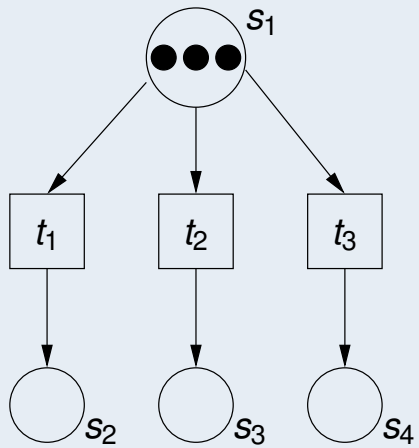
Denn:

$$\bullet t_2 = (0,1,0,0)$$

$$\bullet t_3 = (0,0,1,0)$$

$$\bullet t_2 \oplus \bullet t_3 \leq (0,1,1,0)$$

Petrinetze: Kausalität, Nebenläufigkeit, Konflikt

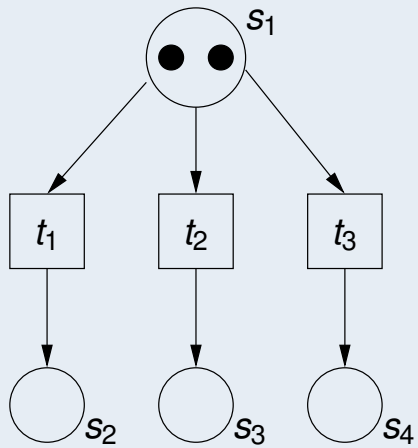


Die Transitionen t_1, t_2 und t_3 sind für die hier gezeigte Markierung nebenläufig aktiviert.

Denn:

$$\begin{aligned} {}^\bullet t_1 &= (1,0,0,0) \\ {}^\bullet t_2 &= (1,0,0,0) \\ {}^\bullet t_3 &= (1,0,0,0) \\ {}^\bullet t_1 \oplus {}^\bullet t_2 \oplus {}^\bullet t_3 &\leq (3,0,0,0) \end{aligned}$$

Petrinetze: Kausalität, Nebenläufigkeit, Konflikt



Die Transitionen der Mengen $\{t_1, t_2\}$, $\{t_2, t_3\}$ und $\{t_1, t_3\}$ sind für die hier gezeigte Markierung jeweils nebenläufig aktiviert. Dies gilt jedoch nicht für die Menge $\{t_1, t_2, t_3\}$ insgesamt.

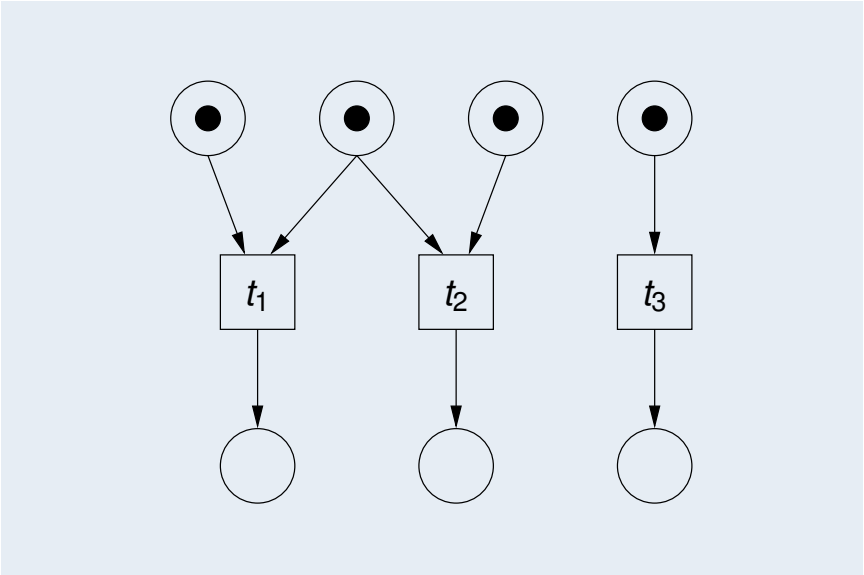
Denn:

$$ {}^\bullet t_1 = {}^\bullet t_2 = {}^\bullet t_3 = (1,0,0,0) $$

$$ {}^\bullet t_1 \oplus {}^\bullet t_2 = {}^\bullet t_2 \oplus {}^\bullet t_3 = {}^\bullet t_1 \oplus {}^\bullet t_3 \leq (2,0,0,0) $$

$$ {}^\bullet t_1 \oplus {}^\bullet t_2 \oplus {}^\bullet t_3 \not\leq (2,0,0,0) $$

Petrinetze: Kausalität, Nebenläufigkeit, Konflikt



Die Transitionen der Mengen $\{t_1, t_3\}$ und $\{t_2, t_3\}$ sind für die hier gezeigte Markierung jeweils nebenläufig aktiviert. Dies gilt jedoch nicht für die Mengen $\{t_1, t_2\}$ und $\{t_1, t_2, t_3\}$.

Dieses Beispiel zeigt auch, dass Nebenläufigkeit nicht transitiv ist: für die angegebene Markierung ist t_1 nebenläufig aktiviert zu t_3, und t_3 ist nebenläufig aktiviert zu t_2, jedoch sind t_1 und t_2 nicht nebenläufig aktiviert.

Konsequenzen von Nebenläufigkeit

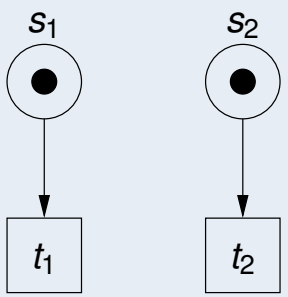
Wenn die Transitionen einer Menge T' für eine Markierung m nebenläufig aktiviert sind, so ist jede Anordnung dieser Transitionen eine Schaltfolge ausgehend von m.

Das heißt, für jede Sequenz $\tilde{t}$, in der jede Transition aus T' genau einmal vorkommt, gibt es eine Markierung m' mit $m\,[\tilde{t}\rangle\,m'$.

Und diese Markierung m' ist durch T' eindeutig bestimmt (also unabhängig von $\tilde{t}$).

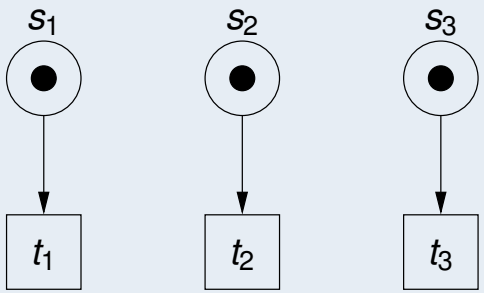
Nebenläufig aktivierte Transitionen führen daher in Erreichbarkeitsgraphen zu Strukturen, die die Form eines Quadrats (oft Diamond genannt) oder (höherdimensionalen) Würfels haben.

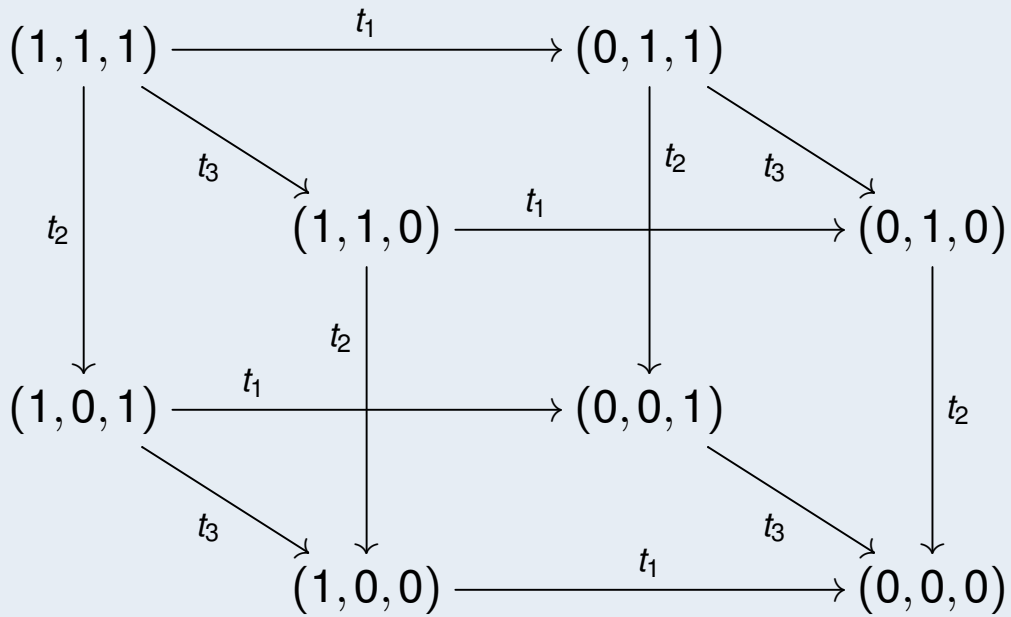
Beispiel für so ein Quadrat:



$$\begin{array}{ccc} (1,1) & \xrightarrow{t_1} & (0,1) \\ \downarrow{\scriptstyle t_2} & & \downarrow{\scriptstyle t_2} \\ (1,0) & \xrightarrow{t_1} & (0,0) \end{array}$$

Beispiel für Entstehen eines Würfels:

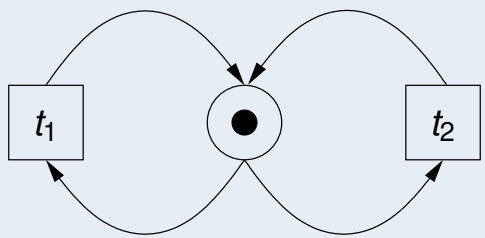




Frage: Wenn ausgehend von einer Markierung m jede Anordnung der Transitionen einer Menge T' eine Schaltfolge darstellt, sind dann die Transitionen aus T' für m nebenläufig aktiviert?

Nein, nicht unbedingt!

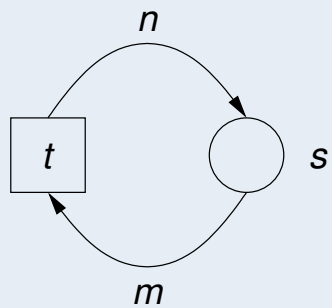
Gegenbeispiel:



Schlinge (Definition)

Eine Schlinge (oder Schleife) in einem Petrinetz besteht aus einer Transition t und einer Stelle s mit ${}^\bullet t(s) > 0$ und $t^\bullet(s) > 0$.

Grafisch:



Für schlingenfreie Petrinetze gilt:

Seien eine Markierung m und eine Menge T' von Transitionen gegeben, so dass jede Anordnung dieser Transitionen von m ausgehend schaltbar ist. Dann sind die Transitionen aus T' für m nebenläufig aktiviert.

Konflikt (Definition)

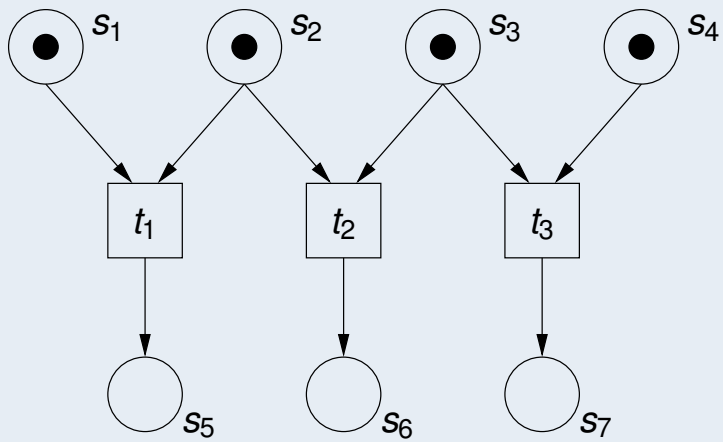
Zwei verschiedene Transitionen $t, t' \in T$ stehen für die Markierung m in Konflikt genau dann, wenn gilt:

- t und t' sind beide für m aktiviert und
- t und t' sind für m nicht nebenläufig aktiviert.

Anschaulich: Jede einzelne der beiden Transitionen könnte schalten, aber nicht tatsächlich beide „gleichzeitig“.

Das liegt immer daran, dass sie eine gemeinsame Stelle in den Vorbedingungen haben. Das heißt, es gibt eine Stelle s mit ${}^\bullet t(s) \geq 1$ und ${}^\bullet t'(s) \geq 1$.

Petrinetze: Kausalität, Nebenläufigkeit, Konflikt



Für die hier gezeigte Markierung steht t_1 in Konflikt mit t_2.

Denn:

$$(1,1,0,0,0,0,0) \leq (1,1,1,1,0,0,0)$$

$$(0,1,1,0,0,0,0) \leq (1,1,1,1,0,0,0)$$

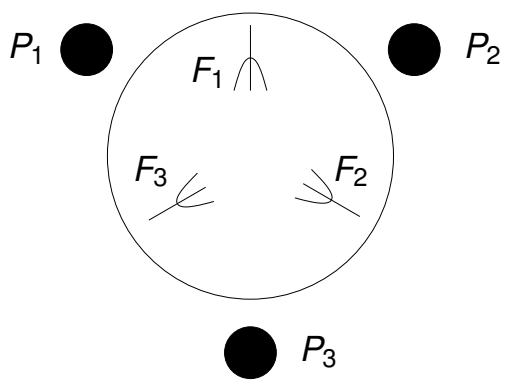
$$(1,1,0,0,0,0,0) \oplus (0,1,1,0,0,0,0) \not\leq (1,1,1,1,0,0,0)$$

Außerdem steht t_2 in Konflikt mit t_3. Jedoch steht t_1 nicht in Konflikt mit t_3 (keine Transitivität der Konfliktrelation).

Beispiel: Dining Philosophers

Wir betrachten nochmals das Beispiel der Dining Philosophers (speisende Philosophen):

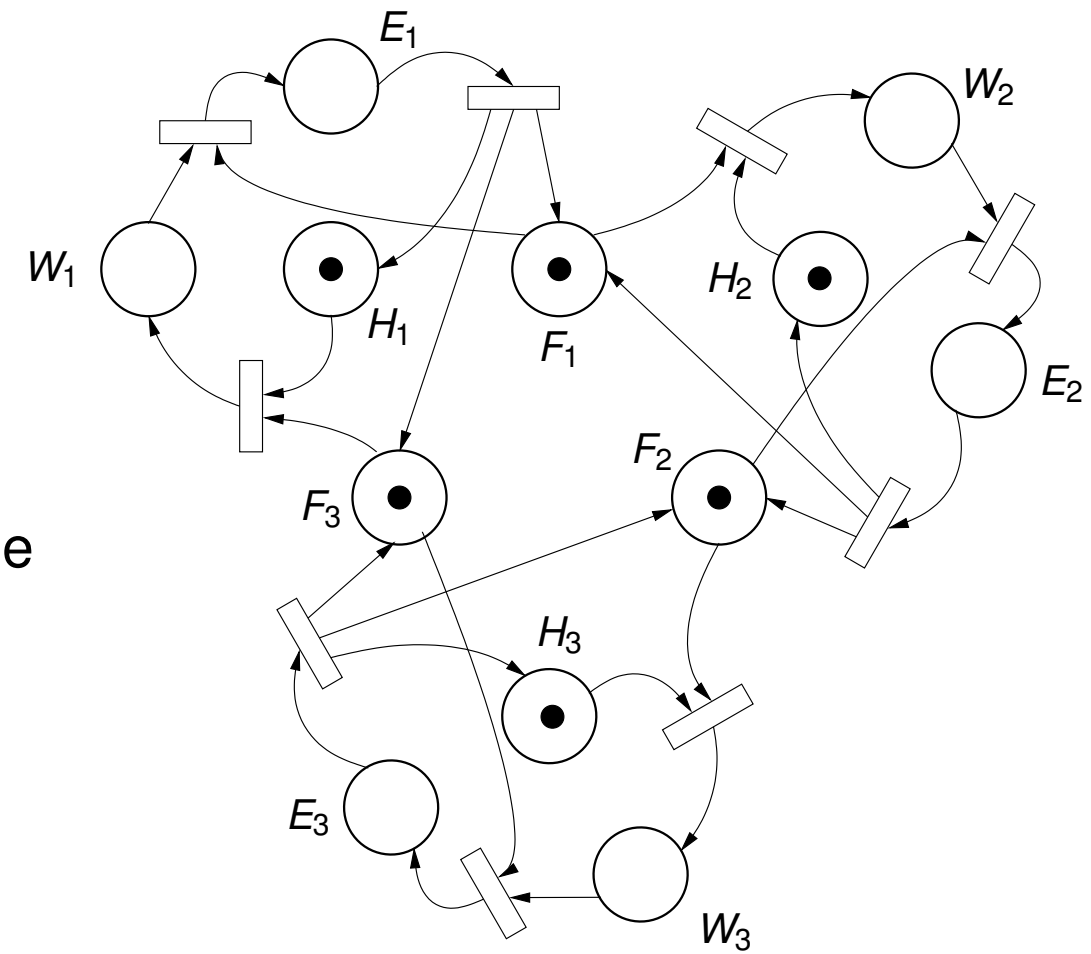
- Es sitzen drei Philosophen P_i um einen runden Tisch, zwischen je zwei Philosophen liegt eine Gabel (fork) F_i.
- Philosophen werden von Zeit zu Zeit hungrig (H_i) und benötigen dann zum Essen (E_i) beide benachbarte Gabeln.
- Jeder Philosoph nimmt zu einem beliebigen Zeitpunkt beide Gabeln nacheinander auf (die rechte zuerst), isst und legt anschließend beide Gabeln wieder zurück (gleichzeitig).



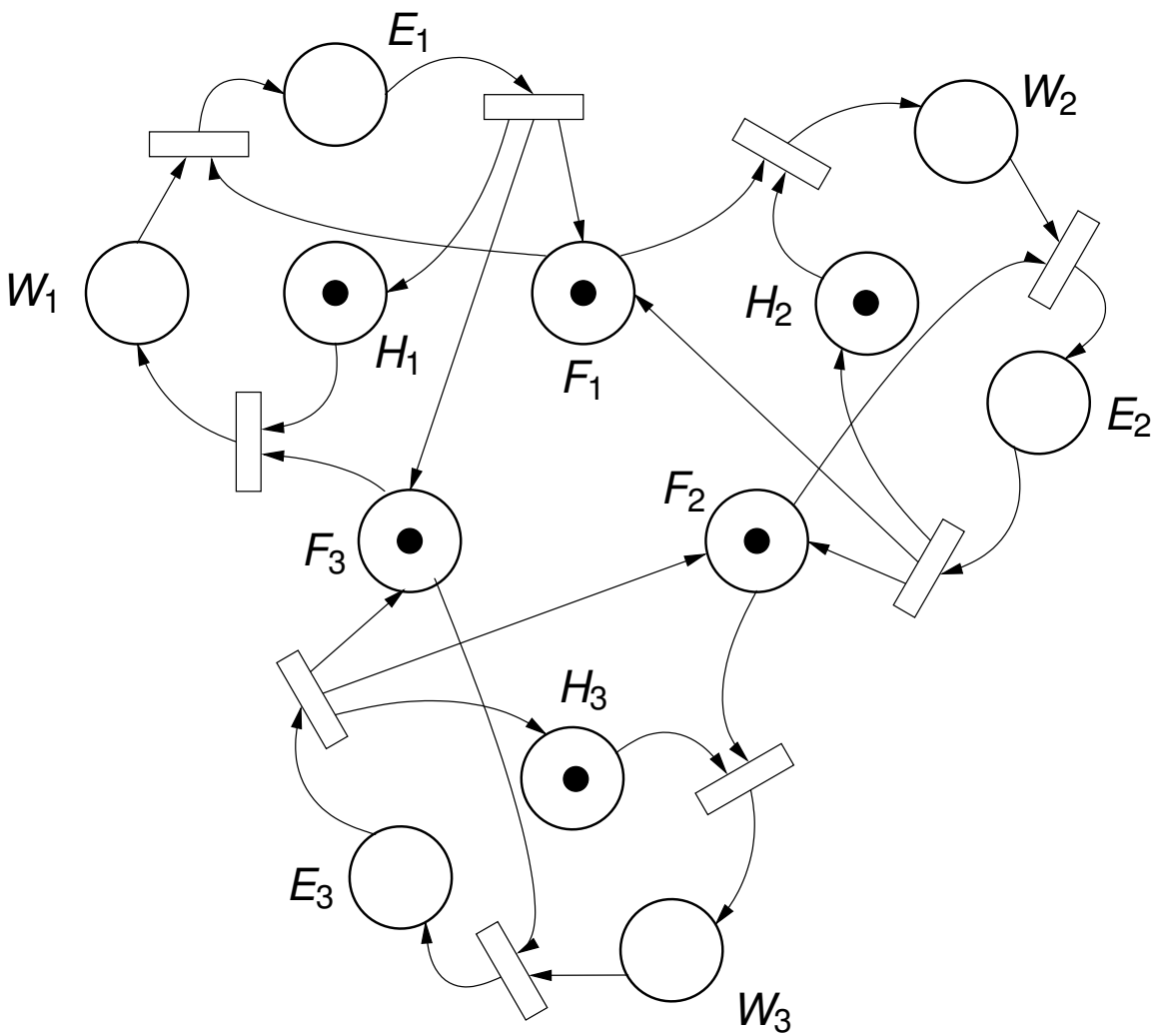
Beispiel: Dining Philosophers

Modellierung als Petrinetz:

- Marke bei H_i bedeutet, Philosoph P_i ist hungrig.
- Marke bei W_i bedeutet, Philosoph P_i wartet auf linke Gabel.
- Marke bei F_i bedeutet, die Gabel F_i liegt auf dem Tisch.
- Marke bei E_i bedeutet, Philosoph P_i isst.

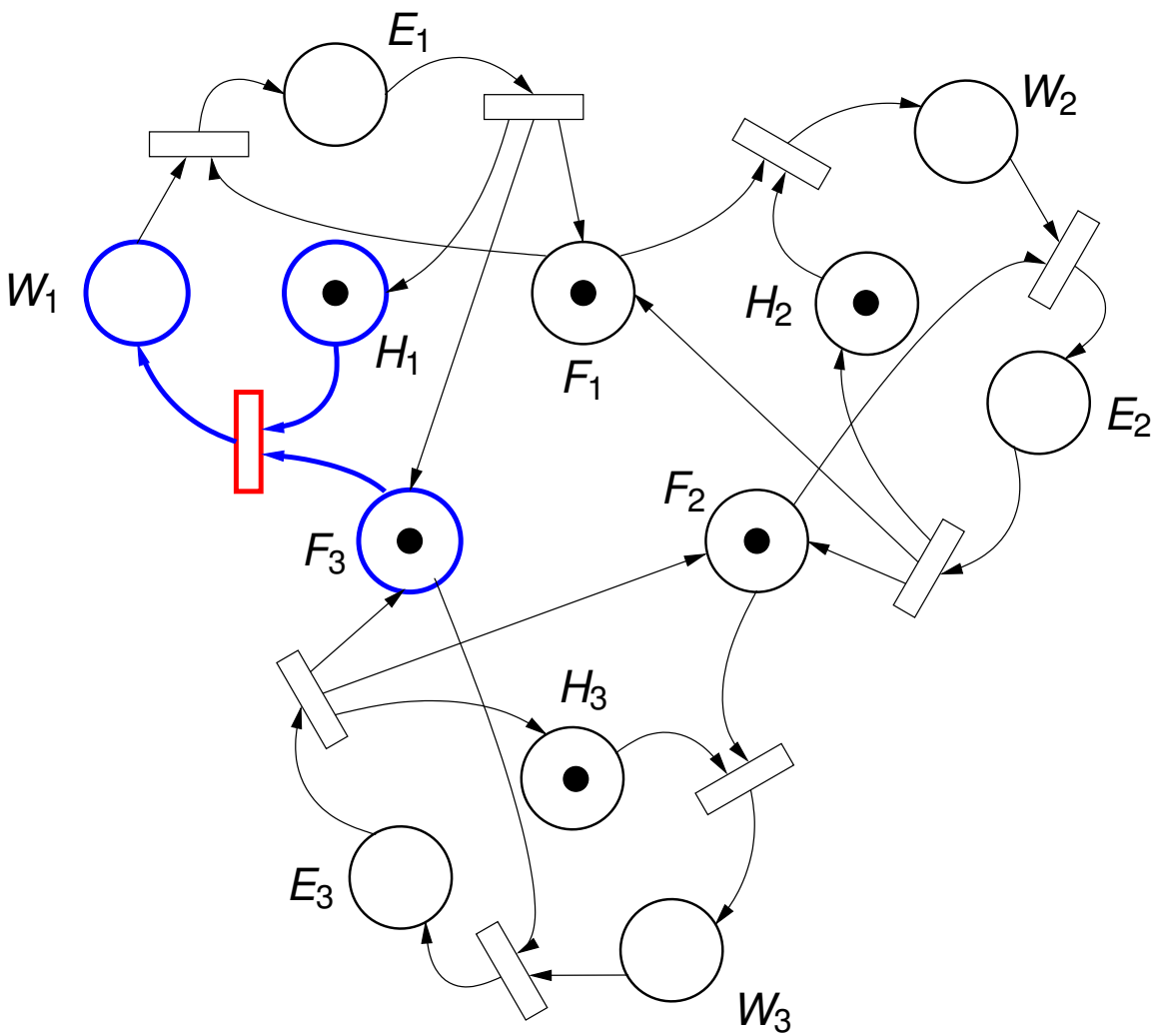


Beispiel: Dining Philosophers



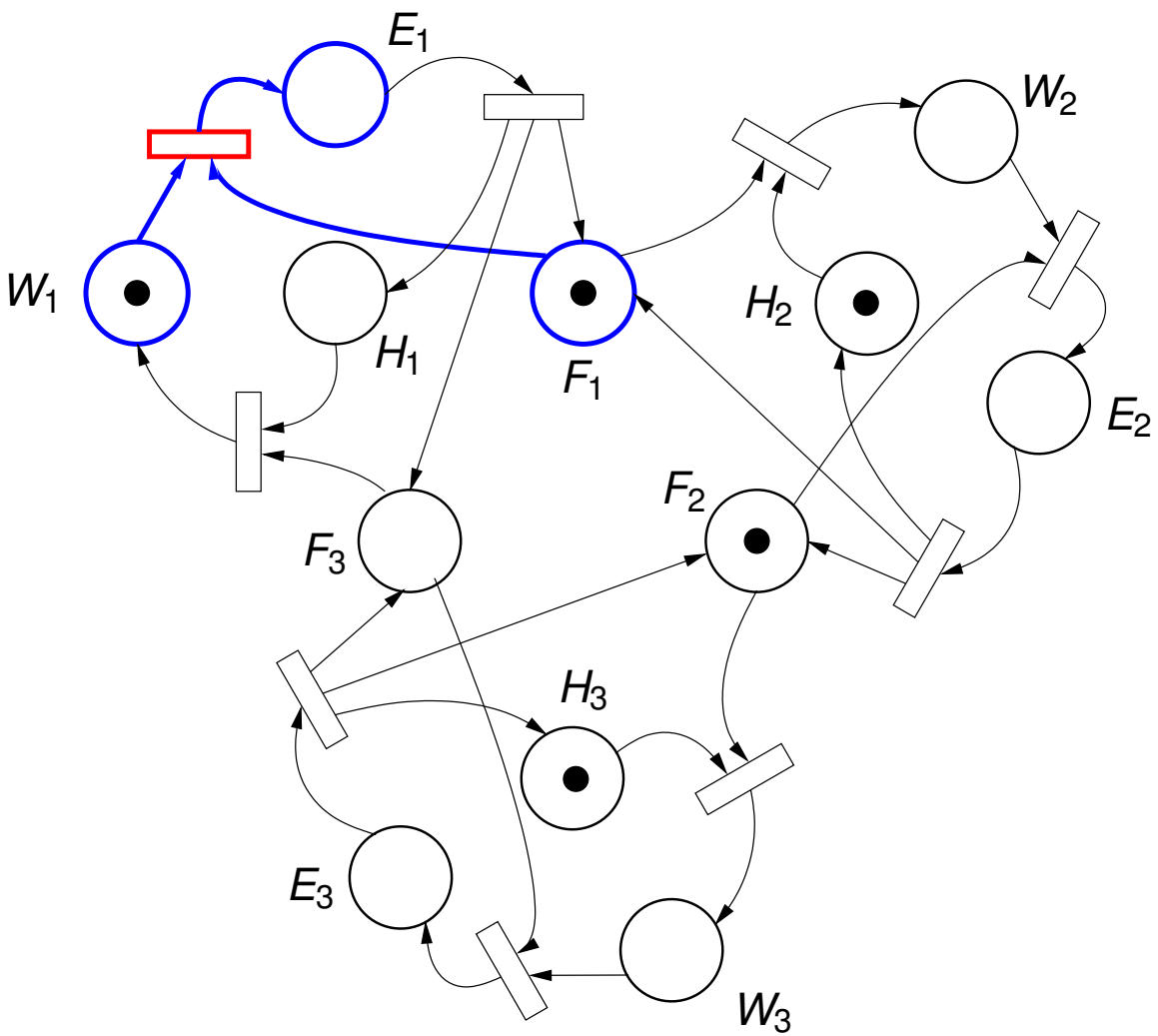
- Der erste Philosoph P_1 möchte essen.

