Einführung in die Softwaretechnik

Vorlesung, Teil 2

Prof. Dr. Bernhard Rumpe
Lehrstuhl Informatik 3 (Software Engineering)
Rheinisch-Westfälische Technische Hochschule Aachen

http://www.se-rwth.de/

Software Engineering (Informatik 3)

Prof. Dr. Bernhard Rumpe

Ahornstr. 55
Zimmer 4303
Email rumpe @ se.rwth-aachen.de
Sprechzeit: N.N. und nach Email-Vereinbarung

Thomas Heer, Jan Oliver Ringert

Email heer @ i3.informatik.rwth-aachen.de
ringert @ se.rwth-aachen.de

Information zur Vorlesung

- Internet-Seite zur Vorlesung: (wie bisher)
  http://se.rwth-aachen.de/tikiwiki/tiki-index.php?page_ref_id=585

- Fragen:
  swt-vorlesung @ se.rwth-aachen.de (wie bisher)

- Übungsbetrieb und Klausuren:
  wie bisher

Inhalt des Teils 2 der Vorlesung

- Vorstellung Arbeitsgruppe Software Engineering
- Modellierung mit der UML
- Architektur / Entwurf
- Formale Spezifikation (Prof. Nagl)
- Kodierung & Dokumentation
- Qualitätssicherung
- Werkzeuge der Softwareentwicklung

0. Preliminiaries

0.1. Vorstellung der Arbeitsgruppe
Software Engineering

Prof. Dr. Bernhard Rumpe
Software Engineering
RWTH Aachen

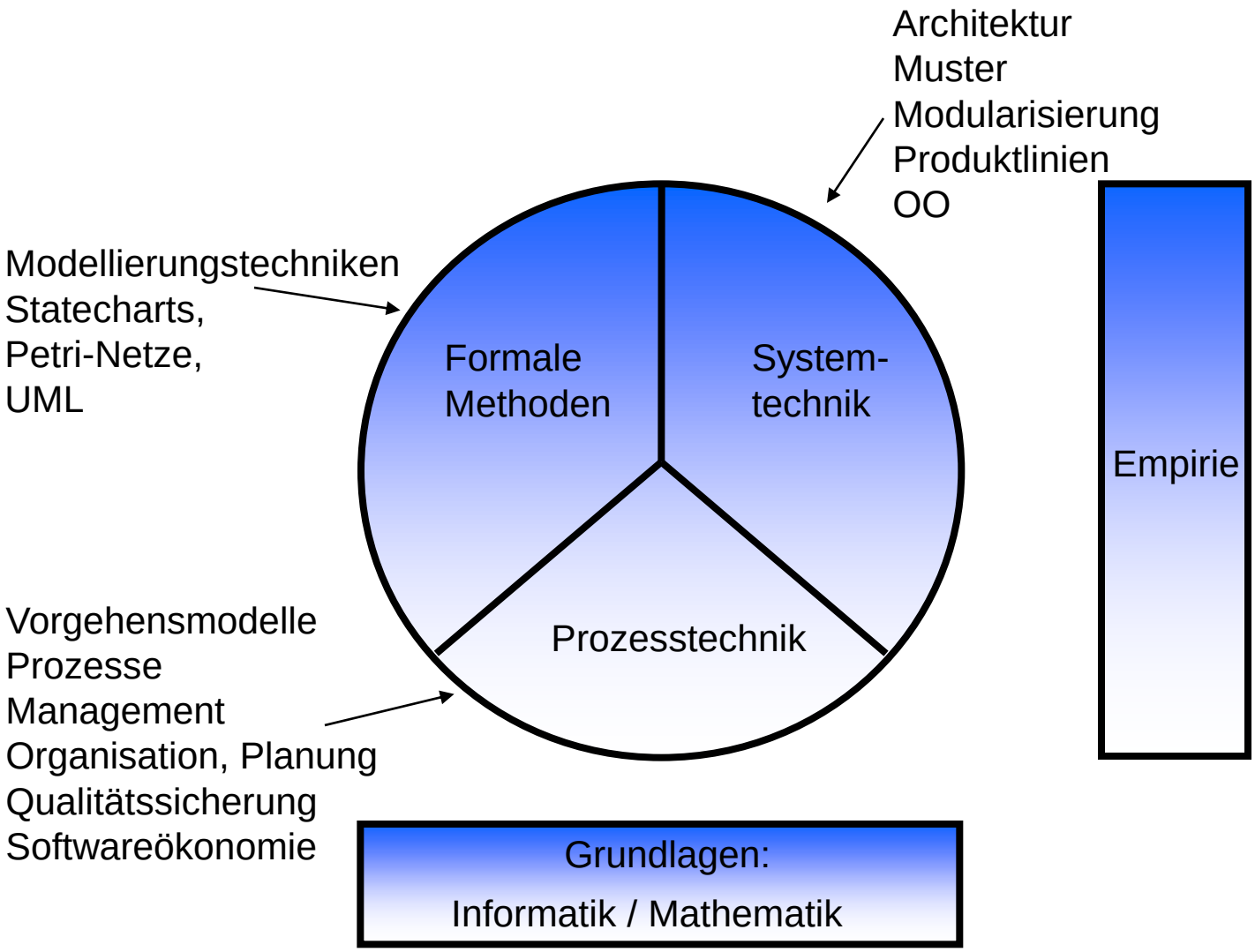
http://www.se-rwth.de/

Software Engineering

- „Software Engineering zielt auf die ingenieurmäßige Entwicklung, Wartung, Anpassung und Weiterentwicklung großer Softwaresysteme unter Verwendung bewährter systematischer Vorgehensweisen, Prinzipien, Methoden und Werkzeuge“
  - (Manifest der Softwaretechnik, 2006)

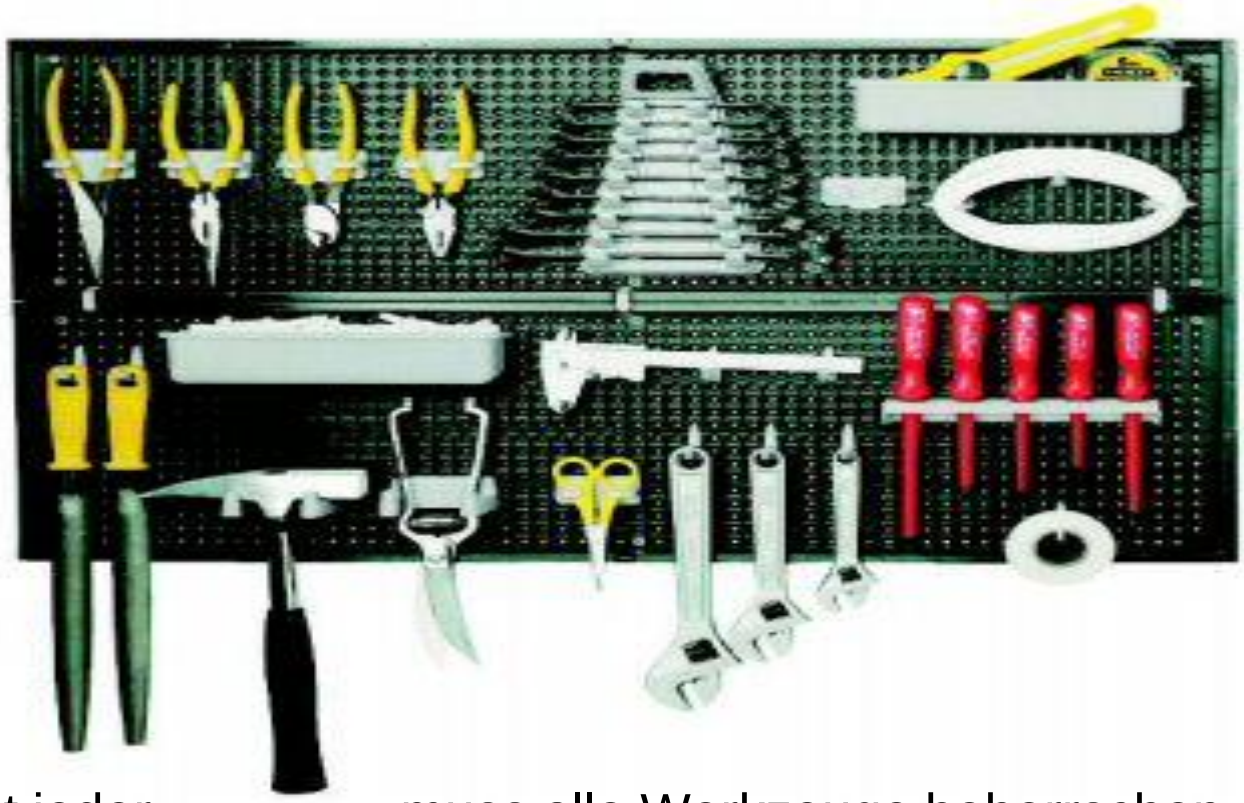
- Berücksichtigung der folgenden Aspekte:
  - Kosten
  - Termine
  - Qualität
    (Korrektheit, Zuverlässigkeit, Performanz, Sicherheit, Nutzbarkeit, Verständlichkeit, Weiterentwickelbarkeit, Anpassbarkeit, Wartbarkeit)

Gliederung des Software Engineering



Portfolio der SE-Techniken

- ... ist vergleichbar mit einer Werkzeugbank:
  - Für jedes Problem das richtige Werkzeug
  - in der Hand eines Experten, der damit umgehen kann



- Nicht jeder muss alle Werkzeuge beherrschen
- Aber: je mehr, um so besser.

Ziel der Arbeitsgruppe SE

- Verbesserung der Softwareentwicklung
- durch die permanente Suche nach

> Methoden, Konzepten und Werkzeugen zur
> - besseren und schnelleren Entwicklung von Softwaresystemen,
> - so dass in kurzer Zeit und mit
> - flexiblen Einarbeitung von sich wandelnden Anforderungen ein
> - qualitativ hochwertiges Ergebnis entsteht.

(Mission-Statement)

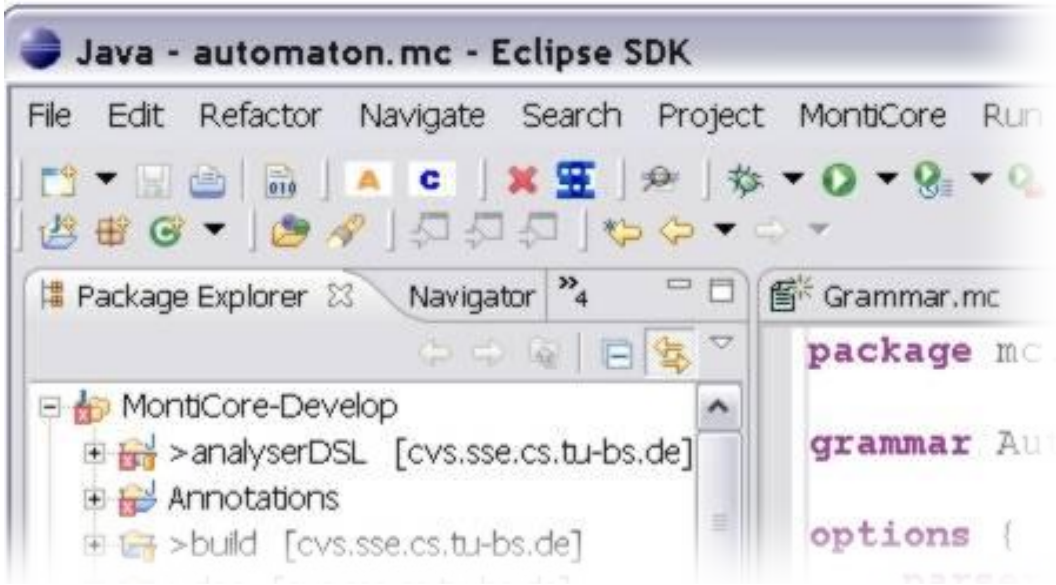
- Nachweis durch Entwicklung geeigneter Software/Systeme

