Vorlesung Softwaretechnik

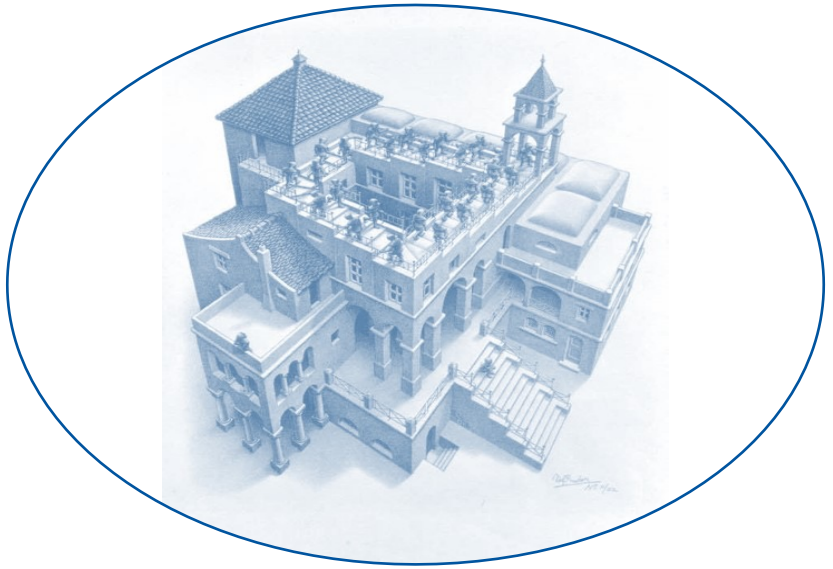
Prof. Bernhard Rumpe
Software Engineering
RWTH Aachen

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

SE Software Engineering | RWTH AACHEN UNIVERSITY

Softwaretechnik

1. Warum, was, wie und wozu Softwaretechnik?
1.1 Einleitung und Motivation



Prof. Bernhard Rumpe
Software Engineering
RWTH Aachen

http://www.se-rwth.de/
@*SE_RWTH*

Literatur:

- Sommerville 1.1
- Balzert Band 1, LE 1

SE Software Engineering | RWTH AACHEN UNIVERSITY

Boeing

19.05.2019

Unglücksmaschine 737 Max

Boeing gesteht erstmals Software-Fehler ein

Teilen:

Montage einer Boeing 737 Max in einem Werk im US-Staat Washington: Diese Flugzeuge dürfen derzeit nicht eingesetzt werden.

Flugsteuerung

Boeing gesteht zweiten Software-Fehler bei 737 Max

Nicht nur das umstrittene MCAS-System hat offenbar Mängel. Boeing entdeckte einen zweiten Software-Fehler bei der Flugsteuerung der 737 Max.

05.04.19 - 11:18 | Felix Stoffels

104 Kommentare

Boeing 737 Max: Neues Softwareproblem. *Boeing*

Quellen: https://www.manager-magazin.de/unternehmen/artikel/boeing-737-max-boeing-gesteht-erstmals-software-fehler-ein-a-1268189.html; https://www.aerotelegraph.com/boeing-gesteht-zweiten-software-fehler-bei-737-max

4. Juni 1996: Erster Start der "Ariane-5"

- Während des Fluges läuft ein unnötiges Kalibrierungsprogramm für die Trägheitssensoren.
- Die gemessenen Werte der Ariane-5 überschreiten die in der Ariane-4-Software vorgesehenen Bereiche.
- Die (Ada-)Exception wird durch Anhalten des Steuerungscomputers behandelt, um auf ein zweites redundantes System umzuschalten.
- Im zweiten System tritt der gleiche Software-Fehler auf und wird identisch behandelt.

- Kosten des Ariane-5-Programms bis 1996: ca. 8 Milliarden US-$
- Wert des zerstörten Satelliten: ca. 500 Millionen US-$

Warum Software Engineering?

Intel Pentium FDIV Fehler (1994)

- Gleitkomma-Divisionen mit bestimmten Werten fehlerhaft
- Schaden: 475 Mio. US-$

NASA Mars Climate Orbiter (1998)

- Fehlerhafte Konvertierung imperialer in metrische Einheiten
- Mars-Orbit verpasst → Orbiter verloren
- Schaden: 125 Mio. US-$

Millennium-Bug (1999)

- Jahreszahlen zweistellig gespeichert
- Berechnungsfehler bei Umstellung: (19)99+1=(19)00
- Schaden: 600 Mil. US-$ (Gartner; geschätzt)

Weitere

https://raygun.com/blog/costly-software-errors-history/
https://www.computerworld.com/article/3412197/top-software-failures-in-recent-history.html
https://en.wikipedia.org/wiki/List_of_software_bugs

Softwaredebakel: Keine Einzelfälle

- Terminliche, finanzielle und technische Katastrophen:
  - 1999: Verlust der Sonde "Mars Climate Orbiter" wegen falscher Einheitenumrechnung
  - 2003-5: Toll Collect: Termin u.a. wegen Softwareproblemen mehrfach verschoben, Einnahmeausfall > 5 Milliarden Euro
  - 2004: Einheitliche Software „Fiskus“ für alle Finanzämter wird nach 13 Jahren und 900 Mio. Euro aufgegeben
  - 2009: Gleichzeitiger Ausfall aller 7000 Fahrkartenautomaten der DB, Kosten ca. 500.000 Euro in vier Stunden
  - 2012: Bundesweites Studienzulassungssystem 2 Jahre verschoben: Chaos bei der Einschreibung
  - 2013: Hunderte Flugausfälle: Telefon-Crash in der europäischen Luftsicherung

- Weltweit jährlicher Schaden durch Softwarefehler 6,2 Billionen US$

Quelle: A White Paper by Roger Sessions: The IT Complexity Crisis: Danger and Opportunity (2009)

Warum Softwaretechnik?

- Größe der Systeme wächst
- Komplexität der Systeme steigt
- Heterogenität nimmt zu
- Software ist lange in Betrieb

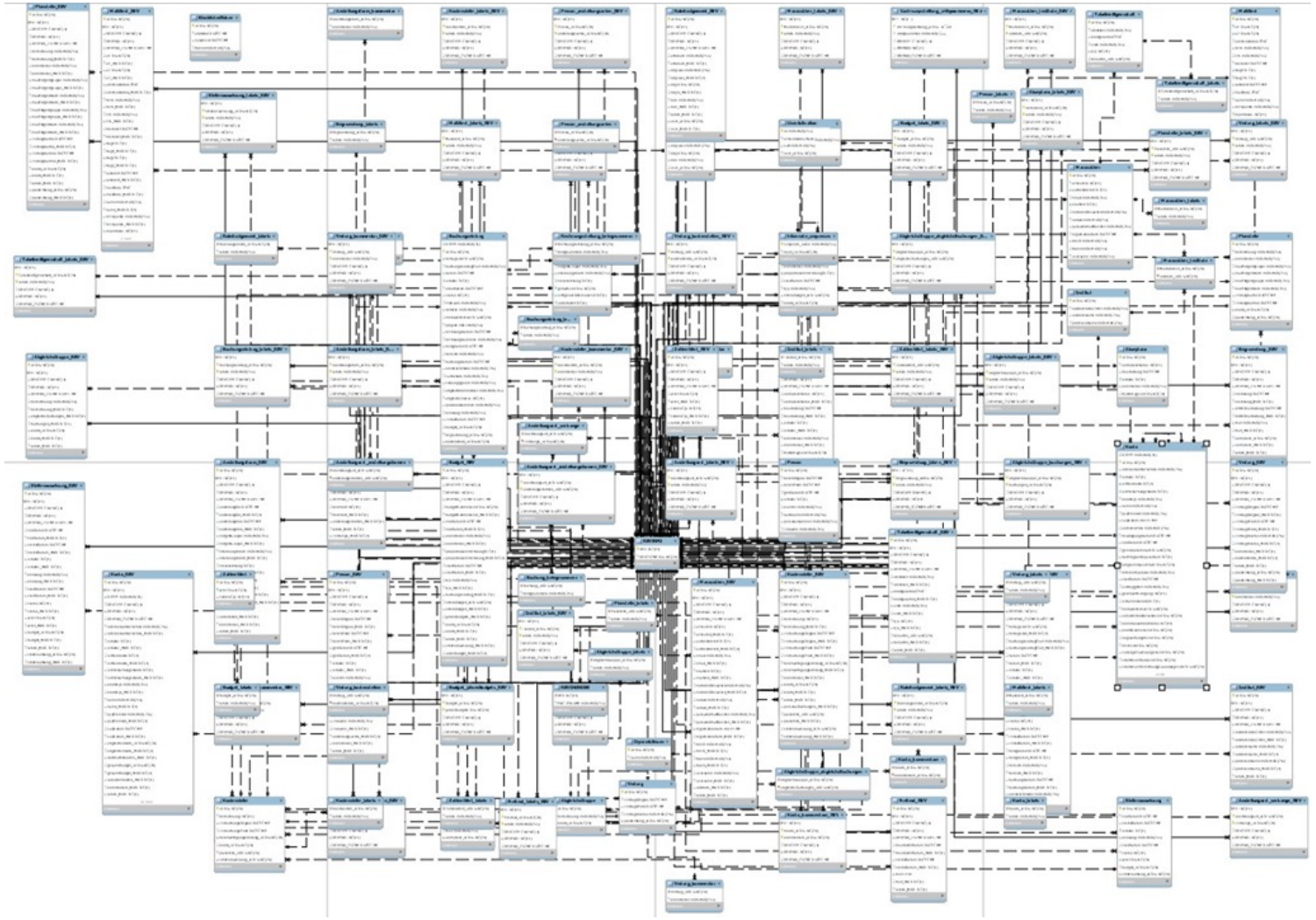
- Softwareprojekte scheitern, kosten mehr als geplant, werden später fertig
- Software ist nicht wiederverwendbar, nur schwer erweiterbar
- Fehler in Sicherheitskritischer Software verursachen große Schäden