Beispiel: Dining Philosophers



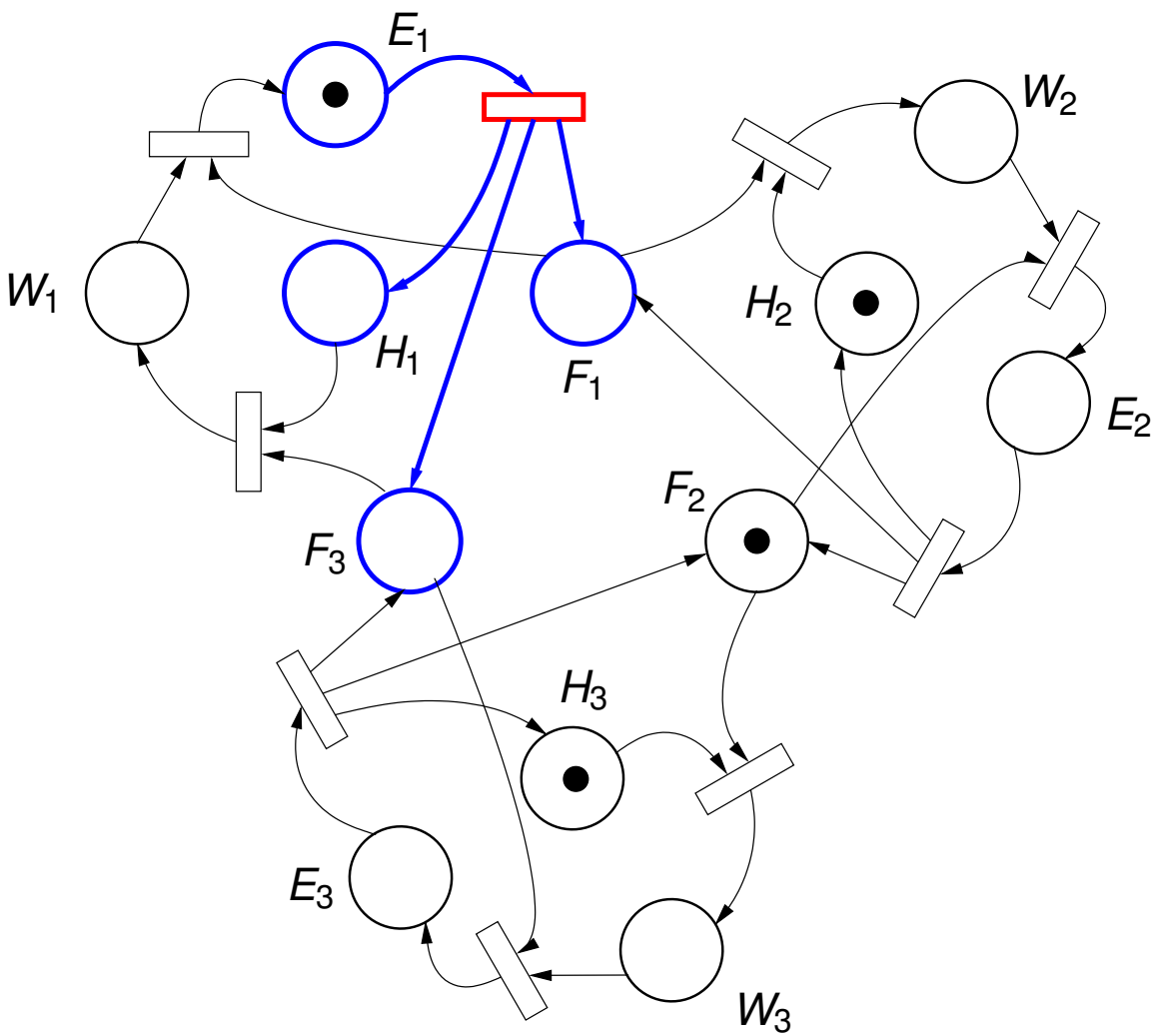
- P_1 nimmt die rechte Gabel F_3.

Beispiel: Dining Philosophers



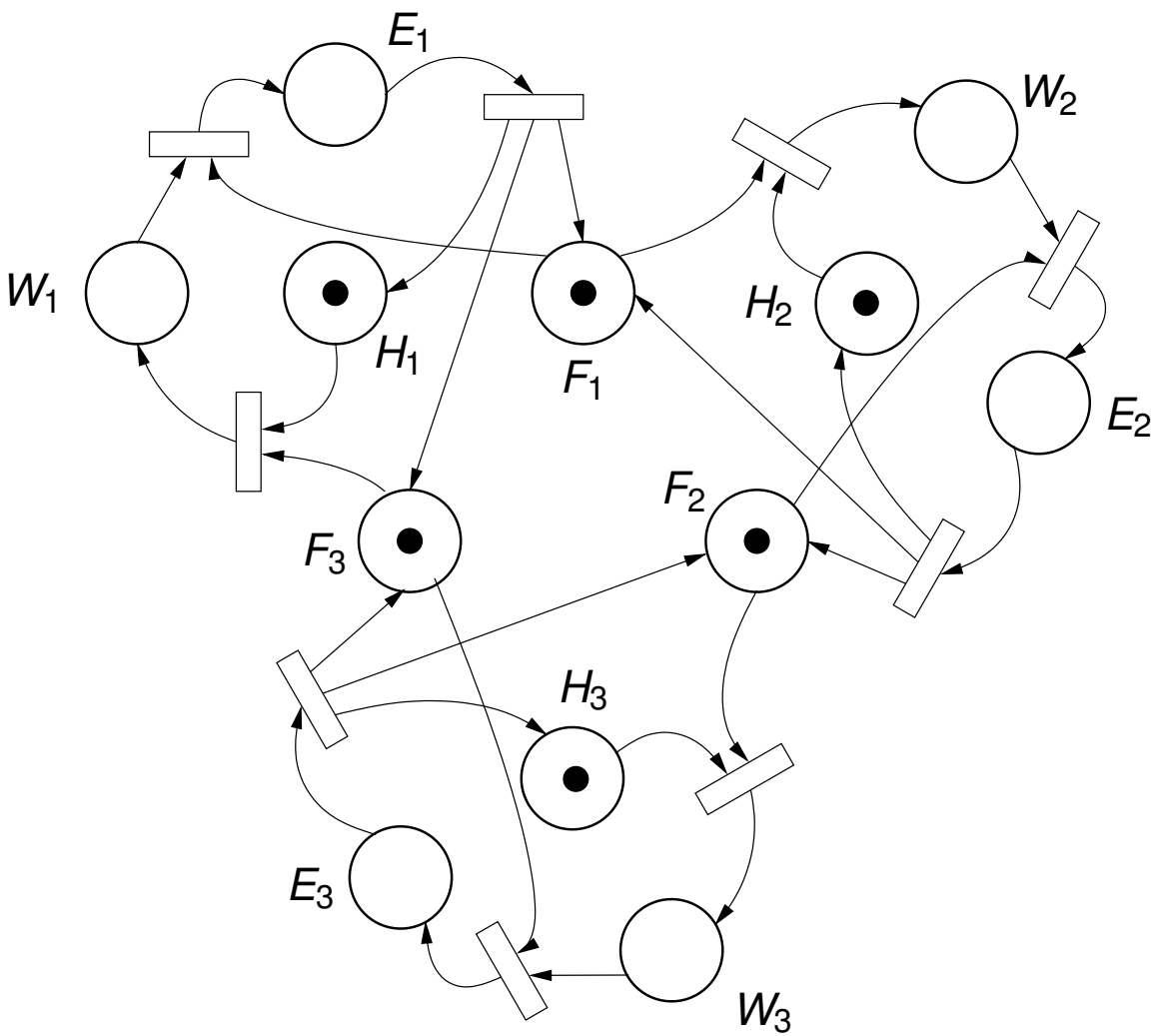
- P_1 nimmt die linke Gabel F_1 und isst.

Beispiel: Dining Philosophers



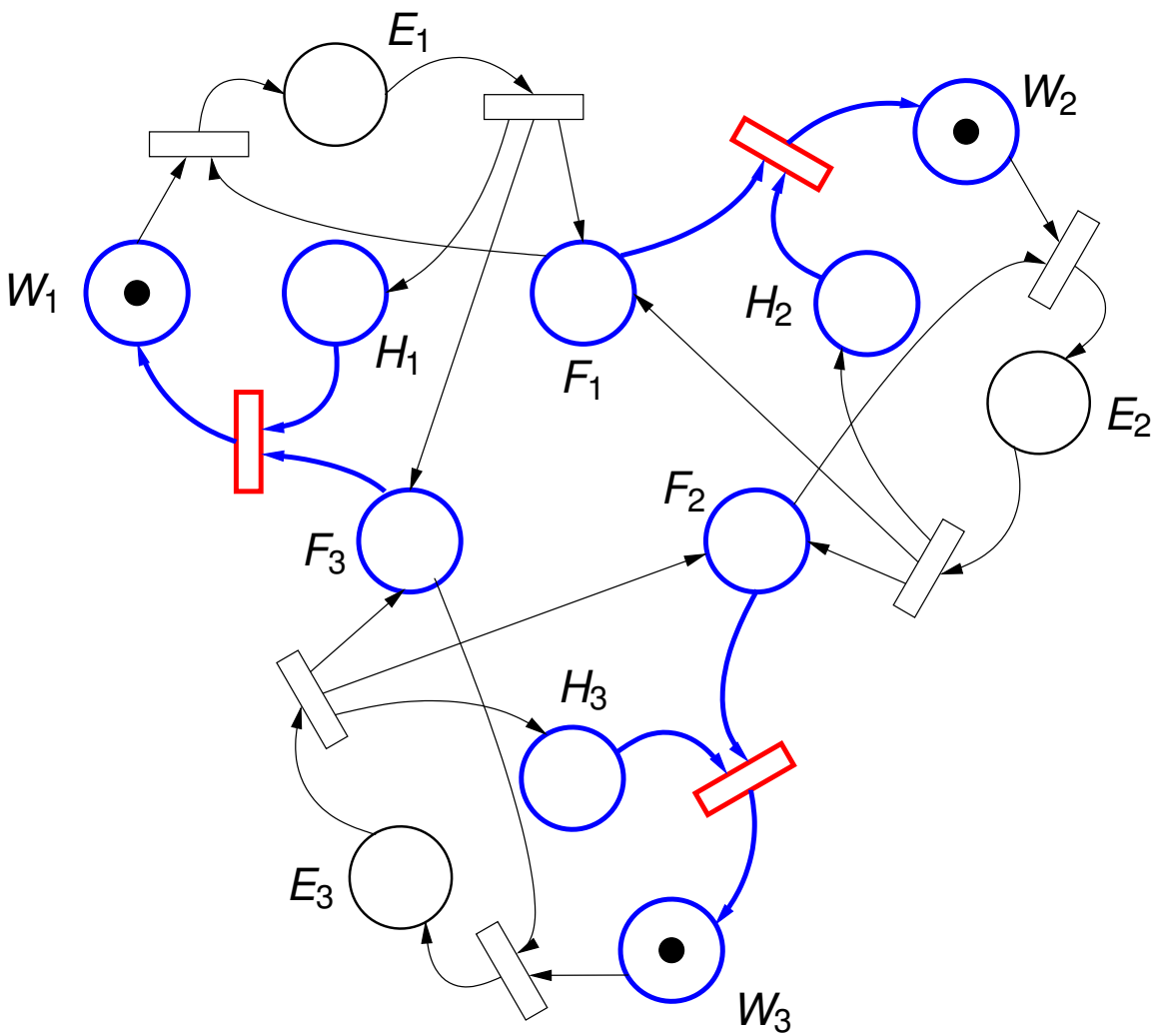
- P_1 legt Gabeln F_3 und F_1 zurück.

Beispiel: Dining Philosophers



- Gibt es Möglichkeit der Verklemmung (Deadlock)?

Beispiel: Dining Philosophers



- Ja! Nachdem alle nebenläufig ihre rechte Gabel nehmen.

Petrinetze: Wechselseitiger Ausschluss

Wir betrachten zum Abschluss des Petrinetz-Teils der Vorlesung noch einige „Fallstudien“, also Beispiele, an denen typische Szenarien und spezielle Modellierungs-„Muster“ nochmals deutlich werden . . .

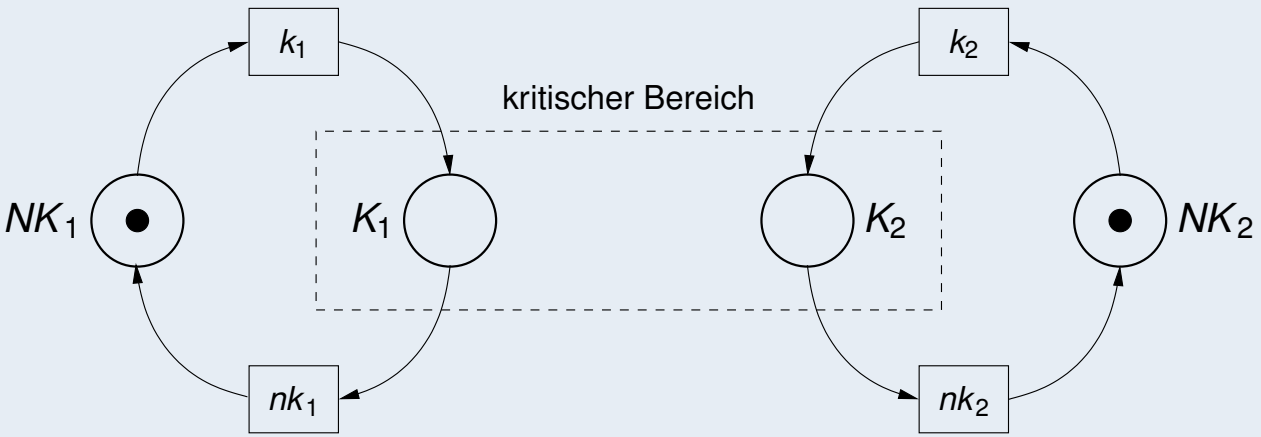
Zunächst behandeln wir das Konzept des wechselseitigen Ausschlusses (engl. mutual exclusion).

- Wir betrachten zwei Akteure, die jeweils einen kritischen Bereich haben.
- Beide Akteure dürfen nicht gleichzeitig in ihren kritischen Bereich kommen, da sie sich dort gegenseitig behindern und unerwünschtes Verhalten auslösen würden (z.B. indem beide Akteure in dieselbe Datei schreiben).

  Es darf sich also immer höchstens ein Akteur im kritischen Bereich befinden.

Petrinetze: Wechselseitiger Ausschluss

Ursprüngliches System:



Bedeutung der Stellen:

K_1: kritischer Bereich Akteur 1

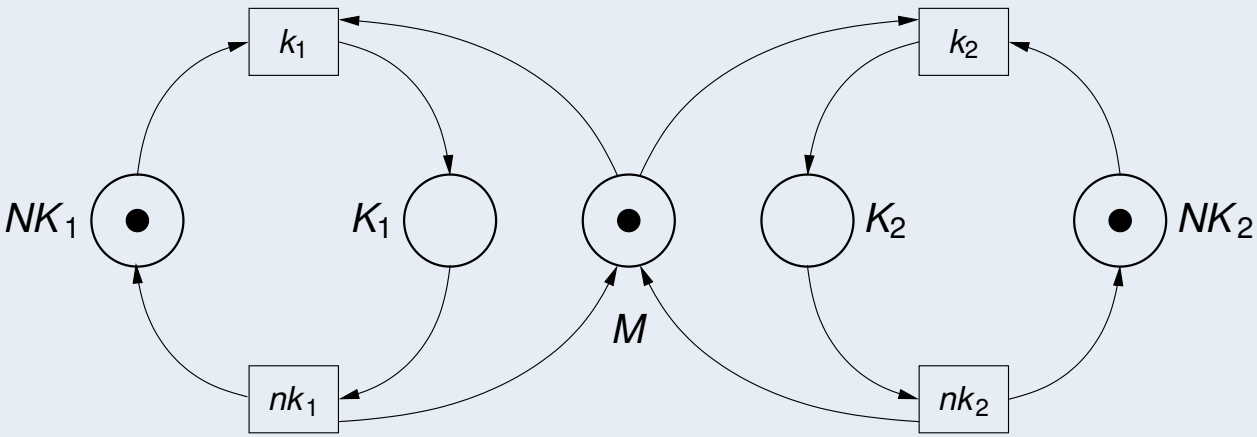
NK_1: nicht-kritischer Bereich Akteur 1

K_2: kritischer Bereich Akteur 2

NK_2: nicht-kritischer Bereich Akteur 2

Petrinetze: Wechselseitiger Ausschluss

Erweitertes System mit Synchronisation:



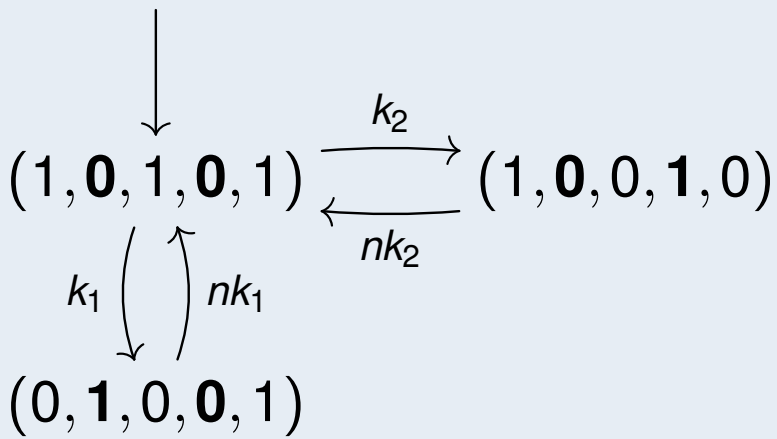
Bedeutung der Stellen:

- K_1: kritischer Bereich Akteur 1
- NK_1: nicht-kritischer Bereich Akteur 1
- K_2: kritischer Bereich Akteur 2
- NK_2: nicht-kritischer Bereich Akteur 2
- M: Hilfsstelle, sogenannter Mutex

Petrinetze: Wechselseitiger Ausschluss

Wir möchten zeigen, dass in den Stellen K_1, K_2 niemals gleichzeitig Marken liegen.

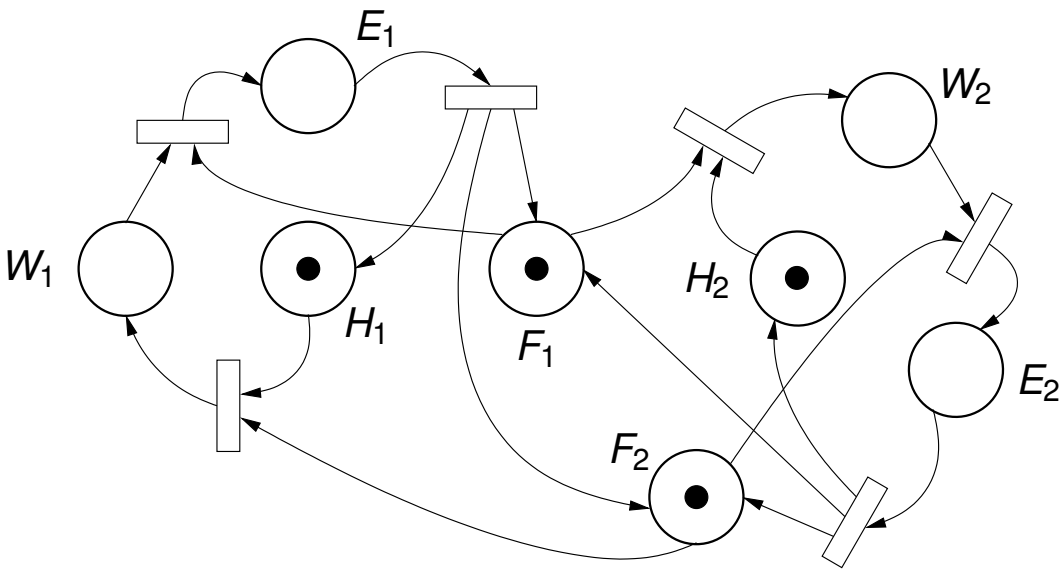
Erreichbarkeitsgraph:

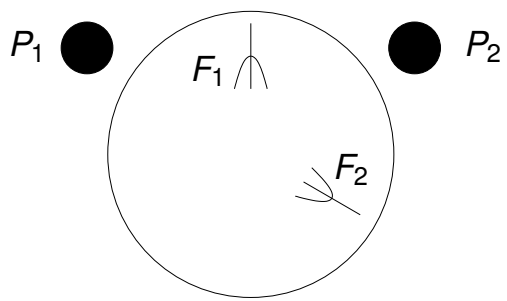


Stellenreihenfolge: NK_1, K_1, M, K_2, NK_2

Petrinetze: Speisende Philosophen

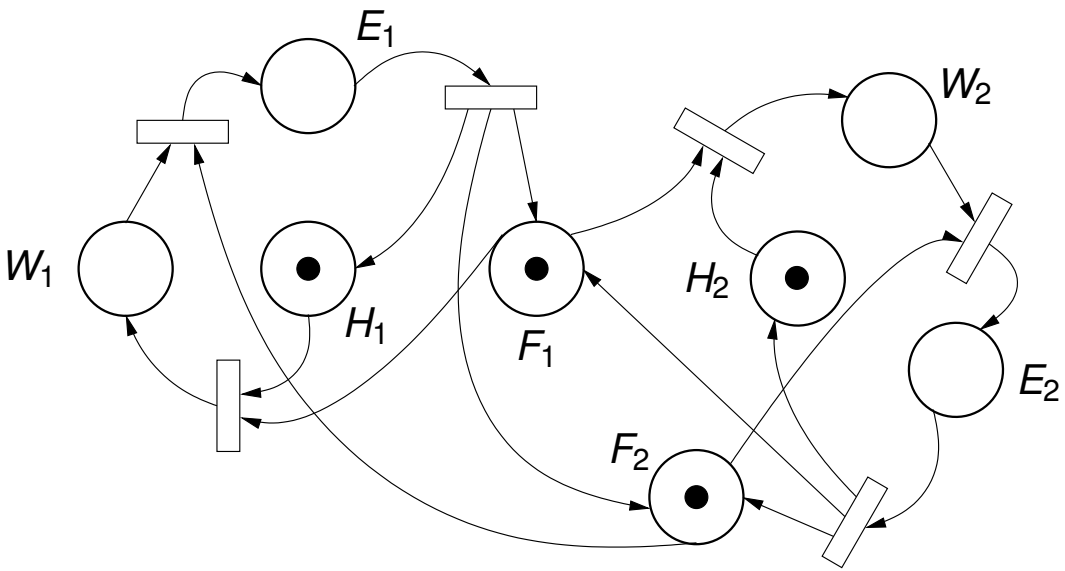
Wir kommen wieder auf die speisenden Philosophen zurück, aber schicken den dritten Philosophen nach Hause:



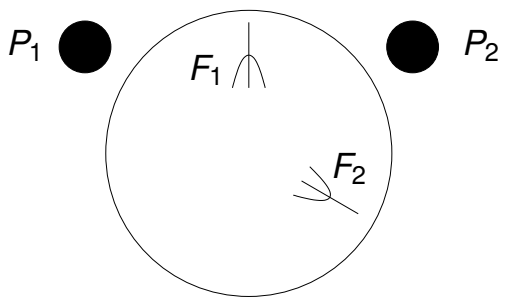


Petrinetze: Speisende Philosophen

Wir kommen wieder auf die speisenden Philosophen zurück, aber schicken den dritten Philosophen nach Hause:



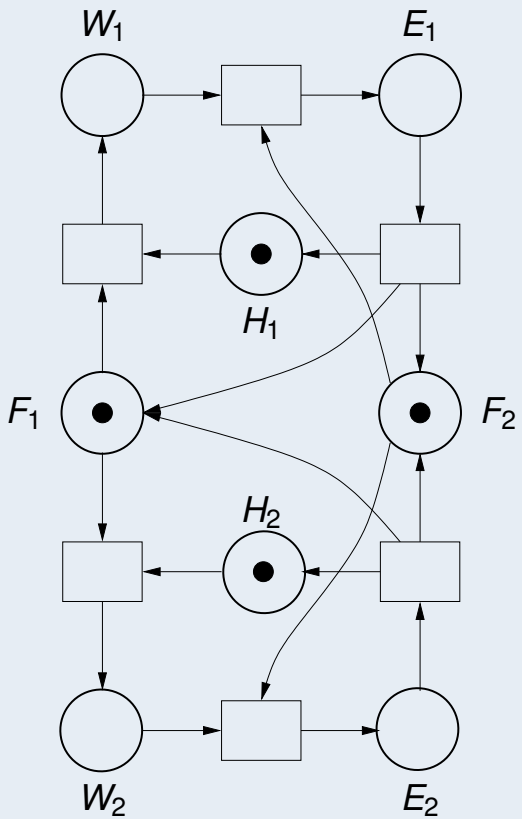
Außerdem machen wir den ersten Philosophen zum Linkshänder (er nimmt die linke Gabel zuerst).



Petrinetze: Speisende Philosophen

Anders dargestellt:

Links- und rechtshändige Philosophen



Verklemmungsfrei durch Synchronisation!

Petrinetze: Kapazitätsbegrenzung

Ein anderes Erfordernis, das wir hin und wieder mal ausdrücken wollen, ist die Begrenzung der Kapazität einzelner Stellen im Petrinetz.

Eine Möglichkeit zum Umgang damit ist die Einführung einer speziellen Art von Petrinetzen:

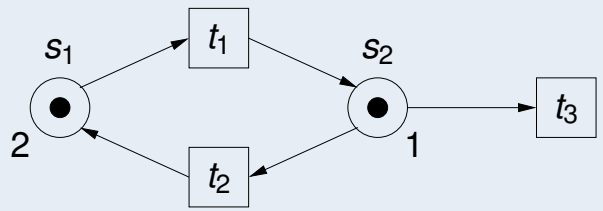
Petrinetz mit Kapazitäten (Definition)

Ein Petrinetz mit Kapazitäten besteht aus einem (herkömmlichen) Petrinetz, mit Stellenmenge S, und einer Kapazitätsfunktion $k : S \to \mathbb{N}$. Für die Anfangsmarkierung m_0 muss gelten: $m_0 \leq k$.

Intuition: Jede Stelle s darf höchstens $k(s)$ Marken enthalten (jemals).

In der grafischen Darstellung werden die Kapazitäten an die Stellen geschrieben.

Beispielnetz mit Kapazitäten



$k(s_1) = 2, k(s_2) = 1$

Natürlich muss die Semantik angepasst werden:

Aktivierung/Schalten bei Petrinetzen mit Kapazitäten (Definition)

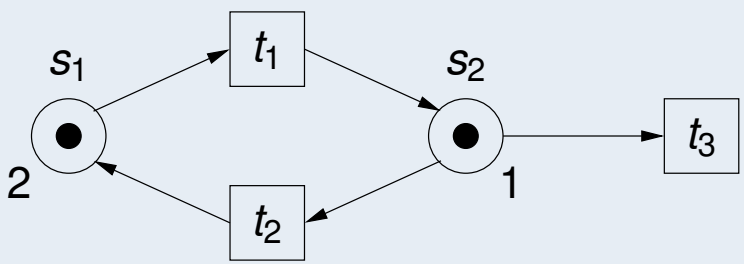
Eine Transition t ist für eine Markierung m aktiviert, wenn gilt:

1. ${}^\bullet t \leq m$
2. und $m \ominus {}^\bullet t \oplus t^\bullet \leq k$.

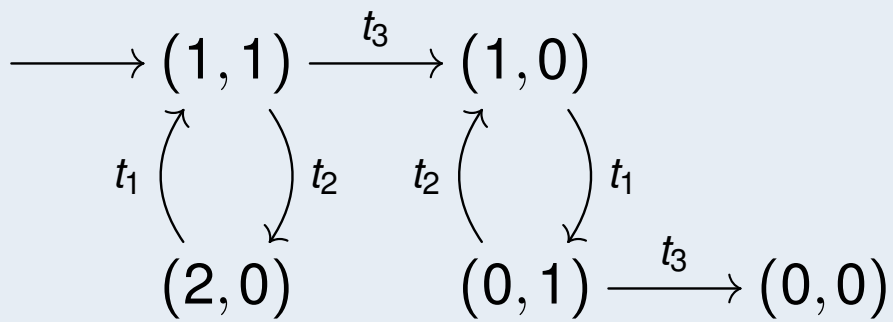
Das heißt, eine Transition darf nur dann schalten, wenn dadurch die Kapazitäten nicht überschritten werden.

Petrinetze: Kapazitätsbegrenzung

Beispielnetz mit Kapazitäten



Erreichbarkeitsgraph



Stellenreihenfolge: s_1, s_2

Insbesondere: Für die Anfangsmarkierung $(1,1)$ ist die Transition t_1 nicht aktiviert.

Es geht jedoch auch ohne Einführung einer neuen Petrinetz-Art!

Umwandlung eines Petrinetzes mit Kapazitäten in eines ohne

1. Füge zu jeder Stelle s eine sogenannte Komplementstelle $\overline{s}$ hinzu. In der neuen Anfangsmarkierung enthält $\overline{s}$ genau $k(s) - m_0(s)$ Marken.