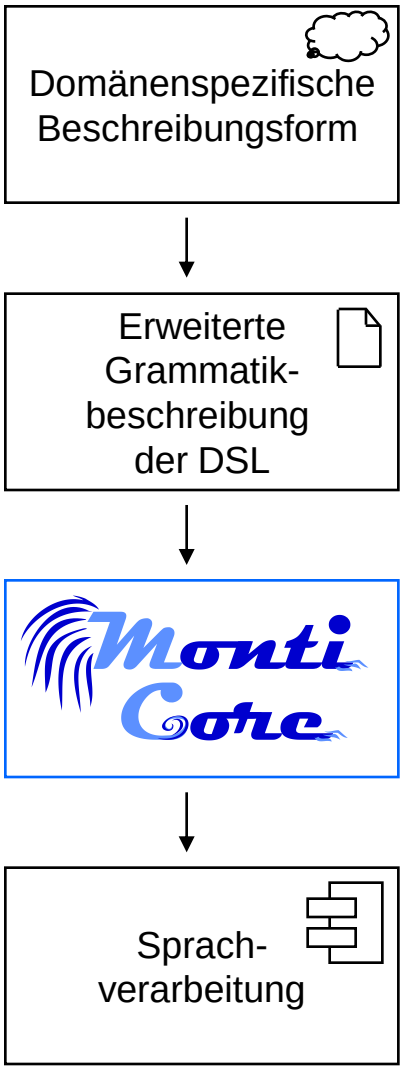
Themengebiete

- Werkzeugbau für Software Entwicklung
- Modellbasierung, vor allem UML
- Domänenspezifische Sprachen
- Softwarearchitektur
- Verifikation und Testen
- Evolution von Software (Anforderungen, Modellen)
- Agile Methoden

MontiCore – Entwicklung von domänenspezifischen Sprachen (DSLs)

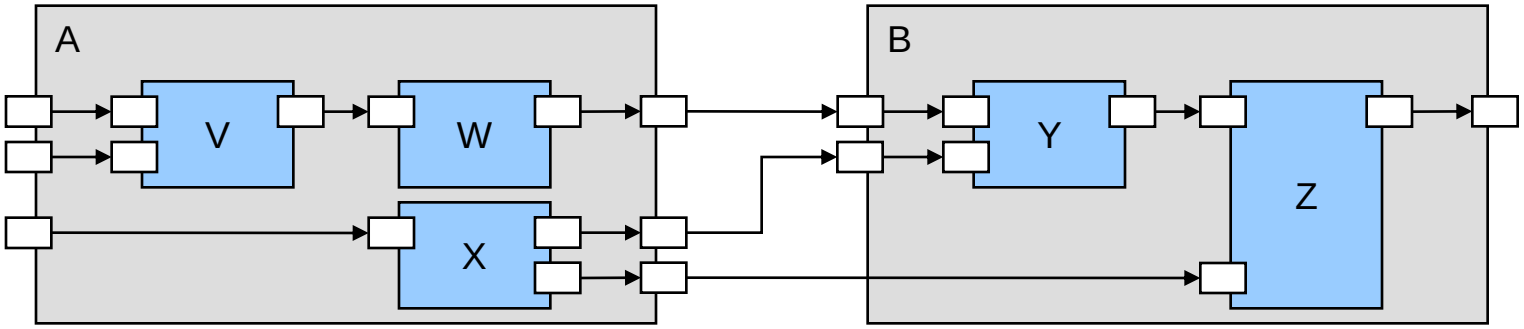
- Kleine effiziente Sprachen für eng umrissene Problemstellungen
- Integration von MontiCore und den erzeugten Komponenten in Eclipse
- Spezifische Codegenerierung für die DSLs





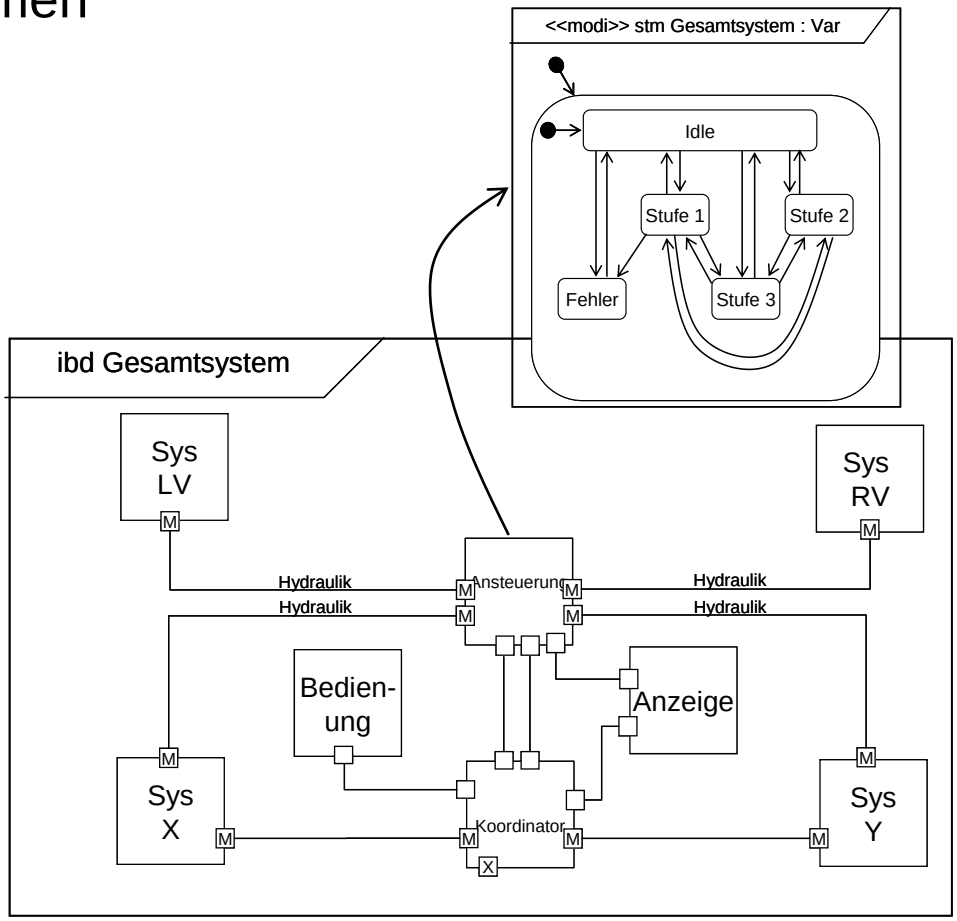
MontiArc

- Modellierungssprache für Systemarchitekturen
- Sprachdefinition mit Hilfe des MontiCore-Frameworks
- Hierarchische Schachtelung von Systemkomponenten
- Darstellung der Kommunikation zwischen Systemkomponenten
- Zeitliche Invarianten (durch OCL-Profil)
- Verfeinerung von Architekturen
- Textuelle Modellierung
- Generierung von Simulationen



Softwarebaukasten für fahrdynamische Regelsysteme

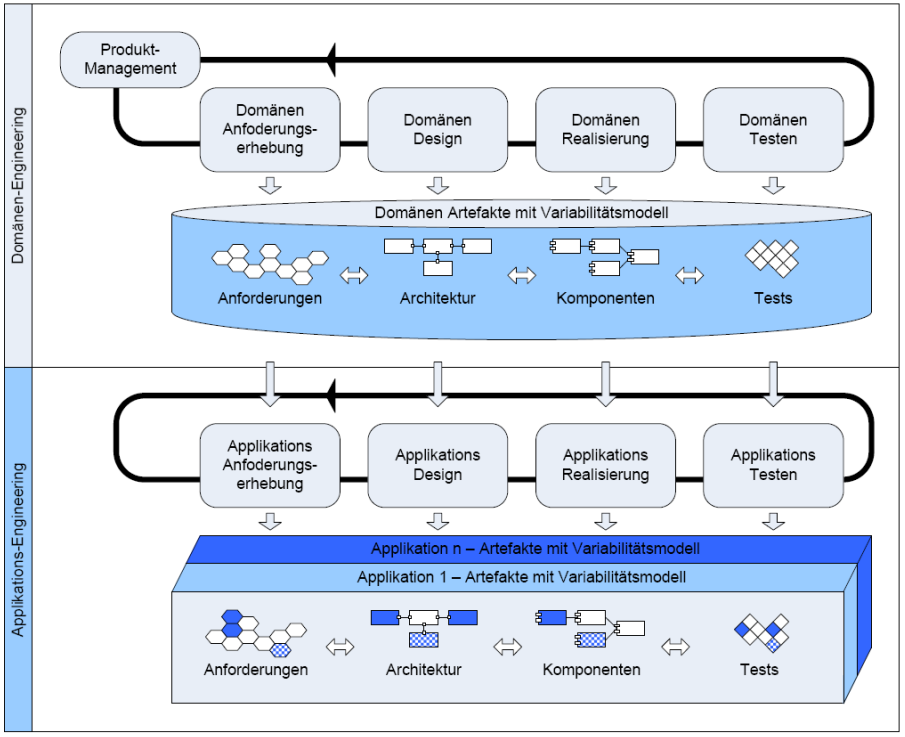
- Unterstützung der Funktions- und Steuergeräteentwicklung
- Bereitstellung einer Referenzmodellsprache zur Beschreibung von fahrdynamischen Regelsystemen
  - kompakte technologieunabhängige Systembeschreibung
  - baureihenübergreifende Darstellung funktionalen Wissens
  - Herstellung von Systemverständnis
  - Wiederverwendung von Modellen aus einer Bibliothek



Software-Produktlinien für Lenkungssteuergerätesoftware

Ziel: Vom Produkt zur Produktlinie

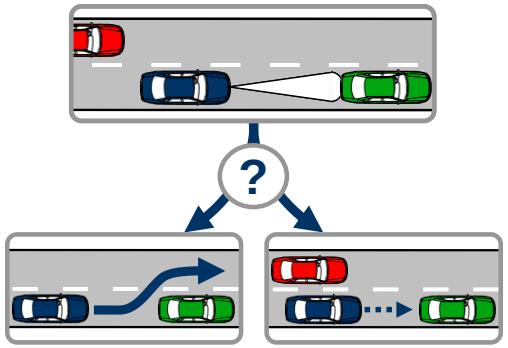
- Entwurf und Fundierung einer adaptiven Wiederverwendungs-strategie für Lenkungssteuergeräte-software
- Definition und Einsatz einer DSL für Generierung von applikationsübergreifendem und –spezifischem Code



intelliCar-Projekt

- Aufbau eines automatisch fahrenden Fahrzeugs für Autobahnen

- Entwurf, Realisierung und Qualitätssicherung einer erweiterbaren „künstlichen Intelligenz“ in den Bereichen
  - Situationserkennung
  - Situationsklassifizierung
  - Verhaltensgenerierung



- Unterstützung und Weiterentwicklung der vorhandenen Toolketten
- Minimierung kosten- und zeitintensiver Erprobungsfahrten durch frühzeitige Absicherung der Software-Qualität unter Verwendung mehrstufiger Software-Tests

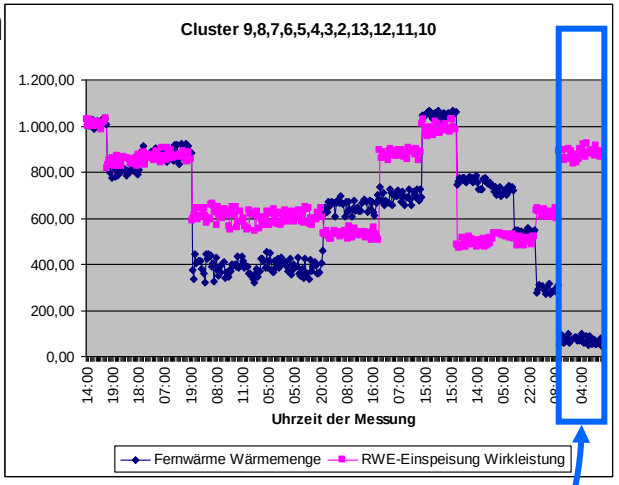
CarOLO-Projekt

- interdisziplinäres Forschungsprojekt für den Aufbau eines automatisch fahrenden Fahrzeugs im städtischen Umfeld
- Teilnahme am Wettkampf DARPA Urban Challenge 2007
- Technologien
  - Laser
  - Kameras
  - Radar
  - GPS + IMU
  - Künstliche Intelligenz