- Menschenleben sind in Gefahr

Worüber wir nicht reden…

animated

Worüber wir reden…



Komplexität beherrschen

z.B. Software MaCoCo, 3/2020: app. 400.000 LOC Code + 8.000LOC Modelle, 11 MA, 4 Jahre Dev.

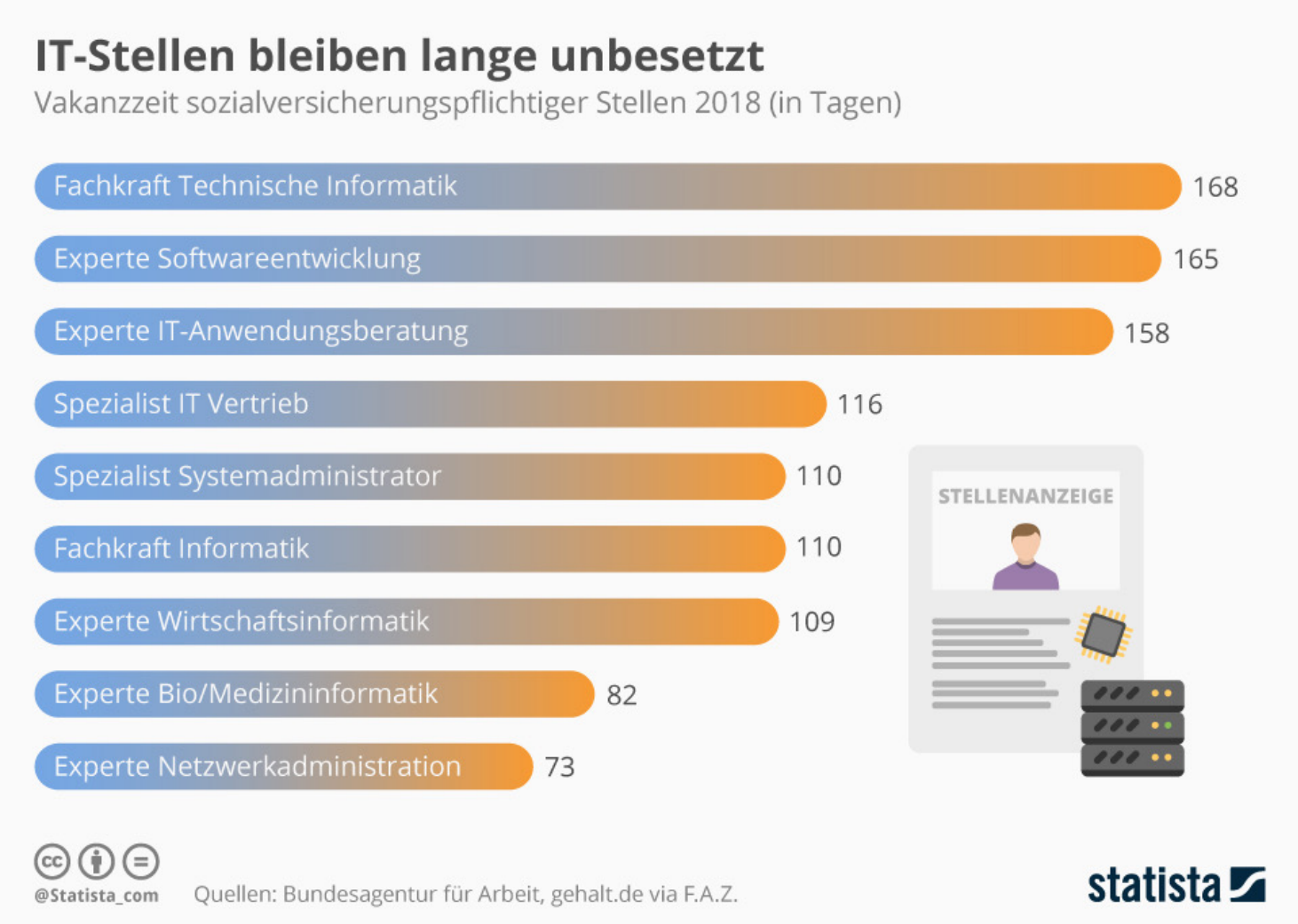
Worüber wir reden… Software Engineering

> „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 drei Aspekte:
  - Kosten
  - Termine
  - Qualität
    (Korrektheit, Zuverlässigkeit, Performanz, Sicherheit, Nutzbarkeit, Verständlichkeit, Weiterentwickelbarkeit, Anpassbarkeit, Wartbarkeit)

Was Sie mit Software Engineering anfangen können ..

- Entwicklung großer komplexer Softwaresysteme
  - an entscheidender Stelle
- Sichere Jobs:
  - Große Auswahl an Stellenangeboten
- Viel Geld verdienen
- Spaß an der Arbeit



Softwaretechnik

1. Warum, was, wie und wozu Softwaretechnik?
1.2. Organisatorisches

Prof. Bernhard Rumpe
Software Engineering
RWTH Aachen

http://www.se-rwth.de/
@*SE_RWTH*

Literatur:

- Sommerville 1.1
- Balzert Band 1, LE 1

SE Software Engineering | RWTH AACHEN UNIVERSITY

Ansprechpartner

- Prof. Bernhard Rumpe
  rumpe @ se-rwth.de
- Hendrik Kausch
  kausch @ se-rwth.de
- Mathias Pfeiffer
  mpfeiffer @ se-rwth.de
- Deni Raco
  raco @ se-rwth.de

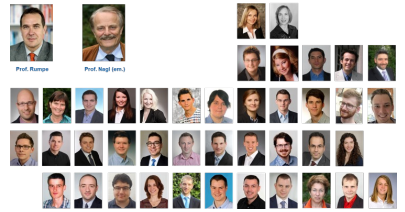



INFORMATIONEN

Moodle: https://moodle.rwth-aachen.de/course/view.php?id=28210

SE - Arbeitsgruppen und Themen

SE Software Engineering



Automotive / Cyber-Physical Systems

- Autonomous Driving
- Simulation
- Deep Learning
- Data Science
- Modeling Embedded Systems
- Generative Test-Driven Development

Model-Driven Systems Engineering

- Systems Modeling Languages
- Domain-Specific Application
- Software Architectures
- Semantic Tool Integration
- Variability & Product Lines
- Industry 4.0, CPPS, Robotics

