Methodische und Praktische Grundlagen der Informatik (MPGI 3) WS 2008/09

Softwaretechnik

Steffen Helke

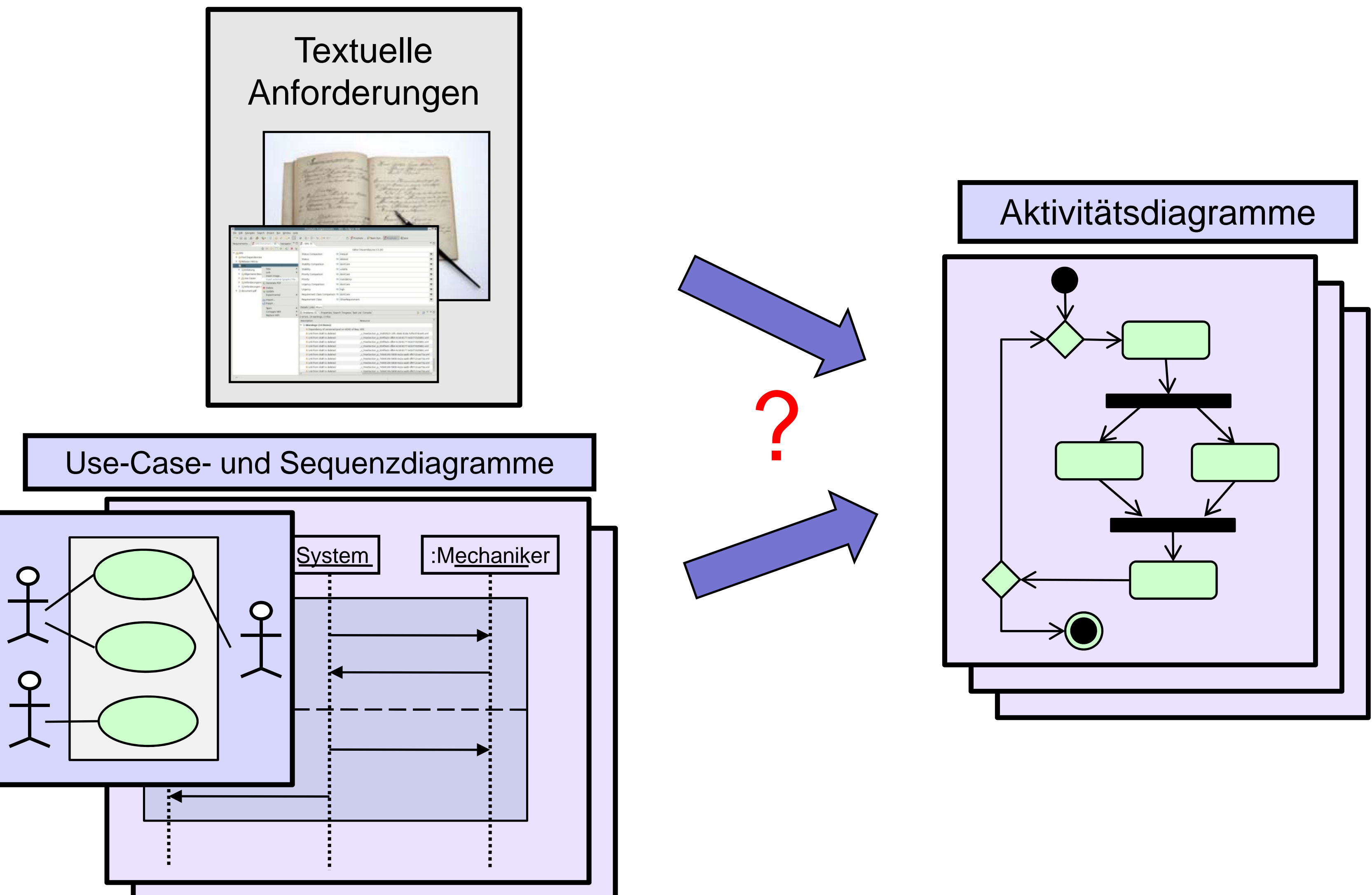
Andreas Mertgen (Organisation)

Rojahn Ahmadi, Georgy Dobrev, Daniel Gómez Esperón, Simon Rauterberg, Jennifer Ulrich (Tutoren)

Was machen wir heute?

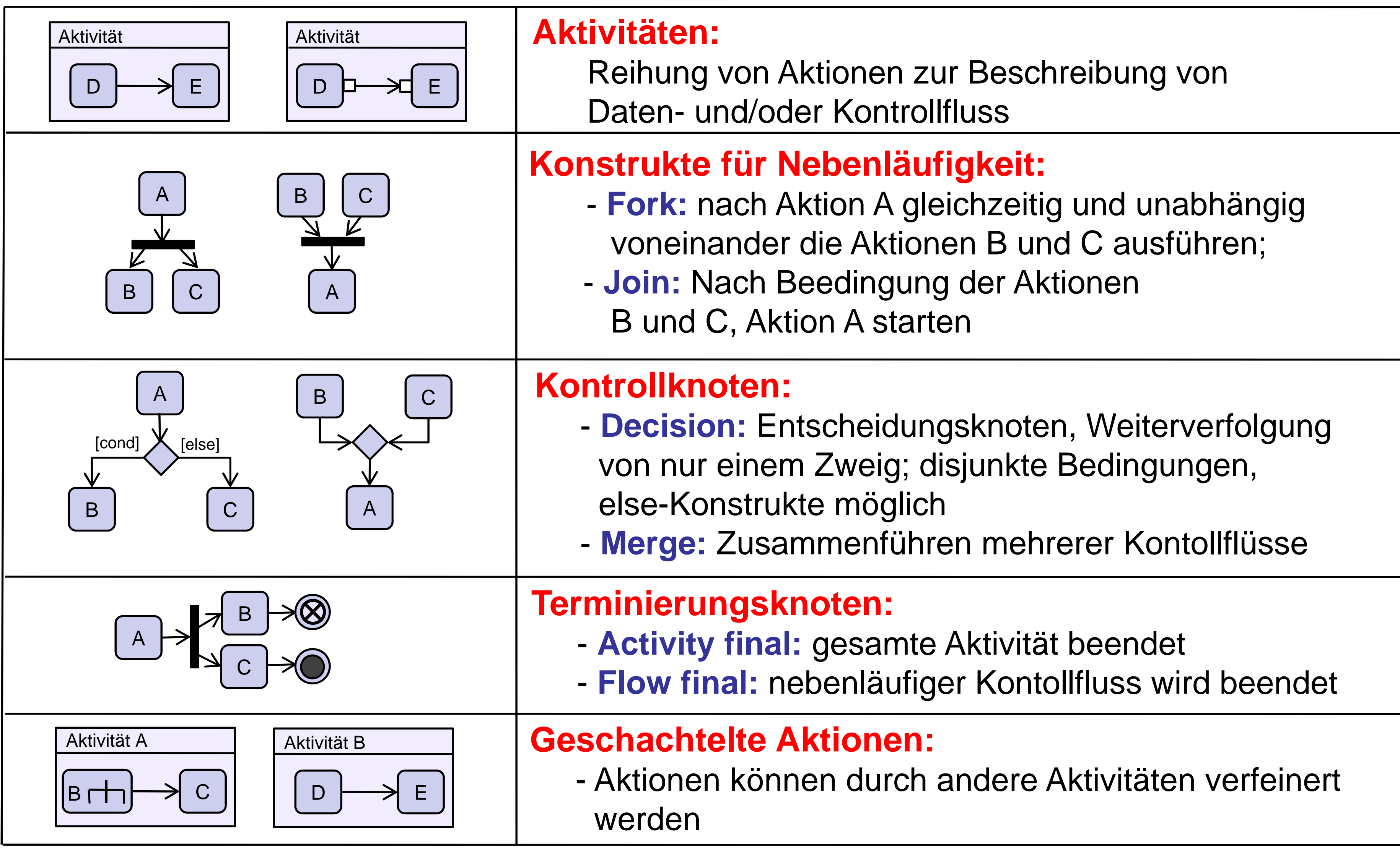
- **Wiederholung**
  - Aktivitätsdiagramme
  - Systemklassenmodell
- **Motivation**
  - Hoare Kalkül
  - **Design by Contract**
- **Operationsschema**
  - **Vor- und Nachbedingungen für Systemoperationen**