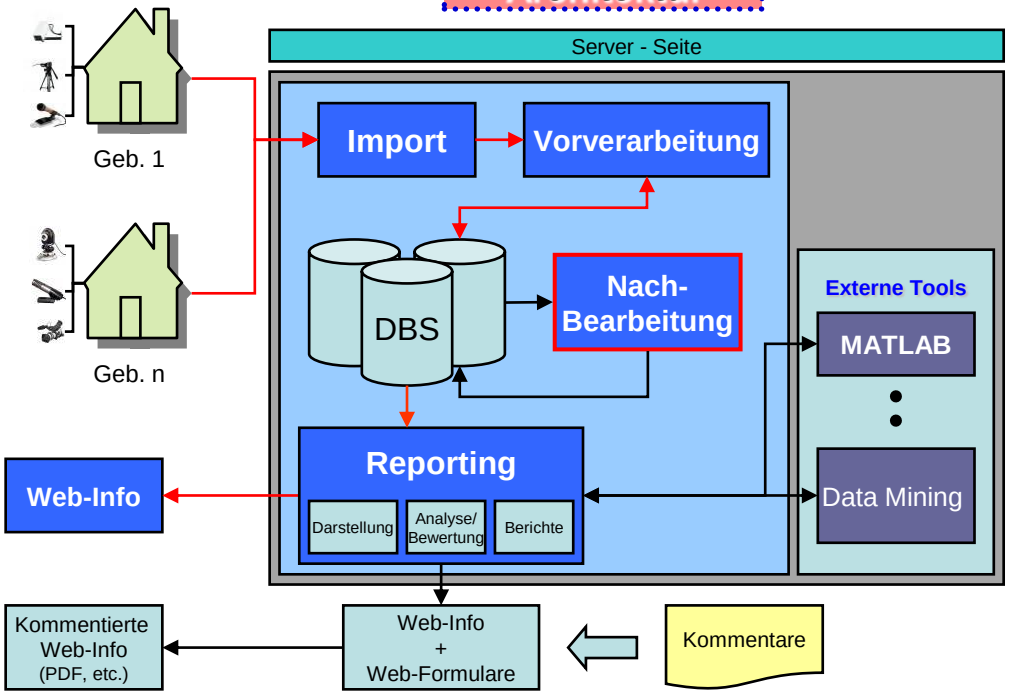
- Langfristige Ziele
  - Entwicklung einer allgemeinen künstlichen Intelligenz für intelligente und sichere Fahrerassistenz und
  - das selbständige Fahrzeug (Taxi, Dienstbote, Transport, ...)

EVA: Evaluierung von Energiekonzepten

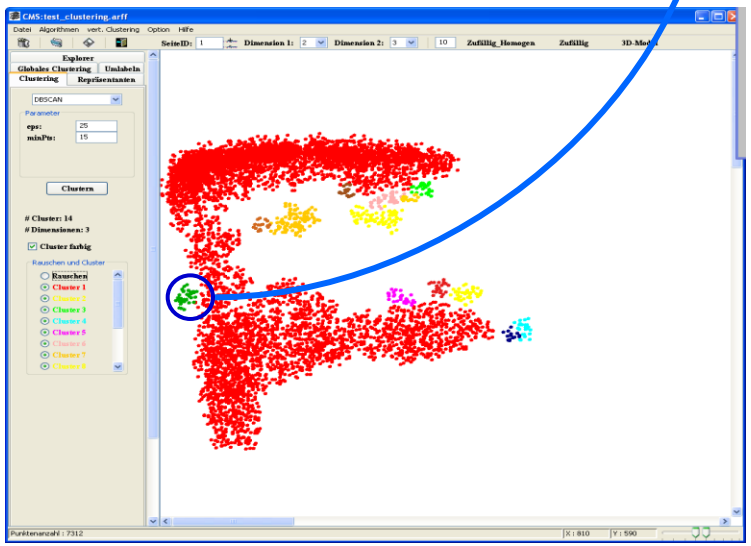
- Ziel: Optimierung des Energiebedarfs von Gebäuden
- Zugriff auf das System übers Web
- Datenauswertung nicht rechner- oder ortsgebunden
- Regel- und musterbasierte Analysen von Verbrauchsdaten



Architektur



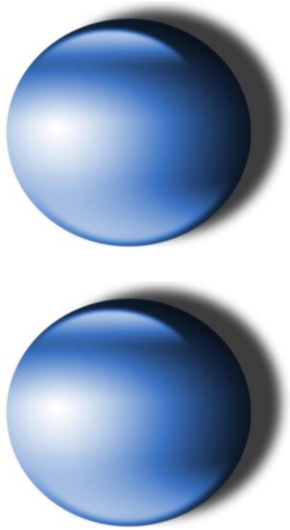
Datenanalyse



StuBS – EDV in Studium und Lehre

- Entwicklung und hochschulweite Einführung einer
- einheitlichen und zeitgemäßen
- EDV-Softwarelandschaft
- für die effiziente Durchführung von Geschäftsprozessen
- von Studium und Lehre
- aus Sicht der
  - Studierenden,
  - Lehrenden und der
  - Verwaltung

- Ziele: EDV Unterstützung bei
  - dezentraler Online-Prüfungsverwaltung
  - dezentraler Online- Lehrveranstaltungsverwaltung
  - Erstellung der TU-weit einheitliche Modulhandbuche
  - Online-Bewerbung für Schüler + aktuelle Bewerberstatistik



0. Preliminiaries

0.2. Einige Essentials aus der Softwaretechnik

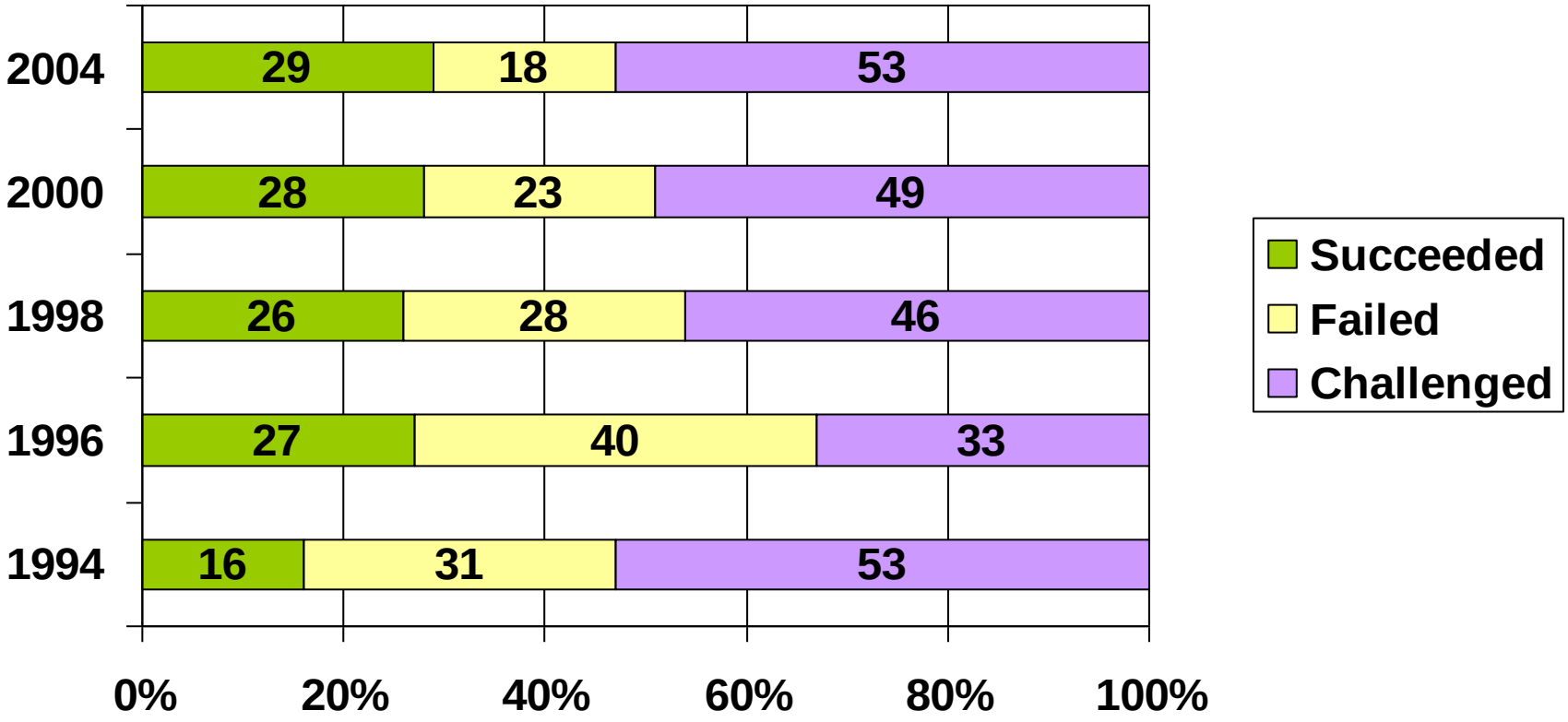
Prof. Dr. Bernhard Rumpe
Software Engineering
RWTH Aachen

http://www.se-rwth.de/

Software-Katastrophen: Kein Einzelfall

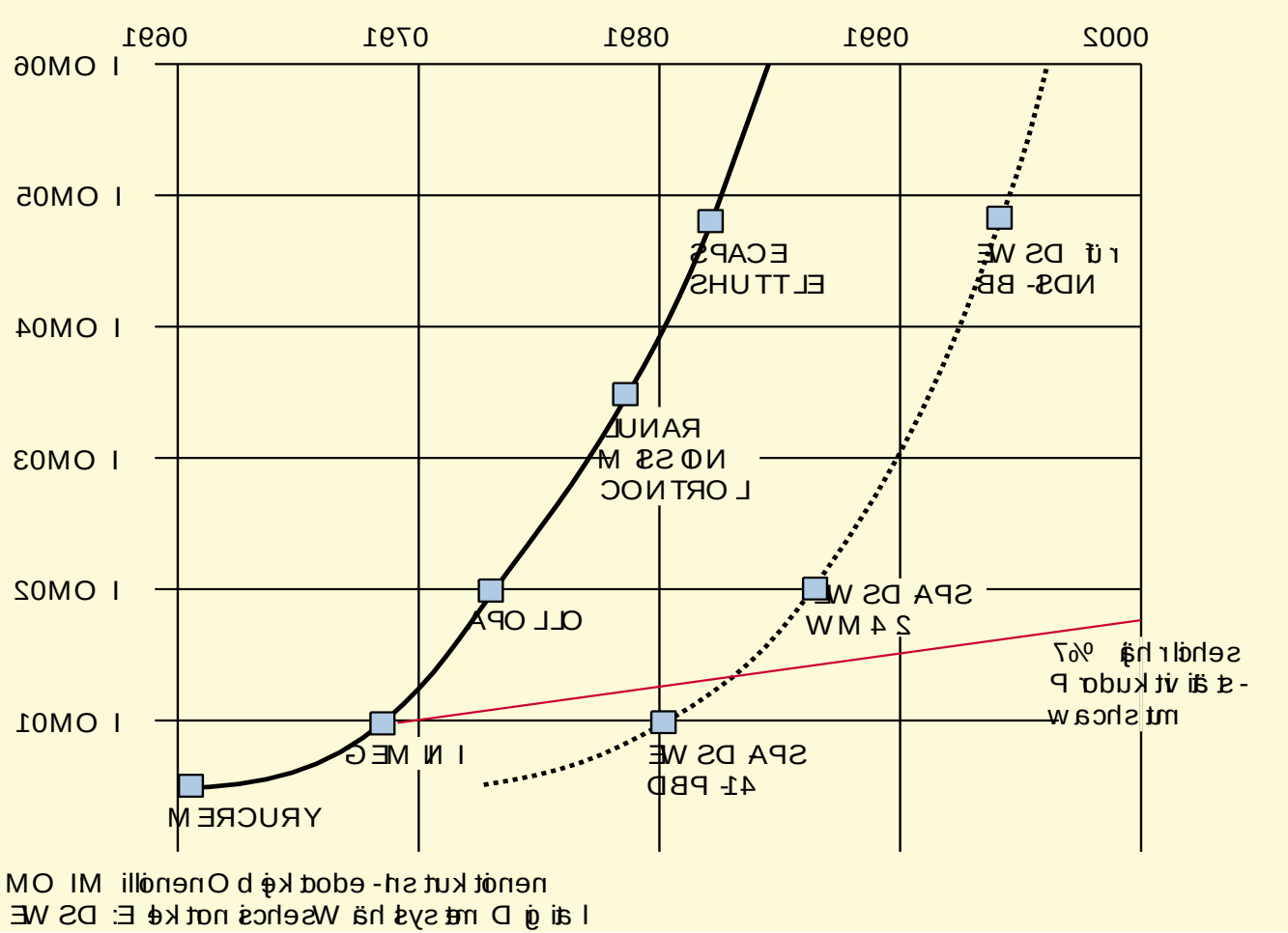
- Terminliche, finanzielle und technische Katastrophen:
  - 1999: Fehlstart einer Titan/Centaur-Rakete wegen falscher Software-Version
  - 1999: Verlust der Sonde "Mars Climate Orbiter" wegen falscher Einheitenumrechnung
  - 1994: Eröffnung des Denver International Airport um 9 Monate verzögert wegen Softwareproblemen im Gepäcktransport-System
  - 2002: Verzögertes und unfertiges Buchhaltungssystem führt zum Zusammenbruch eines US-Lebensmittelsherstellers
  - 2003/4: Toll Collect: Termin u.a. wegen Softwareproblemen mehrfach verschoben, Kosten ***> 1 Milliarde Euro***
  - 2006: Gesundheitskarte, ...
- USA:
  Verlust im Jahr 2004 ca. 160 Milliarden US-$ wegen defekter Software.