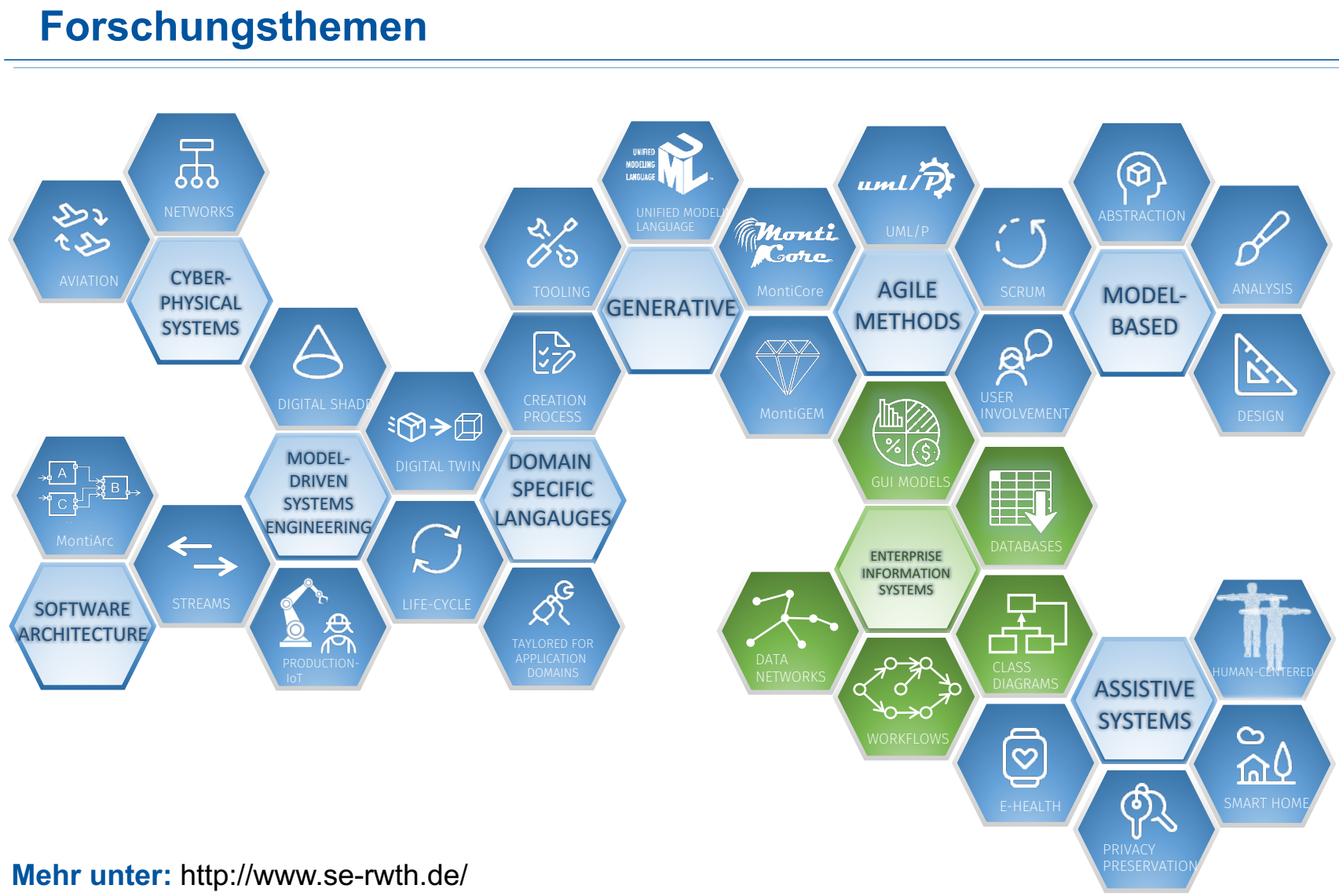
Model-Based Assistance and Information Services

- Models in Information and Workflow Systems
- Human-Centered Assistance
- Behavior/ Process Modeling
- DSLs for rights, roles, privacy
- Digitalization and digital transformation

Modeling Language Engineering

- Development Tools
- Language Workbench MontiCore
- UML, SysML, Architecture DL
- Domain-specific languages (DSL)
- Generation, synthesis
- Testing, analysis, verification
- Software architecture, evolution
- Agile methods

Forschungsthemen



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

Auszug Unserer Kooperationen mit der Industrie

Anwendungsgebiete

- Ambient Assisted Living
- Automotive
- Avionics
- Controlling and Finances
- Cyber-Physical Systems
- Energy
- Health
- Internet of Production (IoP)
- Internet of Things (IoT)
- Industry 4.0
- Logistics
- Robotics
- Smart Homes

Siehe auch: https://www.se-rwth.de/jobs/

Promotion am Lehrstuhl für Software Engineering

- Der Lehrstuhl SE sucht sowohl
  - wissenschaftliche Mitarbeiter:innen (Doktorand:in)
  - als auch HiWis

- **Promotionsstellen für wissenschaftliche Mitarbeiter:innen**
  - Modellbasierte Software-Entwicklung und Analyse
  - Digitale Zwillinge
  - NLP im Automobilen Ziele- und Anforderungsmanagement
  - Modell-basiertes DevOps
  - oder Sie bewerben sich initiativ!

- **HiWi-Stellen: Entwicklung für …**
  - InviDas: Plattform zur Vermittlung von Datenschutzkonzepten für Smart-Wearables
  - MaCoCo: Entwicklung eines webbasierten Management Cockpits
  - MontiCore Neue Features für die Language Workbench MontiCore
  - Tools für Modelle in Wissenschaft und Industrie generell … Basis: Java

Siehe auch: https://www.se-rwth.de/jobs/

SE in der Lehre (B.Sc./M.Sc. Informatik)

- **Softwaretechnik** im 3ten Semester (6 CP)

- Vorlesungen im Bachelor / Master:
  - Model-Based Software Engineering (6 CP, Prof. Rumpe)
  - Model-Based Systems Engineering (6 CP, Prof. Rumpe)
  - Software Language Engineering (6 CP, Prof. Rumpe)

  - Software-Architekturen (6 CP, Prof. Nagl)
  - Die Softwaretechnik-Programmiersprache Ada 95 (6 CP, Prof. Nagl)

  - Der digitale Lebenszyklus von Fahrzeugen als Teil des Internet of Things (IoT), (3 CP oder 6 CP mit Praxis Workshop, Dr. Fischer, DSA)

  - Prozesse und Methoden beim Testen von Software (3 CP oder 6 CP mit Praxis Workshop, Dr. Stefan Kriebel, FEV.io)

  - Praktika
  - Seminare
  - Bachelor-, Masterarbeit

Organisation der SWT Vorlesung

MODUL

Bachelor, vertiefende Vorlesung und Übung (3+2)

TERMINE

Dienstags, 18:30 bis 20:00 Uhr
Mittwochs, 18:30 bis 20:00 Uhr
Donnerstags, 16:30 bis 18:00 Uhr

Raum: **Größer Hörsaal Audimax**

GLOBALÜBUNGEN

Abwechselnd mit VO
Übungsgruppen bilden: 3-4 Studierende – Forum in Moodle nutzen

KLAUSUR

120 Minuten, 120 Punkte
Bestehen der Übungen nicht erforderlich zur Zulassung, verschafft aber Bonus.

INFORMATIONEN

Moodle: https://moodle.rwth-aachen.de/course/view.php?id=28210

Vorlesung Softwaretechnik

VORAUSSETZUNGEN

gute Programmierkenntnisse (ideal ist Java oder ähnliche Sprache)
(Wissen über Programmieren von Algorithmen wird hier kaum mehr vermittelt.)