   Idee: Die Summe der Marken in der Stelle und der Komplementstelle ergibt immer die gewünschte Kapazität.

2. Falls eine Transition t insgesamt Marken aus einer Stelle s herausnimmt, $n = t^\bullet(s) - {}^\bullet t(s) < 0$, füge eine Kante von t nach $\overline{s}$ mit Gewicht $-n$ ein.

3. Falls eine Transition t insgesamt Marken in eine Stelle s hineinlegt, $n = t^\bullet(s) - {}^\bullet t(s) > 0$, füge eine Kante von $\overline{s}$ nach t mit Gewicht n ein.

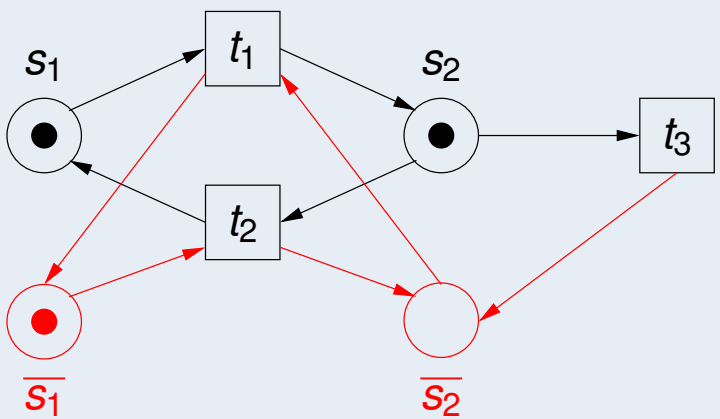
Petrinetze: Kapazitätsbegrenzung

Mit dieser Konstruktion ist für jede erreichbare Markierung m sichergestellt, dass:

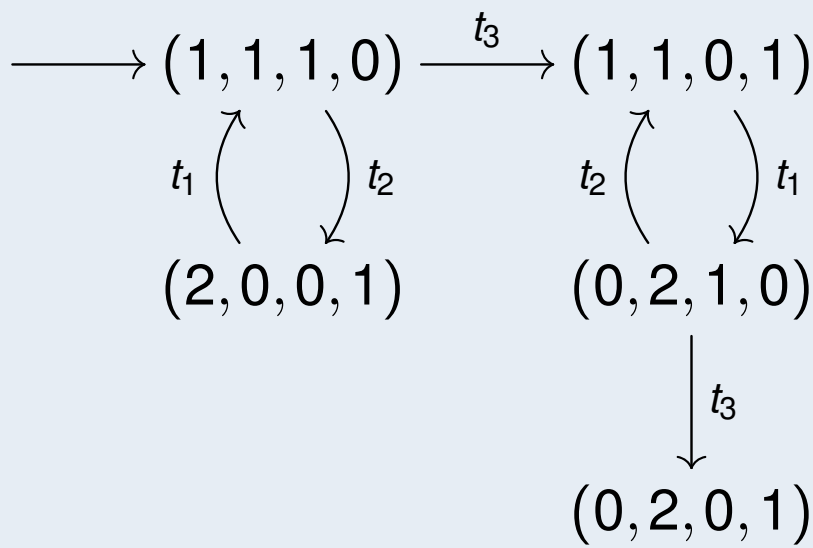
1. $m(s) + m(\overline{s}) = k(s)$ für jedes Paar $s, \overline{s}$;
2. eine Transition t nur schaltbar ist, wenn die Kapazitäten der Stellen in der Nachbedingung noch nicht ausgeschöpft sind. Das wird dadurch überprüft, dass die benötigten Restkapazitäten über die Komplementstellen abgefragt werden.

Petrinetze: Kapazitätsbegrenzung

Umgewandeltes Beispielnetz



Erreichbarkeitsgraph



Stellenreihenfolge: $s_1, \overline{s_1}, s_2, \overline{s_2}$

von Petrinetzen (zurück) zu UML

Bisher haben wir uns hinsichtlich dynamischer Modellierung mit flachen Zustandsdiagrammen (ganz am Anfang) und Petrinetzen (sowie deren Erreichbarkeitsgraphen und verwandten Konzepten) beschäftigt.

Dabei handelte es sich um Modellierung von

- dynamischen Aspekten,
- in qualitativer und quantitativer Hinsicht,
- einer breiten Klasse von Systemen,
- unter „white box“-Sicht,
- mit formaler Syntax und Semantik.

UML dagegen haben wir bisher explizit nur für statische Modellierung benutzt.

Aktivitätsdiagramme

Im Folgenden betrachten wir Aktivitätsdiagramme (activity diagrams). Das sind UML-Diagramme, mit denen man Ablaufpläne, Reihenfolgen von Aktionen, parallele Aktionen, etc. modellieren kann.

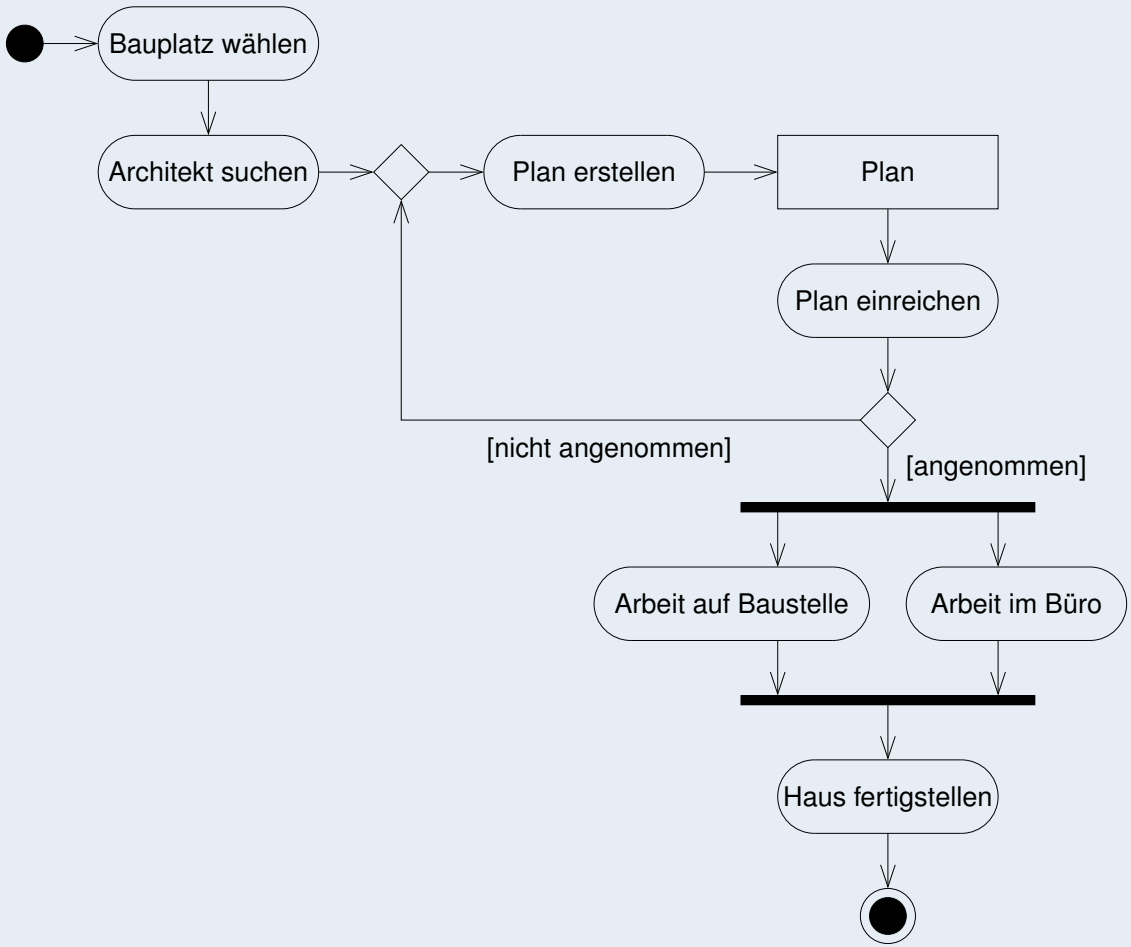
Sie werden beispielsweise verwendet, um Geschäftsprozesse (auch Workflow-Prozesse) zu modellieren. Sie können ebenso eingesetzt werden, um Use Cases oder interne Systemprozesse zu beschreiben. Generell: wann immer man Einzelschritte hat, die sich auf gewisse typische Arten zu Gesamtabläufen zusammenfügen.

Aktivitätsdiagramme sind in vielen Aspekten ähnlich zu Petrinetzen. Im Vergleich zu Petrinetzen bieten sie zusätzliche Modellierungsmöglichkeiten, haben jedoch keine vollständige formale Semantik.

Aktivitätsdiagramme

Aktivitätsdiagramme: Beispiel

Beispiel: Hausbau

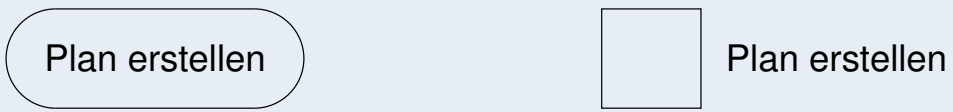


Aktivitätsdiagramme: Elemente

Wir vergleichen im Folgenden Aktivitätsdiagramme und ihre Bestandteile mit Petrinetzen (angelehnt an eine von Störrle aufgestellte Semantik für Teile von Aktivitätsdiagrammen).

Aktionen

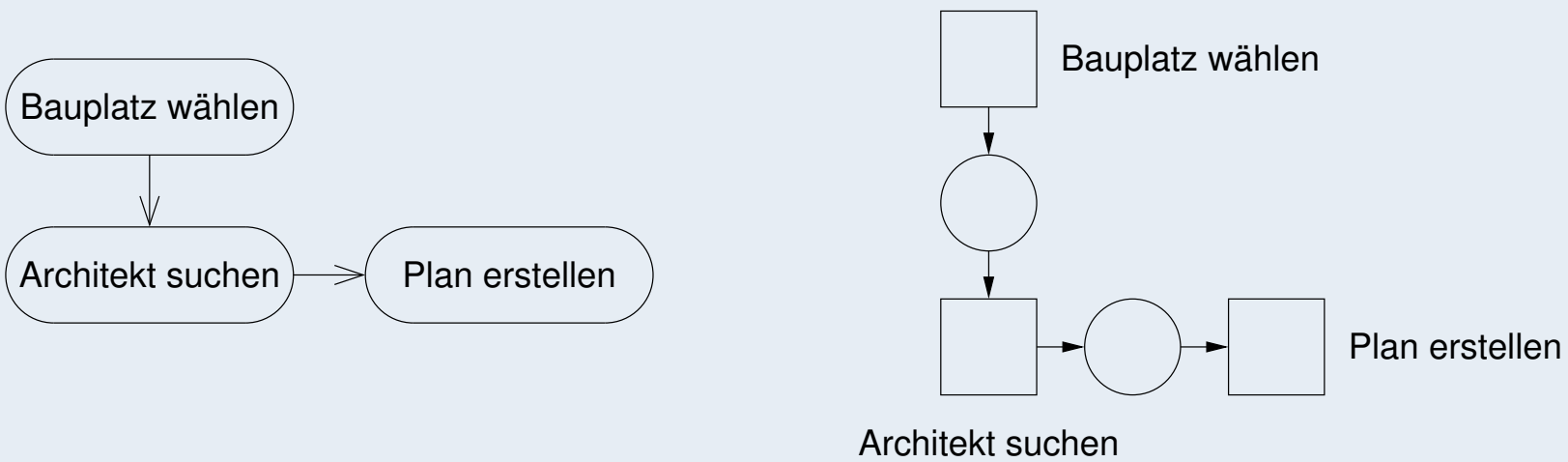
Eine Aktion im Aktivitätsdiagramm wird durch ein Rechteck mit abgerundeten Ecken dargestellt. Es entspricht einer (benannten) Transition eines Petrinetzes.



Intuition: Aktionen stehen für Tätigkeiten, mit Zeitaufwand.

Kontroll- oder Objektfluss

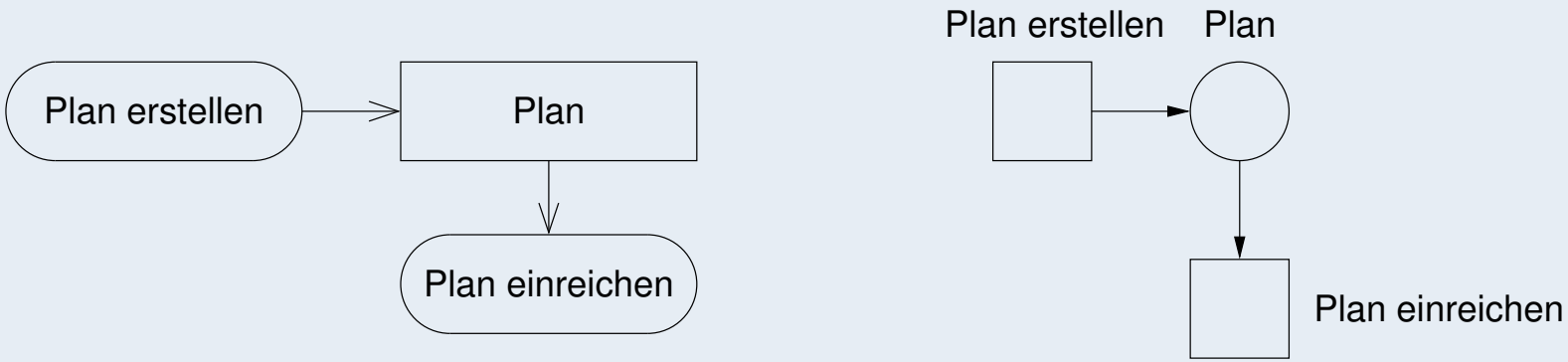
Der Kontroll- oder Objektfluss zwischen Aktionen (sowie Objektknoten) im Aktivitätsdiagramm, dargestellt durch die Kanten/Pfeile dazwischen, wird in dem entsprechenden Petrinetz mittels Hilfsstellen umgesetzt.



Aktivitätsdiagramme: Elemente

Objektknoten

Objektknoten im Aktivitätsdiagramm beschreiben Speicher für die Ablage und Übergabe von konkreten Objekten. Sie entsprechen „normalen“ Stellen im Petrinetz, also solchen, die in irgendeiner Weise Ressourcen abbilden (und einen sinnvollen Namen haben).

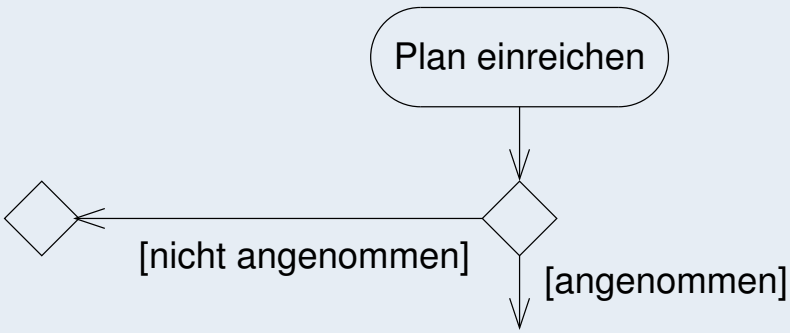


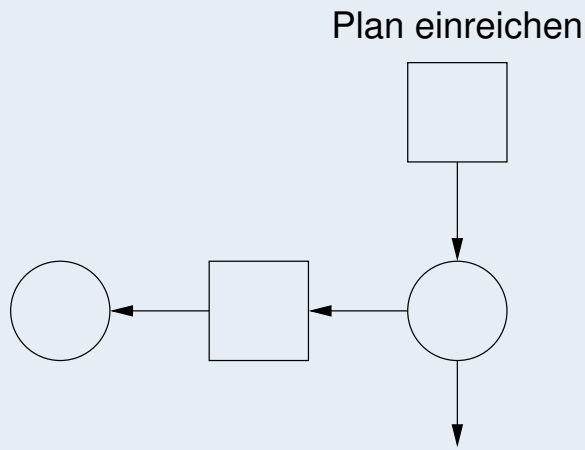
Bemerkung: Objektknoten sind bei Softwaremodellierung in der Regel mit einem Klassennamen beschriftet. Sie können dann (mehrere) Ausprägungen/Instanzen dieser Klasse aufnehmen.

Aktivitätsdiagramme: Elemente

Verzweigungsknoten

Verzweigungsknoten (decision nodes) im Aktivitätsdiagramm beschreiben eine Verzweigung des Kontroll- oder Objektflusses, wobei aus den alternativen Wegen jedes Mal genau einer ausgewählt wird. Sie werden im Petrinetz mittels Hilfsstellen umgesetzt, die sich in der Vorbedingung mehrerer Transitionen befinden.





Bei Bedarf (etwa bei nachfolgenden Verzweigungs- oder Objektknoten) müssen Hilfstransitionen eingeführt werden.

Aktivitätsdiagramme: Elemente

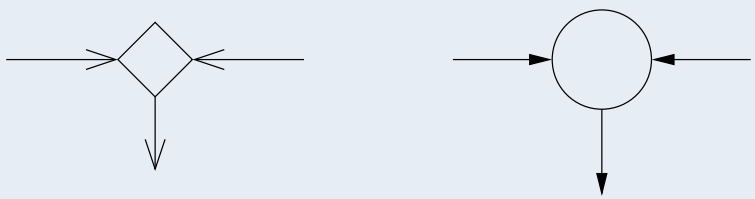
Anmerkungen:

- Wesentlicher Grund für die Einführung von Hilfsstellen und Hilfstransitionen bei der Übersetzung in Petrinetze sind die strukturellen Erfordernisse für ein korrektes Petrinetz (nämlich abwechselndes Auftreten von Stellen und Transitionen).
- Der Kontroll- oder Objektfluss an Verzweigungsknoten wird durch Überwachungsbedingungen (Guards) gesteuert. Sie werden in eckigen Klammern an den ausgehenden Wegen notiert.
- Die Bedingungen dürfen sich logisch nicht überlappen, und sollten insgesamt alle möglichen Fälle abdecken.
- Während Aktionen einen Zeitaufwand haben, werden Kontrollelemente wie Verzweigungsknoten (und weitere gleich gezeigte) als instantan durchlaufen angesehen.

Aktivitätsdiagramme: Elemente

Verbindungsknoten

Es gibt auch Verbindungsknoten (merge nodes), die alternative Kontroll- oder Objektflüsse zusammenführen. Sie werden im entsprechenden Petrinetz mittels Hilfsstellen umgesetzt, die sich in der Nachbedingung mehrerer Transitionen (gegebenenfalls Hilfstransitionen) befinden.

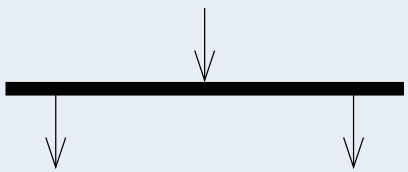


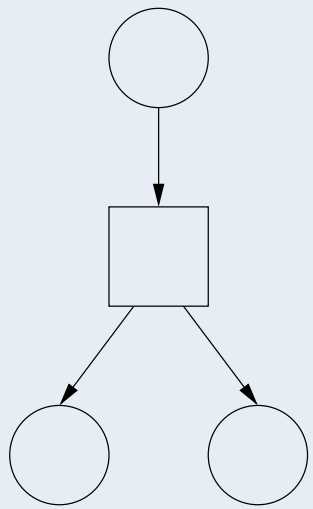
Es gibt auch Knoten (mit gleicher Darstellung), die sowohl Verbindungs- als auch Verzweigungsknoten sind, also mehrere eingehende und mehrere ausgehende Wege haben.

Aktivitätsdiagramme: Elemente

Gabelung (auch: Parallelisierungsknoten)

Eine Gabelung (fork node) im Aktivitätsdiagramm teilt einen Kontroll- oder Objektfluss in mehrere parallele/nebenläufige Flüsse auf. Ihr entspricht im Petrinetz eine Transition mit mehreren Stellen (gegebenenfalls Hilfsstellen) in der Nachbedingung.

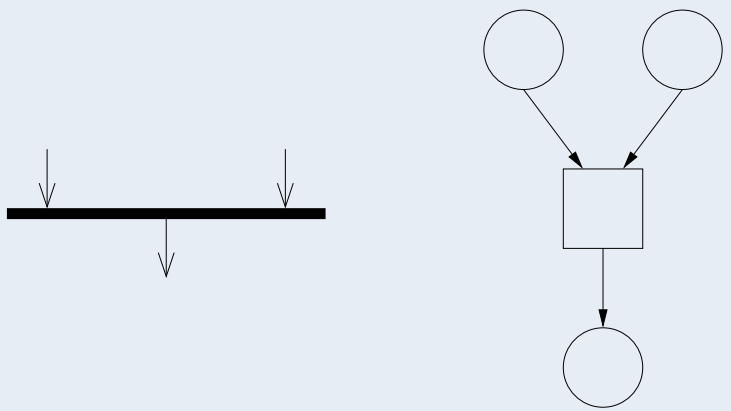




Aktivitätsdiagramme: Elemente

Vereinigung (auch: Synchronisationsknoten)

Dual dazu gibt es die Vereinigung (join node), die mehrere parallele/nebenläufige Kontroll- oder Objektflüsse zusammenführt.
Sie wird im entsprechenden Petrinetz durch eine Transition umgesetzt, die mehrere Stellen (ggfs. Hilfsstellen) in der Vorbedingung hat.

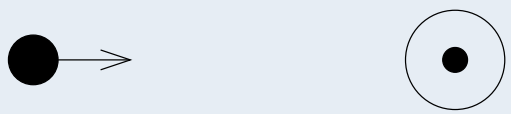


Wie Verbindungs- und Verzweigungsknoten kann man manchmal auch Vereinigungen und Gabelungen zu einem Knoten zusammenfassen.

Aktivitätsdiagramme: Elemente

Startknoten

Ein Startknoten im Aktivitätsdiagramm entspricht einer initial markierten Stelle im Petrinetz. In Startknoten dürfen keine Kanten hineinführen.



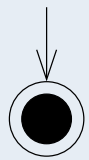
Anmerkungen:

- Im übersetzten Petrinetz wird die entsprechende Stelle initial mit so vielen Marken bestückt, wie es ausgehende Kanten vom Startknoten gibt.
- Diese Kanten führen generell nicht zu Objektknoten.
- Es sind mehrere Startknoten in einem Aktivitätsdiagramm möglich.

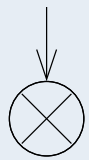
Aktivitätsdiagramme: Elemente

Aktivitätsende

Das Aktivitätsende signalisiert, dass alle Kontroll- und Objektflüsse beendet werden. Es gibt keine allgemeine Entsprechung in Petrinetzen.



Es gibt auch ein Symbol für das Flussende, das nur den in es hineinlaufenden Kontrollfluss beendet.

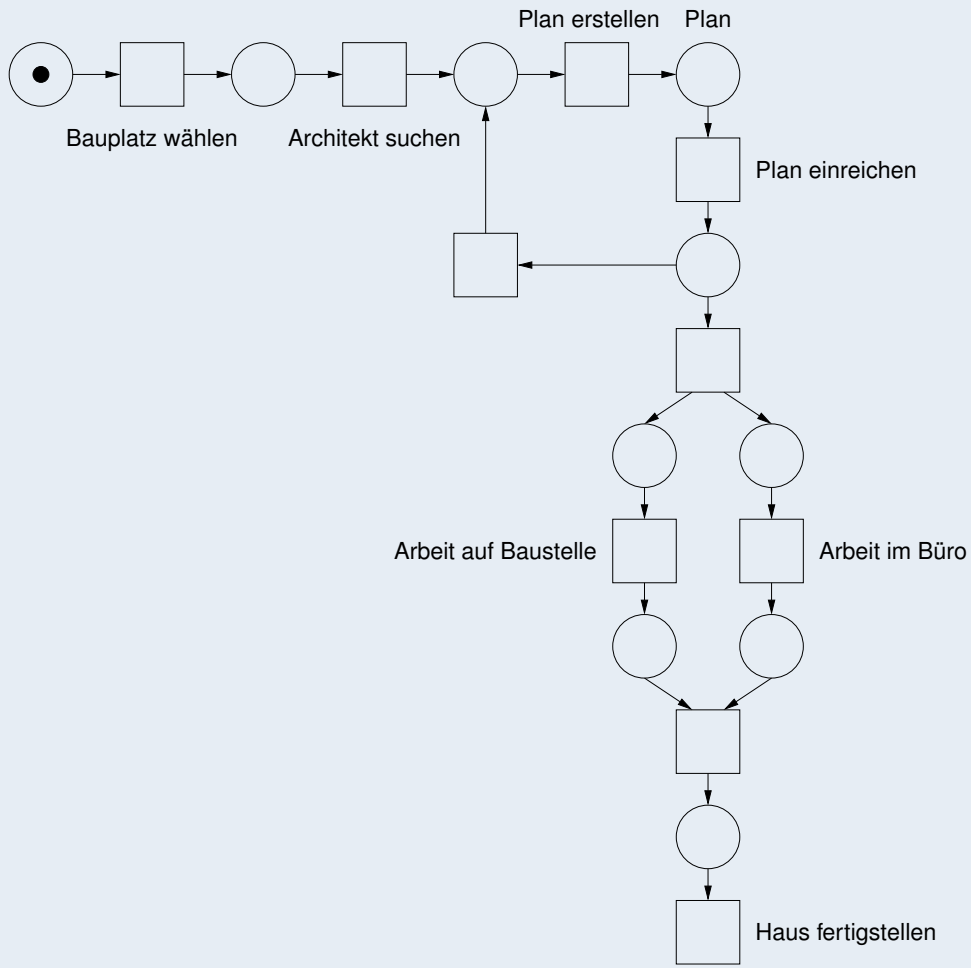


Ausgehende Kanten sind hier jeweils nicht erlaubt.

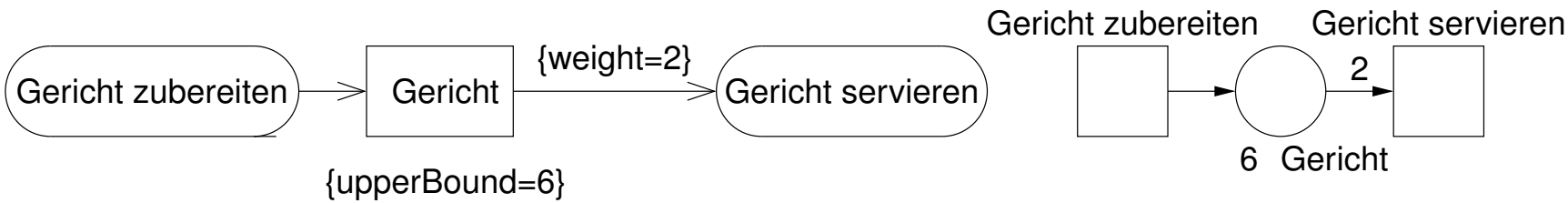
Aktivitätsdiagramme: Beispiel übersetzt

UNIVERSITÄT DUISBURG ESSEN
Offen im Denken

Übersetztes Beispiel, als Petrinetz:



Aktivitätsdiagramme enthalten noch mehr Anleihen aus Petrinetzen. Beispielsweise können auch die Kapazität eines Objektknotens und das Gewicht eines Objektflusses spezifiziert werden.



Dabei beschreibt upperBound die Kapazität (maximal 6 Gerichte dürfen hier gleichzeitig fertig sein) und weight das Gewicht (immer 2 Gerichte werden hier gleichzeitig serviert).

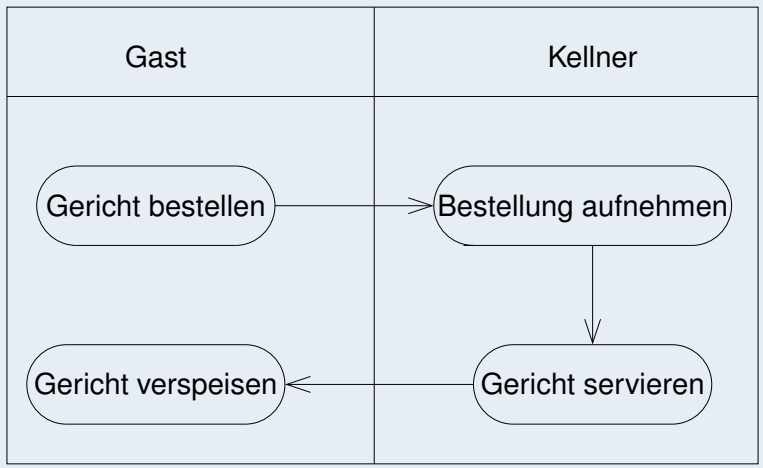
In Aktivitätsdiagrammen darf zusätzlich sogar noch spezifiziert werden, in welcher Reihenfolge die Objekte aus dem Objektknoten genommen werden (unordered, ordered, LIFO, FIFO).

Vorsicht:

- Die Entsprechung zwischen Aktivitätsdiagrammen und Petrinetzen ist nicht immer ganz exakt. Nicht für alle Konzepte gibt es eine direkte Übersetzung.
- Das liegt teilweise auch daran, dass Aktivitätsdiagramme nur ein semi-formales Modellierungsmittel sind und gar nicht alle Aspekte vollständig spezifiziert sind.