Wiederholung: Aktivitätsdiagramme

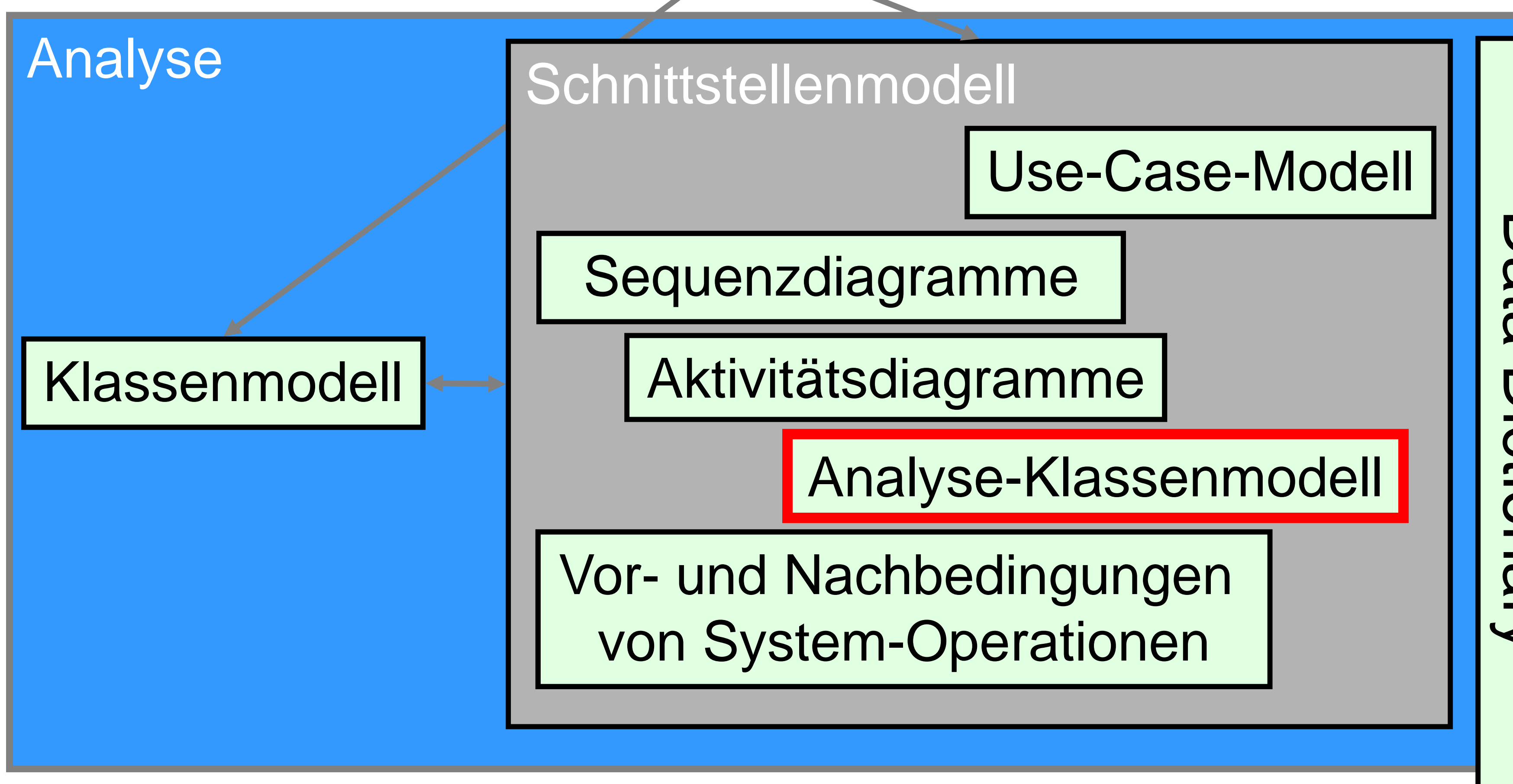


Analyse, 4. Schritt: Aktivitätsdiagramme

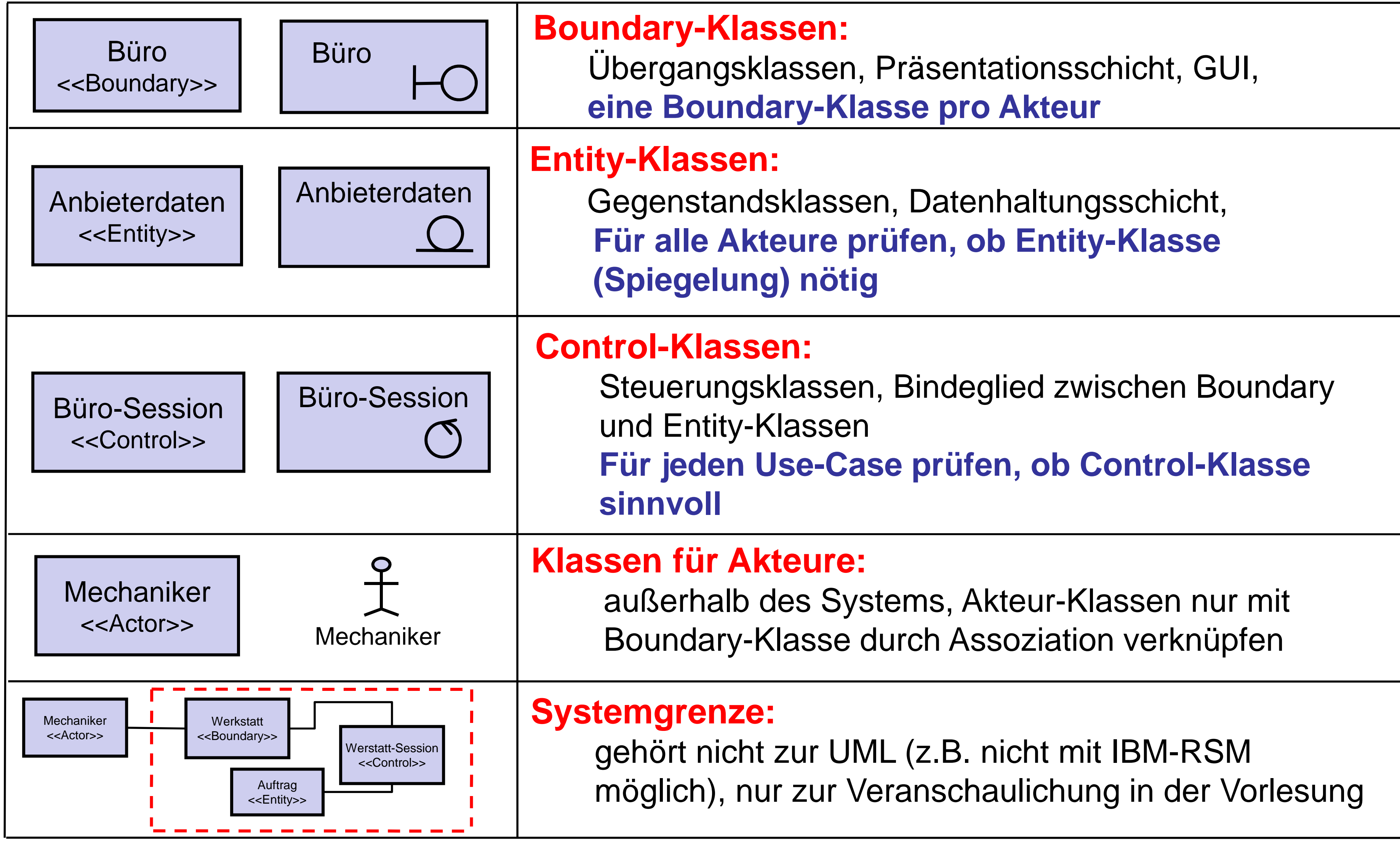
Notationen für Aktivitätsdiagramme



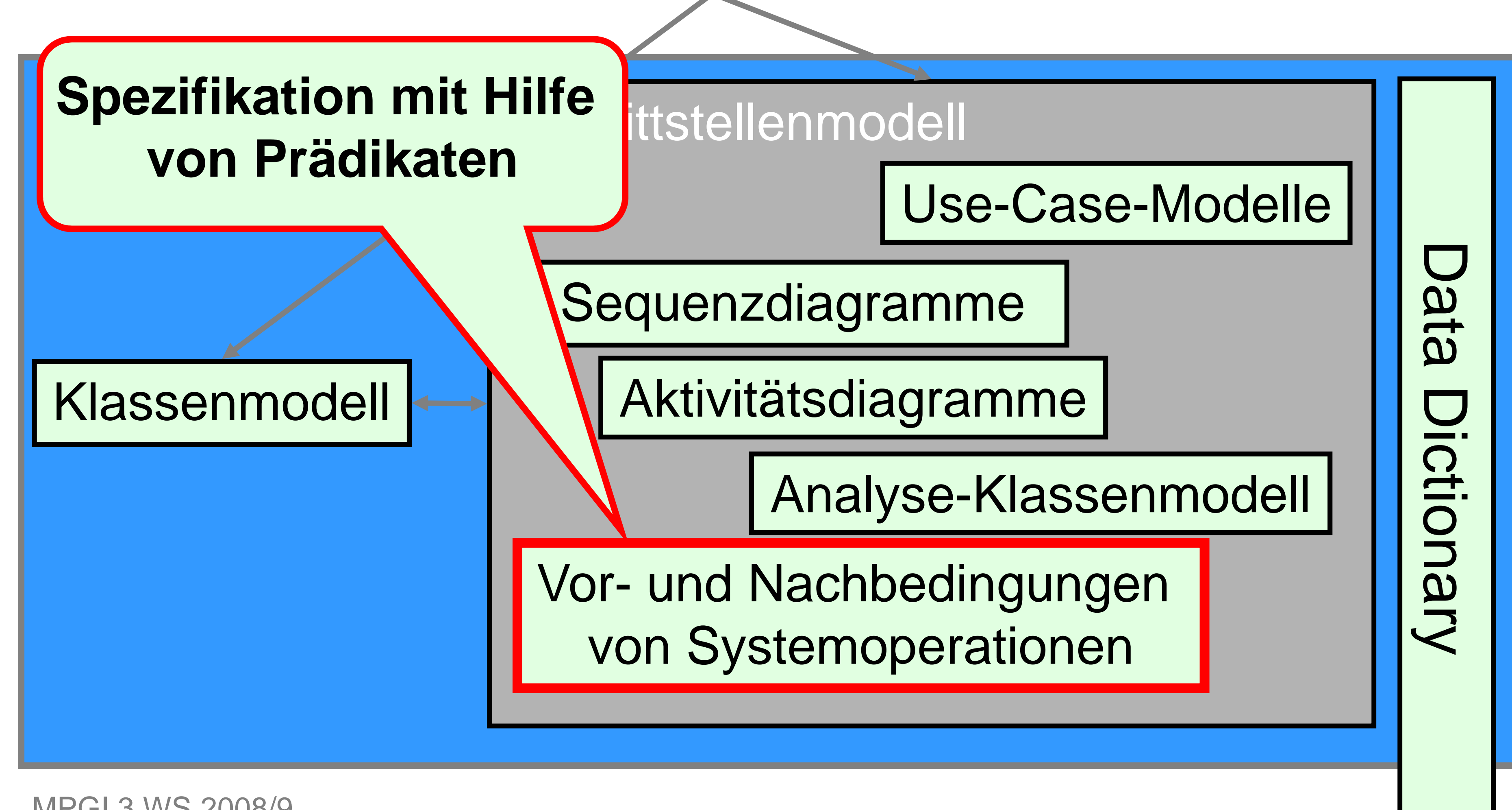
Analyse, 5. Schritt: Systemklassenmodell



Notationen für Systemklassenmodelle



Analyse, letzter Schritt: Vor- und Nachbedingungen von Systemoperationen



Beschreibungen mit Vor- und Nachbedingungen

- Idee aus **MPGI1/2** bekannt: **Hoare-Kalkül** aus dem Jahre 1969

$$\{ \mathbf{P} \} \; \mathbf{S} \; \{ \mathbf{Q} \}$$

  - **S**: Operation (Anweisung des Programms)
  - **P**: Vorbedingung
  - **Q**: Nachbedingung

$$\{ x = y \} \; y := y - x + 1 \; \{ y = 1 \}$$

Verträge bei B. Meyer: Design by Contract

- Bertrand Meyer (1997)

  „Object-Oriented Software Construction“

- Anpassung des Konzepts von Vor- und Nachbedingungen für Klassen und Methoden

Verträge, wie im täglichen Leben aushandeln!

- Vertragspartner im täglichen Leben
  - Kunden
  - Verkäufer/Dienstleister
- Vorteile und Verpflichtungen für beide Parteien

Darstellung *Design by Contract* nach Maritta Heisel (URL: http://swe.uni-duisburg-essen.de/de/members/heisel/)

Vertragsmodelle im täglichen Leben

Stefan Jähnichen möchte von Berlin nach Tokio

Vertrag	Verpflichtungen	Vorteile
Fluggast	• Flugticket kaufen • rechtzeitig zum Abflug da sein ⇨ **Vorbedingung**	• kann in Tokio Kooperations-partner treffen ⇨ **profitiert von Nachbedingung**
Fluggesellschaft	• Fluggast nach Tokio fliegen ⇨ **Nachbedingung**	• macht Umsatz • braucht Jähnichen nur bei Zahlung mitnehmen ⇨ **darf Vorbedingung voraussetzen**

Sinn und Nutzen expliziter Verträge

Bertrant Meyer

„A contract document protects both the client, by specifying how much should be done, and the supplier, by stating that the supplier is not liable for failing to carry out tasks outside of the specified scope.“

Definition: Kontrakte für Software