HÖRER:iNNENKREIS

BA Informatik
BA Mathematik
BA Technik-Kommunikation, NF Informatik
BA Maschinenbau
BA Computational Engineering Science
MA Data Science
MA Automatisierungstechnik
MA Energietechnik
MA Verfahrenstechnik
MA Software Systems Engineering
MA Computational Engineering Science
weitere Nebenfächer

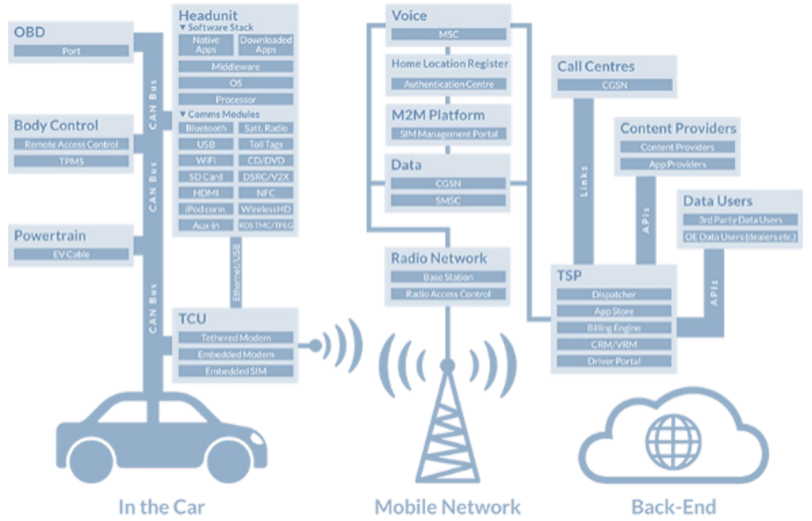
Bonuspunkte aus der Übung

- Geplant sind 10 Übungen mit jeweils 10 Übungs-Punkten → maximal 100 Ü-Punkte in Übungen erreichbar.
- Klausur für die Vorlesung: maximal 120 Klausur-Punkte erreichbar.
- **Ab** 50% aller Übungspunkte **und Bestehen der Klausur** werden Bonuspunkte für die Klausur angerechnet.
- Beispiele für keine Anrechnung von Bonuspunkten, z.B.
  - 49% der Übungspunkte:
  - 95% der Übungspunkte aber Klausur nicht bestanden

Punkte in der Übung	Bonuspunkte in der Klausur
< 50 oder Klausur nicht bestanden	0
=50	+2
>50 bis =60	+3
>60 bis =70	+4
>70 bis =80	+5
>80 bis =90	+7
>90 bis 100	+10

Softwaretechnik

1. Warum, was, wie und wozu Softwaretechnik?
1.3. Softwaresysteme



Prof. Bernhard Rumpe
Software Engineering
RWTH Aachen

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

Literatur:

- Sommerville 1.1
- Balzert Band 1, LE 1

SE Software Engineering | RWTH AACHEN UNIVERSITY

Softwaresysteme

Software: computer programs, procedures, rules, and possibly associated documentation and data pertaining to the operation of a computer system.

(IEEE Standard Glossary of Software Engineering)

Software: Software ist ein Sammelbegriff für

- Programme in menschenlesbarer Quellform,
- ausführbarer Objektcode-Form,
- Dokumentation für Entwickler, Betreiber und Nutzer, einschließlich
- ausführbarer Installations- und Compilations-Skripte, sowie die
- Testumgebung nebst Software-Tests,

die für die Weiterentwicklung, Installation und Nutzung notwendig sind.

(RWTH Software Verträge, Begriffsklärung)

Software-Produkt

- Software als Sammelbegriff bis heute nicht ganz vereinheitlicht, deshalb besser:

- Softwaresystem
  - Ein System (oder Teilsystem), dessen Komponenten aus Software bestehen.

- Software-Produkt
  - Ein Produkt ist ein in sich abgeschlossenes, i. A. für einen Auftraggeber bestimmtes Ergebnis eines erfolgreich durchgeführten Projekts oder Herstellungsprozesses.

    Als Teilprodukt oder Komponente bezeichnen wir einen abgeschlossenen Teil eines Produkts.

Klassifikation von Software?

- Welche Arten von Software nutzen wir?
  - Generisches Produkt oder Einzelanfertigung?
  - Systemsoftware (Betriebssystem, Compiler, Editor, ...) oder Anwendungssoftware (application software)?
  - Produktintegriert (embedded) oder für reine Computersysteme?
  - Echtzeitanforderungen oder flexiblere Zeitanforderungen?
  - Daten-, berechnungs- oder kommunikationsintensiv?
  - Monolithisch oder verteilt?
  - Standalone oder mit anderen Anwendungen integriert?
  - Lokal oder Remote oder in der Cloud?
  - Service eines integrierten Netzwerks?

Besonderheiten von Software

- Software ist immateriell.
- Software wird nicht durch physikalische Gesetze begrenzt.
- Software unterliegt keinem Verschleiß.

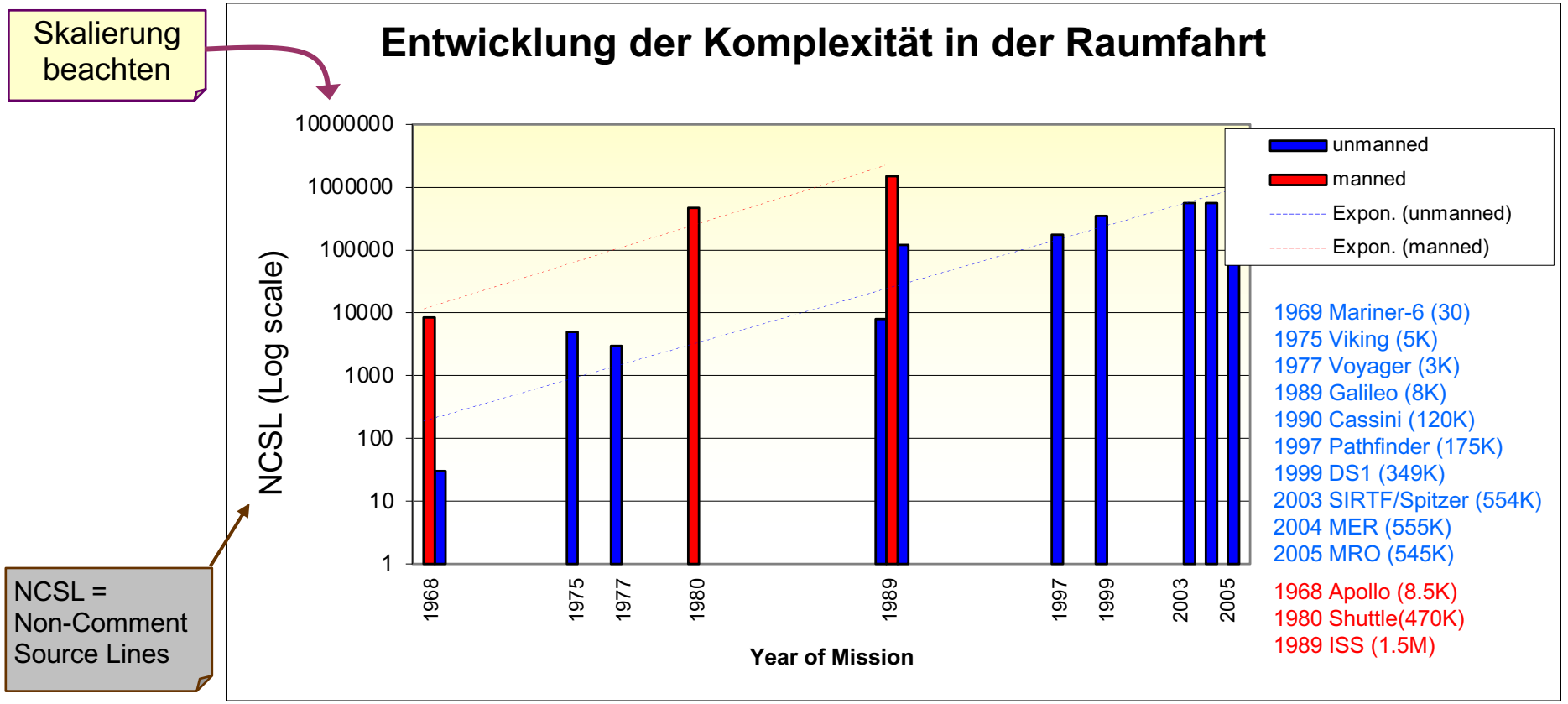
- Defekte sind immer Konstruktionsfehler.
- Software ist schwer zu vermessen
  („Technische Daten“ von Software).

- Software gilt als relativ leicht änderbar
  (im Vergleich zu materiellen technischen Produkten).
- Software unterliegt einem ständigen Anpassungsdruck.
- Software veraltet.