Aktivitätsbereiche (activity partitions)

Zur Strukturierung können Elemente gruppiert werden. Dies dient etwa dazu, die Verantwortung für bestimmte Aktionen festzulegen oder räumliche Verteilung auszudrücken.



Dabei können zum Beispiel Objektknoten auch auf einer Grenze zwischen Bereichen liegen, um eine Übergabe auszudrücken.

Aktivitätsdiagramme: Elemente (Fortsetzung)

Weitere Elemente (hier nicht behandelt) von Aktivitätsdiagrammen:

- Pins: Parameter und Parametersätze für Aktionen
- Senden und Empfang von Signalen
- Kontrollstrukturen: Sprungmarken, Schleifenknoten, Bedingungsknoten
- Unterbrechungsbereiche (interruptible activity region) zur Behandlung von Ausnahmen (Exceptions)
- Expansionsbereiche (expansion region) zur wiederholten Ausführung von Aktivitäten für mehrere übergebene Objekte

Zustandsdiagramme

Zustandsdiagramme

UNIVERSITÄT DUISBURG ESSEN
Offen im Denken

UML-Zustandsdiagramme (state diagrams, auch state machine diagrams oder statecharts genannt), sind eng verwandt mit den bereits eingeführten flachen Zustandsdiagrammen.

Sie werden eingesetzt, wenn bei der Modellierung der Fokus auf die Zustände und Zustandsübergänge des Systems gelegt werden soll.

Im Gegensatz zu Aktivitätsdiagrammen werden auch weniger die Aktionen eines Systems beschrieben, sondern eher die Reaktionen eines Systems auf seine Umgebung.

Zustandsdiagramme

Anwendungen sind die Modellierung von:

- Protokollen, Komponenten verteilter Systeme
- Benutzungsoberflächen
- eingebetteten Systemen
- ...

Zustandsdiagramme

Zustandsdiagramme wurden 1987 von David Harel unter dem Namen Statecharts eingeführt (siehe Artikel zu Beginn der Inhalts-Folien).

Features, mit denen Zustandsdiagramme ausgestattet sind:

- Zustände und Zustandsübergänge
- hierarchische Verfeinerung von Zuständen
- „Parallelschalten“ durch Regionen
- Historien, um sich früher besuchte Zustände zu merken und in diese zurückzukehren
- Kommunikation über Effekte („Fernauslösung“) oder Flags

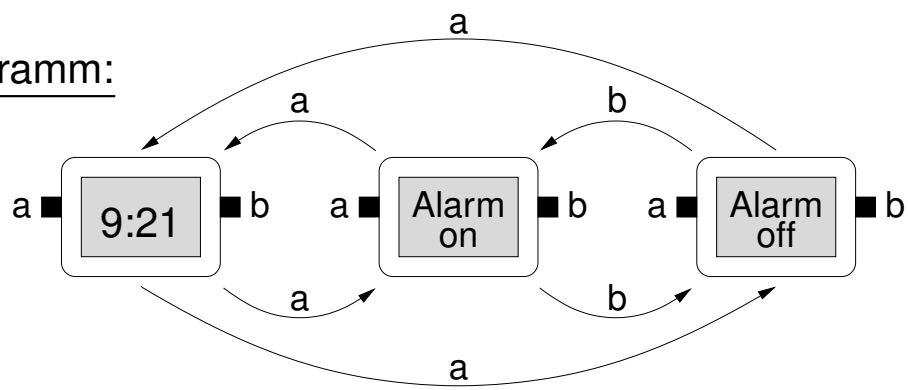
Viele dieser Ausstattungsmerkmale dienen dazu, Diagramme mit vielen Zuständen und Übergängen übersichtlicher und kompakter zu gestalten.

Zustandsdiagramme: Beispiel

Wir lernen Zustandsdiagramme am Beispiel der Modellierung einer Armbanduhr kennen (stark vereinfacht gegenüber einem Beispiel von Harel aus dem ursprünglichen wissenschaftlichen Artikel).

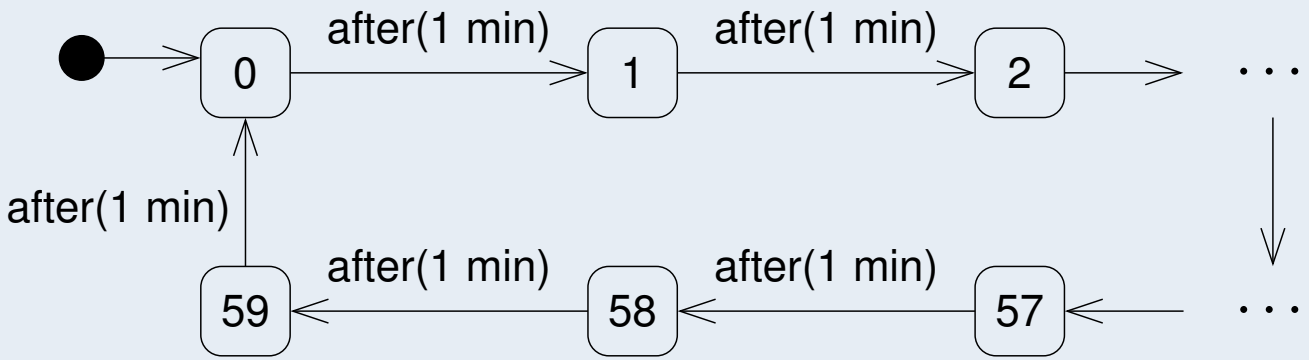
Die Armbanduhr hat zwei Knöpfe (a, b) und zwei Modi (Zeitanzeige, Alarmeinstellung). Zwischen den Modi wechselt man mit Hilfe von Knopf a. Der Alarm kann an (on) oder aus (off) sein. Zwischen den Alarmzuständen wechselt man mit Hilfe von Knopf b (im entsprechenden Modus). Wenn der Alarm an ist, erzeugt die Uhr zu jeder vollen Stunde einen Piepton.

noch kein Zustandsdiagramm:



Zustandsdiagramme: Beispiel

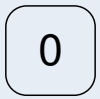
Wir beginnen zunächst mit der Modellierung der Minutenanzeige.



Zustandsdiagramme: Elemente

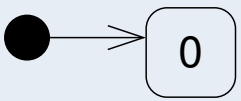
Zustand

Ein Zustand wird durch ein Rechteck mit abgerundeten Ecken dargestellt.



Startzustand

Der Startzustand wird ähnlich zum Startknoten bei Aktivitätsdiagrammen gekennzeichnet.

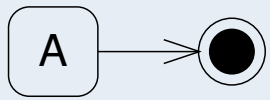


Es dürfen in den schwarzen ausgefüllten Kreis keine Kanten hineinführen, und nur genau eine heraus.

Zustandsdiagramme: Elemente

Endzustand

Endzustände werden wie das Aktivitätsende bei Aktivitätsdiagrammen gekennzeichnet.



Es dürfen aus dem Aktivitätsende-Symbol keine Kanten herausführen.

In Fällen, in denen man (wie in unserem Beispiel) ein System modelliert, das nicht terminieren soll, gibt es keinen Endzustand.

Zustandsdiagramme: Elemente

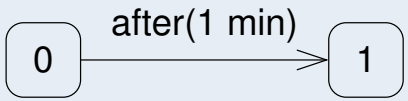
Transition (= Zustandsübergang)

Eine Transition ist nun eine Kante/Pfeil, beschriftet mit

Trigger [Bedingung] / Effekt ,

wobei Bedingung und Effekt optional sind.

- Trigger: Signal oder Nachricht, die die entsprechende Transition auslösen



- Bedingung: Überwachungsbedingung (auch Guard genannt)
- Effekt: Effekt, der durch die Transition ausgelöst wird

In unserem Beispiel bisher (Minutenanzeige) gibt es nur Trigger, die eine Zeitspanne angeben, nach der die Transition auszulösen ist.
Allgemein gibt es auch andere Trigger, beispielsweise Methodenaufrufe oder durch Broadcast-Effekte ausgelöste. (Zu Effekten später mehr.)

Zustandsdiagramme: Aktivitäten

UNIVERSITÄT DUISBURG ESSEN
Offen im Denken

Neben Effekten, die durch Transitionen (also beim Übergang) ausgelöst werden, können in einem Zustand selbst weitere Aktivitäten bei Eintritt, Verweilen oder Verlassen ausgelöst werden.

Diese haben den gleichen Aufbau wie die Beschriftung einer Transition: Trigger [Bedingung] / Effekt.

Dabei kann Trigger dann unter anderem folgendes sein:

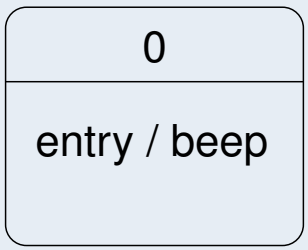
- entry: Der entsprechende Effekt wird bei Eintritt in den Zustand ausgelöst.
- do: Der Effekt ist eine Aktivität, die nach Betreten des Zustands ausgeführt wird und die spätestens dann endet, wenn der Zustand verlassen wird.
- exit: Der entsprechende Effekt wird bei Verlassen des Zustands ausgelöst.

Zustandsdiagramme: Beispiel

Beispiel:

Die Uhr soll zu jeder vollen Stunden ein Tonsignal von sich geben. Daher wird die Aktivität beep bei Eintritt in den Zustand 0 ausgelöst.

Notation:



Zustandsdiagramme: Elemente

Was ist mit der Stundenanzeige?

Diese soll parallel zur Minutenanzeige laufen, das heißt, die Uhr ist eigentlich immer in zwei Zuständen gleichzeitig: ein Zustand für die Stunden, der andere für die Minuten.

Solch eine Situation wird durch Regionen modelliert.

Region

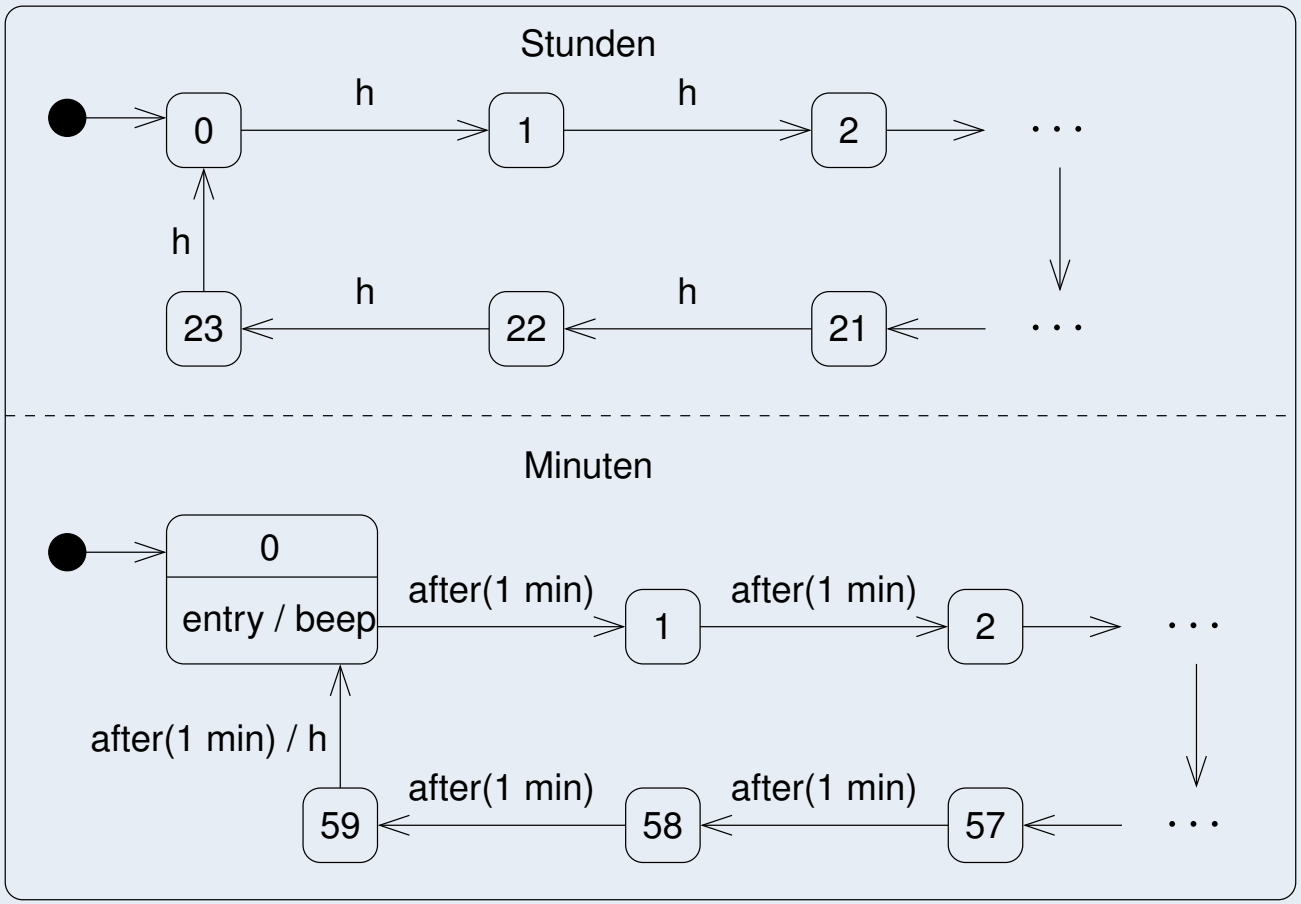
Regionen unterteilen ein Zustandsdiagramm in mindestens zwei Bereiche, die parallel zueinander ausgeführt werden.

Im Beispiel erlaubt uns dies, insgesamt nur $24 + 60 = 84$ Zustände, statt $24 \cdot 60 = 1440$ Zustände, zu zeichnen.

Anmerkung: Es gibt prinzipiell keine Transitionen zwischen zueinander parallelen Regionen.

Zustandsdiagramme: Beispiel

So sieht das modifizierte Zustandsdiagramm für die Uhr jetzt aus:



Zustandsdiagramme: Beispiel

Bemerkungen:

- Es gibt jetzt zwei Startzustände, die beide zu Beginn betreten werden (Uhrzeit: 0:00).

  (Allgemein: höchstens ein Startzustand pro Region.)

- Da Stunden- und Minutenanzeige nicht vollkommen unabhängig voneinander arbeiten sollen, ist eine Synchronisation eingebaut:
  Die Transition in den Minutenzustand 0 löst einen Broadcast-Effekt h aus (h für „hour“).

  Dieser Effekt dient dann in der anderen Region als Trigger, der den Übergang in den nächsten Stundenzustand verursacht.

  Diese beiden Transitionen/Übergänge finden gleichzeitig statt (jeweils in ihrer Region).

Zustandsdiagramme: Hinweise

Weitere Bemerkungen (für uns aber nicht so arg relevant):

- Transitionen verbrauchen im Allgemeinen keine Zeit (im Gegensatz zum Aufenthalt in Zuständen).
- Neben Effekten/Triggern, die direkt ausgeführt werden, gibt es auch solche, die zunächst in einer Event Queue (Warteschlange) abgelegt werden. Diese wird dann schrittweise abgearbeitet, wobei die entsprechenden Trigger ausgelöst werden. Damit kann asynchrone Kommunikation beschrieben werden.
- Effekte können Broadcasts (= Ausstrahlungen) sein, die überall im Zustandsdiagramm sichtbar sind, oder können alternativ direkte Kommunikation bedeuten (beispielsweise Methodenaufrufe). (Die ursprüngliche Statecharts-Semantik von Harel verwendete nur Broadcasts.)

Zustandsdiagramme: Elemente

Nun soll die Möglichkeit hinzugefügt werden, das Stunden-Alarmsignal aus- und wieder einzuschalten.

Dazu führen wir zunächst weitere Zustände (Alarm on, off) ein, zu denen man durch Drücken von Knopf a gelangt (und zwischen denen man mit Knopf b wechselt).

Problem: Wir bräuchten ziemlich viele mit a beschriftete Transitionen, die von den Stunden-/Minutenzuständen ausgehen!

Dafür bieten Zustandsdiagramme folgende Lösung:

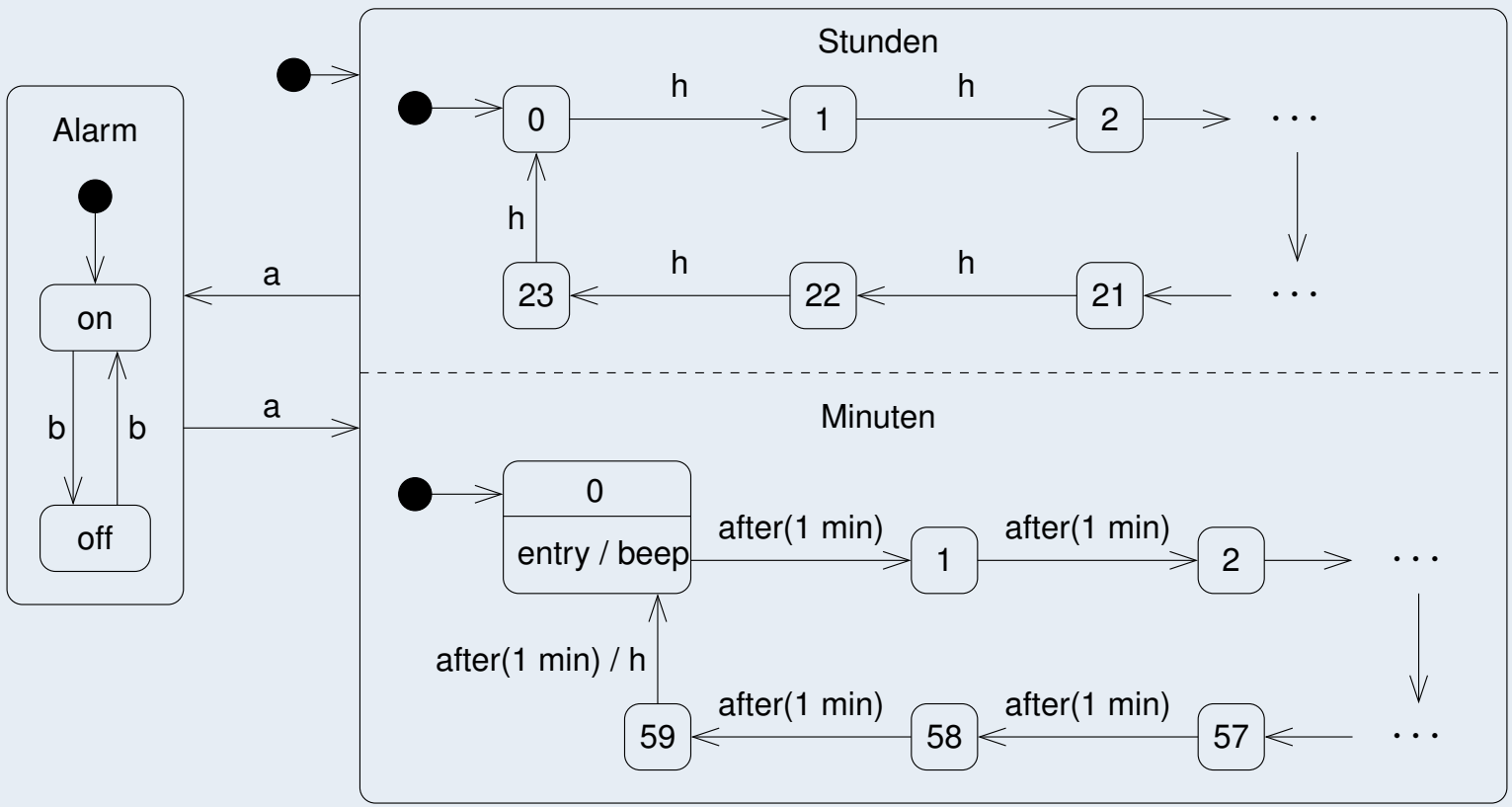
Zusammengesetzte Zustände

Zusammengesetzte Zustände dienen dazu, Hierarchien von Zuständen zu modellieren und dadurch ein- und ausgehende Transitionen zusammenzufassen.

Zustandsdiagramme: Beispiel

UNIVERSITÄT DUISBURG ESSEN
Offen im Denken

Es ergibt sich vorerst folgendes Zustandsdiagramm:



Zustandsdiagramme: Elemente

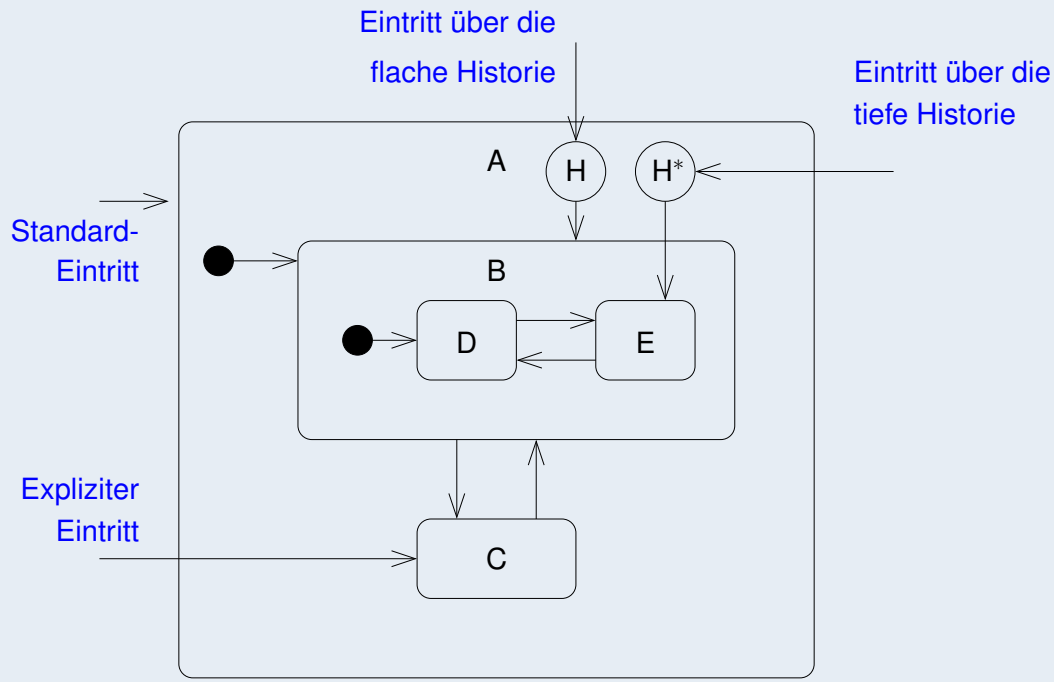
Anmerkungen:

- Zustände mit Regionen sind natürlich auch „zusammengesetzt“, aber in anderem Sinne, nicht nur hierarchisch.
- Wir sprechen daher ab jetzt statt nur allgemein von „zusammengesetzten Zuständen“ auch spezieller von „hierarchischen Zuständen“ einerseits und „Regionen-Zuständen“ andererseits.

Zustandsdiagramme: Eintrittsmöglichkeiten

Um möglichst viel Flexibilität bei der Modellierung zu haben, werden für zusammengesetzte Zustände verschiedene Eintritts- und Austrittsmöglichkeiten bereitgestellt.

Eintrittsmöglichkeiten in einen hierarchischen Zustand (grafisch)



Zustandsdiagramme: Eintrittsmöglichkeiten

Beschreibung der Eintrittsmöglichkeiten (am Beispiel-Diagramm):

- Standard-Eintritt: Dabei wird der Startzustand der obersten Hierarchieebene angesprungen.

  (Fortsetzung bei B, was schließlich zu einer Fortsetzung bei D führt.)

- Expliziter Eintritt: Es wird bei dem explizit angegebenen Unterzustand fortgesetzt.

  (Fortsetzung bei C.)

Zustandsdiagramme: Eintrittsmöglichkeiten

Beschreibung der Eintrittsmöglichkeiten (Fortsetzung):

- Eintritt über die tiefe Historie: Wurde der zusammengesetzte Zustand bereits früher besucht, so wird der letzte vor dem Verlassen des Gesamtzustands aktive Unterzustand (gegebenenfalls pro Region) der tiefstmöglichen Hierarchieebene betreten.

  (Falls also der zusammengesetzte Zustand A das letzte Mal von D aus verlassen wurde, so wird jetzt wieder bei D fortgesetzt.)

  Falls man noch niemals zuvor diesen zusammengesetzten Zustand betreten hat, so wird derjenige Unterzustand betreten, der mit der Kante gekennzeichnet ist, die von dem H^*-Knoten ausgeht.

Zustandsdiagramme: Eintrittsmöglichkeiten

Beschreibung der Eintrittsmöglichkeiten (Fortsetzung):

- Eintritt über die flache Historie: Wurde der zusammengesetzte Zustand bereits früher besucht, so wird der letzte vor dem Verlassen des Gesamtzustands aktive Unterzustand der obersten Hierarchieebene betreten.

  (Falls also der zusammengesetzte Zustand A das letzte Mal von E aus verlassen wurde, so wird jetzt bei B fortgesetzt, was letztendlich zu einer Fortsetzung bei D führt.)

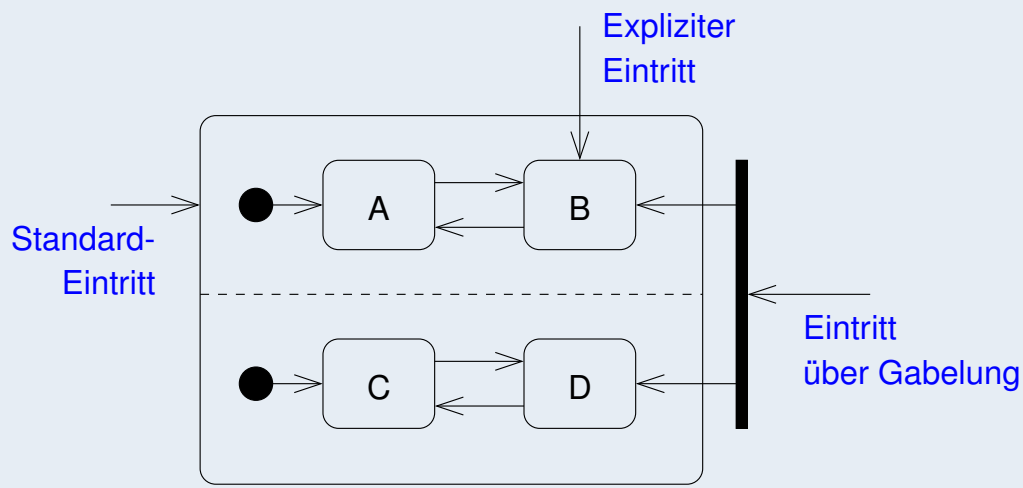
  Falls man noch niemals zuvor diesen zusammengesetzten Zustand betreten hat, so wird analog wie bei der tiefen Historie verfahren.

Außerdem: Eintritt über einen Eintrittspunkt (wird hier nicht behandelt).

Zustandsdiagramme: Eintrittsmöglichkeiten

Wenn ein zusammengesetzter Zustand in mehrere Regionen unterteilt ist, so ergeben sich noch einige Besonderheiten.

Eintrittsmöglichkeiten in einen Regionen-Zustand (grafisch)



Es gäbe zusätzlich auch wieder die Fälle zum Eintritt über flache oder tiefe Historie zu diskutieren, darauf verzichten wir hier jedoch.

Zustandsdiagramme: Eintrittsmöglichkeiten

Beschreibung der Eintrittsmöglichkeiten bei Regionen (am Beispiel-Diagramm):

- Standard-Eintritt: Dabei werden die jeweiligen Startzustände der Regionen angesprungen.

  (Fortsetzung bei A und C.)

- Expliziter Eintritt: Ein Zustand einer Region wird direkt angesprungen. In dazu parallelen Regionen wird beim Startzustand fortgesetzt.

  (Fortsetzung bei B und C.)