Ein Kontrakt ist eine formale Übereinkunft zwischen einem System / einer Klasse und seinen / ihren Akteuren / Klienten, die für beide Seiten Rechte und Pflichten festlegt.

Beispiel: Klasse für einen generischen Stack

class	Stack [T]
attribute	nb_elements: integer
method	create: Stack [T]
	mt(s: Stack [T]): Boolean
	full(s: Stack [T]): Boolean
	push(s: Stack [T], x:T)
	pop(s: Stack [T])
	top(s: Stack [T]): T
end class	Stack [T]

Vor- und Nachbedingungen Stack-Operationen

mt(s: Stack [T]**)**

pre true

post Result = true ⇔ nb_elements = 0

full(s: Stack [T]**)**

pre true

post Result = true ⇔ nb_elements = …

push(s: Stack [T], **x:T)**

pre **not** full

post **not** empty **and** nb_elements‘ = nb_elements +1
and „x ist neues oberstes Element des Stack“

Vor- und Nachbedingungen Stack-Operationen

pop(s: Stack [T]**)**

pre **not** empty

post **not** full **and** nb_elements‘ = nb_elements - 1

top(s: Stack [T]**)**

pre **not** empty

post **noChange and**
Result = „oberstes Element des Stack“

create: Stack [T]

pre **true**

post nb_elements = 0

Verpflichtungen und Vorteile im Stack-Beispiel

Vertrag	Verpflichtungen	Vorteile
Client	• Rufe **push(x)** nur auf, wenn Stack nicht voll ist. ➡ **Vorbedingung**	• Element x wird auf Stack gelegt, **top(x)** ergibt x, nb_elements wird um 1 erhöht. ➡ **profitiert von Nachbedingung**
Server	• Stelle sicher, dass x auf den Stack gelegt wird. ➡ **Nachbedingung**	• unnötig, den Fall zu behandeln, bei dem der Stack voll ist. ➡ **darf Vorbedingung voraussetzen**

Einsatzgebiete von Design bei Contract

1. Analyse

Spezifikation von Systemoperationen (Vertrag zwischen System und Akteuren)

2. Entwurf

Spezifikation aller wichtigen systeminternen Methoden/Operationen (Vertrag zwischen unterschiedlichen Systemkomponenten)

3. Implementierung

Vor- und Nachbedingungen zur Dokumentation des Codes oder mit dynamischer Überprüfung

Vertragsmodelle in Programmiersprachen

1. Explizite Unterstützung

- Eiffel von Bertrand Meyer
- Sather von Steve Omuhundro
- Lisaac, SPARK (Ada) u.a.