Einige Wichtige Angestrebten Eigenschaften von Software

- *Zuverlässigkeit*
  - Software darf im Fall des Versagens keine physischen oder ökonomischen Schäden verursachen.
- *Benutzbarkeit*
  - Software muss sich nach den Bedürfnissen der Benutzer richten.
  - Die Benutzerschnittstelle muss ergonomisch und selbsterklärend sein.
  - Dokumentation muss in allen Detaillierungsgraden ausreichend zur Verfügung stehen.
- *Wartbarkeit*
  - Software muss anpassbar an neue Anforderungen sein.
  - Software sollte möglichst plattformunabhängig sein.
- *Effizienz*
  - Software muss ökonomischen Gebrauch von Ressourcen des unterliegenden Systems machen.

Wachsende Komplexität (2)

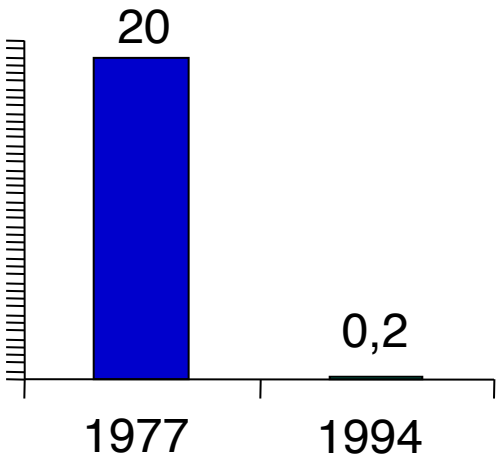


Wachstumsrate ~10x alle 10 Jahre

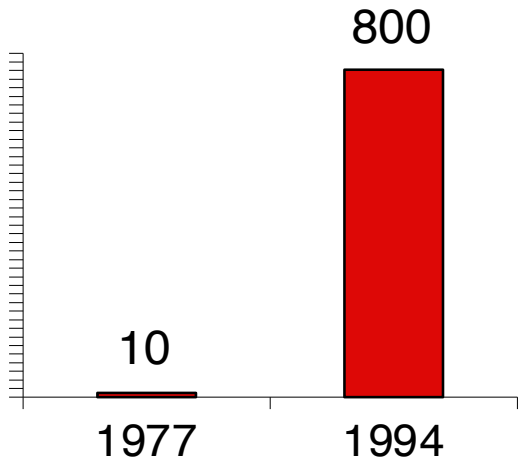
Quelle: Nasa - Flight Software Complexity - 2009

Komplexitätswachstum und Fehlerrate

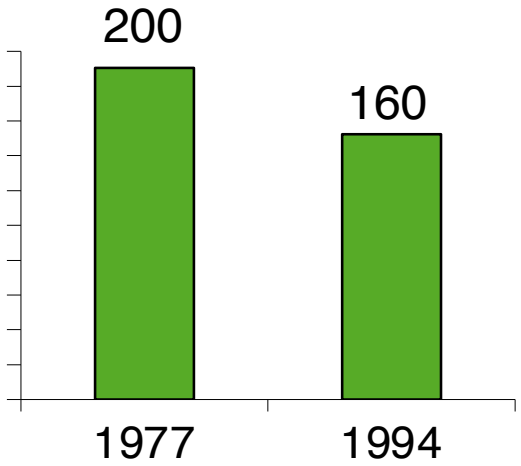
⇨ animated



Anzahl Fehler auf 1000 LOC



Programmgröße (1000 LOC)

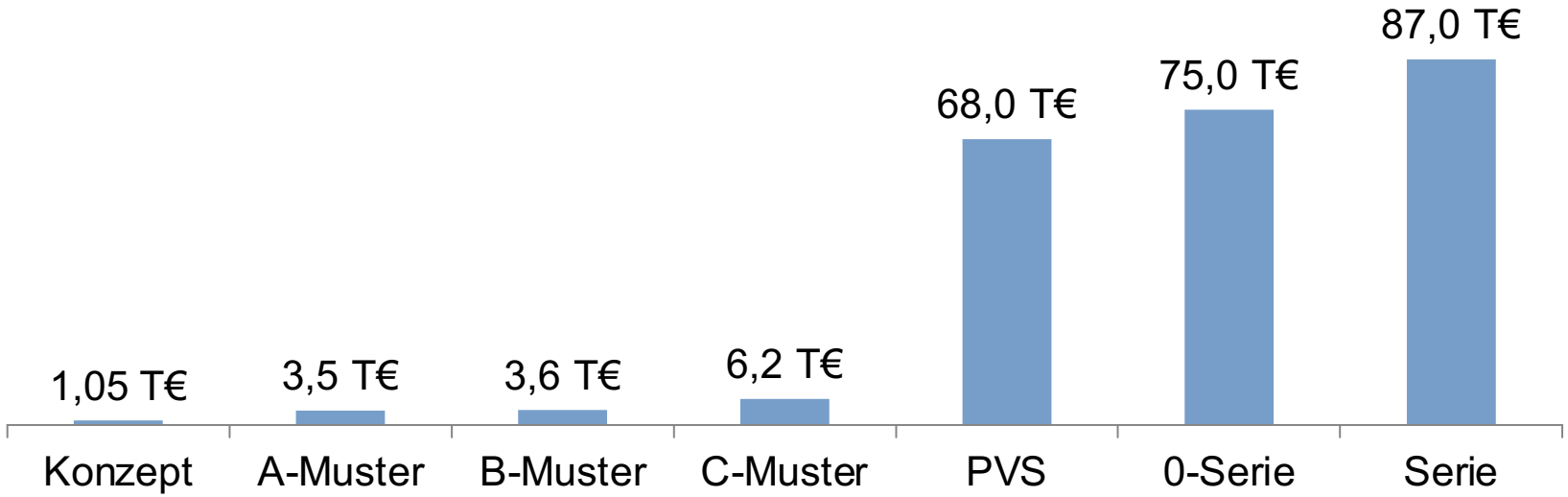


Resultierende absolute Fehleranzahl

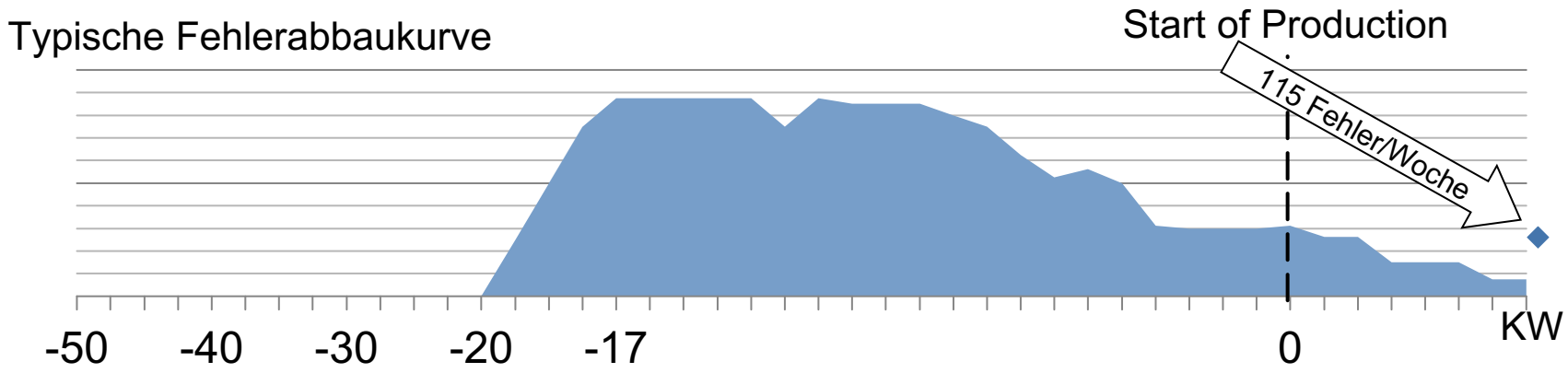
Echte Qualitätsverbesserungen sind nur möglich, wenn die Steigerung der Programmkomplexität **überkompensiert** wird !