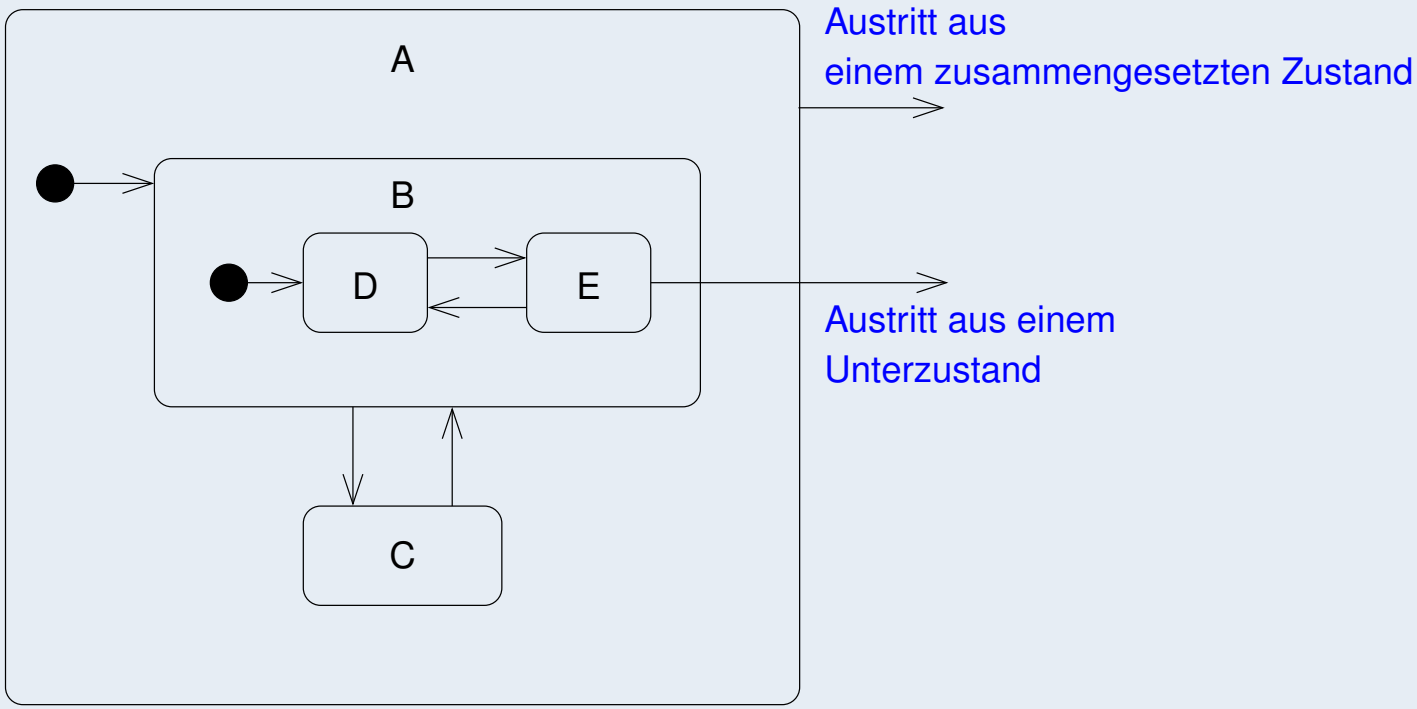
- Eintritt über Gabelung: Die anzuspringenden Zustände in den Regionen werden ähnlich zur Gabelung bei Aktivitätsdiagrammen gekennzeichnet.

  (Fortsetzung bei B und D.)

Zustandsdiagramme: Austrittsmöglichkeiten

Auch für den Austritt aus zusammengesetzten Zuständen gibt es verschiedene Möglichkeiten.

Austrittsmöglichkeiten aus einem hierarchischen Zustand (grafisch)



Zustandsdiagramme: Austrittsmöglichkeiten

Beschreibung der Austrittsmöglichkeiten:

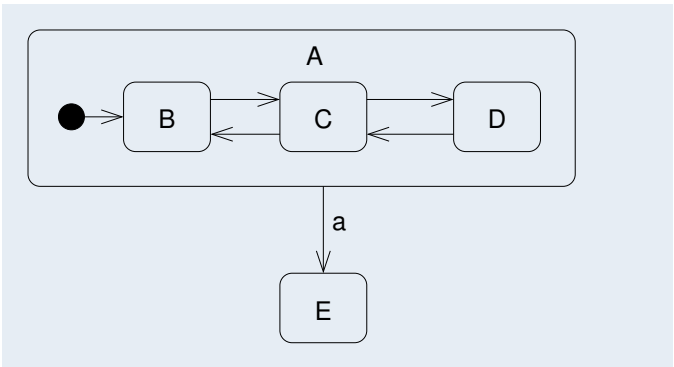
- Austritt aus einem hierarchischen Zustand als Ganzem: Sobald der mit der Transition assoziierte Trigger stattfindet, wird jeder beliebige Unterzustand verlassen.
- Austritt aus einem Unterzustand: Die Transition wird nur genommen, wenn man sich gerade im entsprechenden Unterzustand (im Beispiel-Diagramm: Zustand E) befindet, und der entsprechende Trigger stattfindet.

Außerdem: Austritt über einen Austrittspunkt, Endzustand oder Terminator (hier alle nicht behandelt).

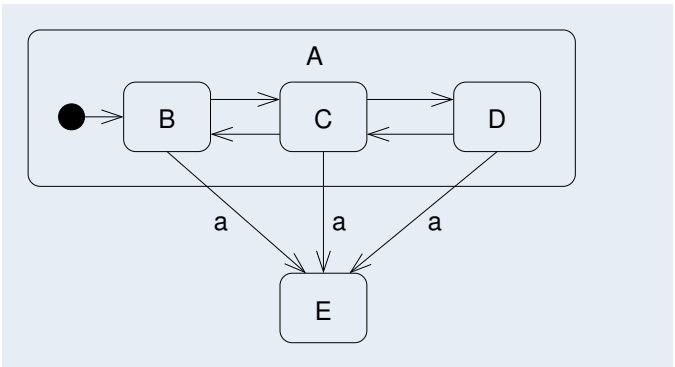
Zustandsdiagramme: Austrittsmöglichkeiten

UNIVERSITÄT DUISBURG ESSEN
Offen im Denken

Beispiel zu Austrittsmöglichkeiten:



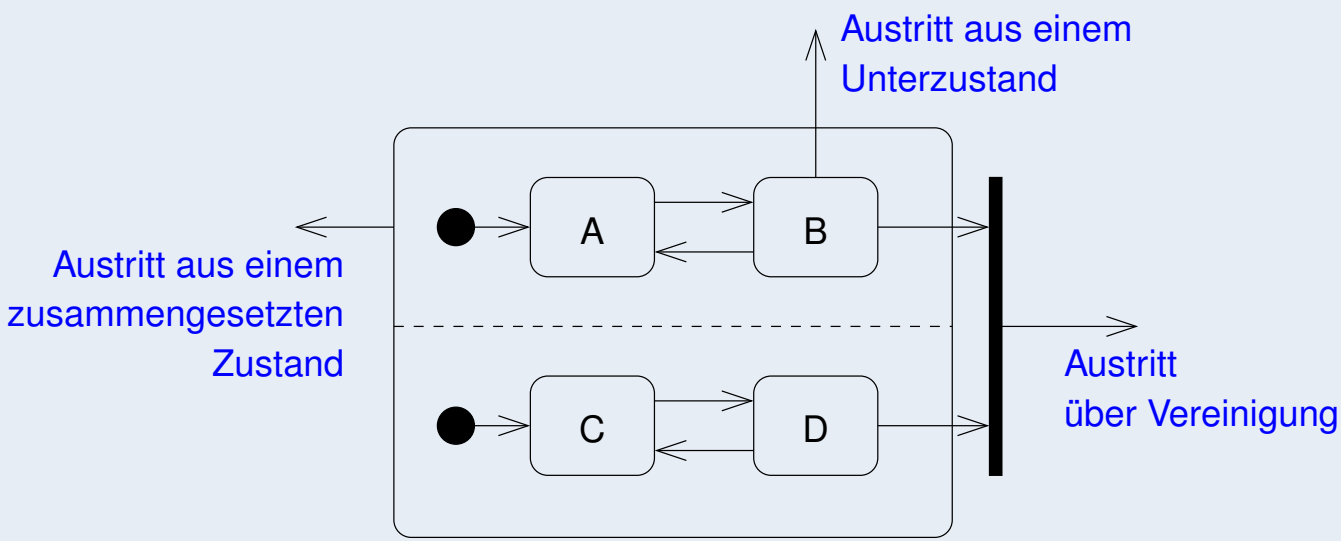
entspricht



Zustandsdiagramme: Austrittsmöglichkeiten

Auch für Austrittsmöglichkeiten müssen wir untersuchen, welche Besonderheiten sich bei Regionen ergeben.

Austrittsmöglichkeiten aus einem Regionen-Zustand (grafisch)



Zustandsdiagramme: Austrittsmöglichkeiten

Beschreibung der Austrittsmöglichkeiten bei Regionen:

- Austritt aus einem Regionen-Zustand als Ganzem: Dabei wird der zusammengesetzte Zustand verlassen, egal in welchen Unterzuständen man sich in den einzelnen Regionen gerade befindet.
- Austritt aus einem Unterzustand: Der zusammengesetzte Zustand wird nur verlassen, wenn man sich in der entsprechenden Region in dem Zustand befindet, der durch den Pfeil verlassen wird. In dazu parallelen Regionen kann man sich in beliebigem Zustand befinden.

  (Hier Austritt nur, wenn man sich in der ersten Region in B befindet.)
- Austritt über Vereinigung: Der zusammengesetzte Zustand kann nur verlassen werden, wenn man sich in den Regionen in den Zuständen befindet, von denen die Pfeile in die Vereinigung hineinführen.

  (Hier Austritt nur, wenn man sich in B und D befindet.)

Zustandsdiagramme: Beispiel

Wieder zurück zur Armbanduhr.

Es gibt noch Probleme:

Wenn man aus der Alarmeinstellung zurückkehrt, ist die Zeit auf 0:00 zurückgesetzt!

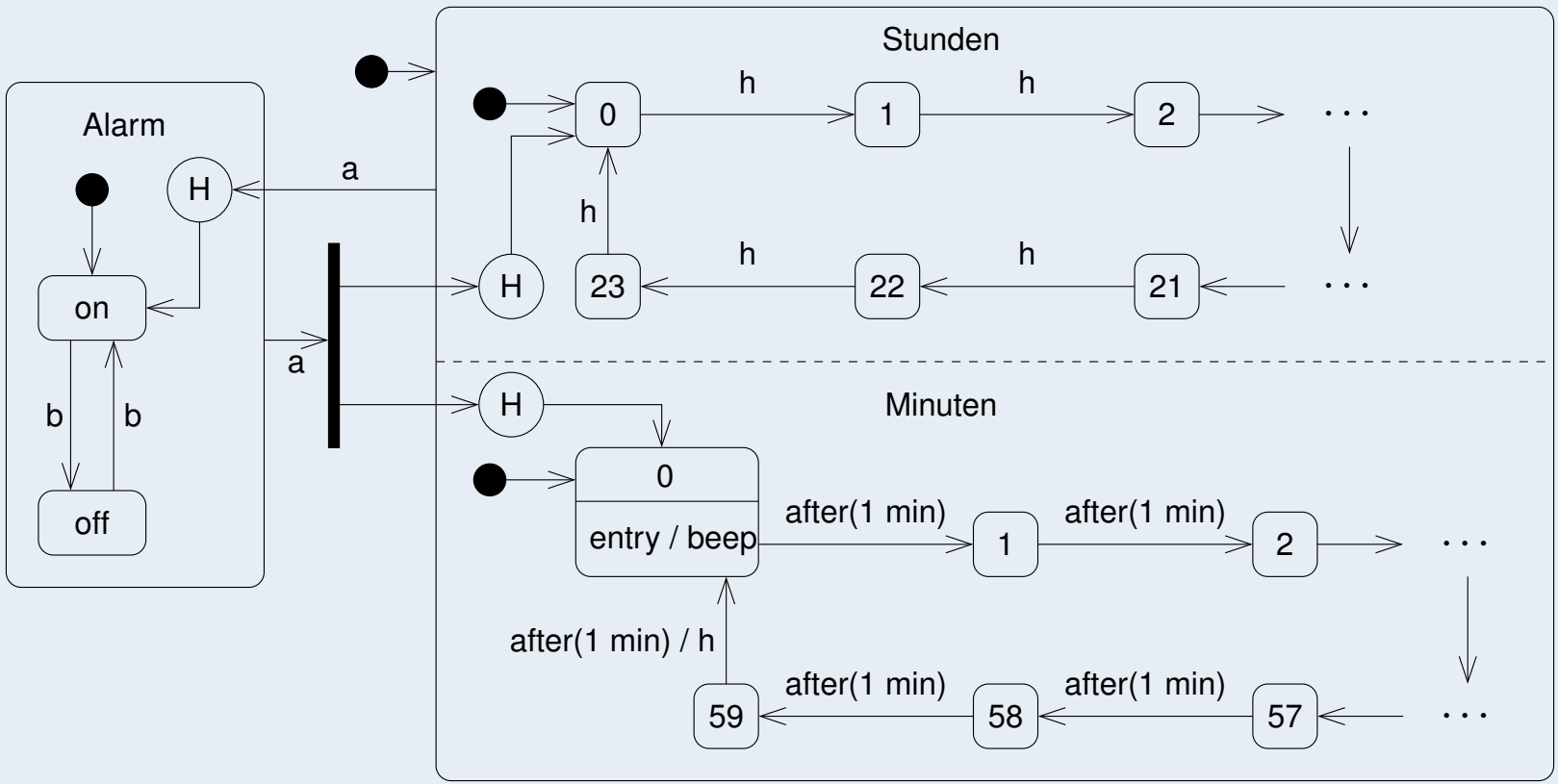
Und umgekehrt, wenn man zur Alarmeinstellung wechselt, wird der Alarm immer auf „on“ gesetzt!

Lösung:

Jeweils Verwendung des Eintritts über die (flache) Historie.

Da man im Fall der Zeitanzeige in zwei Regionen gleichzeitig eintreten muss, verwenden wir dort eine Gabelung.

Zustandsdiagramme: Beispiel



Zustandsdiagramme: Beispiel

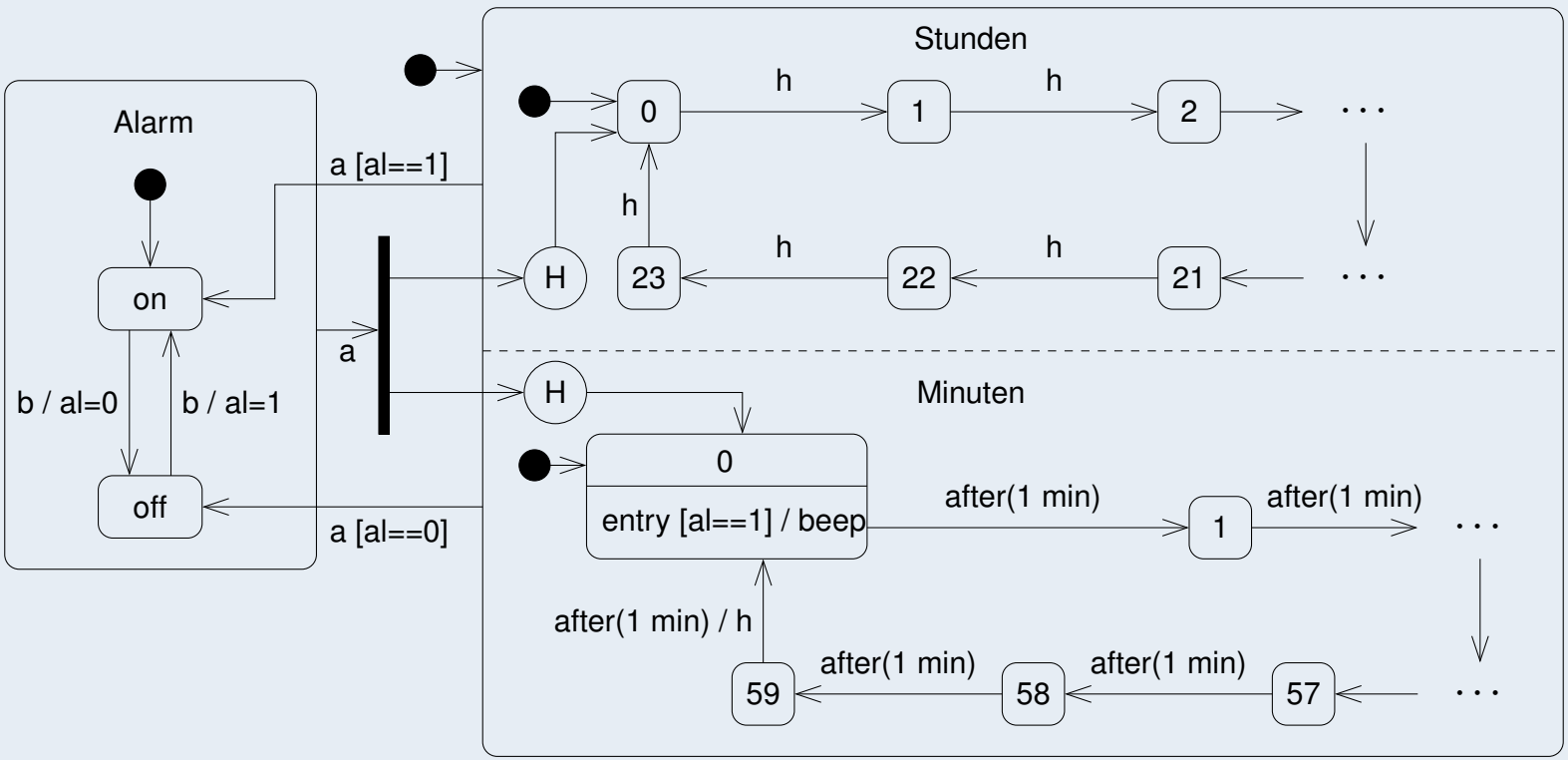
Was fehlt ansonsten noch?

Beim Wechseln zwischen den Alarm-Zuständen (on, off) muss ein Flag (hier al genannt) gesetzt werden, um damit den beep-Effekt unter Kontrolle zu kriegen.

Dieses Flag muss dann mit Hilfe einer Bedingung im Minutenzustand 0 abgefragt werden.

Außerdem betreten wir den Zustand Alarm nun nicht mehr über die flache Historie, sondern fragen mit Hilfe von Bedingungen ab, wie al belegt ist.

Zustandsdiagramme: Beispiel



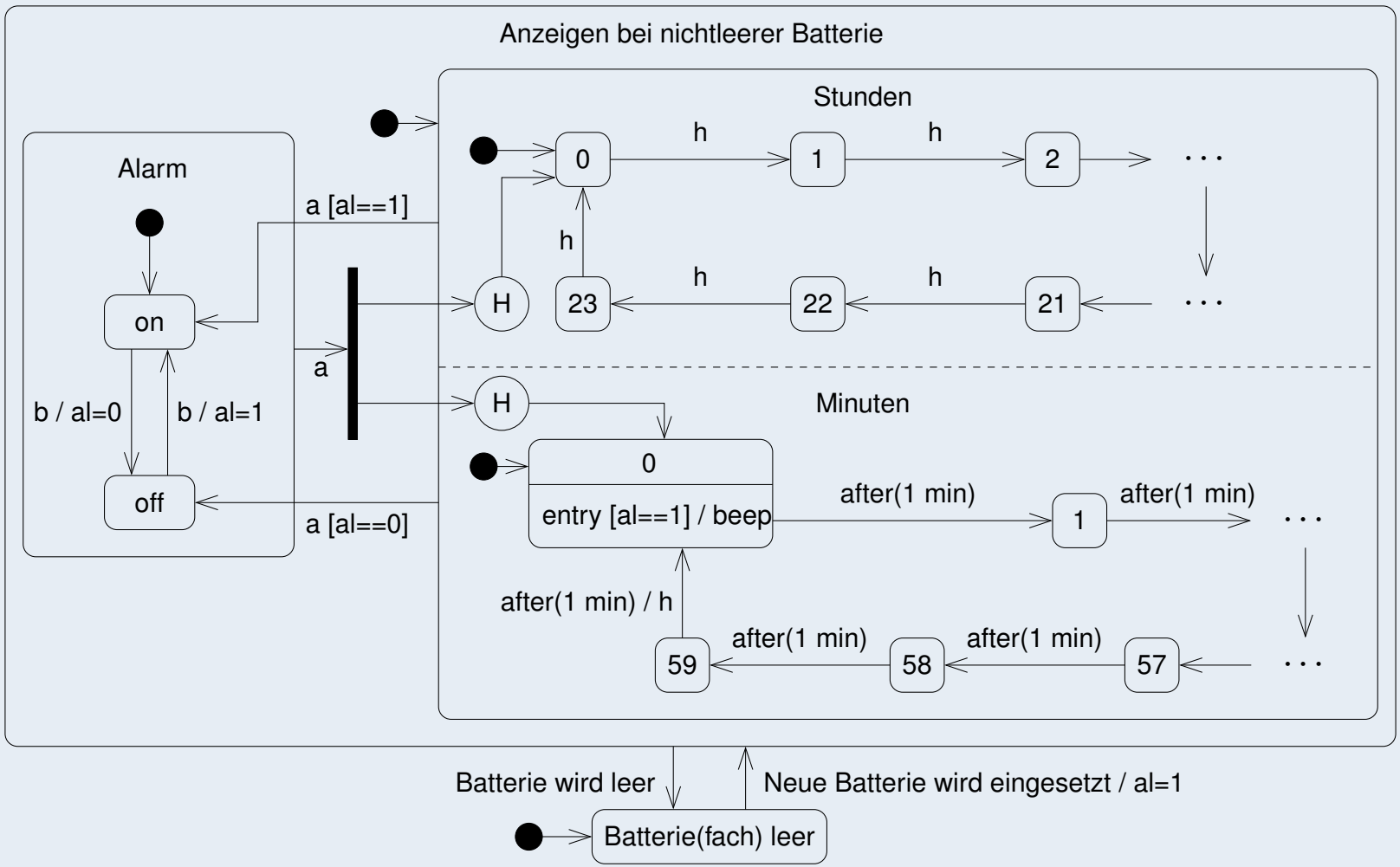
Zustandsdiagramme: Beispiel

Schließlich modellieren wir noch, dass die Batterie der Uhr leer werden kann und gewechselt werden muss.

In diesem Fall will man den zusammengesetzten Zustand für Stunden/Minuten nicht über die flache oder tiefe Historie betreten!
Es wird also dann tatsächlich die Zeit auf 0:00 zurückgesetzt.

Außerdem setzen wir das Flag al beim Einsetzen der Batterie (zurück) auf den Anfangswert 1.

Zustandsdiagramme: Beispiel



Zustandsdiagramme: „Flachklopfen“

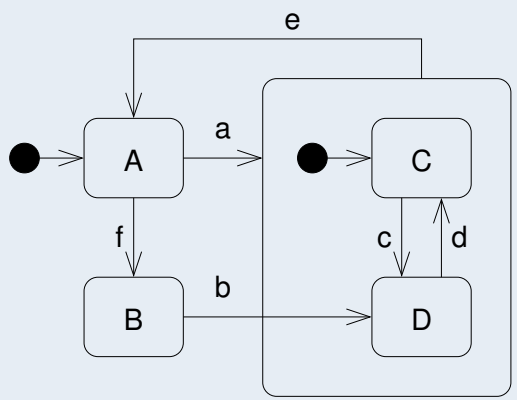
Ein großer Teil der Modellierungsmöglichkeiten von Zustandsdiagrammen in UML dient dazu, diese kompakt und übersichtlich zu notieren.

Umgekehrt kann man UML-Zustandsdiagramme oft „flachklopfen“ und zusammengesetzte Zustände (ob nun hierarchische oder aus Regionen gebildete) auflösen, wodurch man äquivalente flache Zustandsdiagramme erhält, die die gleichen Übergänge erlauben. Dabei erhält man jedoch im Allgemeinen mehr Zustände und/oder Transitionen.

Wir sehen uns einige Beispiele an (und werden bestimmte Features dabei bewusst nicht betrachten) . . .

Zustandsdiagramme: „Flachklopfen“

Beispiel 1: Wandeln Sie folgendes UML-Zustandsdiagramm in ein flaches Zustandsdiagramm um:

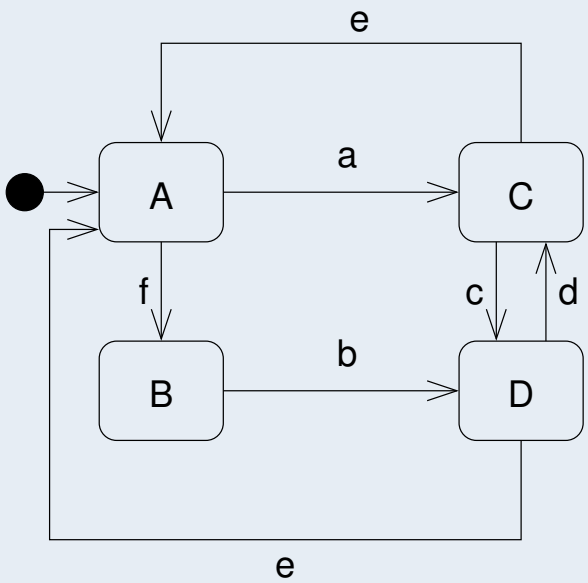


Charakteristika dieses Beispiels: keine Regionen, keine Historie

Ansatz: einfache Zustände behalten, Eintritte/Austritte an Rand hierarchischer Zustände übersetzen, andere Übergänge einfach behalten, und nur äußerster Startzustand bleibt solcher

Zustandsdiagramme: „Flachklopfen“

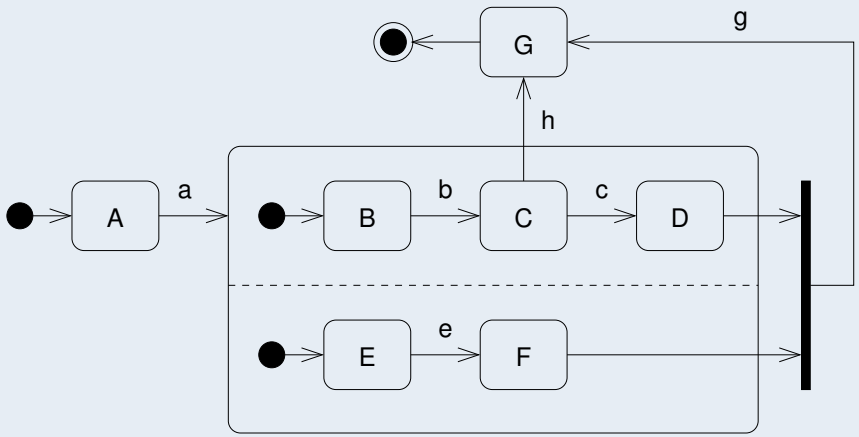
Lösung zu Beispiel 1:



Hauptschritt: Die Transition, die von dem hierarchischen Zustand wegführt, wurde durch mehrere Transitionen ersetzt, die von den Unterzuständen ausgehen.

Zustandsdiagramme: „Flachklopfen“

Beispiel 2: Wandeln Sie folgendes UML-Zustandsdiagramm in ein flaches Zustandsdiagramm um:

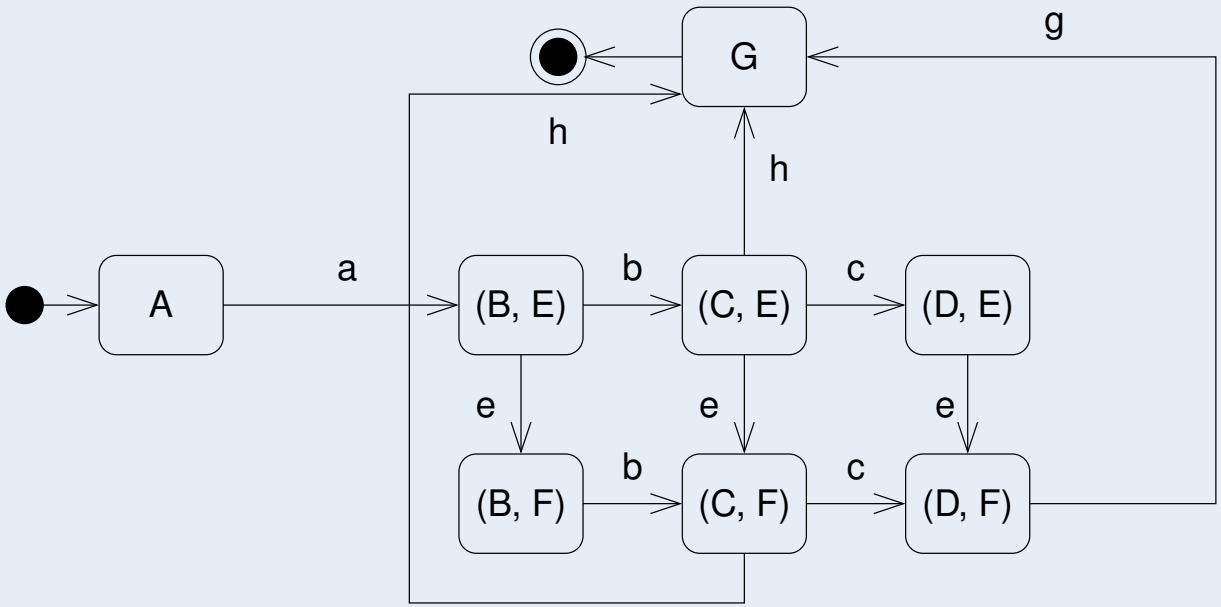


Charakteristika dieses Beispiels: Regionen, aber keine Historie

Ansatz außerhalb Regionen wie bisher, zusätzlich:
Kreuzprodukt zwischen Regionen; parallele Übergänge entsprechend den Regionen; Ein-/Austritte an Rand von, sowie hinein und heraus aus, Regionen übersetzen

Zustandsdiagramme: „Flachklopfen“

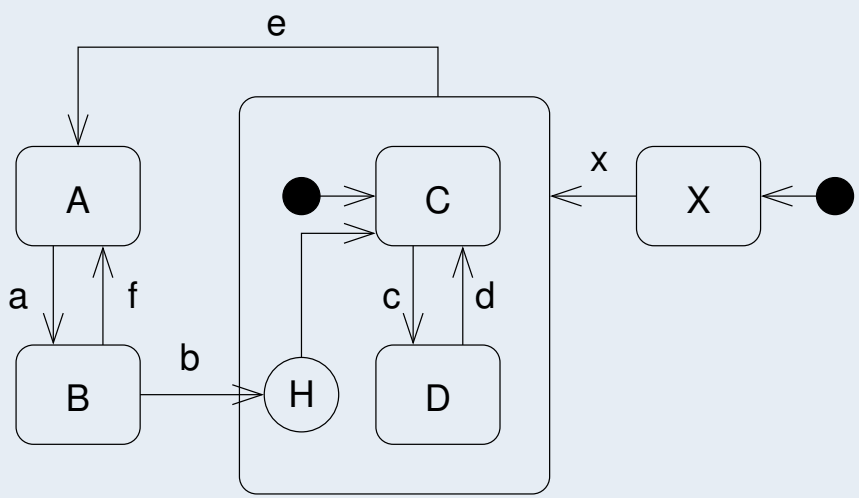
Lösung zu Beispiel 2:



Hauptidee: Kreuzprodukt der Zustandsmengen der Regionen bilden.