2. Eingeschränkte Umsetzung in Java

- Erweiterungen für Java
  - JML (Java Modeling Language)
  - JASS (Java with Assertions) u.a.
- Java ab Version 1.4 direkt

Design by Contract im Code wird auch als Defensive Programmierung bezeichnet!

Verifikation von C-Programmen

European Microsoft Innovation Center

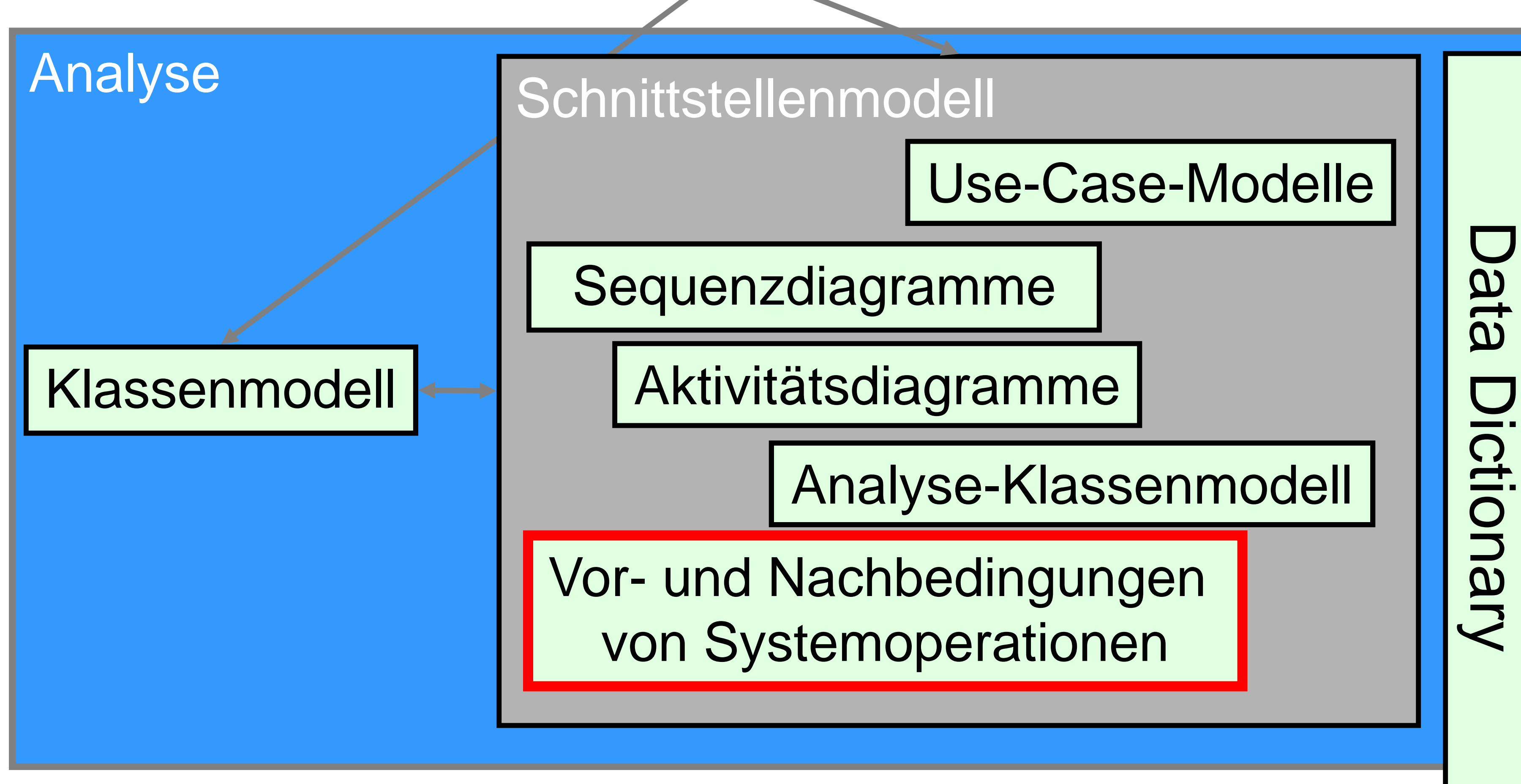
EMIC @ Aachen

- Aktuelle Forschungsarbeiten zur Behandlung von Contracts in C-Programmen
- Ehemaliger Mitarbeiter des Lehrstuhls Softwaretechnik (Thomas Santen)

Vertiefungsveranstaltung im Sommersemester bei SWT (VCC)

- Anwendungsbeispiel Hyper-Visor (Windows Server 2008)
- Virtualisierungsplattform (50000 lines of C, 5000 lines of Assembler)

Analyse, letzter Schritt: Operationsmodell



Operationsmodell

- besteht aus **mehreren Operationsschemata**
- Operationsschema als Spezifikation **für jede Systemoperation,** mit folgenden Informationen
  - informelle Beschreibung der Operation
  - Ein- und Ausgaben, Kommunikation mit der Umwelt
  - Zugriff auf (systeminterne) Objekte
  - Annahmen an den Vorzustand des Systems
  - Zustandsänderung (Effekt) der Operation
- UML-Umsetzung
  - keine Operationsschema enthalten
  - Annotierung z.B. durch **tagged values** möglich

Operationsschema

Ein Operationsschema besteht aus 8 Teilen:

Operation	=	Name der Operation
Description	=	informelle Beschreibung
Input	=	Eingabeparameter
Reads	=	(nur) gelesene Objekte / Attribute
Changes	=	(möglicherweise) geänderte Objekte / Attribute
Sends	=	erzeugte Ausgabeereignisse
Pre	=	Annahmen an den Vorzustand
Post	=	Beziehung zwischen Vor- und Nachzustand

Vorgehen: Erstellung Operationsschema

1. **Gebe informelle Beschreibung der Operation an**
   – Häufig auch verbale Beschreibung von Vor- und Nachbedingung
2. **Beschreibe Ausgabeereignisse**
   – Ableiten aus den Sequenzdiagrammen
3. **Spezifiziere Input, Reads- und Change-Teil**
   – Ableiten aus dem Systemklassenmodell, keine Boundary-Objekte und Links zwischen Boundary und Controller aufführen
4. **Spezifiziere Vorbedingung**
   – Implizite Vorbedingungen berücksichtigen
5. **Spezifiziere Nachbedingung**
   – Nur Variablen verwenden, die auch im Input, Reads- oder Change-Teil eingeführt wurden