Erfolgsstatistik von IT-Projekten



Quelle: CHAOS Report, Standish Group International, Inc.

Wachsende Komplexität (1)



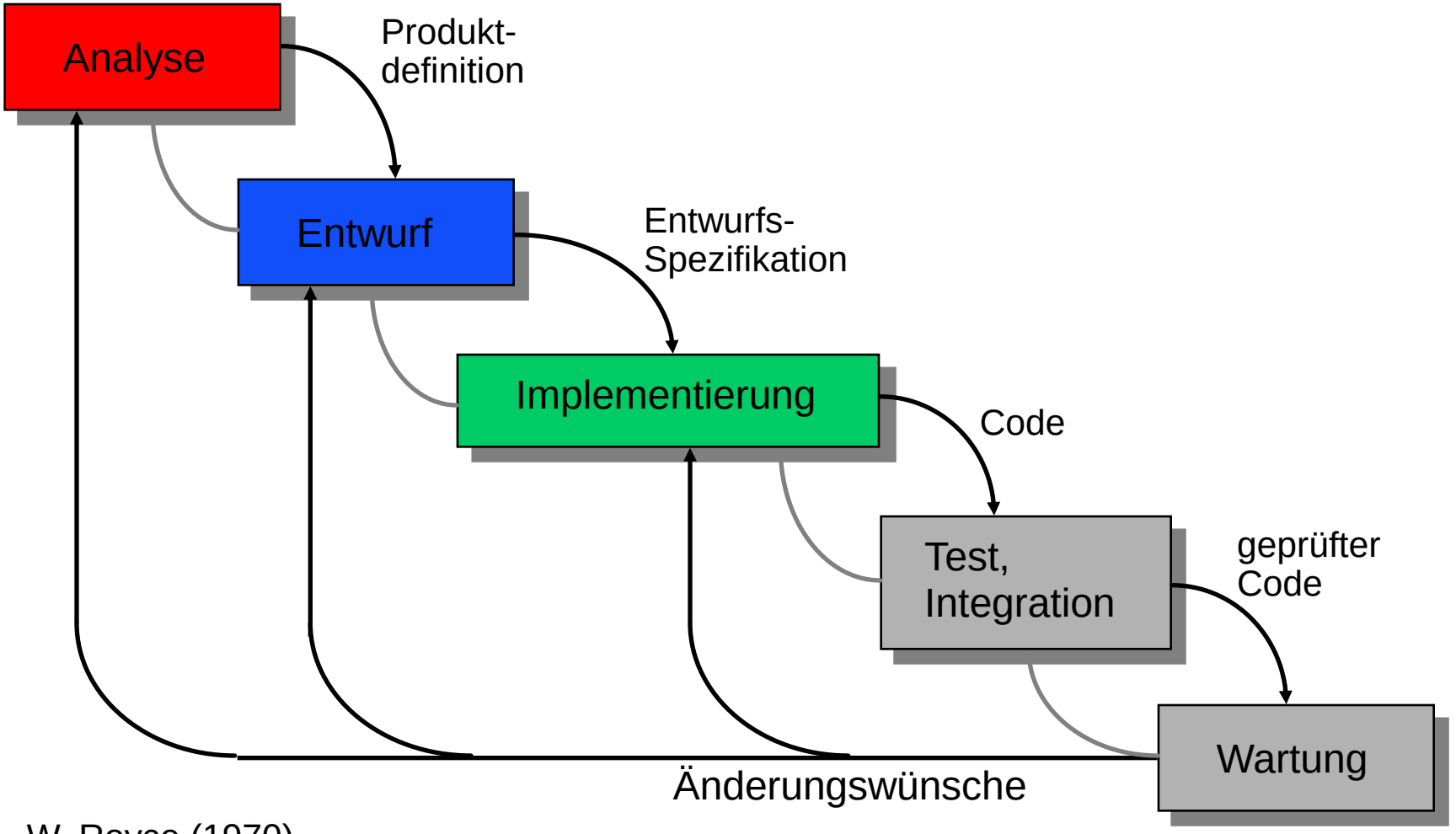
- Siemens EWSD V8.1: 12,5 Millionen LOC, ca.190.000 S. Dokumentation

Software Engineering

- 1968: erste Konferenz zum Thema „Software Engineering“
- Der Begriff wird geprägt:

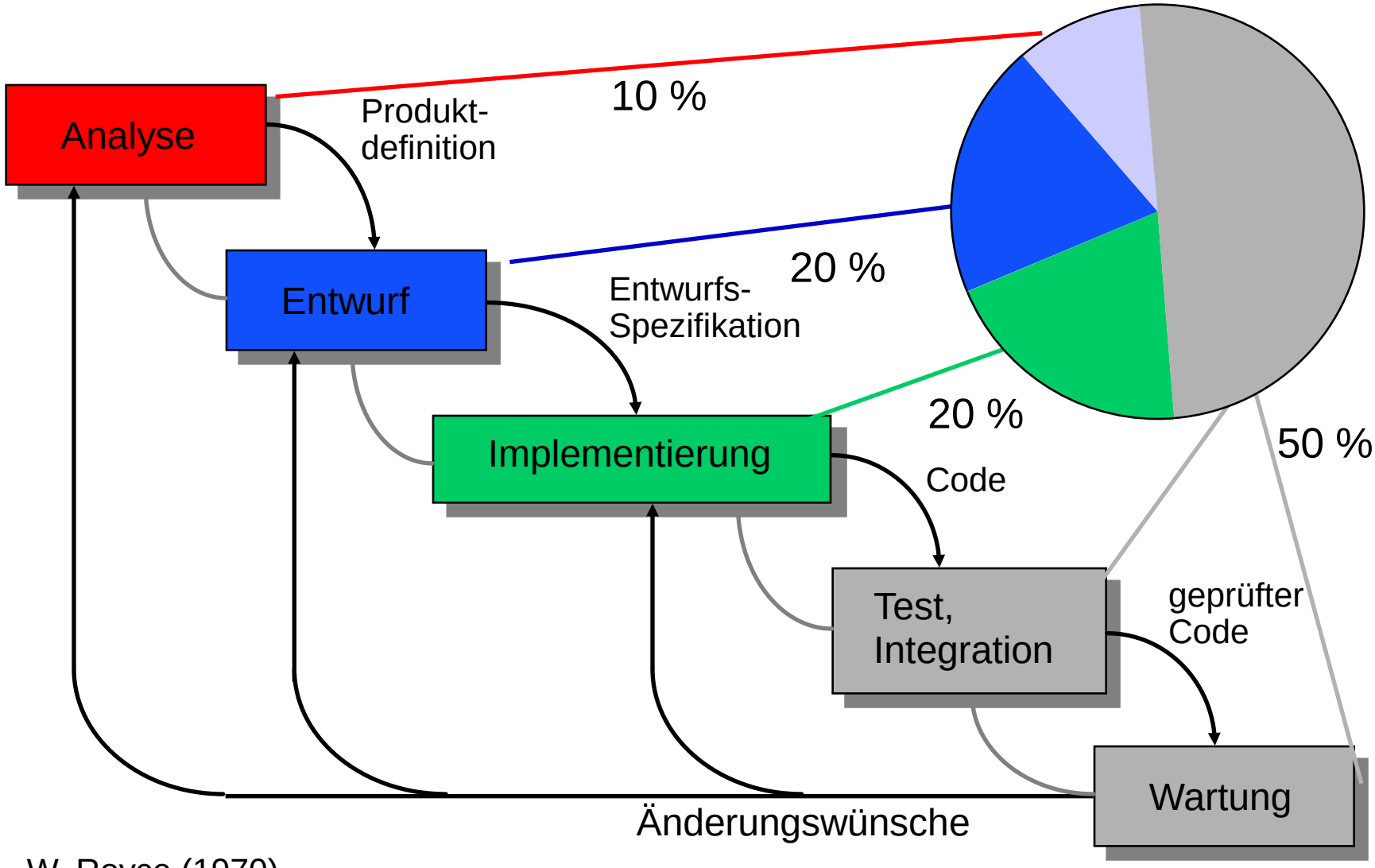
> Software Engineering:
>
> The establishment and use of sound engineering principles in order to obtain economically software that is reliable and runs on real machines.
>
> (F.L. Bauer, NATO-Konferenz Software-Engineering 1968)

Das klassische Wasserfall-Modell



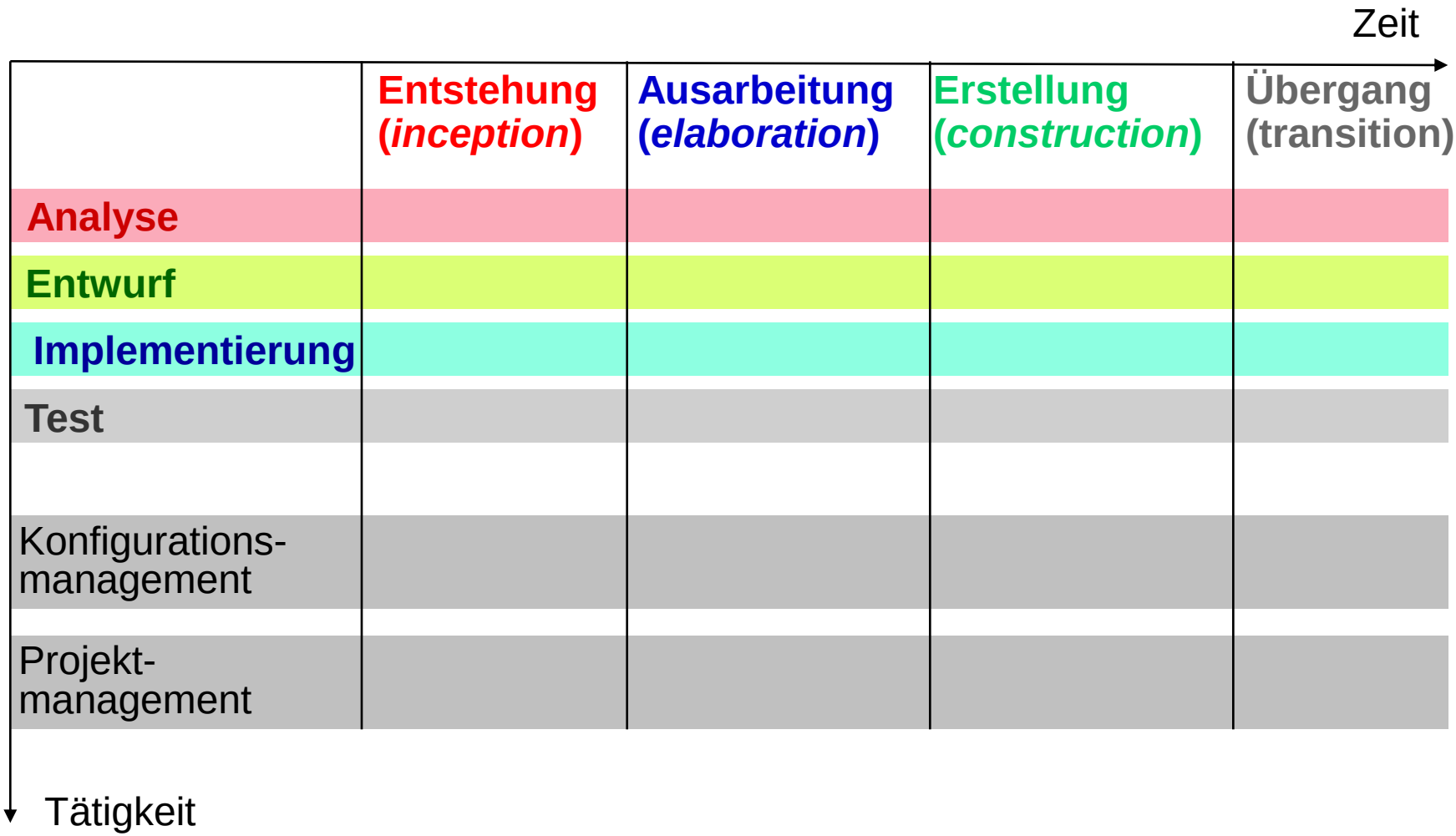
W. Royce (1970)