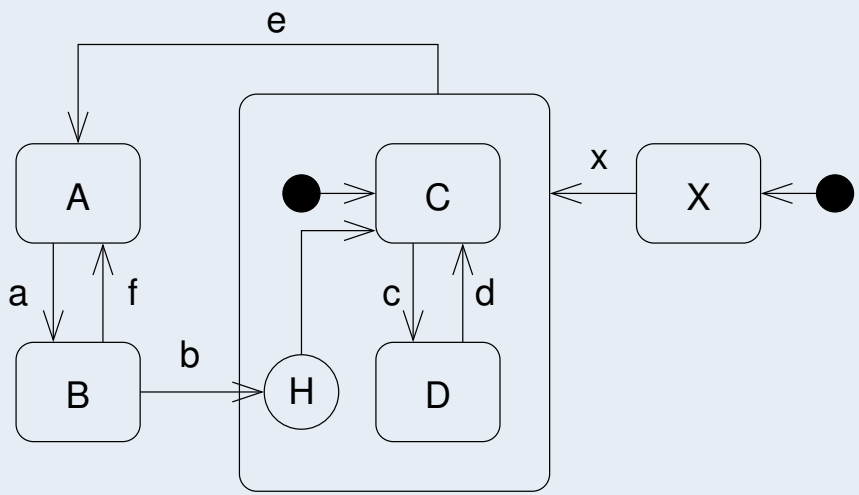
Zustandsdiagramme: „Flachklopfen“

Beispiel 3: Wandeln Sie folgendes UML-Zustandsdiagramm in ein flaches Zustandsdiagramm um:



Charakteristika hier: keine Regionen, aber (flache) Historie

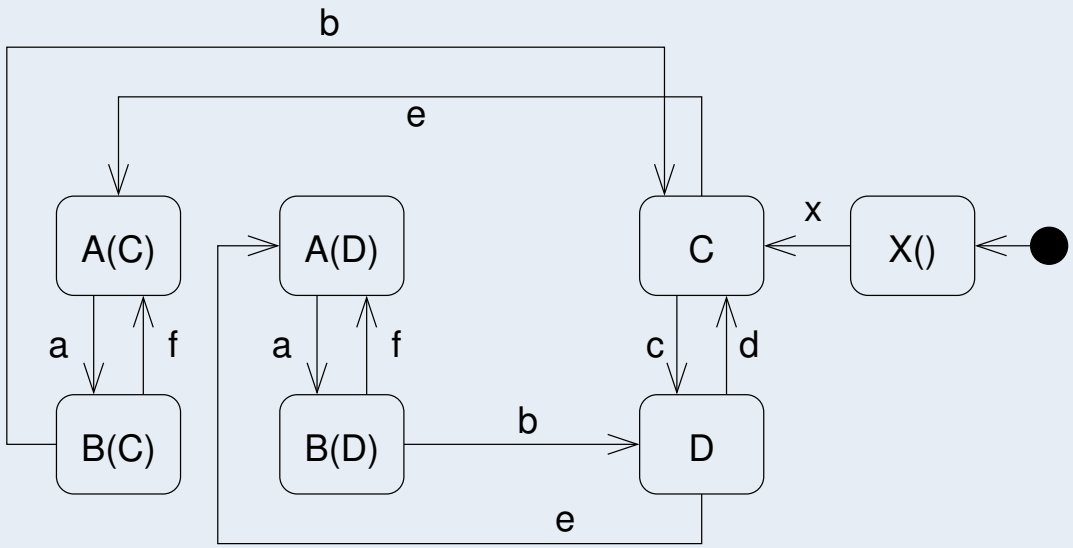
Zustandsdiagramme: „Flachklopfen“



Ansatz jetzt: zunächst wie im einfachen Fall, aber für Verlassen eines hierarchischen Zustands mit Historien-Knoten gegebenenfalls Kopien äußerer Zustände (und ihrer Übergänge) zum Merken des letzten inneren Unterzustands, und Verwendung dieser Information bei Wiedereintritt über die Historie

Zustandsdiagramme: „Flachklopfen“

Lösung zu Beispiel 3:



Hauptidee: In den äußeren Zuständen merkt man sich, ob/aus welchem Unterzustand man den hierarchischen Zustand zuletzt verlassen hat. Dies führt zu zusätzlichen Zuständen.

Zustandsdiagramme: weitere Features

Weitere Features von UML-Zustandsdiagrammen:

- Unterscheidung zwischen verschiedenen Arten von Triggern:
  call event, signal event, change event, time event, any receive event
- Verzögern und Ignorieren von Triggern
- Entscheidungen und Kreuzungen
- Eintritts- und Austrittspunkte, Terminator
- Rahmen und Wiederverwendung von Zustandsdiagrammen

Außerdem: Generalisierung und Spezialisierung von Zustands-diagrammen

Sequenzdiagramme

Sequenzdiagramme

Sequenzdiagramme (sequence diagrams) sind die bekanntesten Vertreter von Interaktionsdiagrammen in UML.

Sie dienen dazu, Kommunikation und Interaktion zwischen mehreren Kommunikationspartnern zu modellieren, und beruhen auf dem Basiskonzept der Interaktion:

Interaktion

Eine Interaktion ist das Zusammenspiel von mehreren Kommunikationspartnern.

Typische Beispiele: Versenden von Nachrichten, Datenaustausch, Methodenaufruf

Sequenzdiagramme

Sequenzdiagramme waren bereits vor Aufnahme in die UML unter dem Namen message sequence charts bekannt.

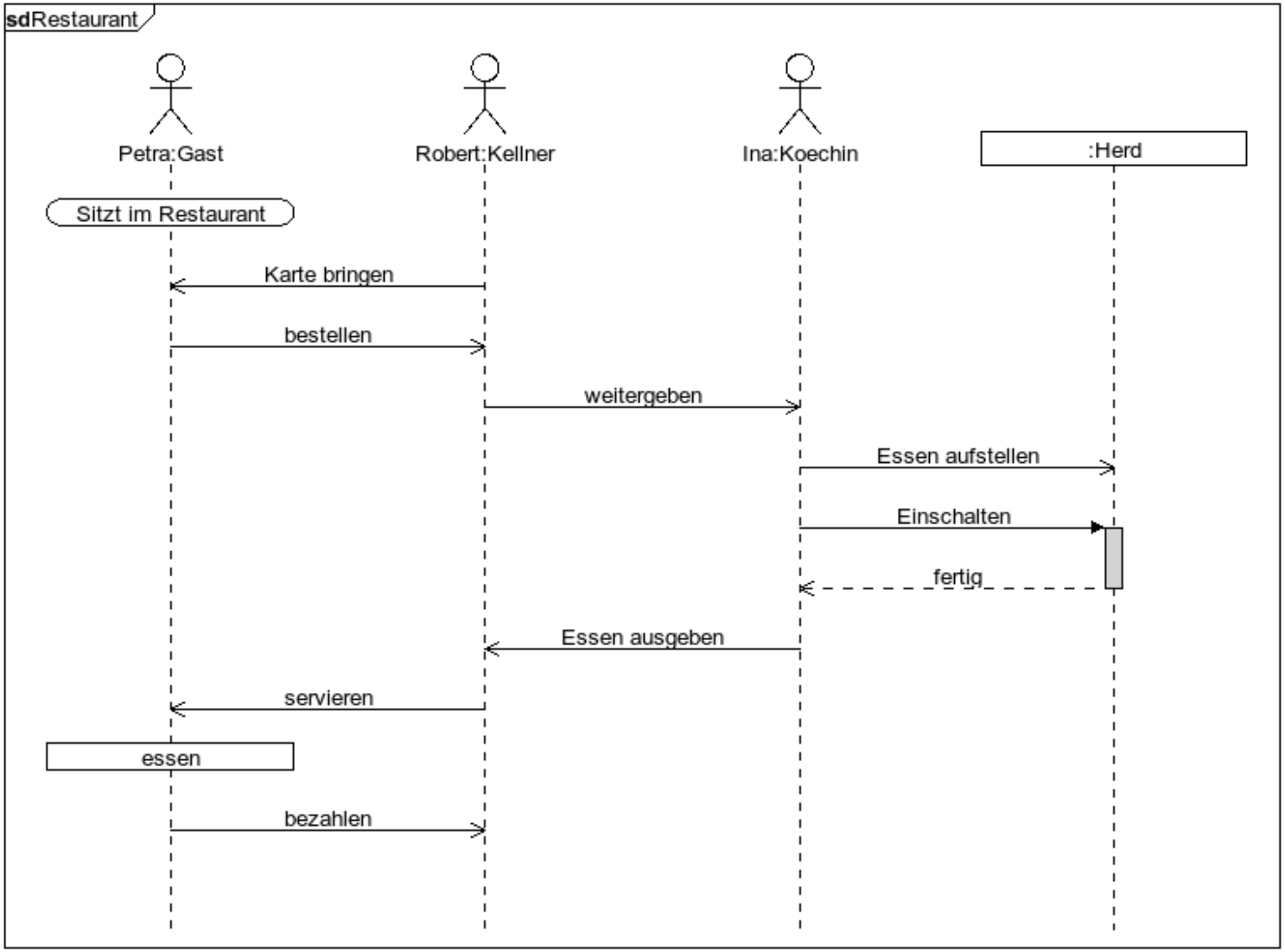
Im Gegensatz zu Aktivitätsdiagrammen oder Zustandsdiagrammen beschreiben sie im Allgemeinen nicht alle Abläufe eines Systems, sondern nur einen oder mehrere mögliche Abläufe.

Sequenzdiagramme

Sequenzdiagramme beschreiben Interaktionen in zwei Dimensionen:

- von links nach rechts: Anordnung der Kommunikationspartner als Lebenslinien. Oft wird der Partner, der den Ablauf initiiert, ganz links angegeben.
- von oben nach unten: Zeitachse

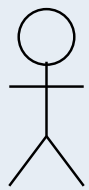
Sequenzdiagramme: Beispiel (Restaurant)



Kommunikationspartner

Die Kommunikationspartner in einem Sequenzdiagramm werden ähnlich wie in Objektdiagrammen als Rechtecke notiert.

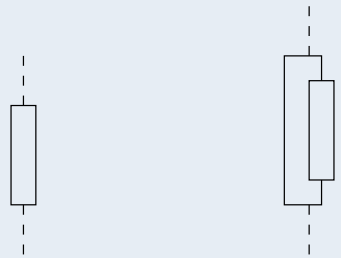
Manchmal werden die Rechtecke auch weggelassen. Menschliche Partner werden auch durch ein Strichmännchen symbolisiert:



Von jedem Kommunikationspartner führt eine gestrichelte Lebenslinie nach unten.

Ausführungsbalken

Aktivitäten eines Kommunikationspartners werden durch sogenannte Ausführungsbalken dargestellt.



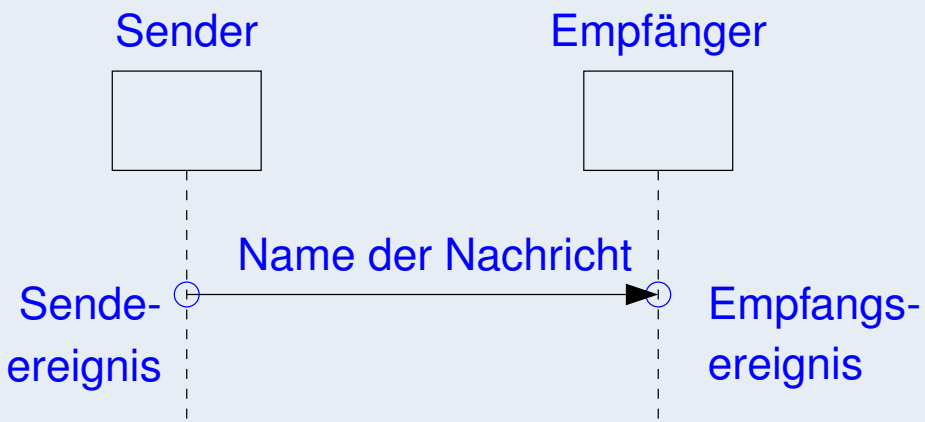
Parallele Tätigkeiten eines Kommunikationspartners werden dabei durch übereinander liegende Ausführungsbalken beschrieben (siehe oben rechts).

Während die Balken aktive Zeit anzeigen, symbolisieren die gestrichelten Linien passive Zeit.

Nachrichten

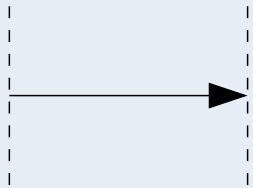
Die Nachrichten beschreiben die Kommunikationen bzw. Interaktionen der Kommunikationspartner und werden durch Pfeile dargestellt. Eine Nachricht hat einen Sender und einen Empfänger.

Die Stellen, an denen die Pfeile auf den Lebenslinien auftreffen, nennt man auch Sendeereignis und Empfangsereignis.



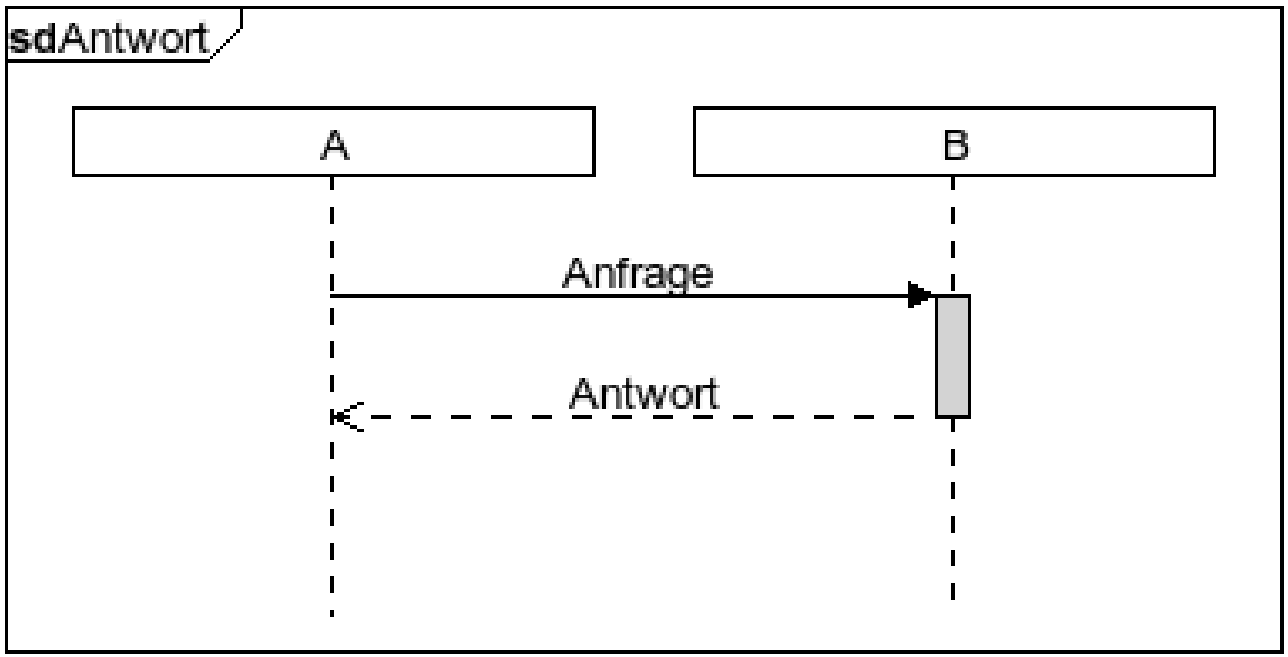
Synchrone Nachrichten

Bei synchroner Kommunikation warten Sender und Empfänger aufeinander. Der Sender macht erst dann weiter, wenn er weiß, dass der Empfänger die Nachricht erhalten hat. Solche Kommunikation wird unter Verwendung einer schwarzen ausgefüllten Pfeilspitze dargestellt.



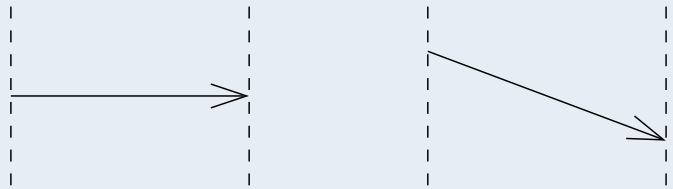
Sequenzdiagramme: Nachrichten

Um zu signalisieren, dass der Empfänger die Nachricht erhalten hat, bzw. die zugehörige Aktion abgeschlossen wurde, wird eine Antwort gesendet. Diese ist als gestrichelter Pfeil dargestellt.



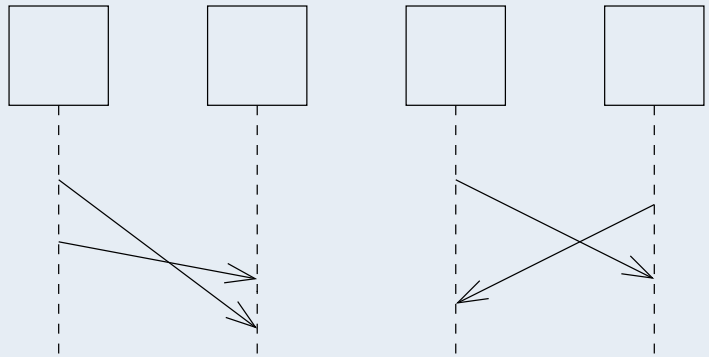
Asynchrone Nachrichten

Bei asynchroner Kommunikation wartet der Sender nicht darauf, dass der Empfänger die Nachricht erhalten hat. Er arbeitet einfach weiter. Solche Kommunikation wird durch eine einfache Pfeilspitze dargestellt.



Sequenzdiagramme: Nachrichten

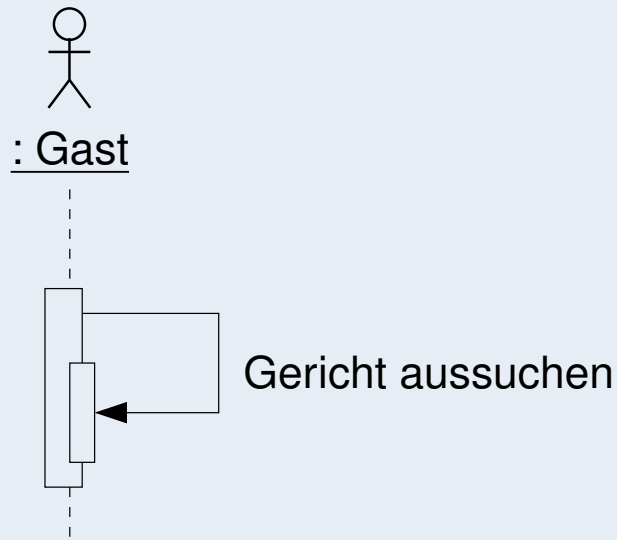
Bei asynchroner Kommunikation (aber nicht bei synchroner Kommunikation) kann auch der Fall eintreten, dass sich Nachrichten überholen oder kreuzen.



Das Überholen von Nachrichten (oben links) ist nur dann nicht möglich, wenn man explizit einen FIFO-Kanal fordert.

Sequenzdiagramme: Nachrichten

Es ist möglich, eine Nachricht an sich selbst zu schicken.



Sequenzdiagramme: Äquivalenz

Äquivalenz von Sequenzdiagrammen

Zwei Sequenzdiagramme sind äquivalent, wenn sie die gleichen Ereignisse enthalten und die Reihenfolge der Ereignisse identisch ist.

Dabei können die Diagramme durchaus verschieden gezeichnet sein (andere Reihenfolge der Kommunikationspartner, verschiedene Abstände zwischen den Ereignissen, . . .).

Durch die Bestimmung der Reihenfolge der Ereignisse kann die Äquivalenz zweier Diagramme nachgewiesen bzw. widerlegt werden.

Sequenzdiagramme: Interaktions-Operatoren

UNIVERSITÄT DUISBURG ESSEN
Offen im Denken

Bisher haben wir gesehen, wie man mit einem Sequenzdiagramm einen möglichen Ablauf beschreiben kann. In manchen Fällen möchte man jedoch mehrere (vielleicht sogar alle) Abläufe beschreiben.

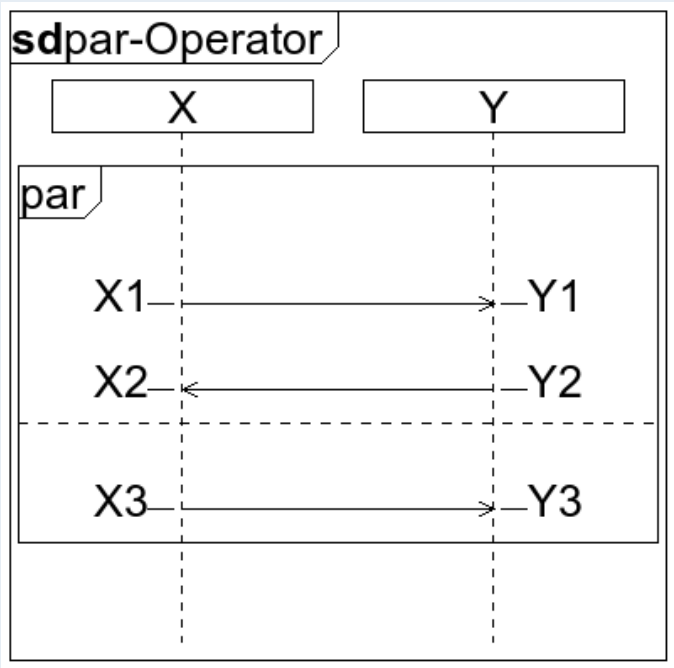
Dazu gibt es die Möglichkeit, kombinierte Fragmente zu verwenden, bei denen mehrere (Interaktions-)Operanden (= Teil-Sequenzdiagramme) mit Hilfe von Interaktions-Operatoren zusammengesetzt werden.

Wir betrachten im Folgenden einige dieser Interaktions-Operatoren.

Sequenzdiagramme: par-Operator

Interaktionsoperator par (Parallelität)

Hier sind die Operanden in beliebiger Reihenfolge ausführbar. Die Reihenfolge der Ereignisse in den Operanden muss aber gewahrt werden. Ansonsten gibt es keine Bedingungen.

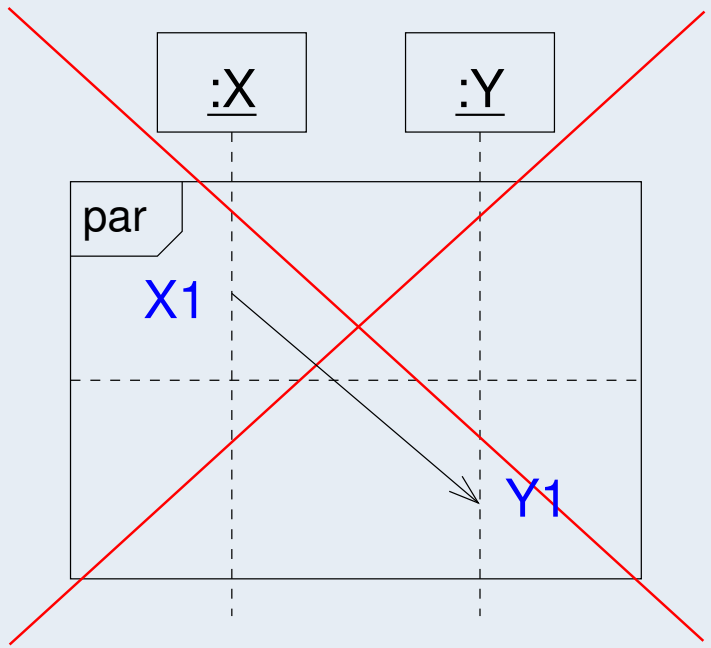


Dabei wird der Operator par links oben in der Ecke angegeben.

Eine gestrichelte waagerechte Linie trennt die verschiedenen Operanden.

Sequenzdiagramme: par-Operator

Nachrichten, die von einem Operanden in einen anderen laufen, sind nicht zugelassen.



Sequenzdiagramme: par-Operator

Ordnung auf den Ereignissen:

- X1 $<$ Y1 $<$ Y2 $<$ X2 (Operand 1)
- X3 $<$ Y3 (Operand 2)

Ansonsten gibt es keine weiteren Einschränkungen.

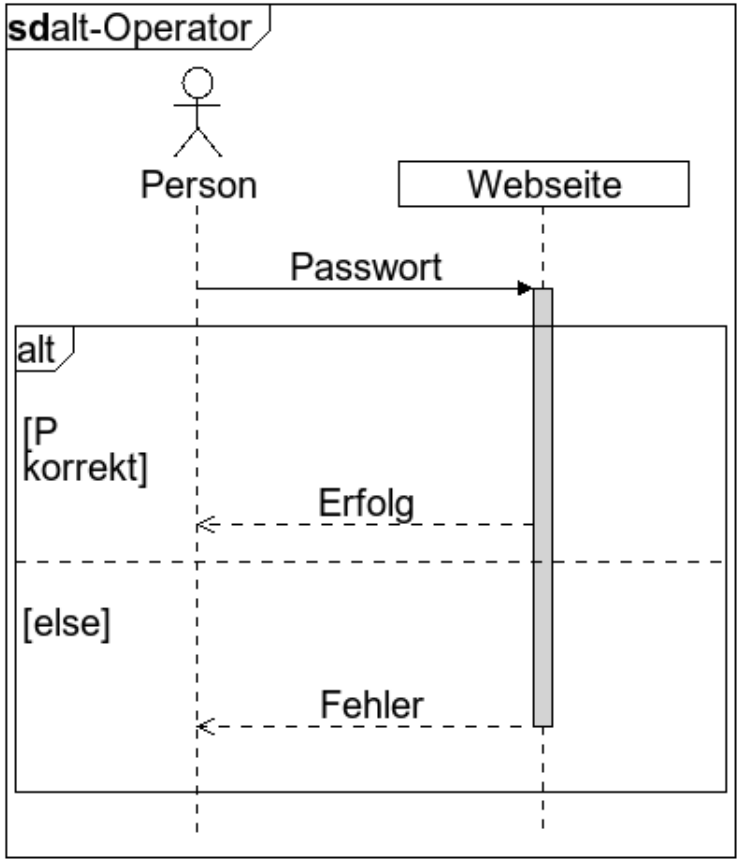
Dieses Sequenzdiagramm beschreibt insgesamt fünfzehn Abläufe, beispielsweise:

- X1, Y1, X3, Y3, Y2, X2
- X3, X1, Y1, Y2, X2, Y3
- ...

Sequenzdiagramme: alt-Operator

Sequenzdiagramme bieten auch die Möglichkeit, alternative Nachrichtenflüsse darzustellen. Zu diesem Zweck wird der alt-Operator verwendet.

Ein möglicher Anwendungsfall ist die unterschiedliche Rückgabe an eine Person bei der Eingabe eines Passwortes.



Sequenzdiagramme: alt-Operator

Dabei ist es nötig, die Eintrittsfälle für die Alternativen durch Guards zu definieren.

Es ist ebenfalls möglich, eine beliebige Anzahl von Alternativen zu modellieren. Diese sollten aber in einem Kontext stehen und nicht unabhängig voneinander sein. Zudem müssen alle Guards disjunkt sein.

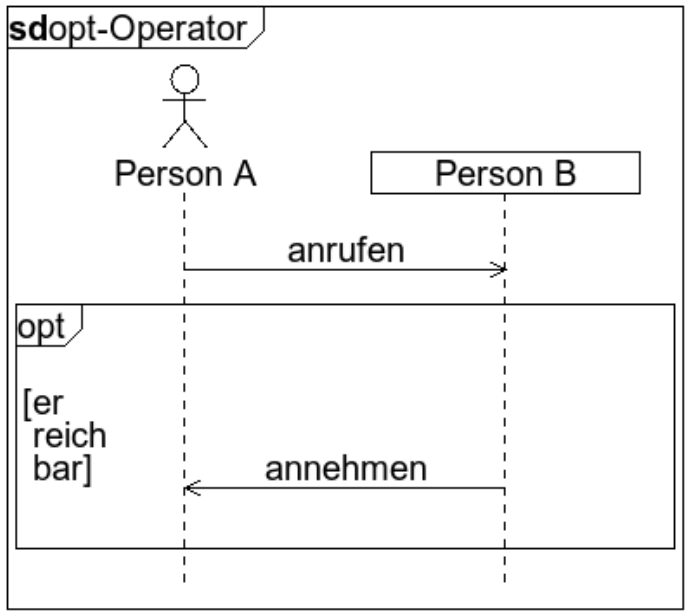
Durch die Verschachtelung der Blöcke ist es aber möglich, auch unabhängige Alternativen zu modellieren.