(Durchschnittswerte, aus Balzert 96)

Geschätzte relative Abstellkosten je Software-Fehler bei der Fahrzeugentwicklung



Quelle: HIS, Stand 1999



Quelle: G.Wagner, M. Liebler, 2007

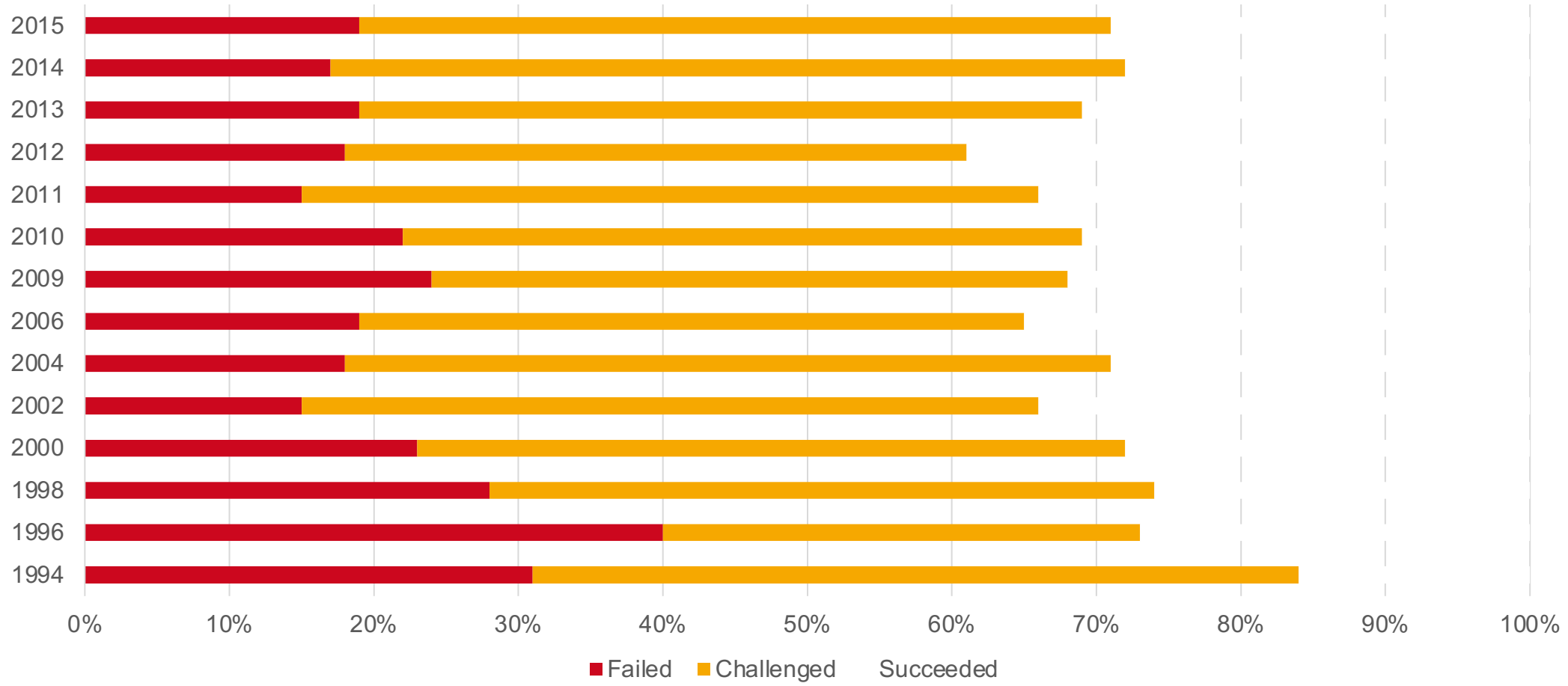
Zunehmende Qualitätsanforderungen

- Gefundene Defekte in 1000 Zeilen Quellcode (M. Cusumano, MIT):
  - 1977: 7 - 20 Defekte
  - 1994: 0,05 - 0,2 Defekte
  - 2003: 0,02 - 0,04 Defekte

- Steigerung des Qualitätsniveaus um den Faktor 100 in 13 Jahren.
- Aber: Komplexitätssteigerung muss kompensiert werden !
  - Komplexitätssteigerung ca. Faktor 10 in 5 Jahren

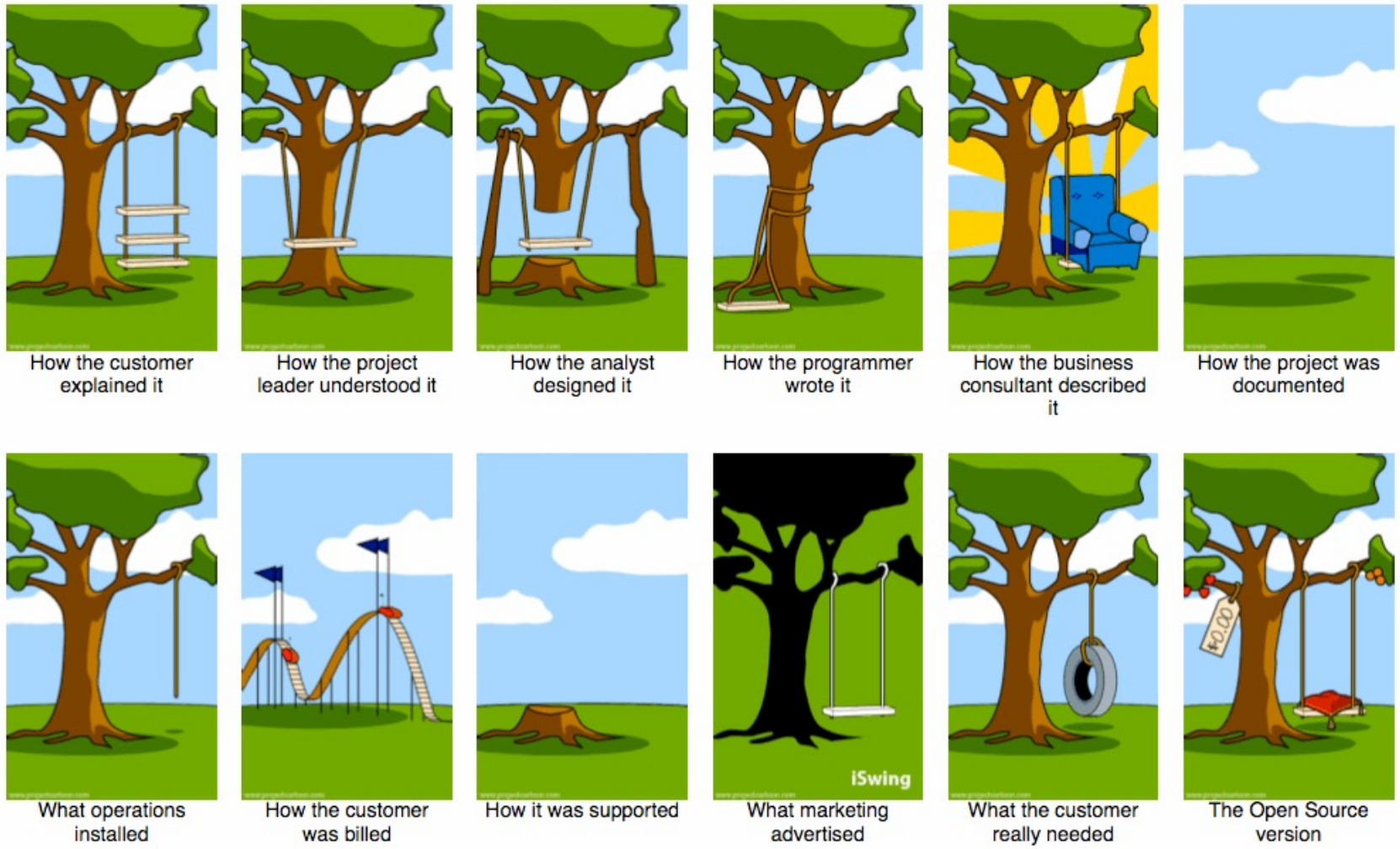
- Zunehmende „Altlasten“:
  - Anwendungssoftware wird oft 20 Jahre und länger eingesetzt.
  - In manchen Betrieben sind 60-70% der Softwarekosten für Anpassung von Altsoftware!