Aufstellen eines Operationsschemas

Schritt 1: informelle Beschreibung der Operation

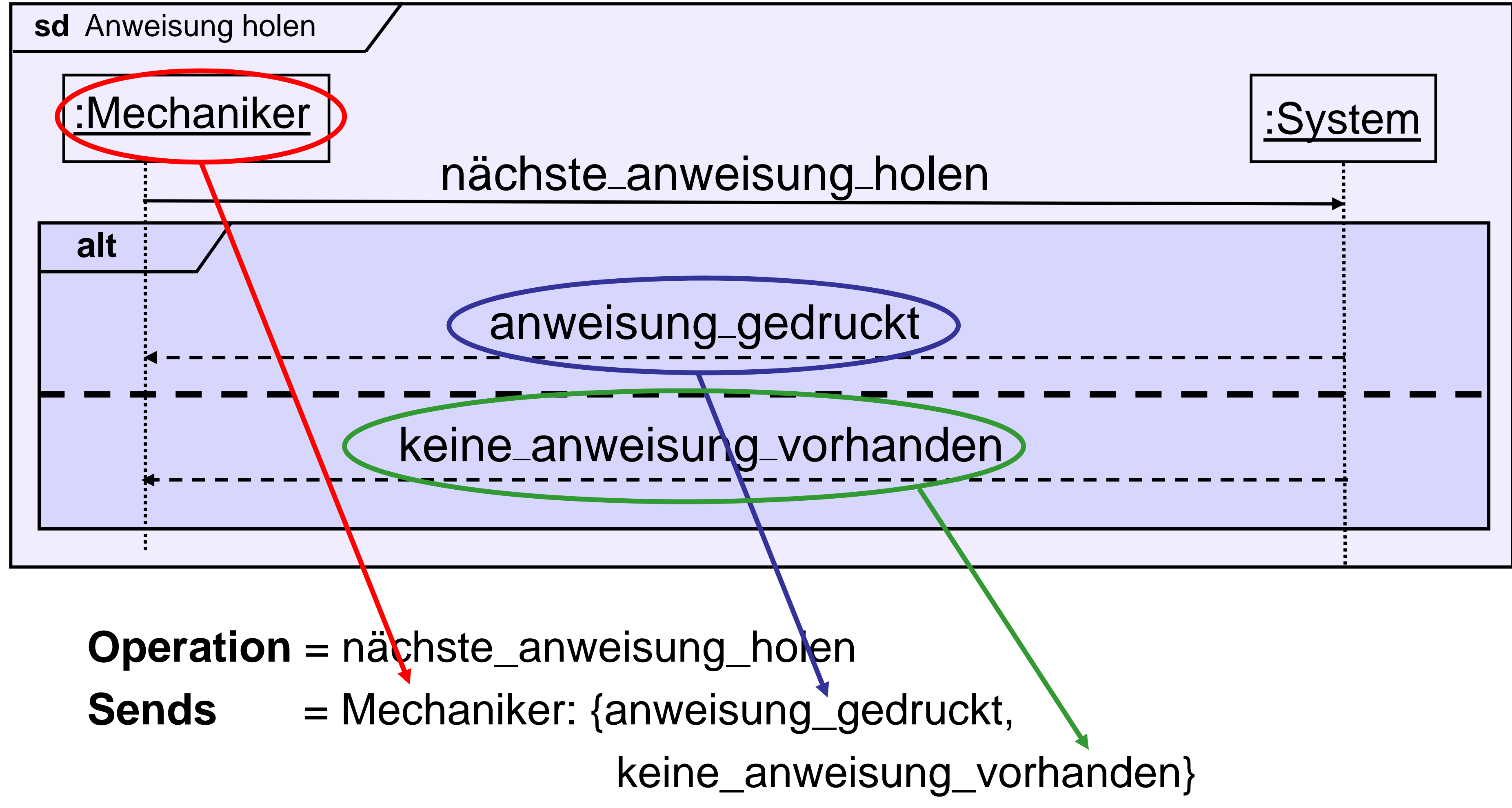
Operation = anweisung_generieren

Description = Nach der Erstellung eines Auftrags wird eine Reparaturanweisung generiert. Dieses kann auch nach Rücksprache mit dem Kunden erfolgen, falls ein Mechaniker zusätzliche Schäden festgestellt hat.

Schritt 2: Ableitung aus Sequenz-Diagrammen

- zum System **hinführende** Pfeile sind Aufrufe von Systemoperationen
- jeder Aufruf einer Systemoperation in einem Sequenz-Diagramm beschreibt einen „Fall“ für die Spezifikation dieser Operation
- nachfolgende Ausgabeereignisse und ihre Zielakteure werden im **sends**-Teil gesammelt

Schritt 2: Beispiel nächste_anweisung_holen



Operation = nächste_anweisung_holen

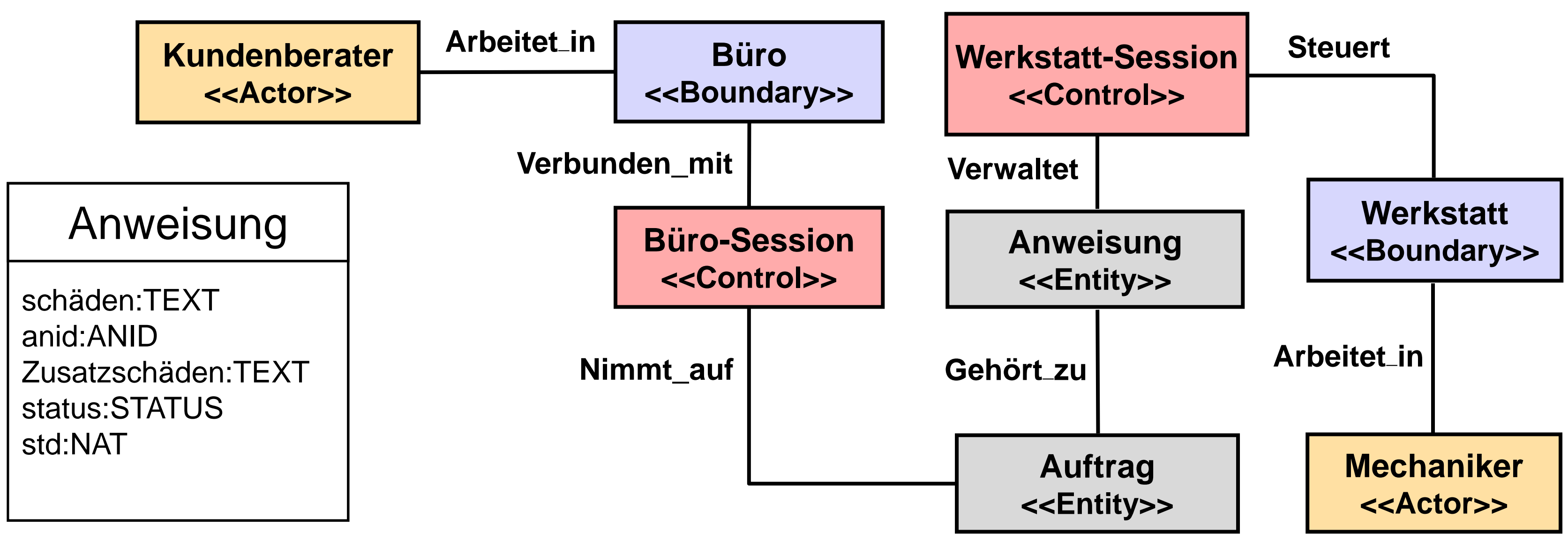
Sends = Mechaniker: {anweisung_gedruckt, keine_anweisung_vorhanden}

Schritt 3: Ableitung aus Systemklassenmodell

- Zugriff der Operation auf systeminterne Objekte wird in **Reads**- und **Changes**-Teilen spezifiziert
- möglichst disjunkte Objektlisten in **Reads**- und **Changes**-Teilen
- Objekte aus dem **Reads**-Teil dürfen nur gelesen werden
- Objekte aus dem **Changes**-Teil dürfen sowohl gelesen als auch verändert werden
- Benennung der Eingabeparameter im **Input**-Teil