Es gibt keine Beschränkung der Anzahl Nachrichten, die in einem Block enthalten sein dürfen.

Wichtig: Bei der Nutzung eines alt-Operators müssen alle Fälle abgedeckt sein. Dies kann zum Beispiel durch einen Fall *else* beschrieben werden.

Sequenzdiagramme: opt-Operator

Der opt-Operator dient dazu, optionale Nachrichtenflüsse zu definieren.
Im Gegensatz zum bereits vorgestellten alt-Operator ist es damit möglich, diesen Block auch zu überspringen, falls die Nachricht nicht gesendet werden soll.
Ein Beispiel ist ein Telefonanruf, bei dem nur abgehoben wird, wenn die andere Person erreichbar ist.



Sequenzdiagramme: opt-Operator

Auch bei einem opt-Operator ist es nötig, durch Guards die Eintrittsbedingung zu definieren. Ist diese nicht erfüllt, wird der Teil innerhalb des Operators übersprungen.

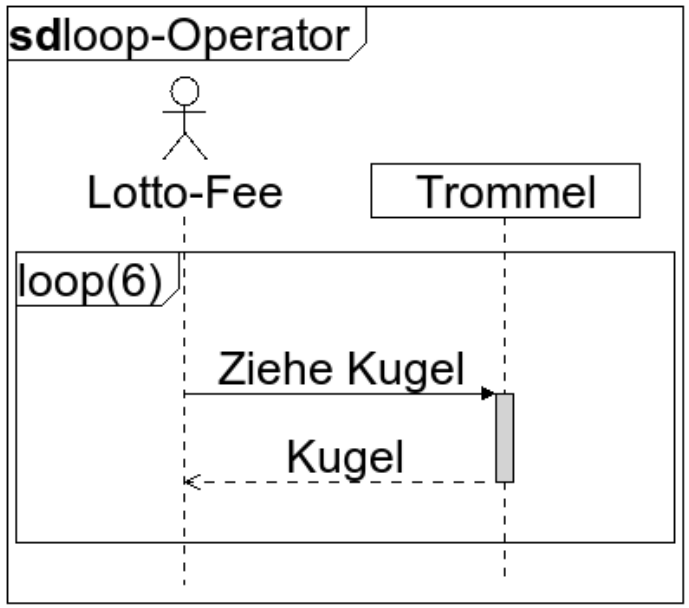
Im Gegensatz zum alt-Operator gibt es keine Möglichkeit, verschiedene Fälle in einem Operator zu definieren. Für mehrere verschiedene Bedingungen muss daher ein eigener opt-Operator eingeführt werden. Innerhalb eines Operators kann ein beliebiger Nachrichtenfluss mit einer beliebigen Anzahl an Nachrichten stattfinden.

Auch dieser Operator kann beliebig verschachtelt und mit anderen Operatoren kombiniert werden.

Sequenzdiagramme: loop-Operator

Der loop-Operator ermöglicht es, einen definierten Nachrichtenfluss zu wiederholen.

Ein Beispiel ist die Ziehung der Lottozahlen *6 aus 49*, bei der die Lottofee 6 mal eine Kugel aus der Trommel zieht.



Sequenzdiagramme: loop-Operator

In dem gezeigten Beispiel ist eine feste Anzahl von Abläufen gegeben. Es ist auch möglich, eine minimale und eine maximale Zahl der Form loop(min,max) zu definieren, wobei „*“ für eine beliebige Zahl einschließlich 0 stehen kann. In diesem Fall ist normalerweise zusätzlich ein Guard angegeben, der angibt, wann abgebrochen wird. Ist die min-Anzahl durchlaufen und die Bedingung erfüllt, wird die Schleife verlassen.

Auch der loop-Operator ist nicht in der Lage, verschiedene Fälle zu definieren. Dies geschieht, indem innerhalb die bereits bekannten Operatoren opt und alt verwendet werden.

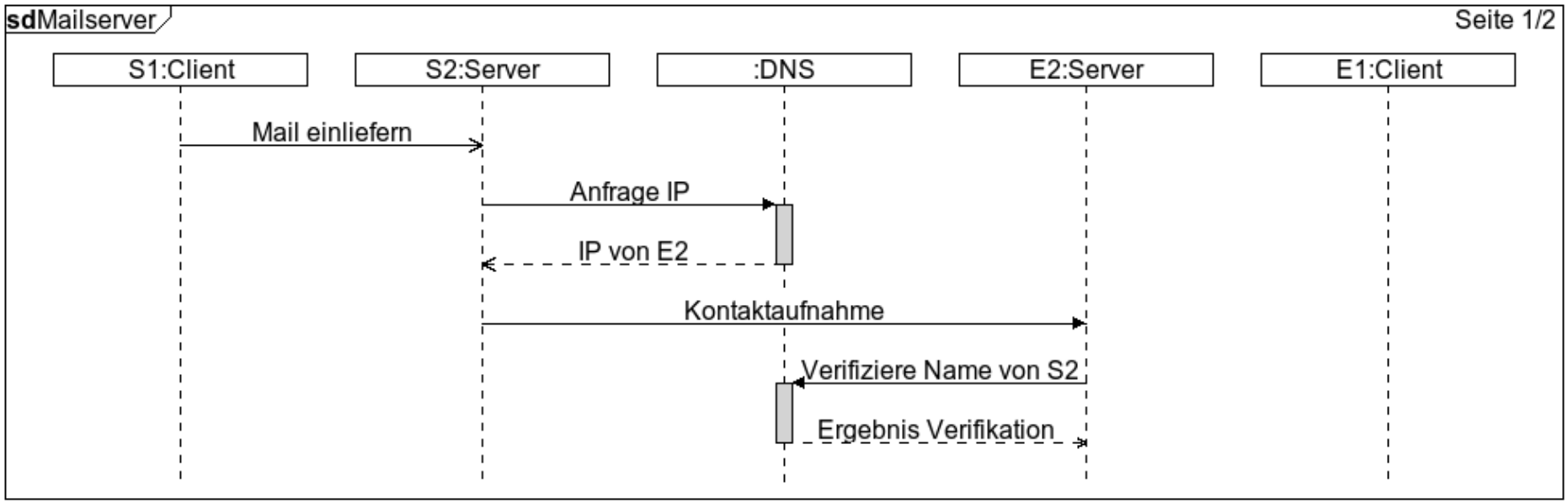
Sequenzdiagramme: Mailserver (Beispiel)

Beispiel: Wir modellieren ein Protokoll, bei dem ein Client-Rechner S eine E-Mail an einen anderen Client R verschickt.

Weitere Beteiligte sind der Mail-Server von S, der Mail-Server von R und der DNS-Server, der benötigt wird, um Mail-Adressen in die IP-Adressen des entsprechenden Servers umzuwandeln (DNS = domain name system).

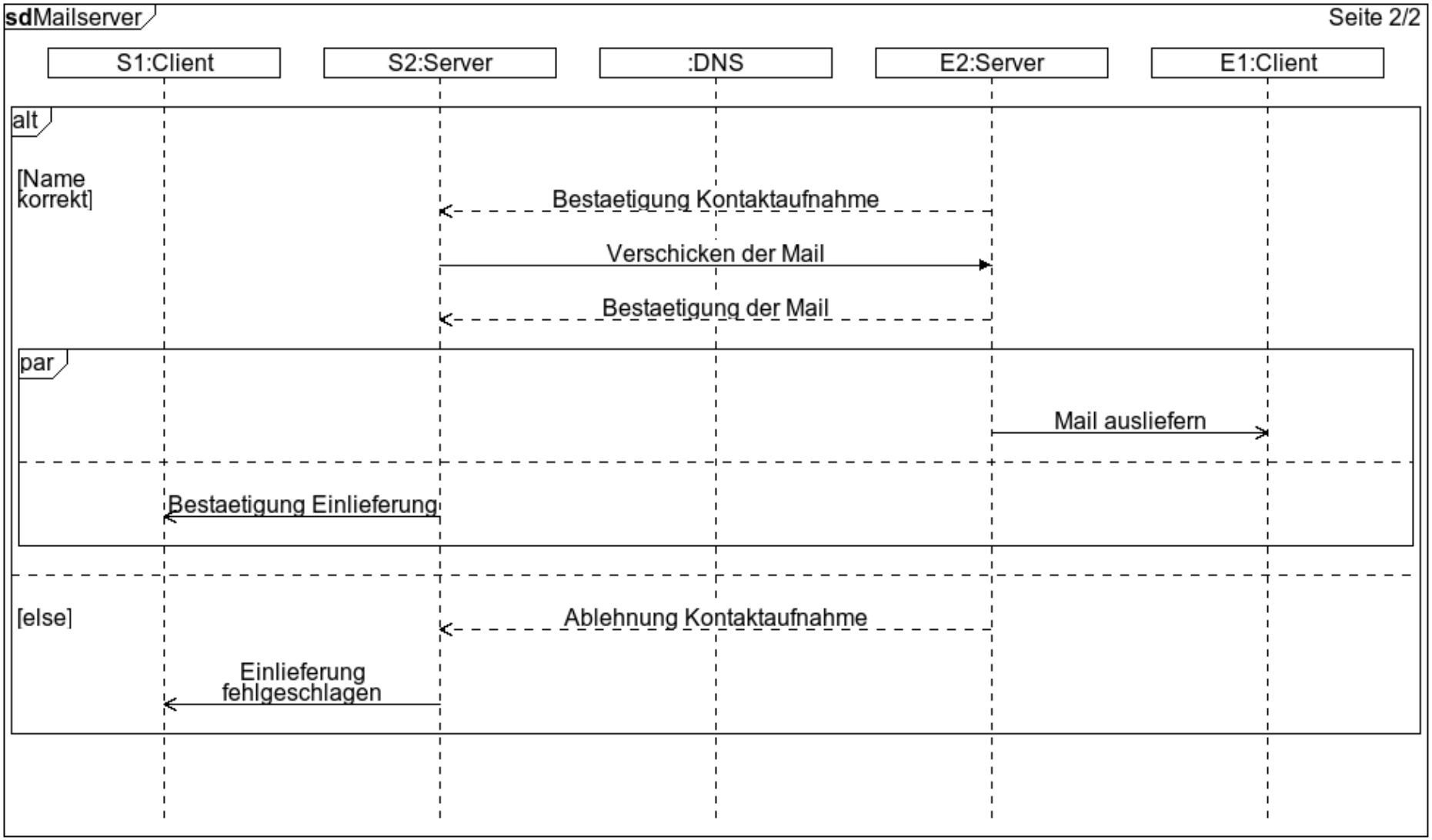
Sequenzdiagramme: Mailserver (Beispiel)

UNIVERSITÄT DUISBURG ESSEN
Offen im Denken



Sequenzdiagramme: Mailserver (Beispiel)

UNIVERSITÄT DUISBURG ESSEN
Offen im Denken



Sequenzdiagramme: Mailserver (Beispiel)

Bemerkung: Dieses Sequenzdiagramm ist an den Ablauf im SMTP-Protokoll angelehnt (SMTP = send mail transport protocol), ist jedoch erheblich vereinfacht.

Sequenzdiagramme zu vielen TCP/IP-Netzwerkprotokollen findet man unter:

`http://www.eventhelix.com/Realtimemantra/Networking/`

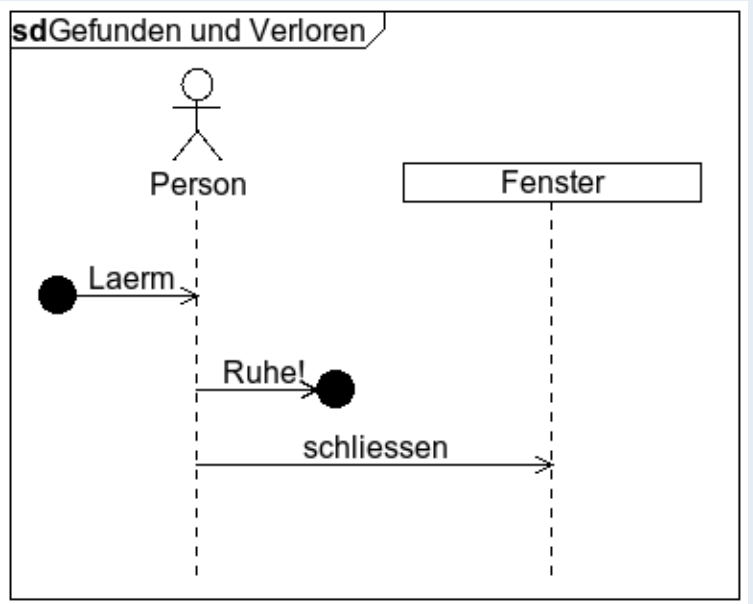
Sequenzdiagramme: weitere Arten von Nachrichten

Es gibt noch einige weitere Arten von Nachrichten:

Gefundene und verlorene Nachrichten

Bei gefundenen und verlorenen Nachrichten werden das Sende- bzw. das Empfangsereignis nicht explizit modelliert. Die Nachrichten tauchen quasi aus der Umgebung auf und verschwinden wieder dorthin.

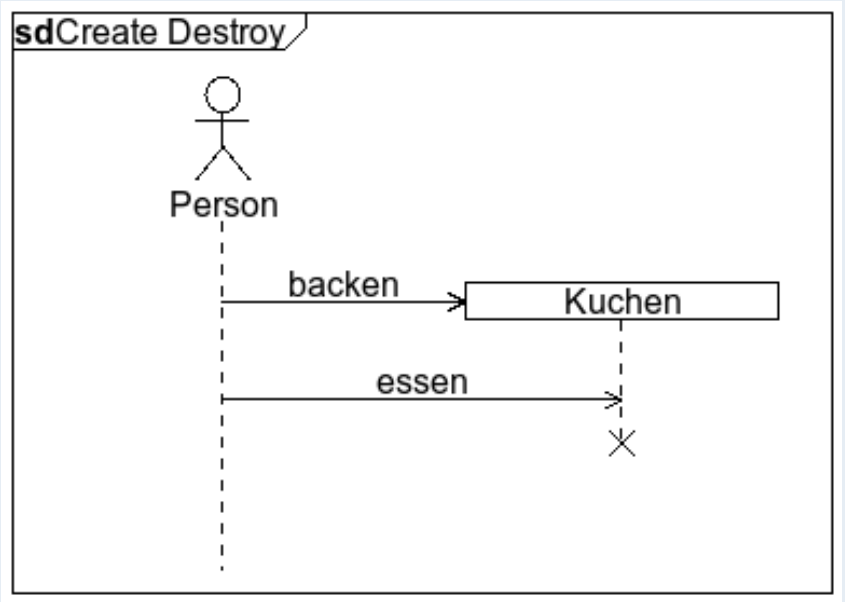
Solche Nachrichten werden benötigt, wenn die entsprechenden Kommunikationspartner nicht mitmodelliert werden.



Erzeugung und Löschung von Objekten

Außerdem kann es passieren, dass Objekte nicht während des ganzen Ablaufs zur Verfügung stehen. Sie können während des Ablaufs dynamisch erzeugt und wieder gelöscht werden.

Dies erfolgt zumeist durch sogenannte Erzeugungs- und Löschnachrichten und wird folgendermaßen dargestellt:



Anwendungsfalldiagramme

Anwendungsfalldiagramme

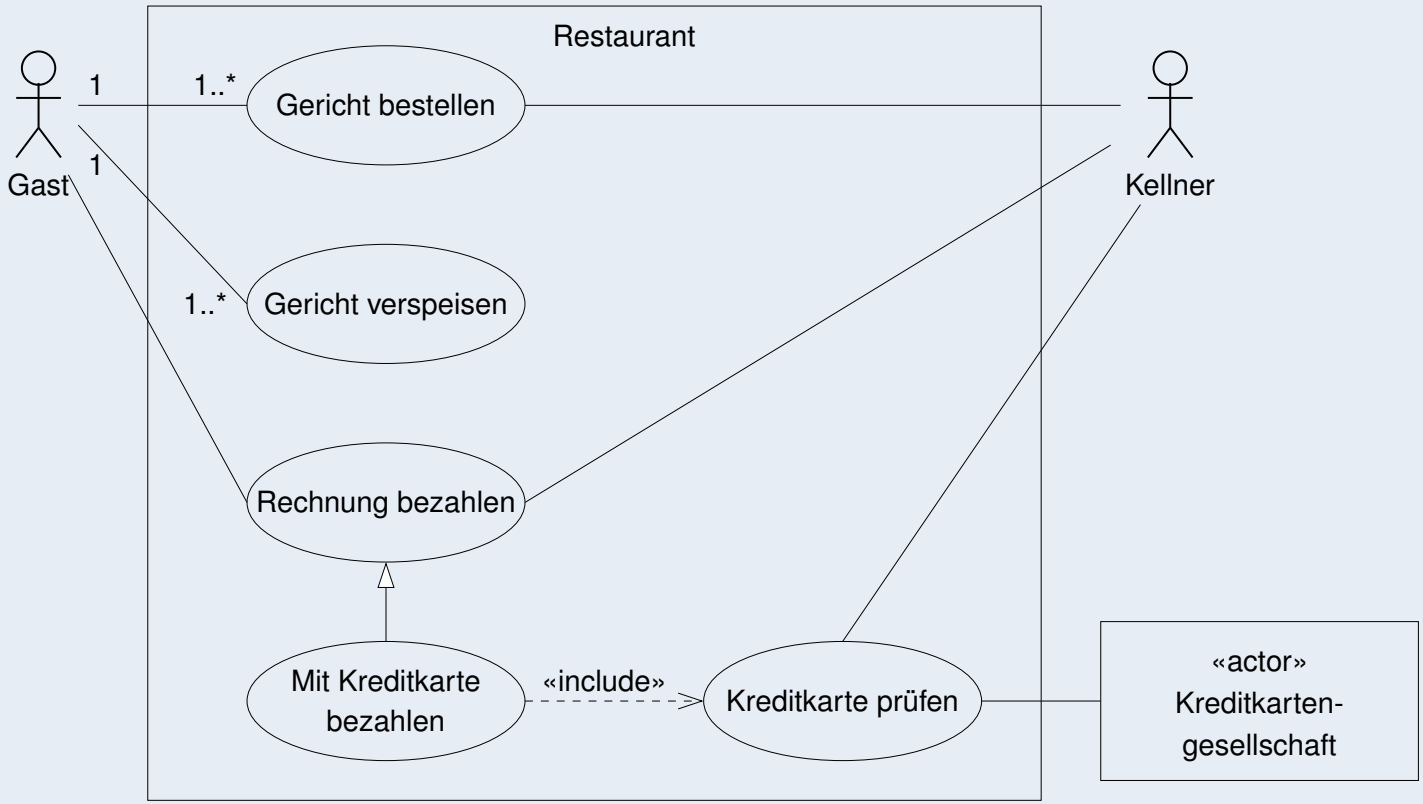
In frühen Stadien der Entwicklung und bei der Kommunikation mit dem Auftraggeber spielen auch Anwendungsfalldiagramme (engl. use case diagrams) eine große Rolle.

Anwendungsfalldiagramme modellieren die Funktionalität des Systems,

- auf einem hohen Abstraktionsniveau;
- aus der Black-Box-Sicht des Anwenders (das heißt, nur das von außen Sichtbare soll beschrieben werden, nicht die interne Realisierung);
- durch Spezifikation der Schnittstellen.

Die Anwender bzw. Nutzer tauchen als sogenannte Akteure in den Diagrammen auf.

Beispiel: Restaurant



Anwendungsfalldiagramme

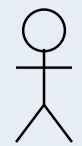
Anwendungsfalldiagramme bestehen aus im Folgenden beschriebenen Komponenten.

Systemgrenze

Die Systemgrenze ist ein Rechteck, das beschreibt, was sich außerhalb und was sich innerhalb des zu erstellenden Systems befindet.

Akteur

Ein Akteur ist ein Typ oder eine Rolle, die ein externer Benutzer oder ein externes System während der Interaktion mit dem System einnimmt. Menschliche Akteure werden durch Strichmännchen symbolisiert, andere Akteure durch ein Rechteck, das mit dem Schlüsselwort «actor» gekennzeichnet ist.

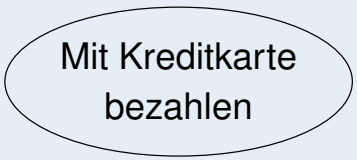


Gast

«actor»
Kreditkarten-
gesellschaft

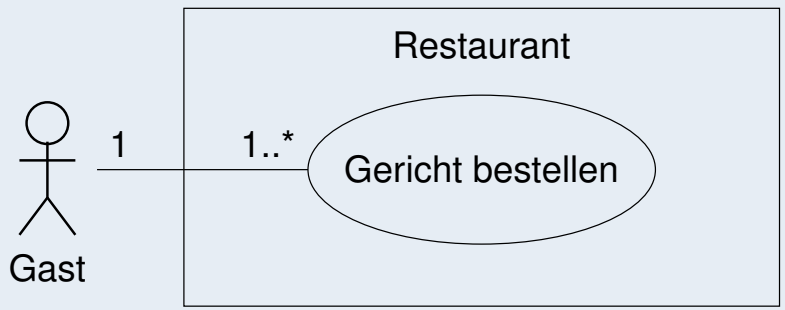
Anwendungsfall

Ein Anwendungsfall ist eine Menge von Interaktionsfolgen, die von dem System bereitgestellt werden und die einen Nutzen für einen oder mehrere Akteure bringen. (Das heißt, es handelt sich dabei um eine Art „Service“ des Systems.) Ein Anwendungsfall wird durch eine Ellipse dargestellt.



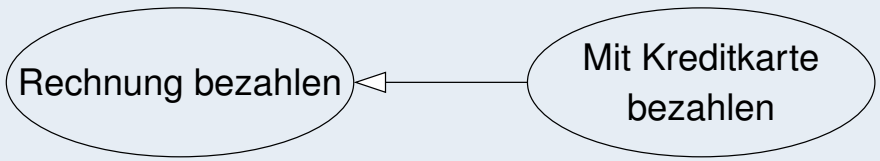
Assoziation

Wie bei Klassendiagrammen gibt es Assoziationen, vor allem zwischen Akteuren und Anwendungsfällen. (Welcher Akteur ist an welchem Anwendungsfall beteiligt?)
Assoziationen dürfen die üblichen Beschriftungen besitzen (Multiplizitäten etc.).



Generalisierung/Spezialisierung

Anwendungsfälle können andere Anwendungsfälle spezialisieren, das heißt, sie können von ihnen erben. Dabei werden – wie bei Klassen – auch alle Assoziationen geerbt.



Auch Spezialisierungsbeziehungen zwischen Akteuren sind möglich.

Include-Beziehung

Bei einer Include-Beziehung wird modelliert, dass ein Anwendungsfall die Funktionalität eines anderen Anwendungsfalles auf jeden Fall beinhaltet. Das heißt, der zweite Anwendungsfall wird immer als eine Art „Unterprozedur“ aufgerufen.

Es gibt auch sogenannte Extend-Beziehungen, bei denen ein Anwendungsfall nur unter bestimmten Bedingungen in einen anderen Anwendungsfall eingebunden wird.

Anwendungsfalldiagramme

UNIVERSITÄT DUISBURG ESSEN
Offen im Denken

Bemerkungen:

- Anwendungsfalldiagramme sollten nicht zu viele Details enthalten.
- Sie sind ein einfaches Mittel, um Anwenderwünsche zu diskutieren, und sollten nur eine grobe Sicht auf die Funktionalität des Systems darstellen.
- Bei Bedarf müssen bestimmte Anwendungsfälle dann noch textuell oder mit Hilfe anderer UML-Diagramme genauer beschrieben werden.

Weitere UML-Diagramme

Überblick über weitere UML-Diagramme

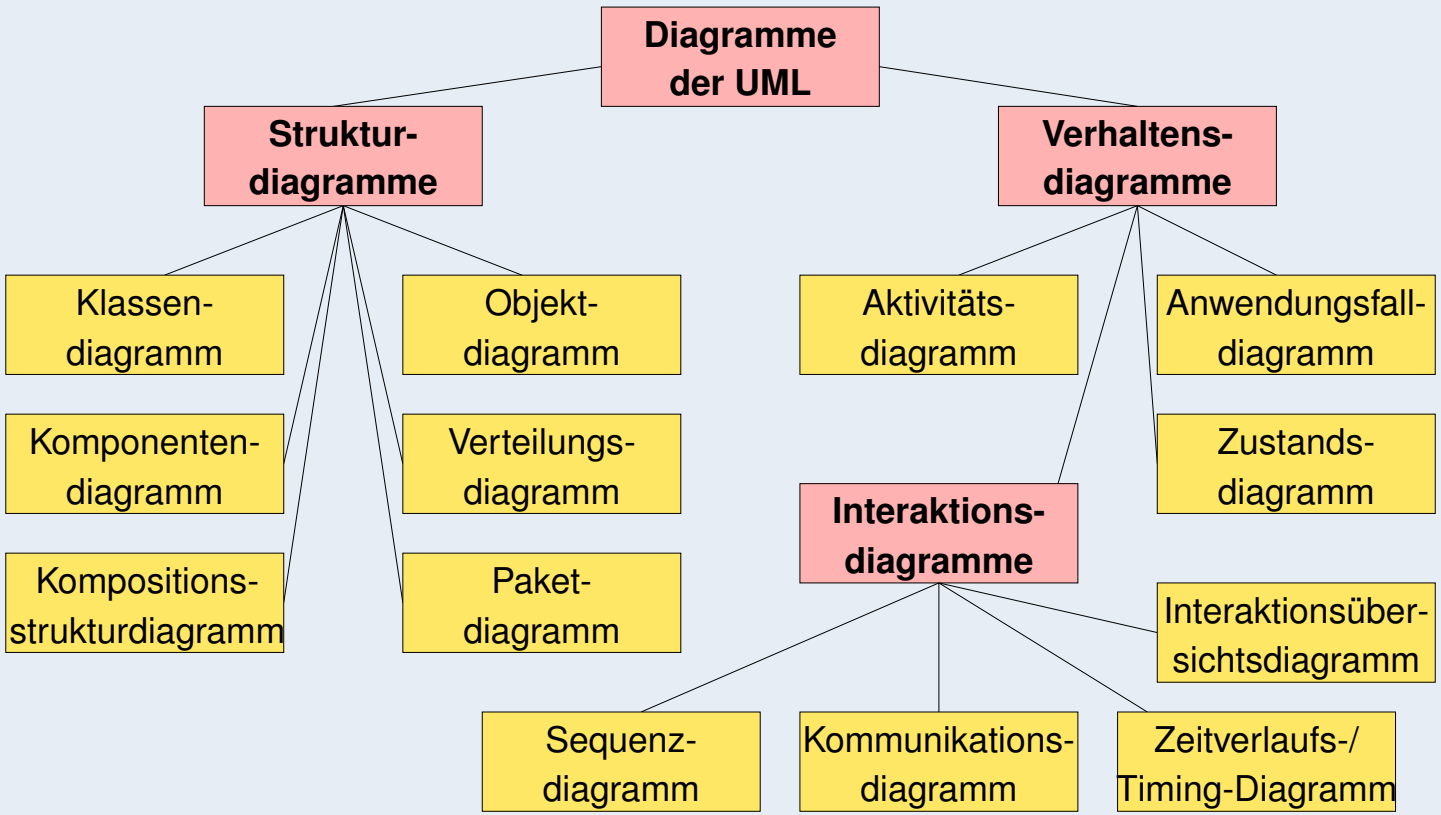
UNIVERSITÄT DUISBURG ESSEN
Offen im Denken

Wir schließen die Vorlesung mit einem kurzen Überblick über die noch fehlenden Typen von UML-Diagrammen ab.

Zuletzt gibt es dann noch ein paar Abschlussbemerkungen.