Ungefähre Verteilung des Arbeitsaufwands



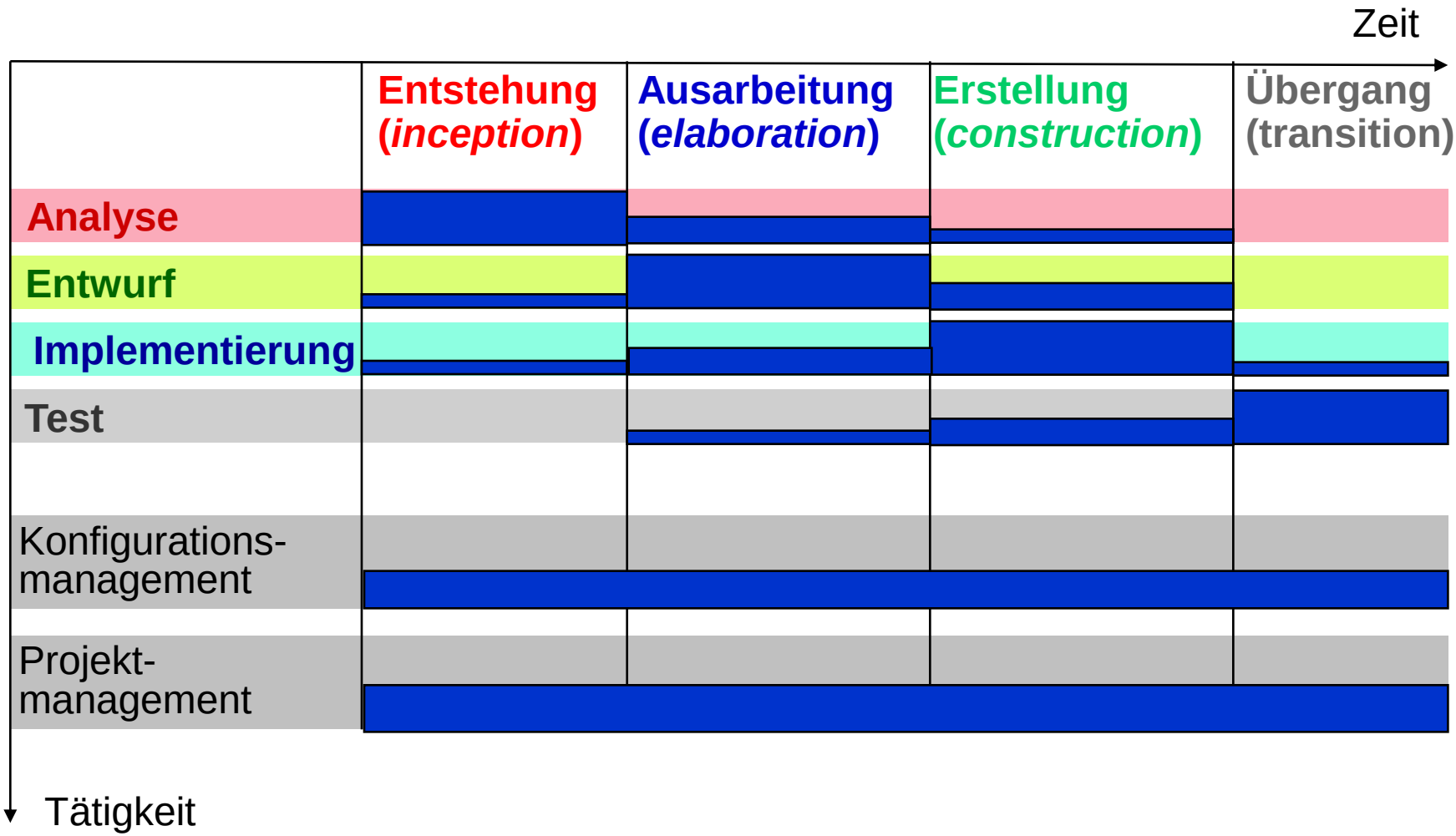
W. Royce (1970)

Zweidimensionales Modell



Rational Unified Process 1999 (Jacobson et al., Kruchten)

Aufwandsverteilung und Schwerpunkte



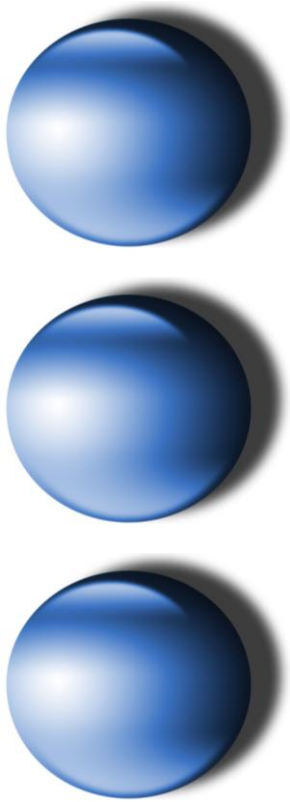
Rational Unified Process 1999 (Jacobson et al., Kruchten)

eXtreme Programming (XP)

- Entwicklungsmethodik für kleinere Projekte
- Konsequente evolutionäre Entwicklung in sehr kleinen Inkrements
- Tests + Programmcode sind das Analyseergebnis, das Entwurfsdokument und die Dokumentation.
- Code wird permanent lauffähig gehalten
- Diszipliniertes und automatisiertes Testen als Qualitätssicherung
- Paar-Programmierung als QS-Maßnahme
- Refactoring zur evolutionären Weiterentwicklung
- Codierungsstandards

- Aber auch: Weglassen von traditionellen Elementen
  - kein explizites Design, ausführliche Dokumentation, Reviews

- „Test-First“-Ansatz
  - Zunächst Anwendertests definieren, dann den Code dazu entwickeln



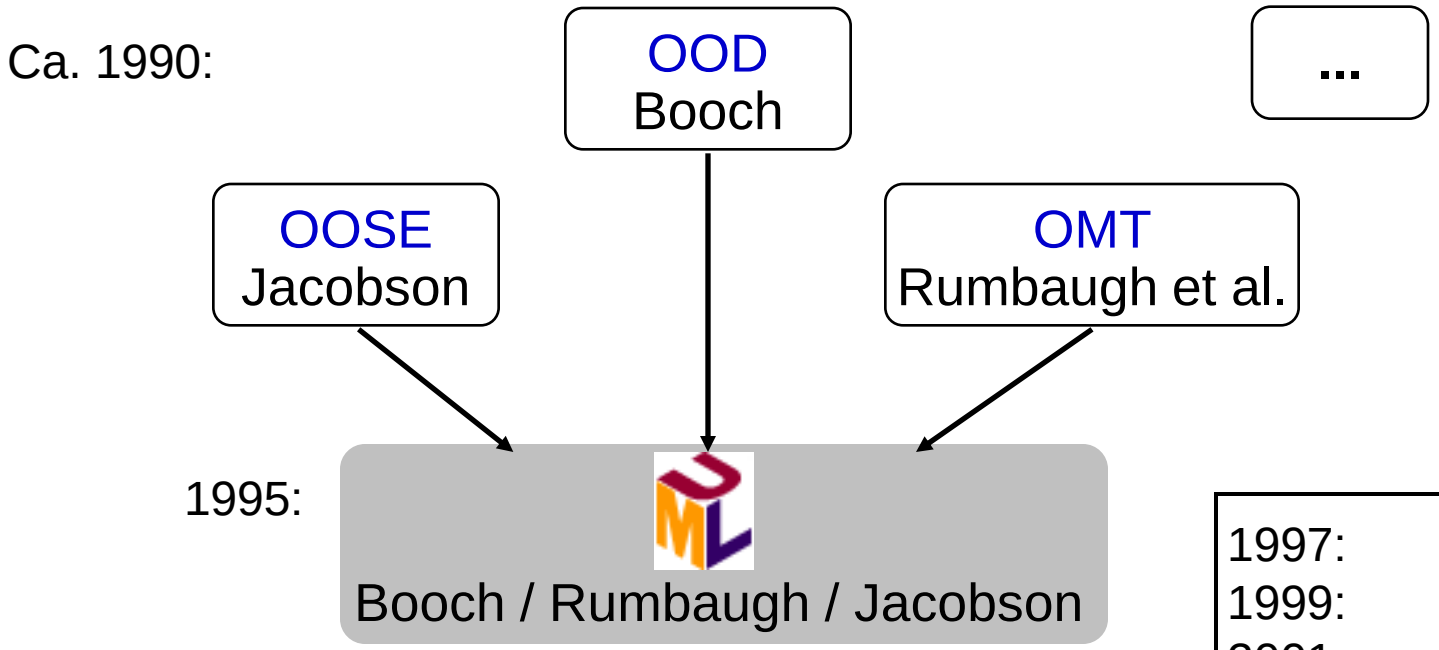
0. Preliminiaries

0.3. Kernelemente der UML
(in der Systemanalyse und -modellierung)

Prof. Dr. Bernhard Rumpe
Software Engineering
RWTH Aachen

http://www.se-rwth.de/

Unified Modeling Language UML



1997:	UML 1.1
1999:	UML 1.3
2001:	UML 1.4
2002:	UML 1.5
2004:	UML 2.0

- UML ist eine Notation der zweiten Generation für objektorientierte Modellierung
- UML ist Industriestandard der OMG (Object Management Group)

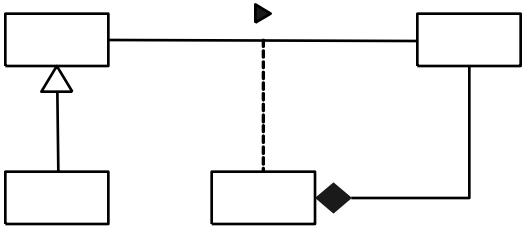
Unified Modeling Language

- Graphische Modellierungssprache für Software-Systeme
- Sprachmittel zur Spezifikation, Kommunikation und Dokumentation
  - zwischen Entwicklern
  - Entwicklern mit Anwendern
  - Vereinigung mehrerer Vorgänger-Methoden
- Standardisiert seit September 1997 von der OMG
- Entwickelt von

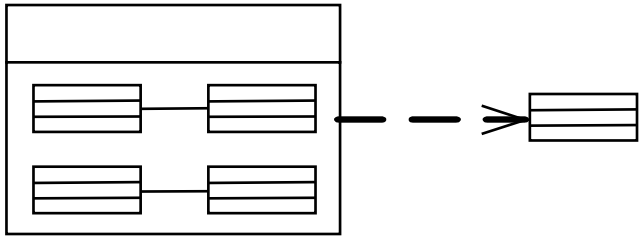
  Booch, Rumbaugh, Jacobson, Selic, Kobryn, Cook
  und vielen anderen ...
- Besteht aus:
  - Einer Menge von Modellierungskonzepten
  - Einer konkreten Notation