Schritt 3.1: Eingaben und Reads-Objekte definieren

Beispiel Operation: anweisung_generieren



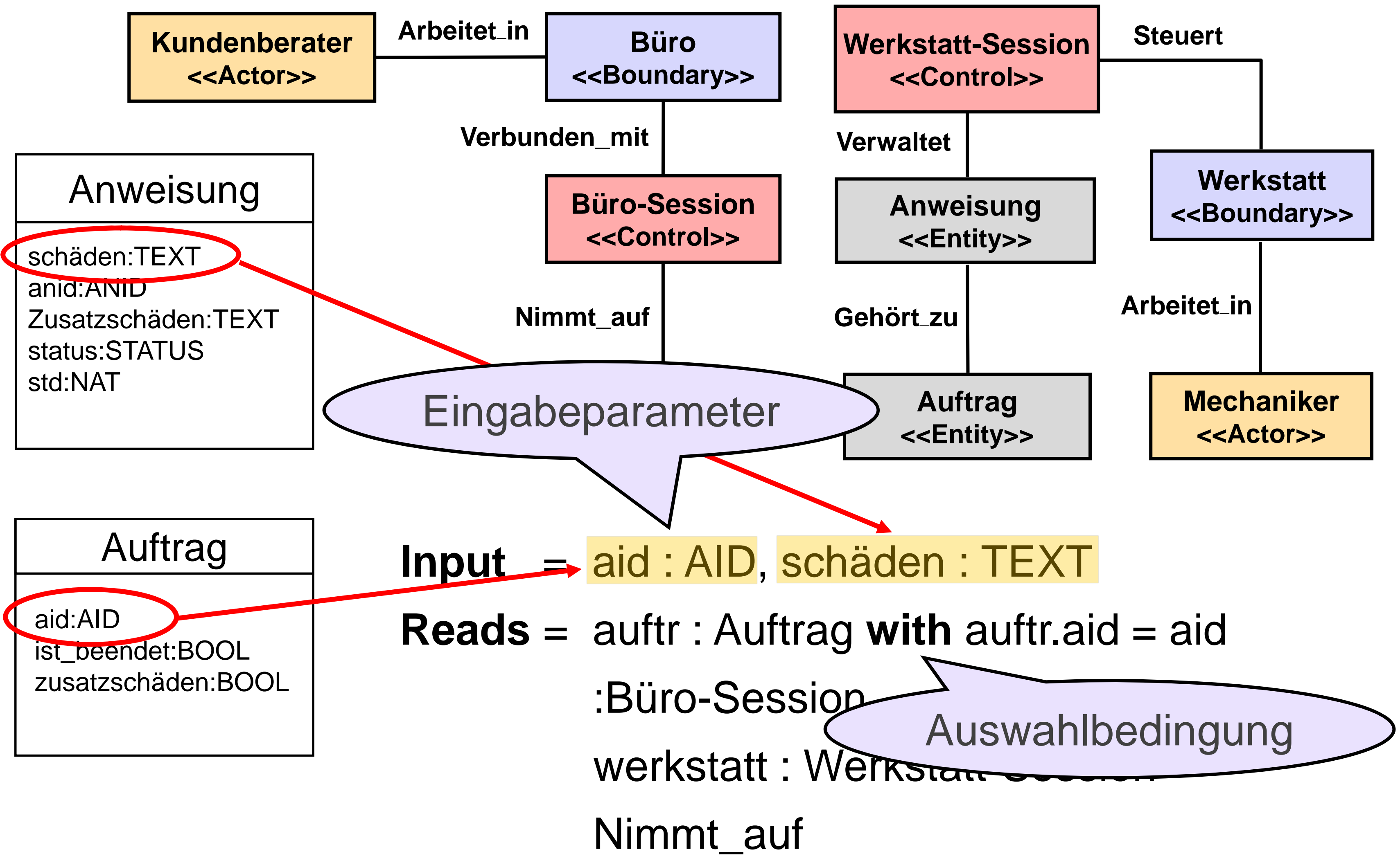
Auftrag

aid:AID
ist_beendet:BOOL
zusatzschäden:BOOL

Input = aid : AID, schäden : TEXT

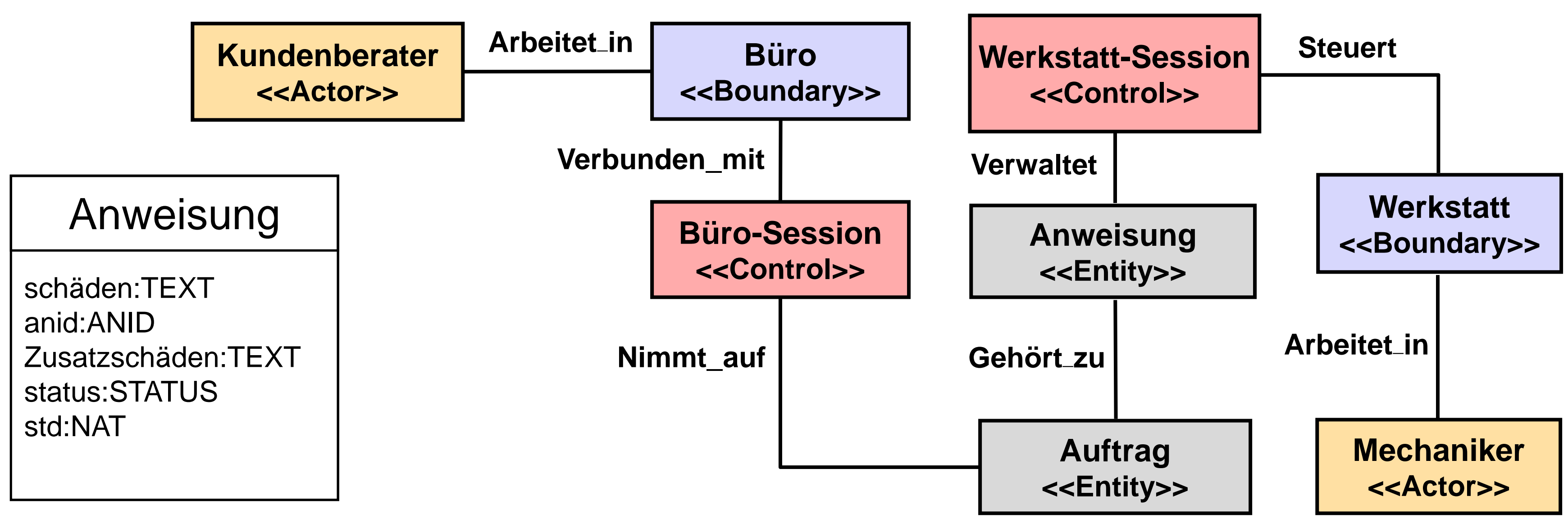
Reads = auftr : Auftrag **with** auftr.aid = aid
:Büro-Session, Auftrag,
werkstatt : Werkstatt-Session
Nimmt_auf

Schritt 3.1: Eingaben und Reads-Objekte definieren
Beispiel Operation: anweisung_generieren