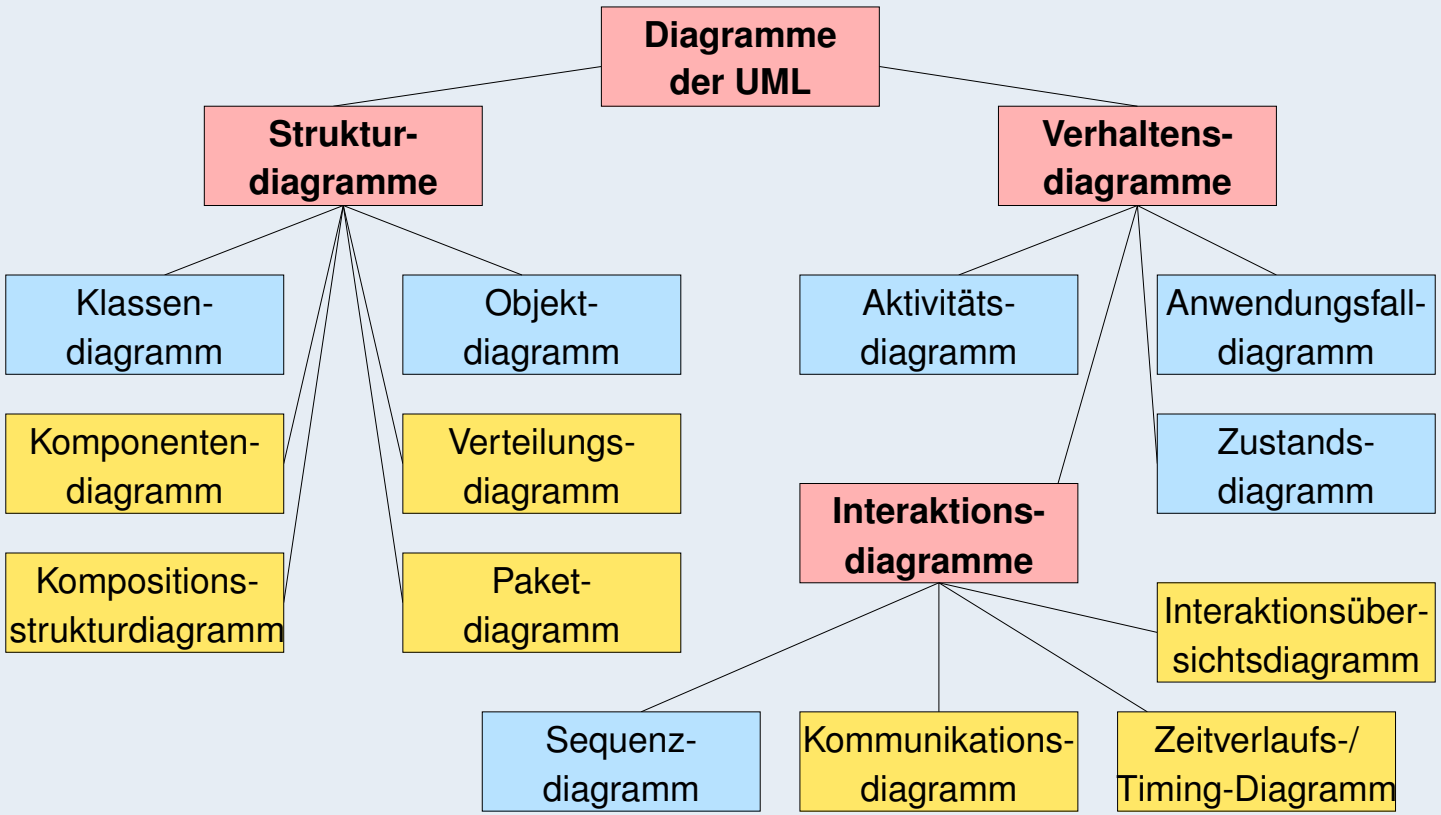
Überblick über weitere UML-Diagramme

UML-Diagramme ▸ Liste



Überblick über weitere UML-Diagramme

UML-Diagramme (blau: bereits behandelte Diagramme)



Überblick über weitere UML-Diagramme

Wir beginnen zunächst mit den drei noch fehlenden Arten von Interaktionsdiagrammen:

- Kommunikationsdiagramme
- Timing-Diagramme bzw. Zeitverlaufsdiagramme
- Interaktionsübersichtsdiagramme

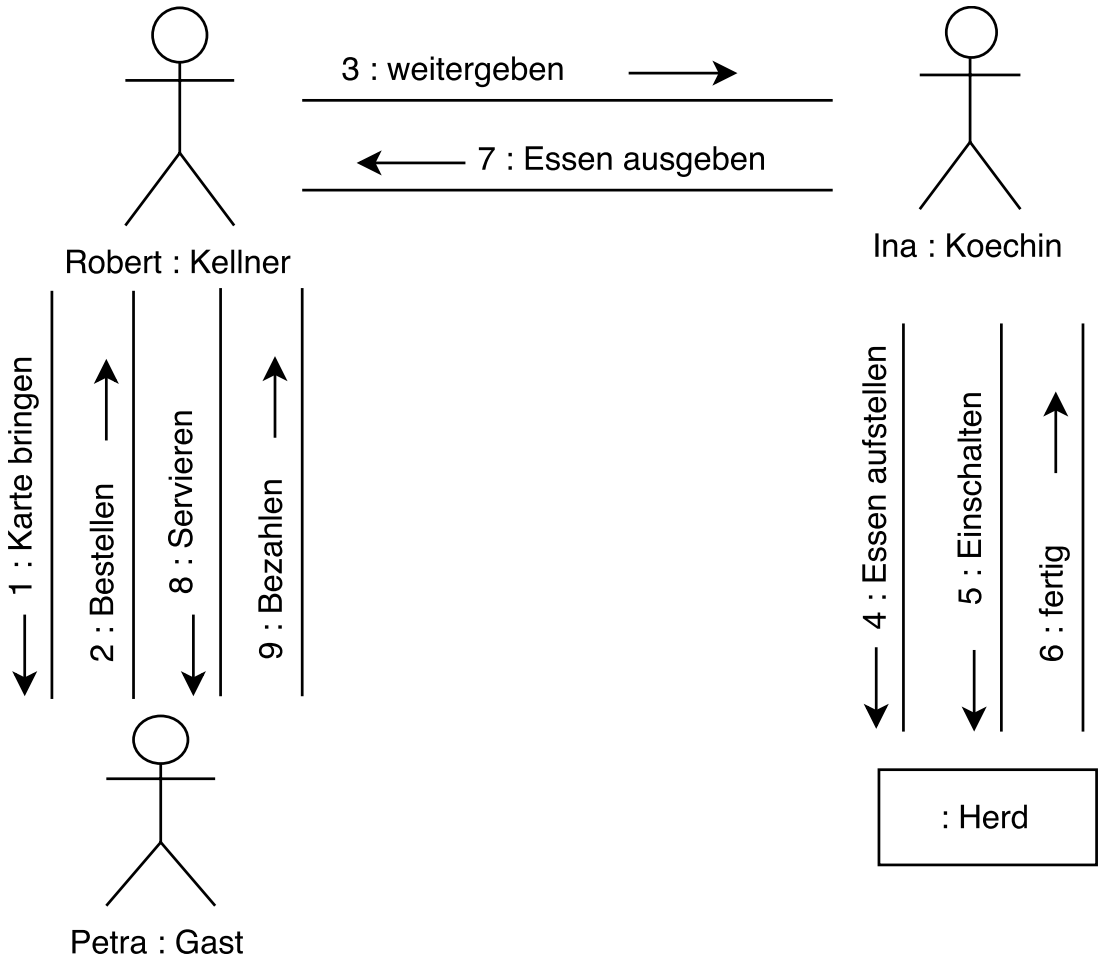
Kommunikationsdiagramme

Kommunikationsdiagramme enthalten dieselbe Information wie Sequenzdiagramme, werden jedoch anders dargestellt.

Während bei Sequenzdiagrammen der Fokus eher auf dem zeitlichen Ablauf liegt, heben Kommunikationsdiagramme eher die Kommunikationsbeziehungen der Teilnehmer hervor.

Kommunikationsdiagramme gehören – genau wie Sequenzdiagramme – zur Klasse der Interaktionsdiagramme.

Das Sequenzdiagramm Restaurantbesuch ▸ Diagramm wird folgendermaßen als Kommunikationsdiagramm dargestellt:



Kommunikationsdiagramme

Dabei werden die Kommunikationspartner wie bisher durch Rechtecke oder Strichmännchen dargestellt. Es wird jedoch kein zeitlicher Ablauf mehr dargestellt.

Die Interaktionen bzw. Nachrichten werden durch Linien notiert, an denen die Namen der Nachrichten und die Senderichtung ($\rightarrow$) stehen.

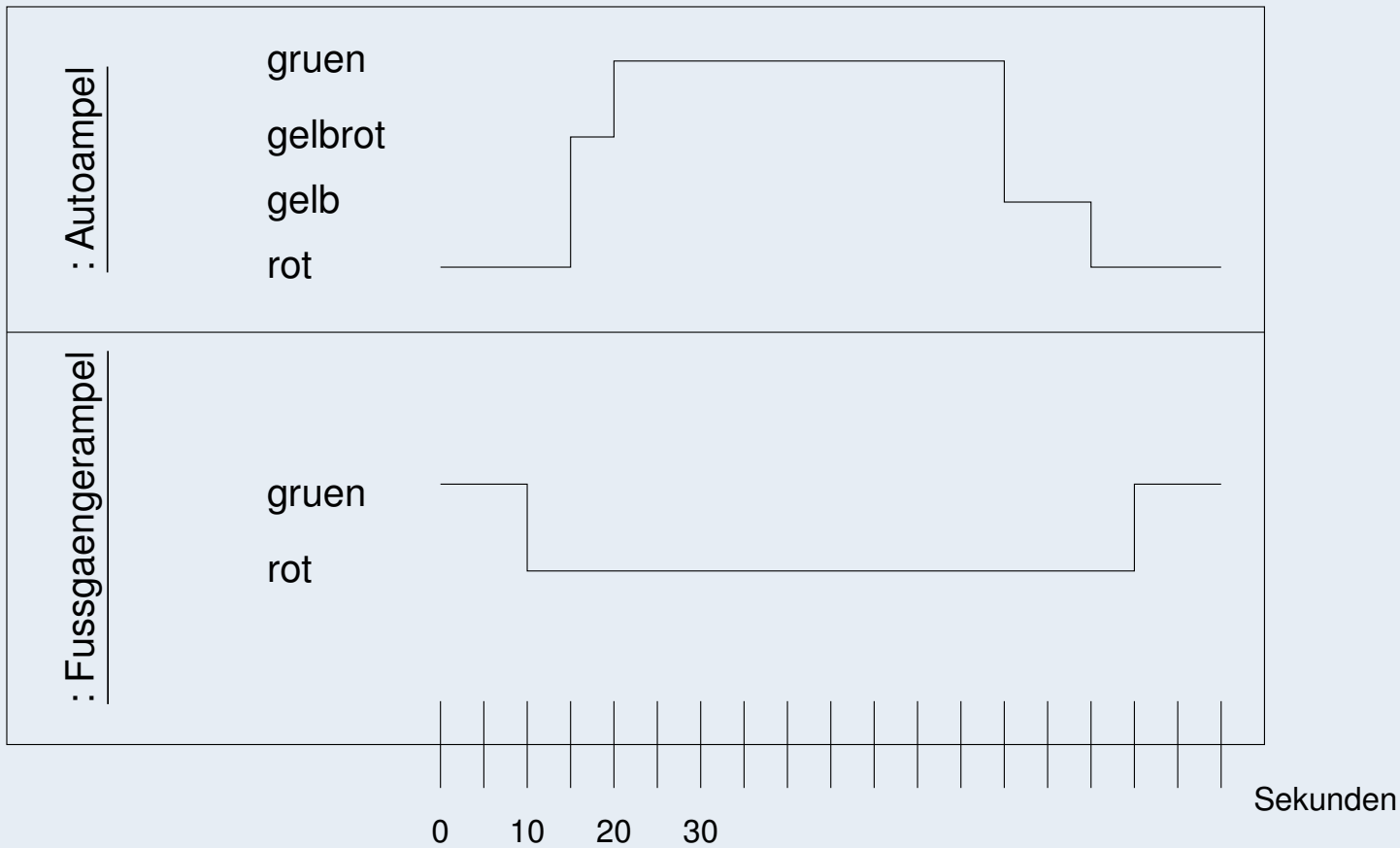
Die Nummerierung der Nachrichten (1, 2, 3, . . .) gibt die Reihenfolge an. Durch Buchstaben hinter den Nummern (2a, 2b, . . .) beschreibt man parallele Nachrichten, die in beliebiger Reihenfolge angeordnet sein können.

Timing-Diagramme

Timing-Diagramme sind in der Elektrotechnik weit verbreitet und zeigen an, zu welchem Zeitpunkt welcher Kommunikationspartner welchen Zustand einnimmt.

Dabei wird von links nach rechts (horizontal) die Zeit aufgetragen und von oben nach unten werden die Kommunikationspartner und deren Zustände aufgetragen.

Beispiel: Ampelschaltung



Interaktionsübersichtsdiagramme

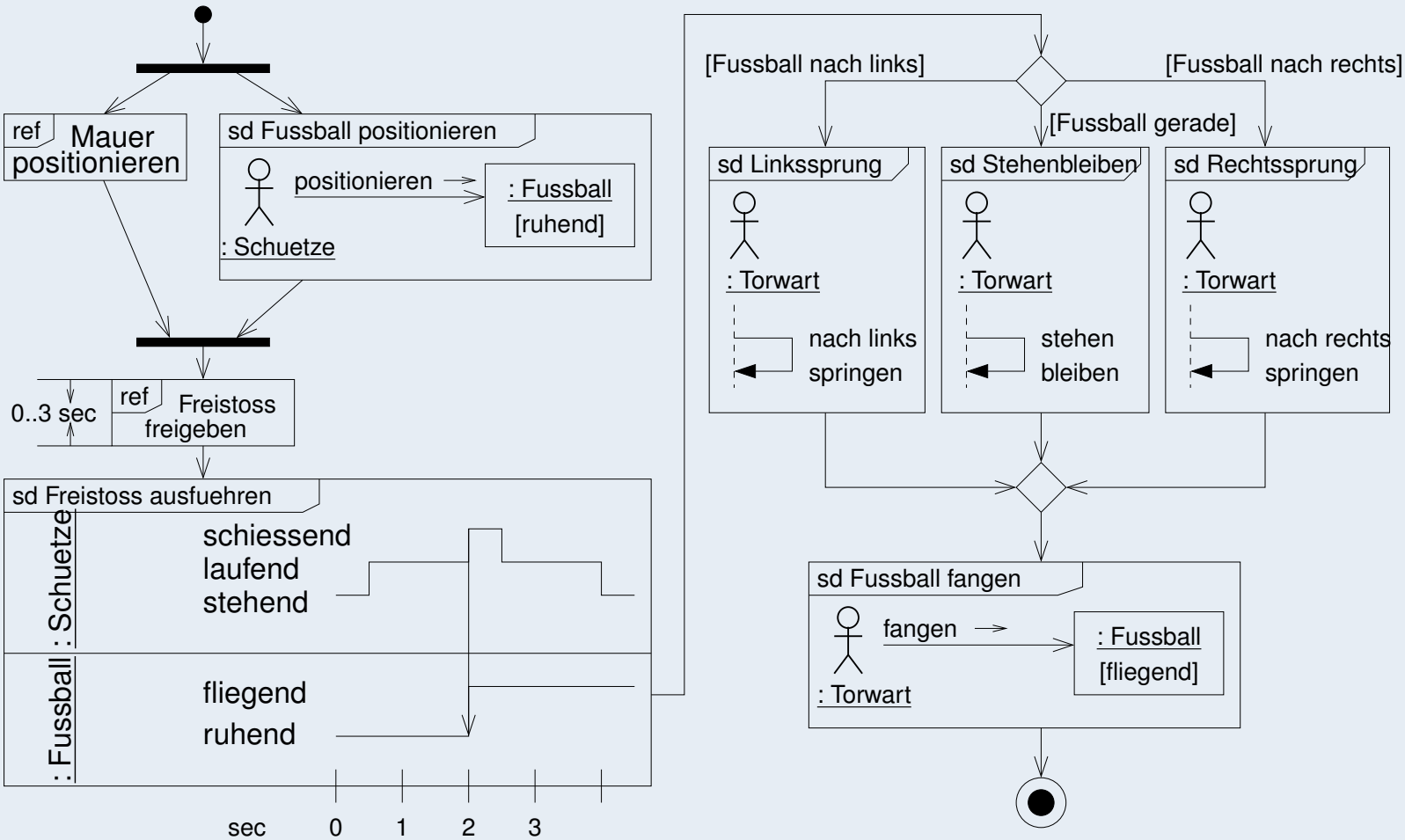
UNIVERSITÄT DUISBURG ESSEN
Offen im Denken

Interaktionsübersichtsdiagramme sind im Wesentlichen Aktivitätsdiagramme, die – anstatt der Aktionen – hierarchisch weitere Interaktionsdiagramme (Sequenzdiagramme, Kommunikationsdiagramme, Timing-Diagramme) enthalten können.

Sie werden eingesetzt, wenn es eine größere Menge verschiedener Arten von Aktionen gibt, über die man sonst nur schwer den Überblick behalten kann.

Als Beispiel betrachten wir ein Interaktionsübersichtsdiagramm, das den Ablauf eines Freistoßes in einem Fußballspiel modelliert.

Interaktionsübersichtsdiagramme



Interaktionsübersichtsdiagramme

UNIVERSITÄT DUISBURG ESSEN
Offen im Denken

Bemerkungen:

- Interaktionsreferenzen (Schlüsselwort ref) werden verwendet, um an anderer Stelle definierte Interaktionsdiagramme wiederzuverwenden.
- Anstatt der Aktionen wie in Aktivitätsdiagrammen werden ganze Interaktionsdiagramme verwendet, die alle sd als Diagrammtyp für Interaktionsdiagramme erhalten.

Überblick über weitere UML-Diagramme

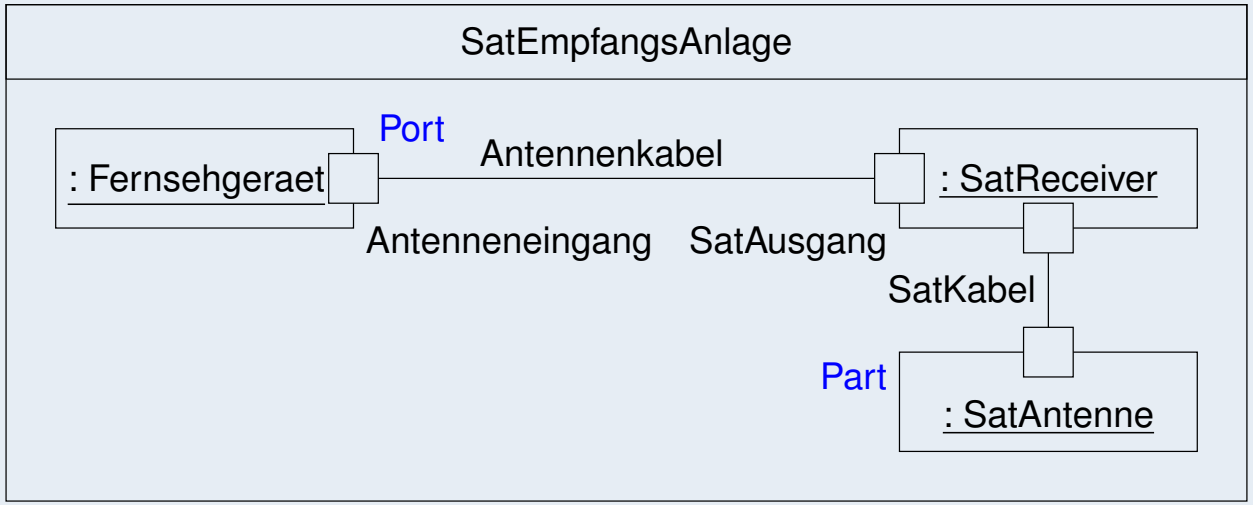
Wir sehen uns nun noch (ganz kurz) die fehlenden vier Typen von Strukturdiagrammen an. ▸ UML-Übersicht

- Kompositionsstrukturdiagramme
- Paketdiagramme
- Verteilungsdiagramme
- Komponentendiagramme

Im weitesten Sinne dienen sie alle dazu, die (übergeordnete) Struktur bzw. Architektur eines Systems darzustellen.

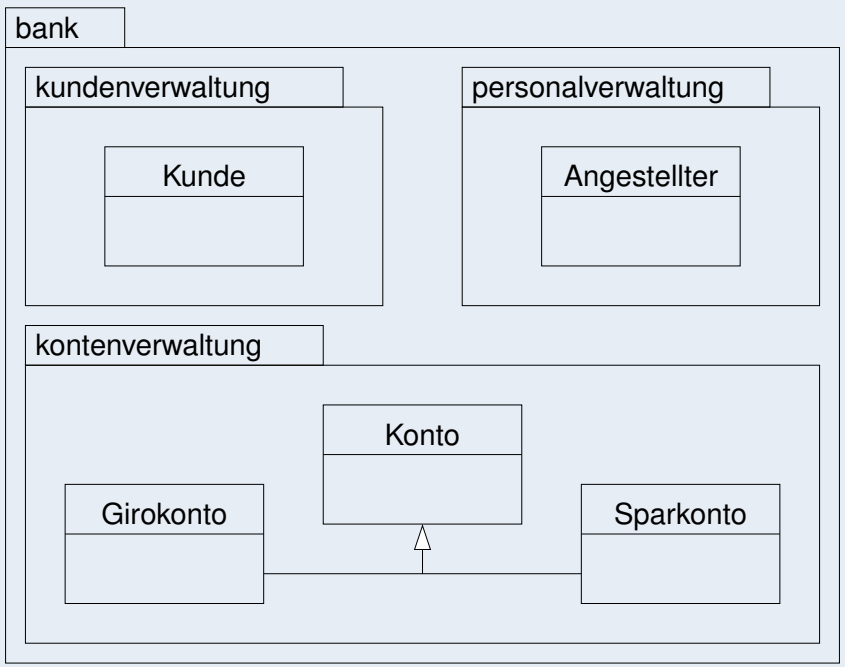
Kompositionsstrukturdiagramme

Kompositionsstrukturdiagramme (engl. composite structure diagrams) beschreiben die interne Struktur von Komponenten (z.B. Klassen) in einer White-Box-Darstellung. Instanzen von durch Komposition verbundenen Klassen werden dabei als sogenannte Parts innerhalb der Klasse dargestellt. Ports, dargestellt als kleine Quadrate, sind Verbindungsstellen zwischen Parts und beinhalten Schnittstellen.



Paketdiagramme

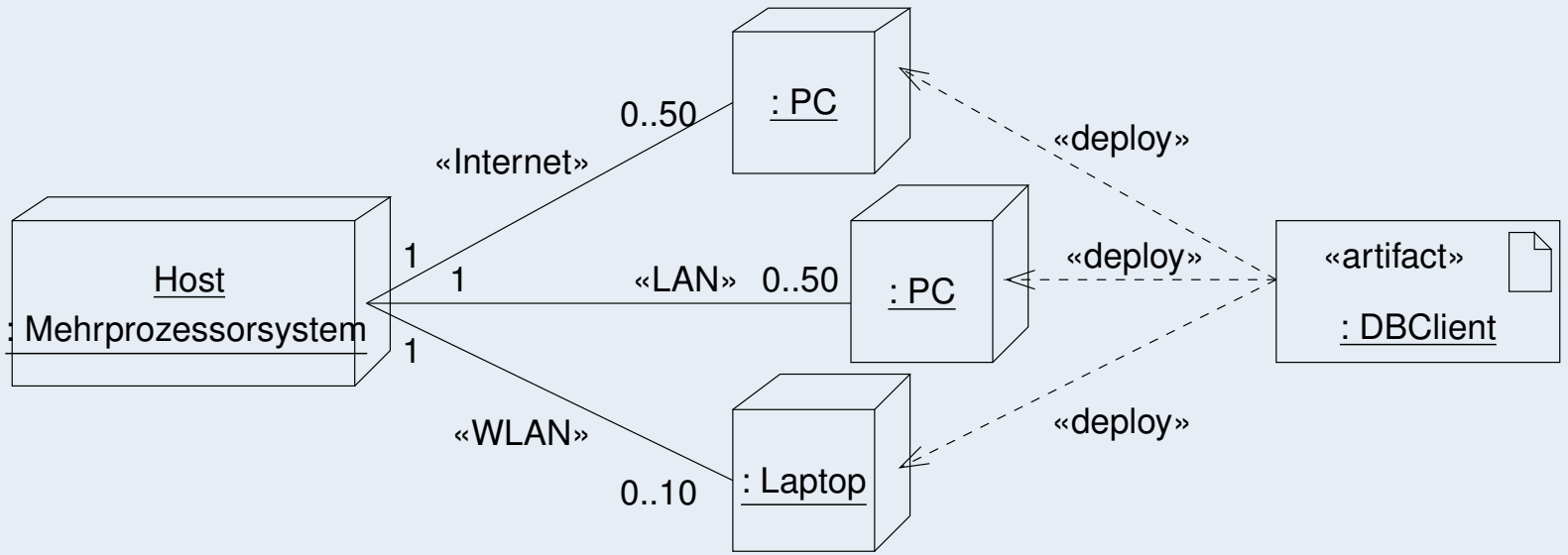
Ein großes Softwaresystem muss in Pakete bzw. Module gegliedert werden. Paketdiagramme (engl. package diagrams) beschreiben die statische Struktur eines großen Systems durch Zusammenfassen von Klassen in Paketen.



Verteilungsdiagramme

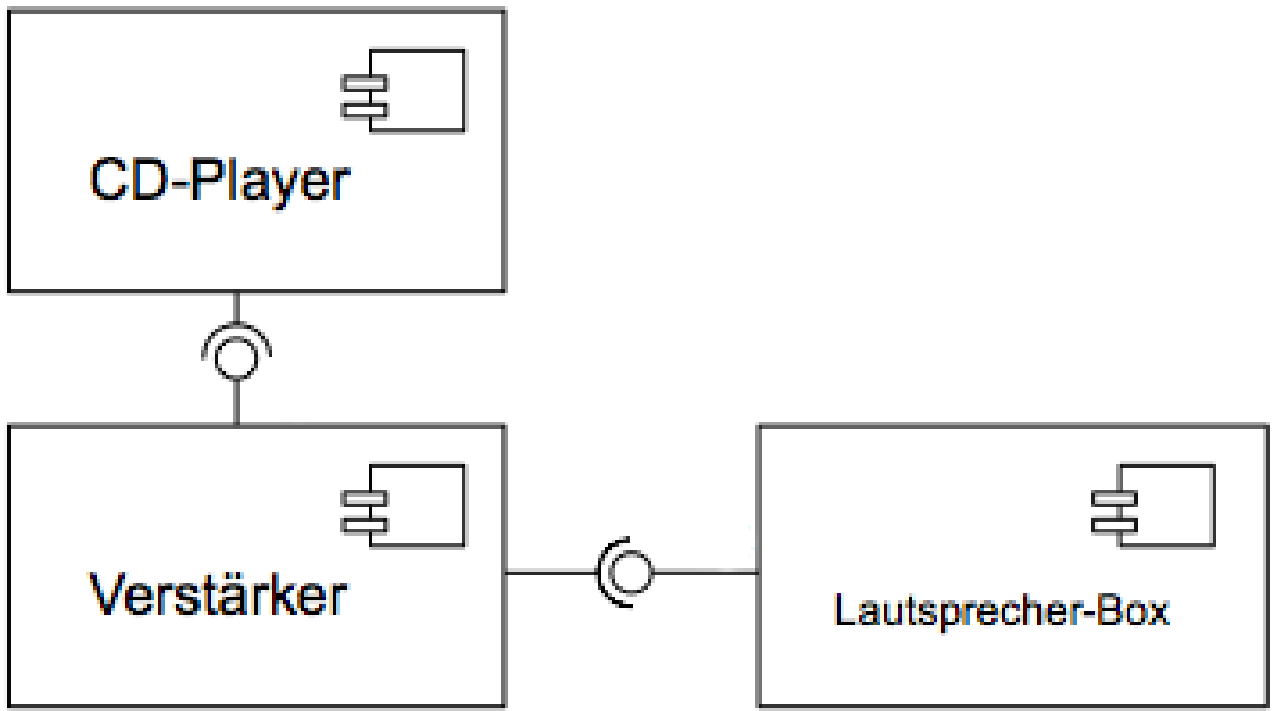
Verteilungsdiagramme (engl. deployment diagrams) beschreiben die Hardware-Komponenten, die in einem System benutzt werden, und wie diese in Beziehung stehen.

Sie können auch konkrete physische Informationseinheiten (Dateien, etc.) enthalten, die als Artefakte bezeichnet werden.



Komponentendiagramme

Ein Komponentendiagramm (engl. component diagram) beschreibt im Gegensatz dazu die Software-Architektur eines Systems. Es beantwortet die Fragen, wie Klassen zur Laufzeit zu größeren Komponenten zusammengefasst werden, und welche Schnittstellen (Services) angeboten und genutzt werden.



Abschließende Bemerkungen

UML als semi-formale Modellierungssprache

UML ist eine semi-formale Modellierungssprache, das heißt, nicht bei jedem Sprachelement gibt es vollständige Einigkeit über die Bedeutung.

Trotz dieser Kritik ist die UML ein großer Fortschritt gegenüber früher genutzten Modellierungssprachen, da sie vereinheitlichte Diagramme bereitstellt, die von jedem Beteiligten am Softwareentwicklungsprozess verstanden werden können.

Abschließende Bemerkungen

Konsistenz von Modellen

Bei der Beschreibung eines großen Systems durch mehrere Modelle ist auch darauf zu achten, dass die Modelle untereinander konsistent sind. (Beispielsweise: Klassennamen, die in einem Sequenzdiagramm verwendet werden, tauchen auch im Klassendiagramm auf.)

Es gibt Ansätze, solche Konsistenz (halb-automatisch) zu erreichen, indem man sogenannte Modelltransformationen durchführt und verschiedene Diagramme ineinander übersetzt.

Abschließende Bemerkungen

Es gibt noch viele Aspekte in der UML, die wir nicht betrachtet haben.

Weitergehende Informationen über UML gibt es in zahllosen Büchern (siehe Literaturliste) und auf den Seiten der OMG (Object Management Group):

```
http://www.omg.org/technology/documents/formal
/uml.htm
```

Und natürlich gibt es neben UML und Petrinetzen noch zahlreiche andere Modellierungssprachen!