Strukturelle Notationen der UML

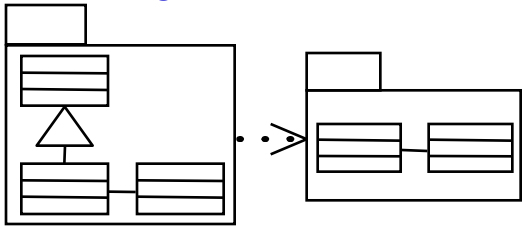
Klassendiagramm



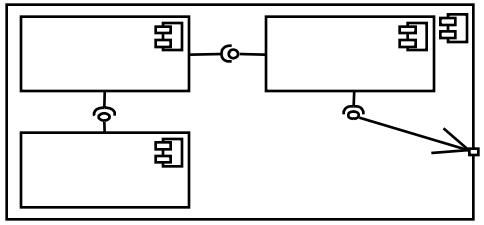
Kompositionsstrukturdiagramm



Paketdiagramm

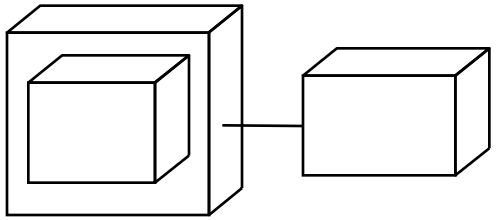


Komponentendiagramm



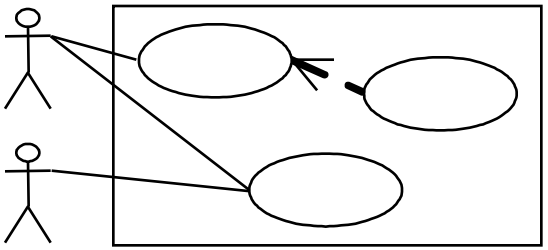
Objektdiagramm

Verteilungsdiagramm

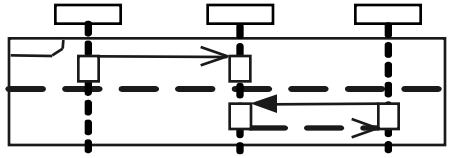


Verhaltensorientierte Notationen der UML

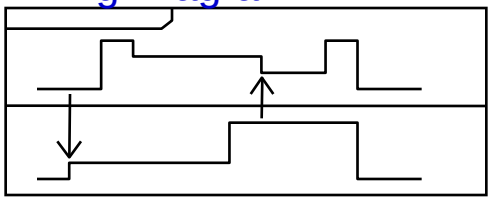
Use-Case-Diagramm



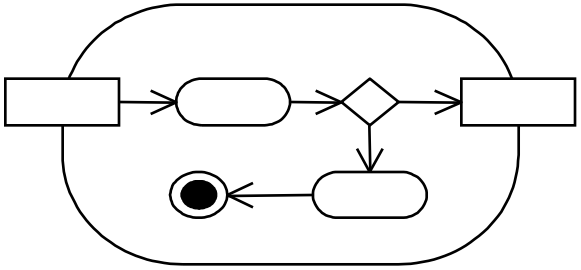
Sequenzdiagramm



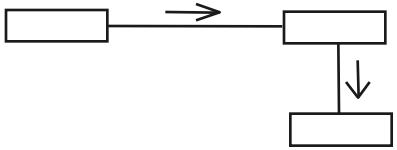
Timing-Diagramm



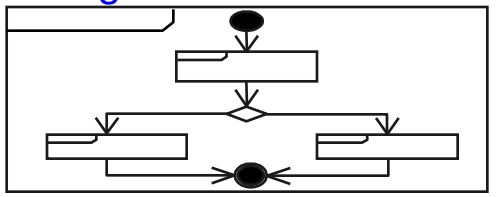
Aktivitätsdiagramm



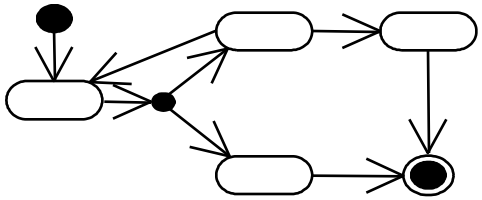
Kommunikationsdiagramm



Interaktionsübersichts-diagramm

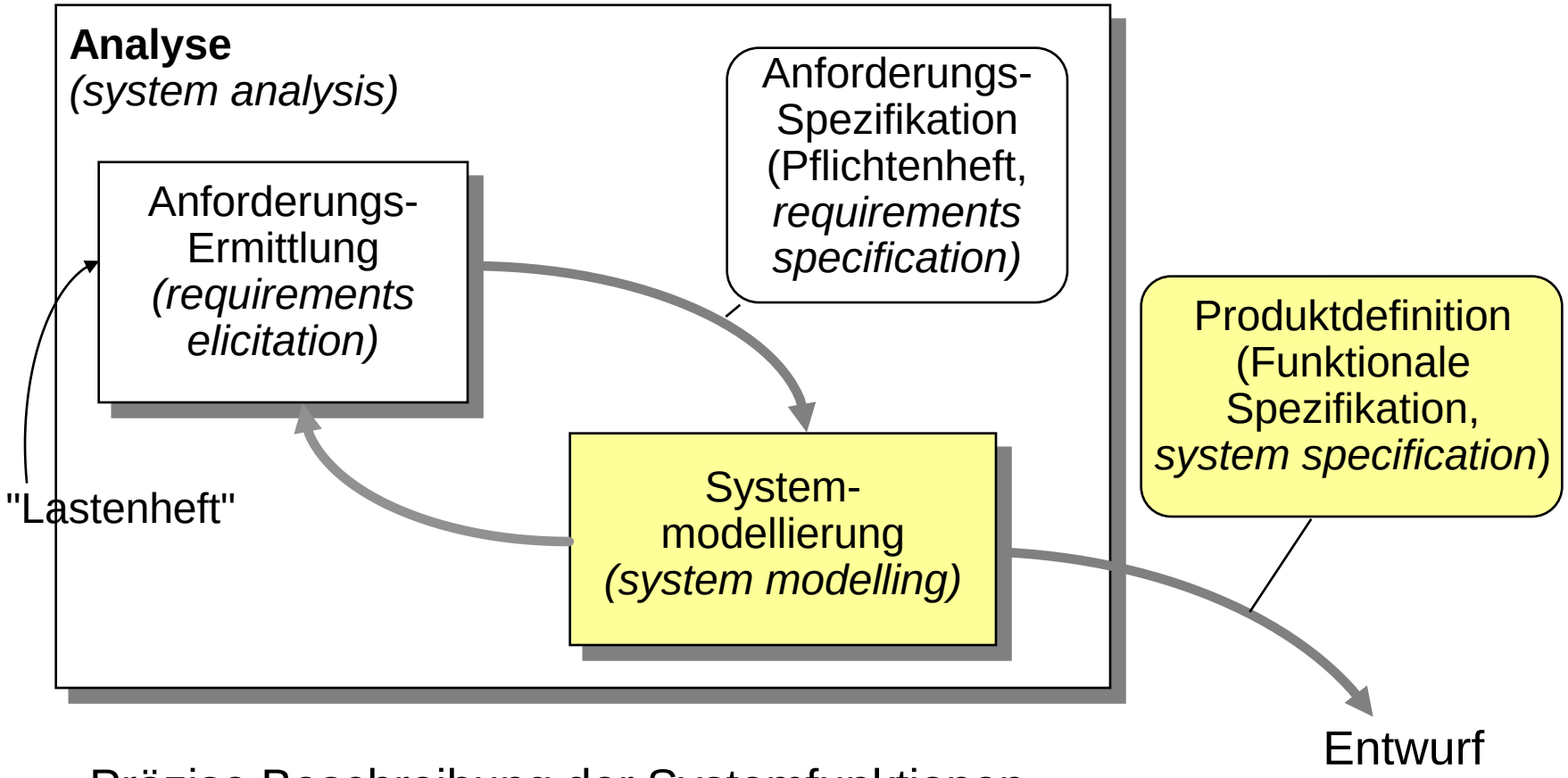


Zustandsautomat



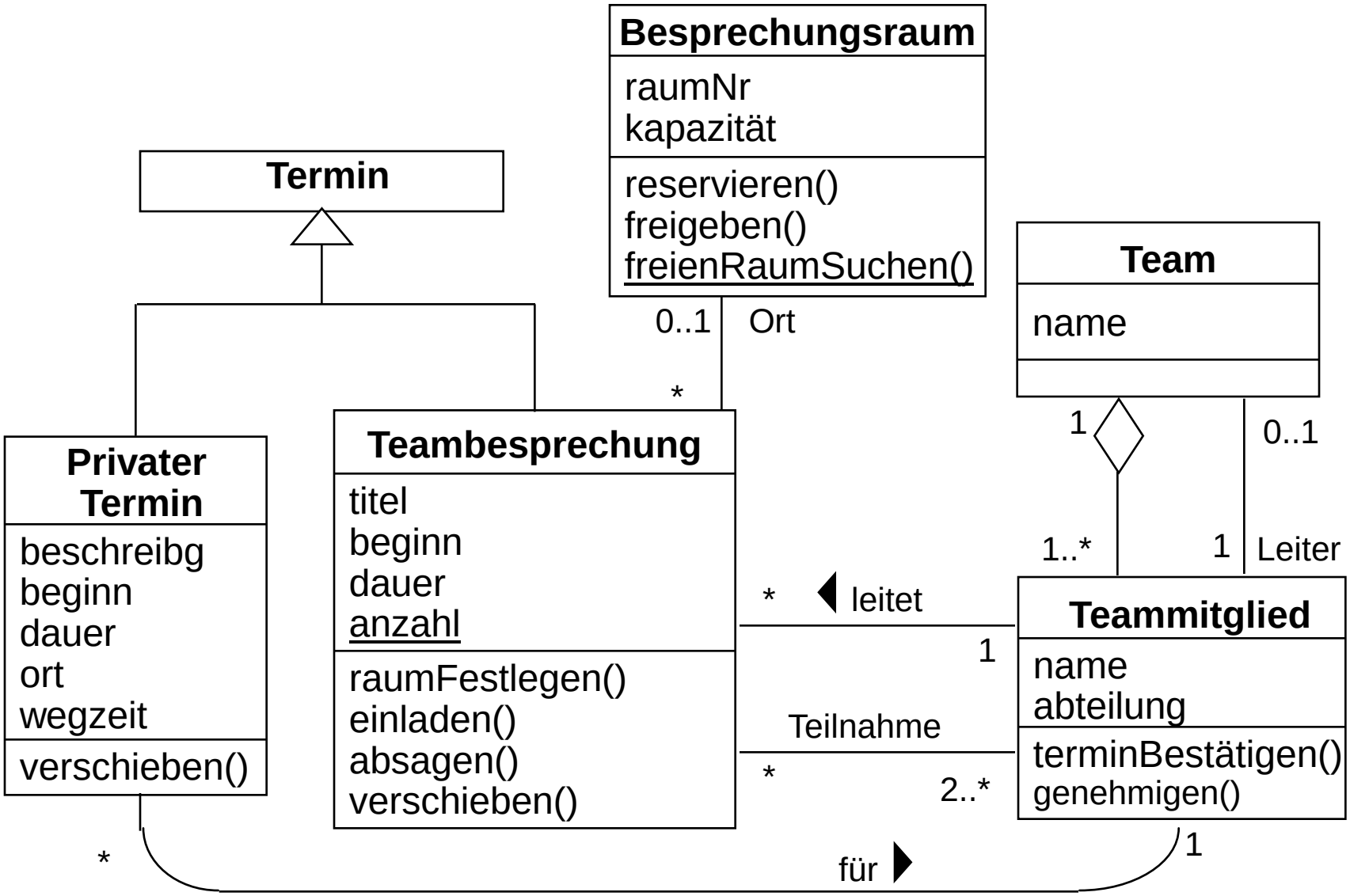
+ Textueller Teil:
  Object Constraint Language (OCL)

Systemmodellierung



- Präzise Beschreibung der Systemfunktionen
- „Was“ ist zu realisieren, ohne das „Wie“ vorherzubestimmen

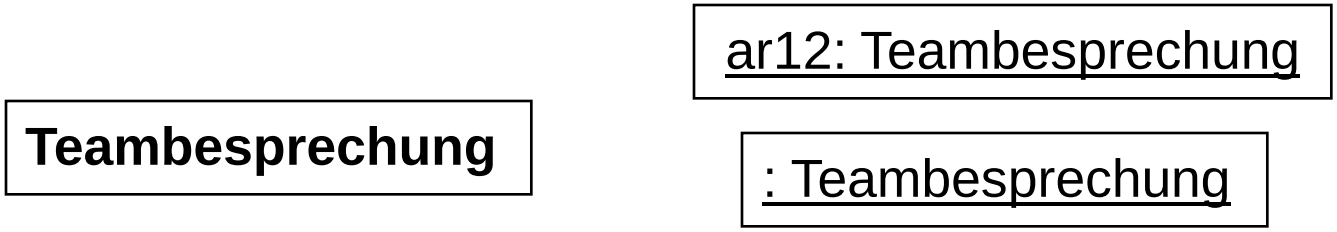
Beispiel für ein Klassendiagramm der UML: Seminar-Organisation