Schritt 3.2: sich veränderndе Objekte finden

Beispiel Operation: anweisung_generieren



Auftrag

aid:AID
ist_beendet:BOOL
zusatzschäden:BOOL

Changes = anw : Anweisung **type**

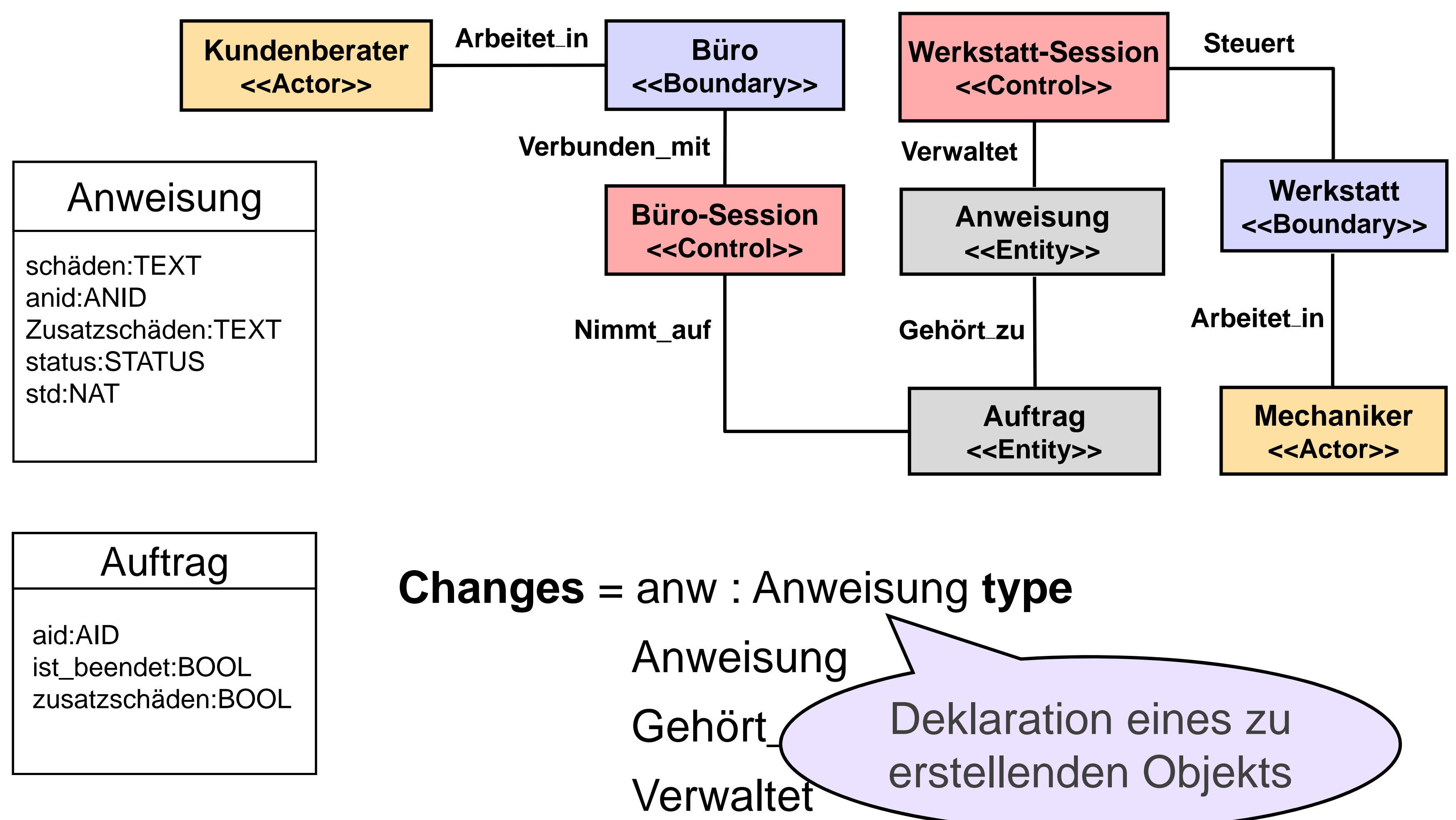
Anweisung

Gehört_zu

Verwaltet

Schritt 3.2: sich verändernde Objekte finden

Beispiel Operation: anweisung_generieren



Erklärungen zur verwendeten Syntax

Notation	Bedeutung
:Büro-Session	**unbenanntes Objekt**
auftr:Auftrag **with** auftr.id = aid	**benanntes Objekt** mit besonderer Eigenschaft (implizite Vorbedingung)
Nimmt_auf	**Assoziation**, die alle vor Ausführung der Operation existierenden Links beschreibt.
Anweisung	**Klasse**, die alle vor Ausführung der Operation existierenden Objekte beschreibt.
Anweisung **type**	**Klasse als Typ**, die zur Deklaration neuer Objekte verwendet wird.

Schritt 4 und 5: Vor- und Nachbedingungen definieren

- **Vorbedingung (Pre-Teil)**
  - eine **Aussage** über die **Eingabeparameter** und die Zustände der Objekte aus **Reads**- und **Changes**-Teil **vor** Ausführung der Operation
  - **keine** Aussage über dem Operationsaufruf vorhergegangene Aktivitäten!
- **Nachbedingung (Post-Teil)**
  - eine **Beziehung** zwischen Eingabeparametern und Zuständen der Objekte **vor** Ausführung der Operation und den Zuständen der Objekte **nach** Ausführung der Operation
  - **kein** „Programm“: nur der Effekt wird spezifiziert, aber nicht, wie er erreicht wird!

Schritt 4 und 5: Vor- und Nachbedingung spezifizieren

Beispiel Operation: anweisung_generieren (Link erzeugen)

Op. = anweisung_generieren

Reads = auftr : Auftrag **with** auftr.aid = aid
:Büro, Auftrag,
werkstatt : Werkstatt-Session
Nimmt_auf

Changes = anw : Anweisung **type**
Anweisung
Gehört_zu
Verwaltet

Post = …

$\wedge$ Gehört_zu‘ = Gehört_zu $\cup$ {(anw, auftr)} $\wedge$

…

Schritt 4 und 5: Vor- und Nachbedingung spezifizieren

Beispiel Operation: anweisung_generieren (Link erzeugen)

Op. = anweisung_generieren

Reads = auftr : Auftrag **with** auftr.aid = aid
:Büro, Auftrag,
werkstatt : Werk…
Nimmt_auf

implizite Vorbedingung:
∃ auftr : Auftrag • auftr.aid = aid

Changes = anw : Anweisung **type**
Anweisung
Gehört_zu
V…

Wert „nachher“

Link als Paar von Objekten

Post = …

∧ Gehört_zu‘ = Gehört_zu ∪ {(anw, auftr)} ∧

…

Schritt 4 und 5: Vor- und Nachbedingung spezifizieren

Beispiel Operation: anweisung_generieren (Neues Objekt erzeugen)

Pre = **implicit**

Changes = anw : Anweisung **type** Anweisung

Post = **let**

anw_ids == {a : Anweisung • a.anid};

•

anw **new** ∧ anw.schäden‘ = schäden ∧

(∃ anid:ANID • anid ∉ anw_ids ∧ anw.anid‘ = anid) ∧

anw.status‘ = bereit_zur_bearbeitung ∧

is_sent {anweisung_erstellt} ∧

Gehört_zu‘ = Gehört_zu ∪ {(anw, auftr)} ∧ …

Schritt 4 und 5: Vor- und Nachbedingung spezifizieren

Beispiel Operation: anweisung_generieren (Neues Objekt erzeugen)