Erfolgsstatistik von IT-Projekten



Quelle: CHAOS Report, 1994-2015, Standish Group International, Inc.

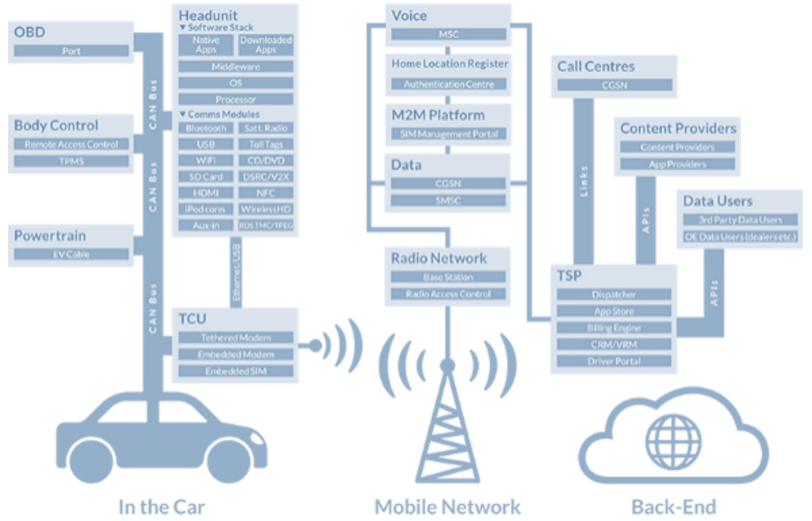
Herausforderungen



Quelle: https://www.conversationagent.com/2010/01/what-really-affects-behavior.html

Softwaretechnik

1. Warum, was, wie und wozu Softwaretechnik?
1.4. Definition, Übersicht



Prof. Bernhard Rumpe
Software Engineering
RWTH Aachen

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

Literatur:

- Sommerville 1.1
- Balzert Band 1, LE 1

SE Software Engineering | RWTH AACHEN UNIVERSITY

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)

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

Aufgabenstellungen der Softwaretechnik

- Softwareentwicklung ist mehr als nur Programmieren!

- Dazu gehören auch:
  - Vorgehensmodelle
  - Analyse
    - Anforderungen | Modellierung
    - System | Modellierung | Muster
  - Softwareentwurf (Design)
    - Systementwurf | Oberflächen | Muster
  - Generative Entwicklung
  - Test

- Und dazu gehören auch:
  - Management großer und komplexer Projekte
  - Schätzung von Terminen und Kosten
  - Erfassung von Kunden- und Marktanforderungen
  - Änderungsmanagement
  - Sicherstellung eines hohen Qualitätsniveaus
  - Wartung und Weiterentwicklung von Altsystemen
  - Softwareproduktlinien

  - Guter Programmierstil
  - Entwicklungswerkzeuge
  - Prinzipien wie
    Abstraktion, Strukturierung, Hierarchisierung,
    Modularisierung, Wiederverwendung & Variabilität

Softwaretechnik vs. Programmieren

- Softwaretechnik
  - Größe & Komplexität der Entwicklungsprojekte
  - Teams, Zusammenarbeit, Komponenten, Qualität,…
- Margaret Hamilton
  - Direktorin der Softwareentwicklungs-Abteilung des Instrumentation Laboratory des MIT
  - On-Board-Flugsoftware für das Apollo-Raumfahrtprogramm
  - 420,837 lines of code (LOC)
- Große Software
  - Windows 7 (40 Mio), Facebook (61 Mio)

Quelle: https://t3n.de/news/codezeilen-facebook-windows-software-546933/codezeilen_code-zeilen_infografik/

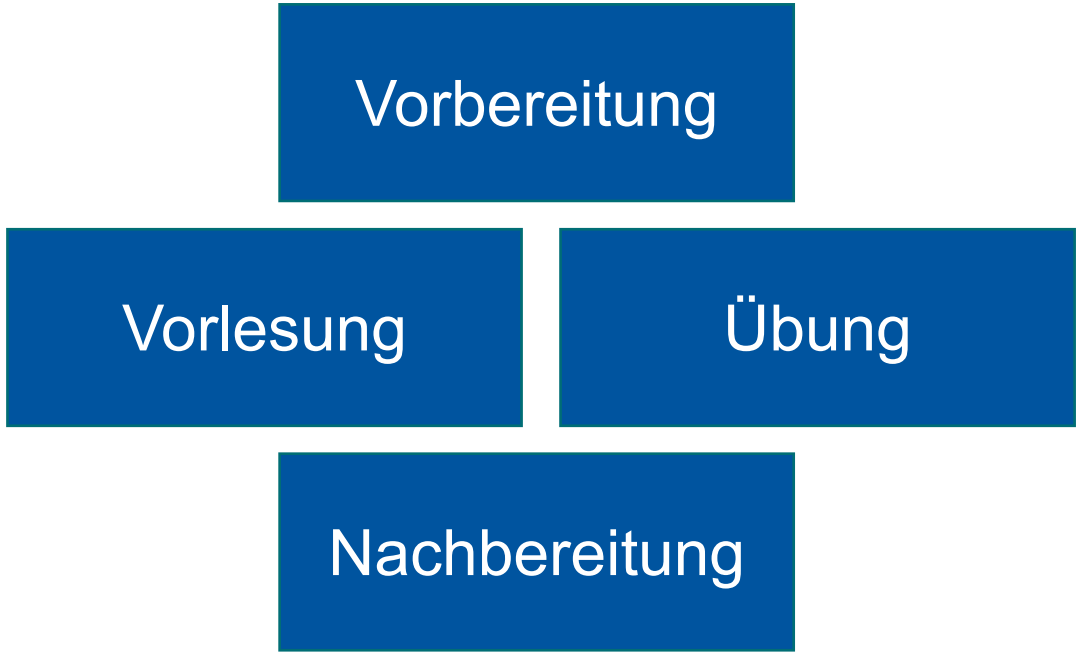
Quelle: https://de.wikipedia.org/wiki/Datei:Margaret_Hamilton_-_restoration.jpg

Portfolio der SE-Techniken

- ... ist vergleichbar mit einer Werkzeugkasten:
  - Für jedes Problem das richtige Werkzeug
  - in der Hand eines Experten, der damit umgehen kann
- Nicht jeder muss alle Werkzeuge beherrschen
- Aber: je mehr man beherrscht, um so besser.

Themen Überblick

- Organisatorisches
- **Vorgehensmodelle**
- **Analyse**
  - Anforderungen | Modellierung
  - System | Modellierung | Muster
- **Softwareentwurf (Design)**
  - Systementwurf | Oberflächen | Muster
- **Implementierung & Generative Entwicklung**
  - Implementierung | Generative SWT
  - Werkzeuge
- **Test**
  - Qualität in der SWT und Testen im Speziellen
- **Wiederverwendung & Variabilität**
  - Komponenten und Wiederverwendung
  - Softwareproduktlinien und Variabilität
- Ausblick

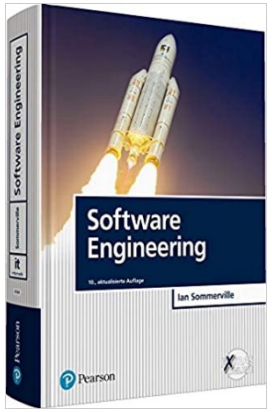


Wozu? Lernziele

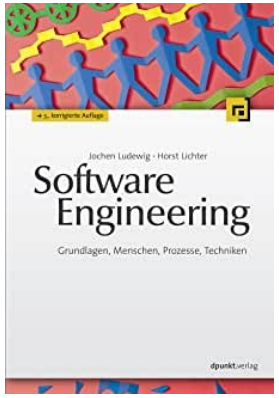
- Kenntnis über
  - Welche Vorgehensmodelle werden in der Praxis angewandt
  - Phasen der Softwareentwicklung & konkrete Schritte
  - Unterschiedliche Familien an Programmiersprachen
  - Prinzipien der Generativen Entwicklung
  - Die wichtigsten Werkzeuge
  - Prozesse von Softwareproduktlinien
- Praktische Anwendung von
  - Modellierungstechniken in der Analyse und Design Phase, z.B. Use Case, Aktivitätsdiagramme, Klassendiagramme, Objektdiagramme, Sequenzdiagramme, Zustandsdiagramme, Feature Diagramme, erkennen, anwenden und analysieren
  - Muster für den Entwurf, die Analyse, die Architektur
  - Versionskontrolle z.B. git und SVN
  - Ableitung von Testdaten und Teststrategien
  - Definition von Komponentenspezifikationen

Literatur (generell für alle Themen der SWT)

- *I. Sommerville*
  - Software Engineering
  - 19. Oktober 2018, 10. Auflage



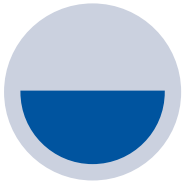
- *J. Ludewig und H. Lichter*
  - Software Engineering: Grundlagen, Menschen, Prozesse, Techniken



- *H. Balzert:*
  - Lehrbuch der Software-Technik Bd. 1

- *Manfred Broy, Marco Kuhrmann*
  - Einführung in die Softwaretechnik, Springer 2021

Was haben wir heute gelernt?