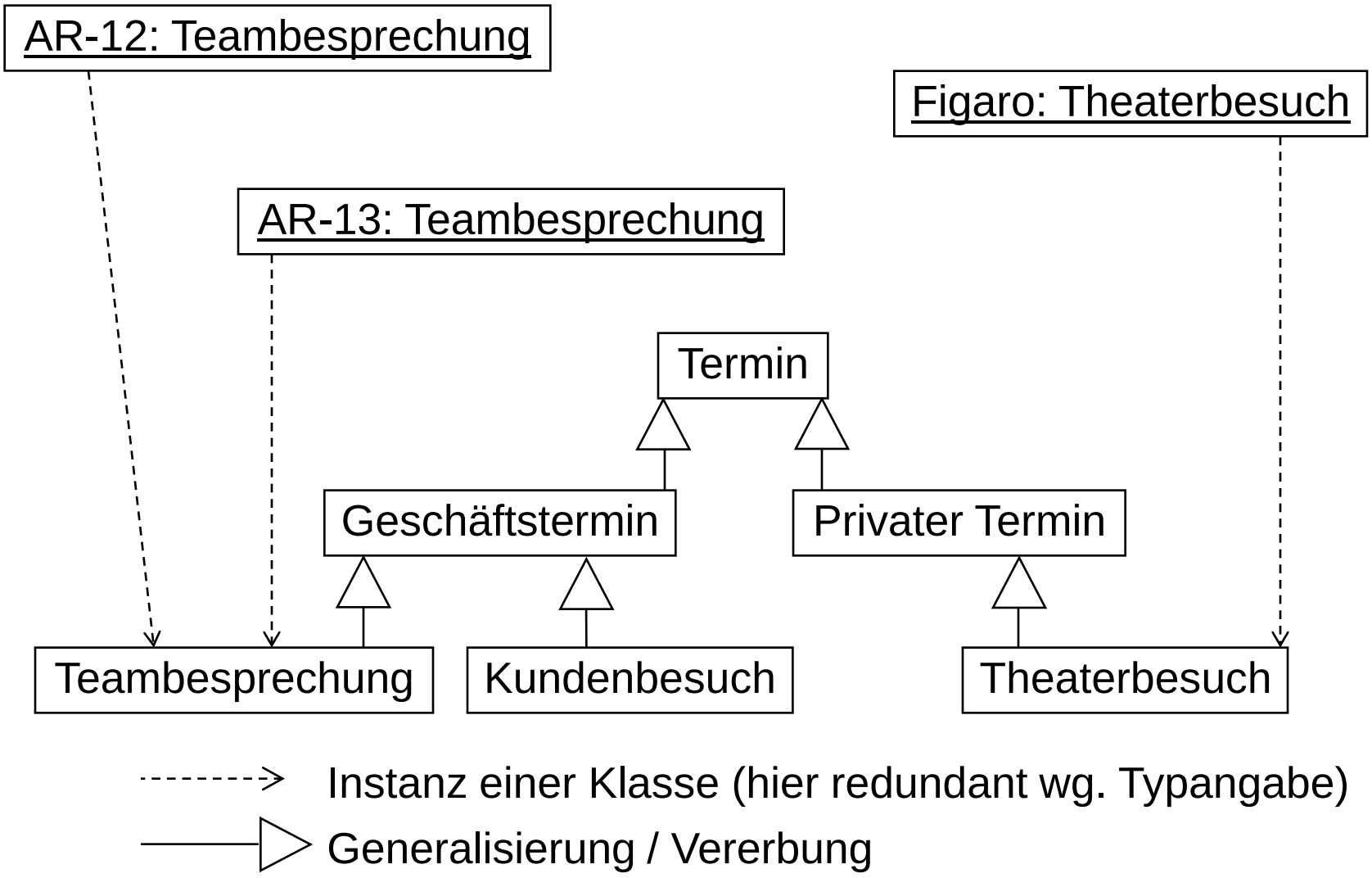
Klassen und Objekte in der Analyse

- Definition: Ein ***Objekt*** ist ein elementarer Bestandteil des betrachteten Fachgebiets.
  Ein Objekt wird erzeugt und behält eine unveränderliche Objektidentität bis zu seiner Löschung.
- Definition: Eine ***Klasse*** ist eine Beschreibung gleichartiger Objekte.
  Jedes Objekt gehört zu (ist Instanz von) genau einer Klasse.
- Notation:

- Beispiele:



Beispiel: Termin-Klasse und Termin-Objekte

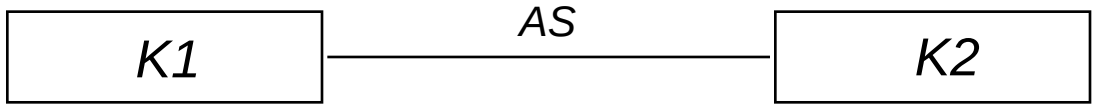


Assoziation in der Analyse

- Definition: Eine (binäre) ***Assoziation*** *AS* zwischen zwei Klassen *K1* und *K2* beschreibt, dass die Instanzen der beiden Klassen in einer fachlich wesentlichen Beziehung zueinander stehen.

- Semantik: Für jedes Objekt *O1* der Klasse *K1* gibt es eine individuelle, veränderbare und endliche Menge *AS* von Objekten der Klasse *K2*, mit dem die Assoziation *AS* besteht.
  Analoges gilt für Objekte von *K2* .

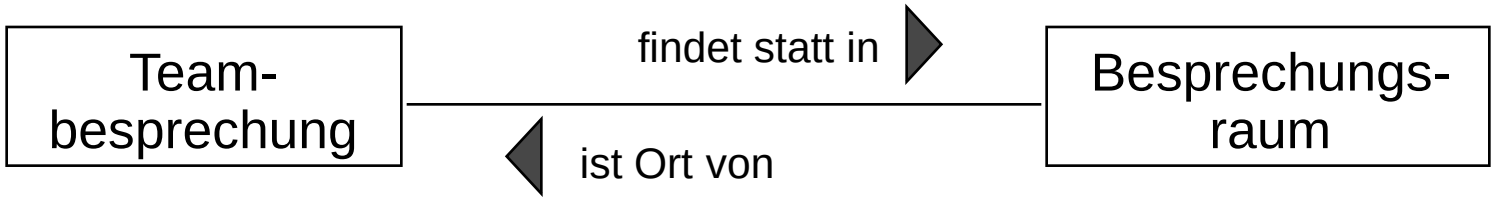
- Notation:



- Beispiel:

Leserichtung und Assoziationsenden

- Für Assoziationsnamen kann die ***Leserichtung*** angegeben werden. Es ist möglich, mehrere Namen für eine Assoziation anzugeben.



- Ein Name für ein ***Assoziationsende*** bezeichnet die Assoziation (evtl. zusätzlich) aus der Sicht einer der teilnehmenden Klassen.

Semantik (bidirektionaler) Assoziationen

- Eine Assoziation ist ähnlich zu einer *Tabelle*:

Teilnahme-Assoziation	
Teambesprechung	Teammitglied
ar12	tm1
ar12	tm3
pbX1	tm1
pbX1	tm2

- Von einem beteiligten Objekt aus betrachtet, gibt eine Assoziation eine *Menge* von assoziierten Objekten an:

  Objekt ar12:Teambesprechung

    Teammitglied-Objekte in Teilnahme-Assoziation: {tm1, tm3}

  Objekt tm1:Teammitglied

    Teambesprechung-Objekte in Teilnahme-Assoziation:
      {ar12, pbX1}

- Beide Sichtweisen sind gleichberechtigt und äquivalent.

Multiplizität bei Assoziationen

- Definition: Die ***Multiplizität*** einer Klasse *K1* in einer Assoziation *AS* mit einer Klasse *K2* begrenzt die Anzahl der Objekte der Klasse *K2*, mit denen ein Objekt von *K1* in der Assoziation *AS* stehen darf.

- Notation:

Multiplizität *Mult:*

n	(genau *n* Objekte der Klasse *K2*)
n..m	(*n* bis *m* Objekte der Klasse *K2*)
n1, n2	(*n1* oder *n2* Objekte der Klasse *K2*)

Zulässig für *n* und *m* :

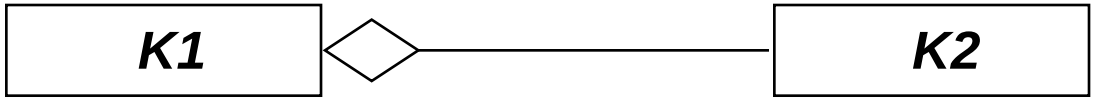
Zahlenwerte	(auch 0)
*	(d.h. beliebiger Wert, einschließlich 0)

- Beispiel:

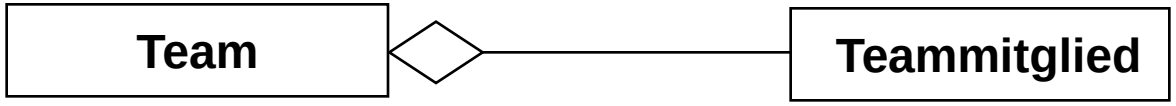
Aggregation

- Definition: Ein Spezialfall der Assoziation ist die *Aggregation*.
- Regel: Wenn die Assoziation den Namen "besteht aus" tragen könnte, handelt es sich um eine Aggregation.
  - Eine Aggregation besteht zwischen einem *Aggregat* und seinen *Teilen.*
  - Die auftretenden Aggregationen bilden auf den Objekten immer eine transitive, antisymmetrische Relation (einen gerichteten zyklenfreien Graphen).
  - Mit der Aggregation sind oft gemeinsame Lebensdauer der Teile mit dem Aggregat impliziert.
  - Umhängen ist allerdings erlaubt (Beispiel: Reifen eines Autos)

- **Notation:**



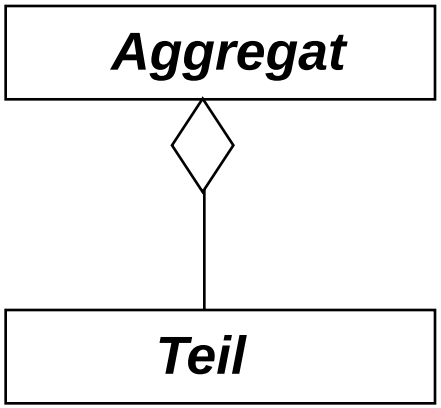
- **Beispiel:**



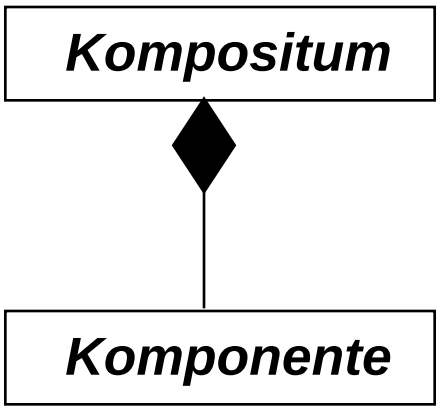
Komposition

- Definition: Ein Spezialfall der Aggregation ist die *Komposition*. Eine Komposition besteht zwischen einem *Kompositum* und seinen *Komponenten*.
  - Ein Objekt kann Komponente höchstens eines Kompositums sein.
  - Das Kompositum hat die alleinige Verantwortung für Erzeugung und Löschung seiner Komponenten.
  - Wenn ein Kompositum gelöscht wird, werden alle seine Komponenten gelöscht.
  - Es herrscht also eine starke auch zeitliche Bindung.
  - Beispiel: Fahrgestell eines Autos

- Notation:



Aggregation



Komposition

Attribute

- Definition: Ein ***Attribut*** *A* einer Klasse *K* beschreibt ein Datenelement, das in jedem Objekt der Klasse vorhanden ist.

  Jedes Objekt der Klasse *K* trägt für jedes Attribut *A* von *K* einen individuellen und veränderbaren Attributwert.