Pre = **implicit**

Changes = anw : Anweisung **type** Anweisung

Post = **let**

anw_ids

•

anw **new** ∧ anw.schäden‘ = schäden ∧

(∃ anid:ANID • anid ∉ anw_ids ∧ anw.anid‘ = anid) ∧

anw.status‘ = bereit_zur_bearbeitung ∧

is_sent {anweisung_erstellt} ∧

Gehört_zu‘ = Gehört_zu ∪ {(a

implizite Vorbedingung

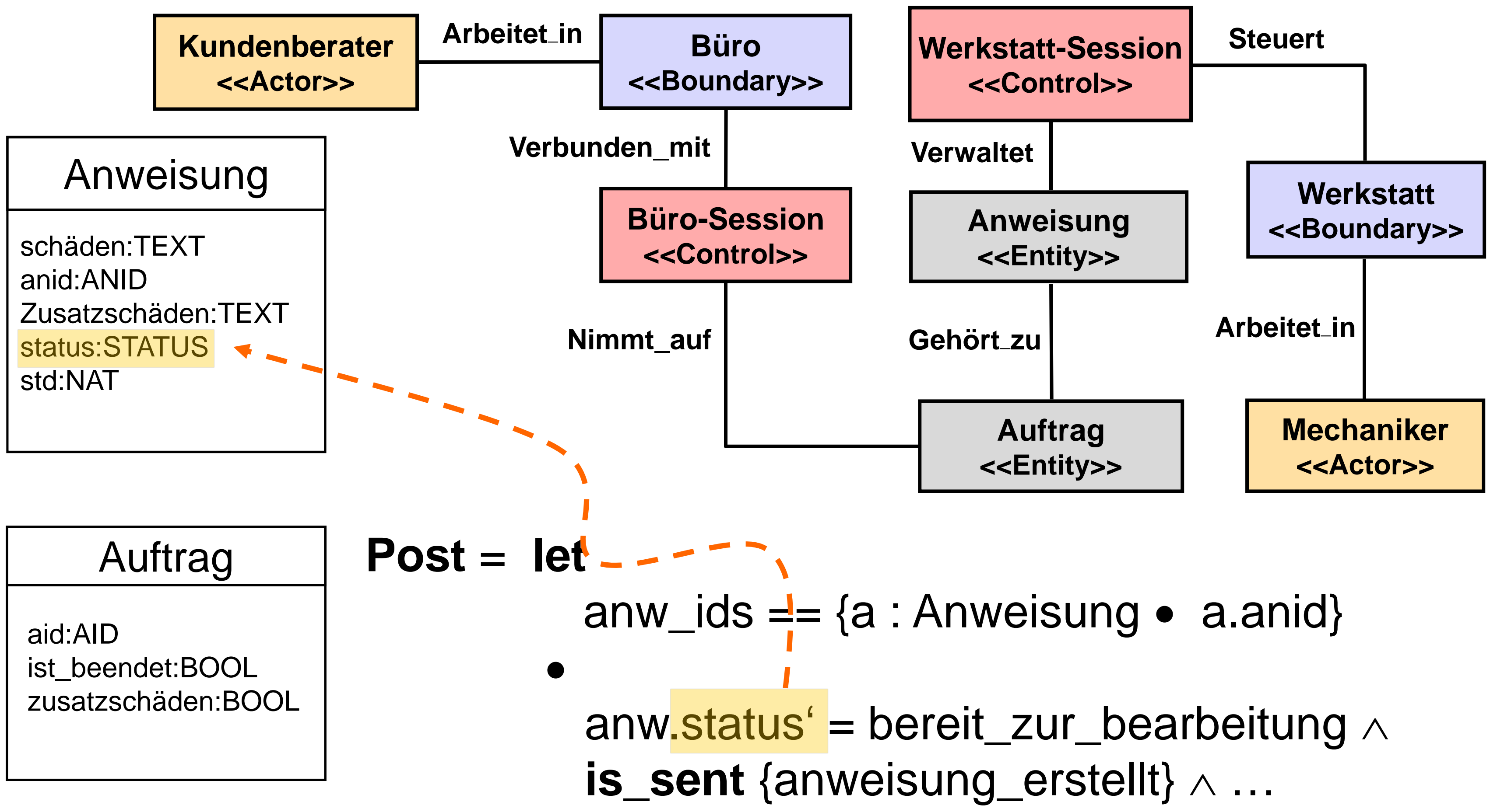
lokale Definitionen

Abkürzung für:
Anweisung‘ = Anweisung ∪ {anw}

Ereignisausgabe

Schritt 4 und 5: Vor- und Nachbedingung spezifizieren

Beispiel Operation: anweisung_generieren (Attribute abgleichen)



Das Operationsmodell bestimmt die im Systemklassenmodell notwendigen Attribute

Beispiel: anweisung_generieren (Zusammenfassung)

Input = aid : AID, schäden : TEXT

Reads = auftr : Auftrag **with** auftr.aid = aid, Auftrag,
:Büro-Session, werkstatt : Werkstatt-Session, Nimmt_auf

Changes = anw : Anweisung **type,** Anweisung, Gehört_zu, Verwaltet

Sends = Kundenberater:{anweisung_erstellt}

Pre = **implicit**

Post = **let**

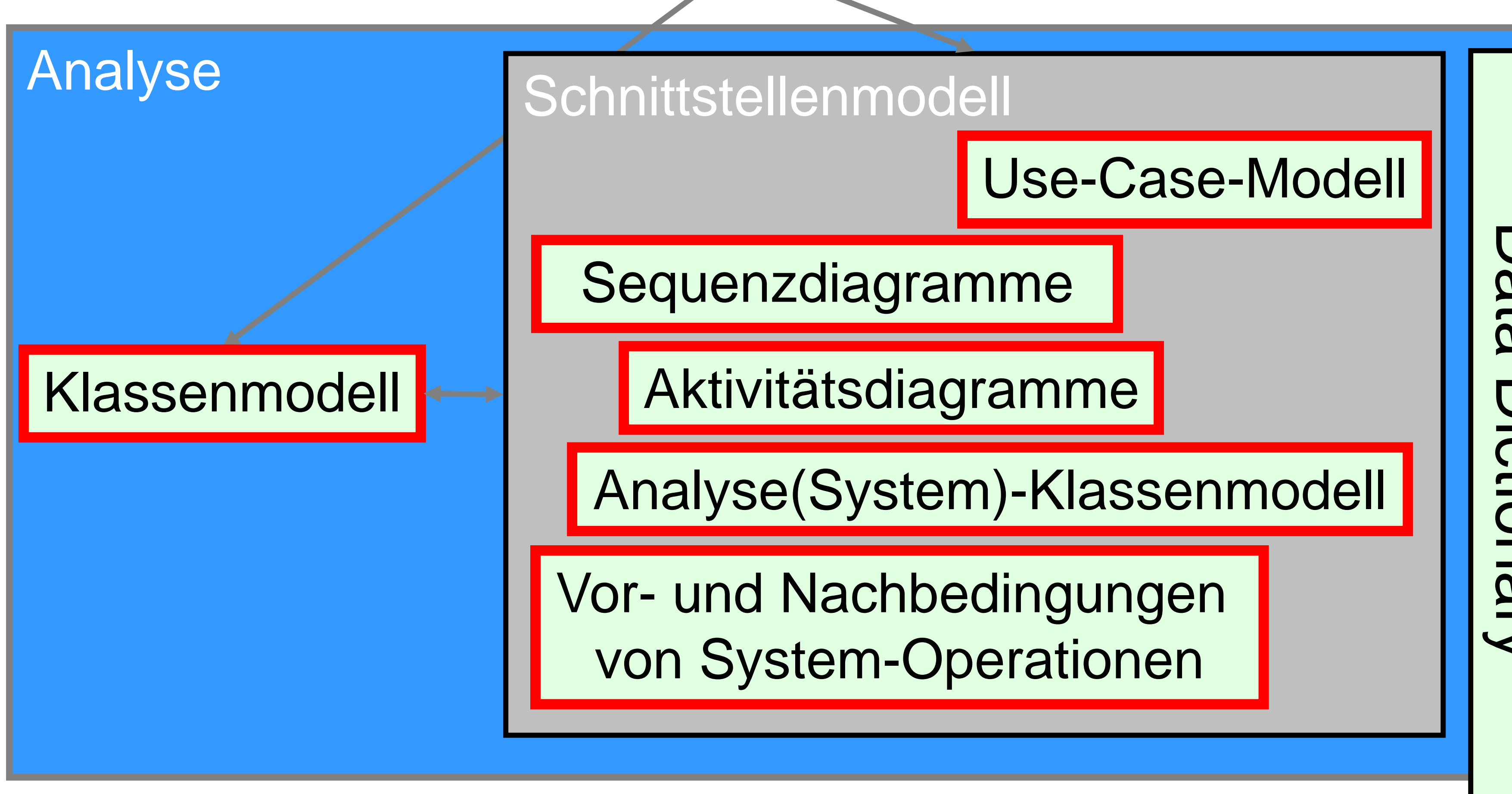
anw_ids == {a : Anweisung • a.anid}

•

Erzeugung und Belegung neuer Objekte einer Klasse	anw **new** $\wedge$ ($\exists$ anid:ANID • anid $\notin$ anw_ids $\wedge$ anw.anid' = anid) $\wedge$ anw.schäden' = schäden $\wedge$ anw.zusatzschäden' = " " $\wedge$ anw.status' = bereit_zur_bearbeitung $\wedge$ anw.std' = 0 $\wedge$
Erweiterung von Assoziationen mit Links	Gehört_zu' = Gehört_zu $\cup$ {(anw, auftr)} $\wedge$ Verwaltet' = Verwaltet $\cup$ {(werkstatt,anw)} $\wedge$

is_sent {anweisung_erstellt}

Was haben wir bis jetzt erreicht?



Aufbau des Datenlexikons (Data Dictionary)

Name	Art	Beschreibung	Quelle
Büro	Boundary-Klasse	Schnittstelle, über die die Bürokräfte mit dem System kommunzieren	Systemklassen-modell
Werkstatt	Boundary-Klasse	Schnittstelle, über die die Meachaniker mit dem System kommunzieren	Systemklassen-modell
Auftrag	Entity-Klasse	Datenklasse, in der Informationen über einen Auftrag abgelegt sind	Systemklassen-modell
Status	Aufzählungs-typ	Status einer Reparaturanweisung (bereit zur Bearbeitung, momentan in Bearbeitung, Bearbeitung beendet)	Systemklassen-modell
…			

Tut alles, damit die TutorInnen sehr leicht und schnell alle Begriffe finden!