- Notation:

K
A1
...
An

- Beispiel:

Teambesprechung
titel
beginn
dauer

ar12: Teambesprechung
titel = "12.Abteilungsrunde"
beginn = 10.10.2002 09:00
dauer = 60

Datentypen für Attribute

- Definition: Eine Klasse *K* kann für ein Attribut *A* einen bestimmten ***Datentyp*** vorschreiben. In allen Objekten der Klasse *K* sind dann die Attributwerte von *A* von diesem Datentyp.
- Die Syntax für Datentypen ist in UML nicht festgelegt.
  Häufig verwendete Datentypen sind:
  - Einfache Standard-Datentypen (z.B. Boolean, Char, Int)
  - Klassennamen (Dann ist der Attributwert eine *Objektreferenz*.)
- Notation:
  A oder
  A : *Typ* oder
  A : *Typ* = *Anfangswert*
- Beispiel:

Teambesprechung
titel: String beginn: Date dauer: Int = 60

Klassenattribute

- Ein ***Klassenattribut*** *A* beschreibt ein Datenelement, das genau einen Wert für die gesamte Klasse annehmen kann. (Gewöhnliche Attribute heißen auch ***Instanzattribute***, weil sie für jede Instanz individuelle Werte annehmen.)

- Notation: Unterstreichung

  A : *Typ*

- Beispiel:

Teambesprechung
titel: String beginn: Date dauer: Int anzahl: Int

- Aber: Klassenattribute verursachen sehr(!) viel Test- und Wartungsprobleme. Deshalb soweit wie möglich vermeiden!

Operation

- Definition: Eine ***Operation*** einer Klasse *K* ist die Beschreibung einer Aufgabe, die jede Instanz der Klasse *K* ausführen kann. In der Beschreibung der Klasse wird der Name der Operation angegeben.

- Notation:

Klasse
Attribut_1 ... Attribut_n
Operation_1 ... Operation_m

- Beispiel:

Teambesprechung
titel beginn dauer
raumFestlegen() einladen() absagen()

Zur besseren Unterscheidung von Attributnamen werden meist Klammern hinter Operationsnamen gesetzt, auch wenn über die Parameterliste noch keine Aussagen gemacht werden sollen.

Parameter und Datentypen für Operationen

- Detaillierungsgrad:
  - Analysephase: meist Operationsname ausreichend
  - Parameternamen und Datentypen *können* angegeben werden (manchmal auch Parametername ohne Datentyp)
  - Später (Entwurfsphase) sind vollständige Angaben nötig.

- Notation:

  Operation (*Parameter:ParamTyp*, ...): *ResultTyp*

- Beispiele (Klasse Teambesprechung)**:**

  raumFestlegen (wunschRaum: Besprechungsraum): Boolean

  absagen (grund: String);

Klassenoperation

- Definition: Eine ***Klassenoperation*** *A* einer Klasse *K* ist die Beschreibung einer Aufgabe, die nur unter Kenntnis der aktuellen Gesamtheit der Instanzen der Klasse ausgeführt werden kann. Gewöhnliche Operationen heißen auch ***Instanzoperationen.***

- Notation:

  Unterstreichung analog zu Klassenattributen.

- Beispiel:

Besprechungsraum
raumNr kapazität
reservieren() freigeben() freienRaumSuchen()

Konstruktor

- Definition: Ein ***Konstruktor*** *C* einer Klasse *K* ist eine spezielle Klassenoperation, die eine neue Instanz der Klasse, d.h. ein neues Objekt, erzeugt und initialisiert.

  Ergebnistyp von *C* ist immer implizit die Klasse *K*.

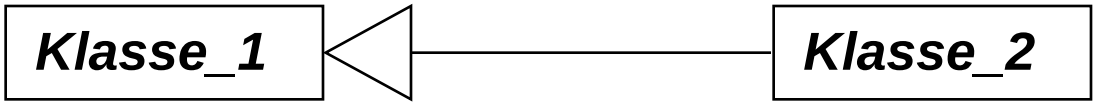
  Ein Konstruktor ohne Parameter wird implizit für jede Klasse angenommen. Explizite Konstruktoren können mit "<<constructor>>" markiert werden.

- Beispiel:

Besprechungsraum
raumNr kapazität
reservieren() freigeben() freienRaumSuchen() Besprechungsraum (raumNr, kapazität) <<constructor>>

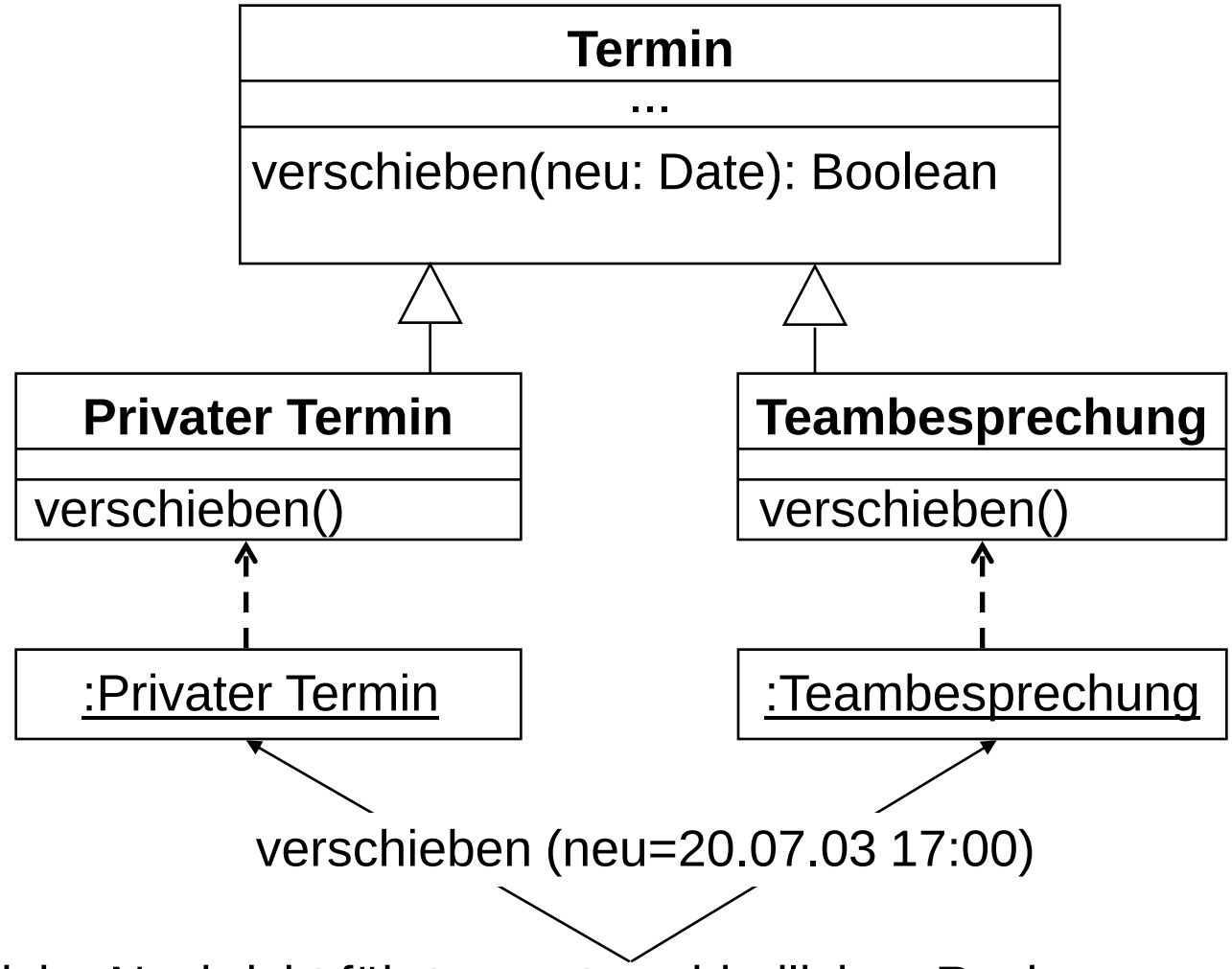
Vererbung

- Definition: Eine ***Vererbungsbeziehung*** von einer Klasse *K1* zu einer Klasse *K2* ist eine Beschreibung der Tatsache, dass alle Objekte der Klasse *K2* zusätzlich zu den in der Klasse *K2* beschriebenen Eigenschaften auch alle Eigenschaften der Klasse *K1* haben.
  'Eigenschaften' sind hier
  - die Liste der Attribute
  - die Teilnahme an Assoziationen, Aggregationen und Kompositionen
  - die Liste der Operationen.
- Notation:



- Umgangssprachlich: Jedes *Klasse_2* **ist ein** *Klasse_1*.
  Klasse_2 ist **Spezialfall** von *Klasse_1*.

Polymorphie

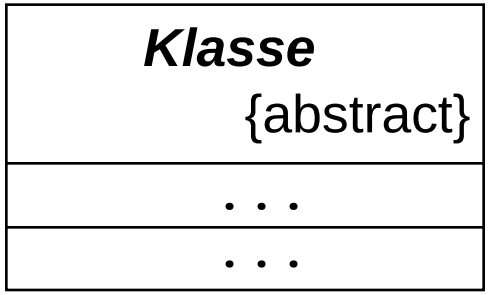


- Die gleiche Nachricht führt zu unterschiedlichen Rechenvorschriften, abhängig vom Empfänger (*dynamische Bindung*).

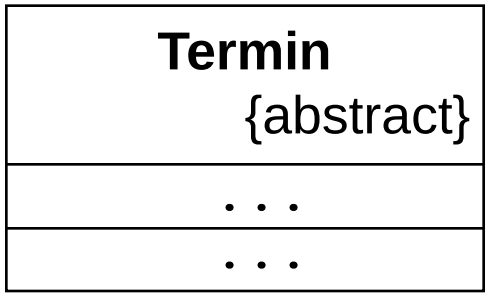
Abstrakte und konkrete Klassen

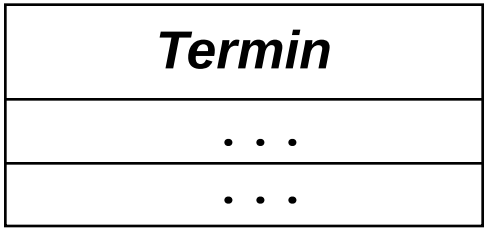
- Definition: Eine Klasse kann als ***abstrakt*** deklariert werden. In diesem Fall ist es nicht zulässig, Instanzen der Klasse zu bilden.
  Abstrakte Klassen dienen als "Schema" in der Vererbung und als Gruppierungsmittel, zum Beispiel für gemeinsame Funktionalität.
  Eine Klasse, von der Instanzen gebildet werden können, heißt ***konkret***.

- Notation:



- Beispiel:

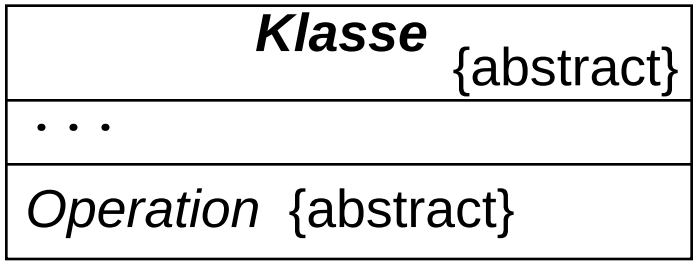




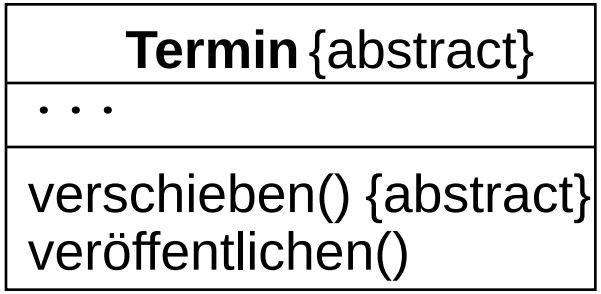
Abstrakte und konkrete Operationen

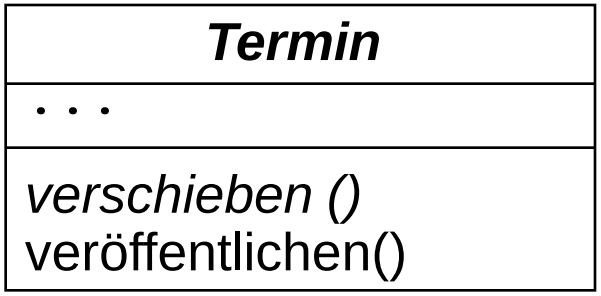
- Definition:
- Eine Operation *OP* ist ***abstrakt***, wenn sie keine Implementierung besitzt. Abstrakte Operationen treten nur in abstrakten Klassen auf. Das Verhalten von *OP* muss dann in Unterklassen definiert werden.

- Notation:



- Beispiel:





Beispiel für ein Klassendiagramm der UML: Seminar-Organisation