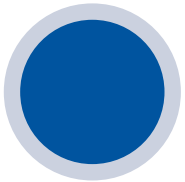
Software

erfüllt kritische Aufgaben in jedem Bereich des Lebens

ist von großer wirtschaftlicher Bedeutung

ist ein schwer zu fassender Werkstoff

entsteht durch Zusammenarbeit mehrerer/vieler Personen



Software Engineering

ist die Lehre von der ingenieurmäßigen Entwicklung von Software und softwarebasierten Systemen

strebt die schnelle und effiziente Herstellung qualitativ hochwertiger Softwareprodukte an

entwirft dafür Prinzipien, Methoden, Werkzeuge und Prozesse

strebt nach Wiederverwendung theoretisch und empirisch geprüfter Ergebnisse

ist anwendungsnah und erfordert, dass ihre Methoden und Techniken geübt werden

Vorlesung Softwaretechnik

2. Vorgehensmodelle

Prof. Bernhard Rumpe
Software Engineering
RWTH Aachen

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

@SE_RWTH

SE Software Engineering | RWTH AACHEN UNIVERSITY

Warum, was, wie und wozu?

Warum?

Es werden verschiedene, problemadäquate Vorgehensmodelle in der Praxis angewandt

Was?

Die wichtigsten Vertreter kennenlernen

Wie?

Betrachten Prozessschritte, Akteure, zeitliche Verläufe

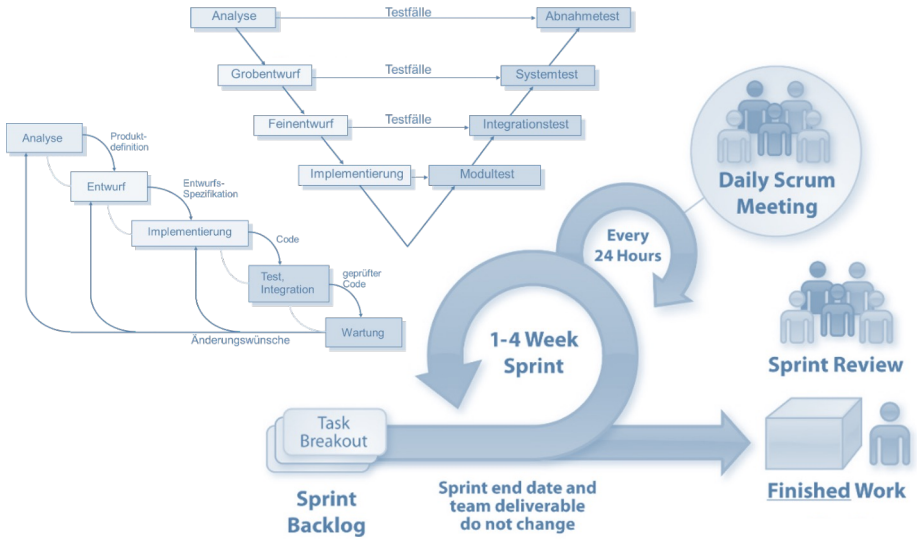
Wozu?

Selbst überlegen, welches Modell für Sie gut anwendbar ist

Fragen stellen beim Bewerbungsgespräch und beurteilen ob Sie damit arbeiten können/wollen

Softwaretechnik

2. Vorgehensmodelle
2.1 Einleitung



Prof. Bernhard Rumpe
Software Engineering
RWTH Aachen

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

Literatur:

- Sommerville 1.2

Software Engineering | RWTH AACHEN UNIVERSITY

Aktivitäten in der Softwareentwicklung

- Trennung der
  - zeitlich abgegrenzten Phasen
    und der darin stattfindenden
  - inhaltlich bestimmten Aktivitäten.

- Aktivitäten:
  - Analyse
  - Entwurf
  - Implementierung
  - Test (einschließlich Integration, synonym: Validierung)
  - Deployment (Installation, Schulung)
  - Evolution (incl. Wartung)
  - und viele weitere
    (Versionsmanagement, Reviews, Tooling,
    Variantenmanagement, Prozessoptimierung, ...)

Vorgehensmodelle

- …organisieren einen Entwicklungsprozess in strukturierte Abläufe
  - Methoden und Techniken
  - auftretende Aufgabenstellungen und Aktivitäten in einer sinnfälligen logischen Ordnung darzustellen.
- Vorgehensmodelle sind organisatorische Hilfsmittel, die für konkrete Projekte individuell angepasst werden und in die konkrete Maßnahmenplanung überleiten.

- Bekannte Vorgehensmodelle?
  - V-Modell, RUP
  - Agile Softwareentwicklung, Extreme Programming, Scrum, ...
  - OMT, OOSE (Jacobson),
  - Open Source (als Vorgehensweise), Hacking

⇨ discuss

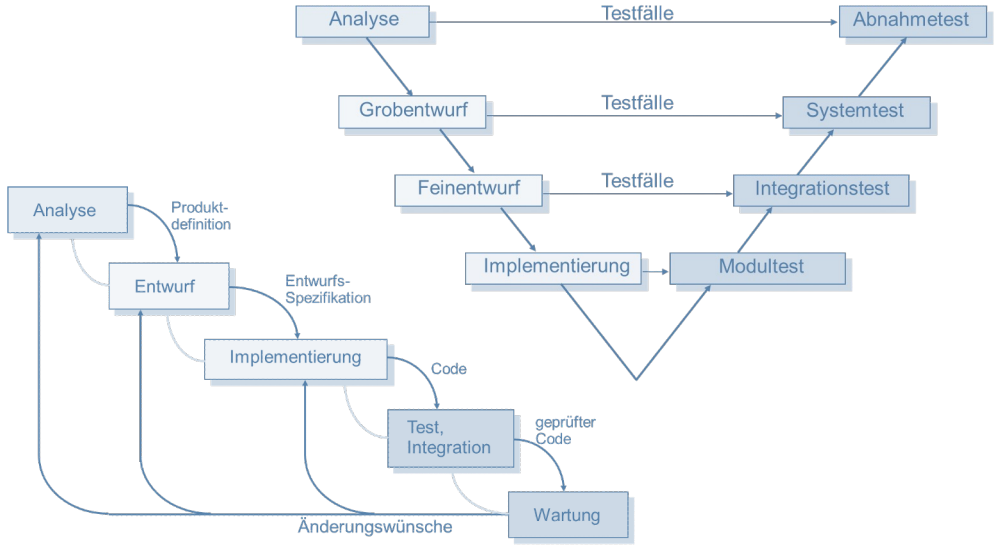
Arten von Vorgehensmodellen

- Klassische Vorgehensmodelle
  - Wasserfall Model
  - V-Modell (Berry Boehm + Deutsche Norm)
  - Rational Unified Process (RUP)
  - Spiral-Modell

- Agile Vorgehensmodelle
  - Extreme Programming (XP) (Kent Beck)
  - Crystal (Alistair Cockburn)
  - Kanban (Taiichi Ohno - Toyota)
  - Scrum (Ken Schwaber, Mike Beedle)
  - Agile Modeling (Scott Ambler)
  - Adaptive Software Development (Peter Coad)

Softwaretechnik

2. Vorgehensmodelle
2.2 Klassische Vorgehensmodelle

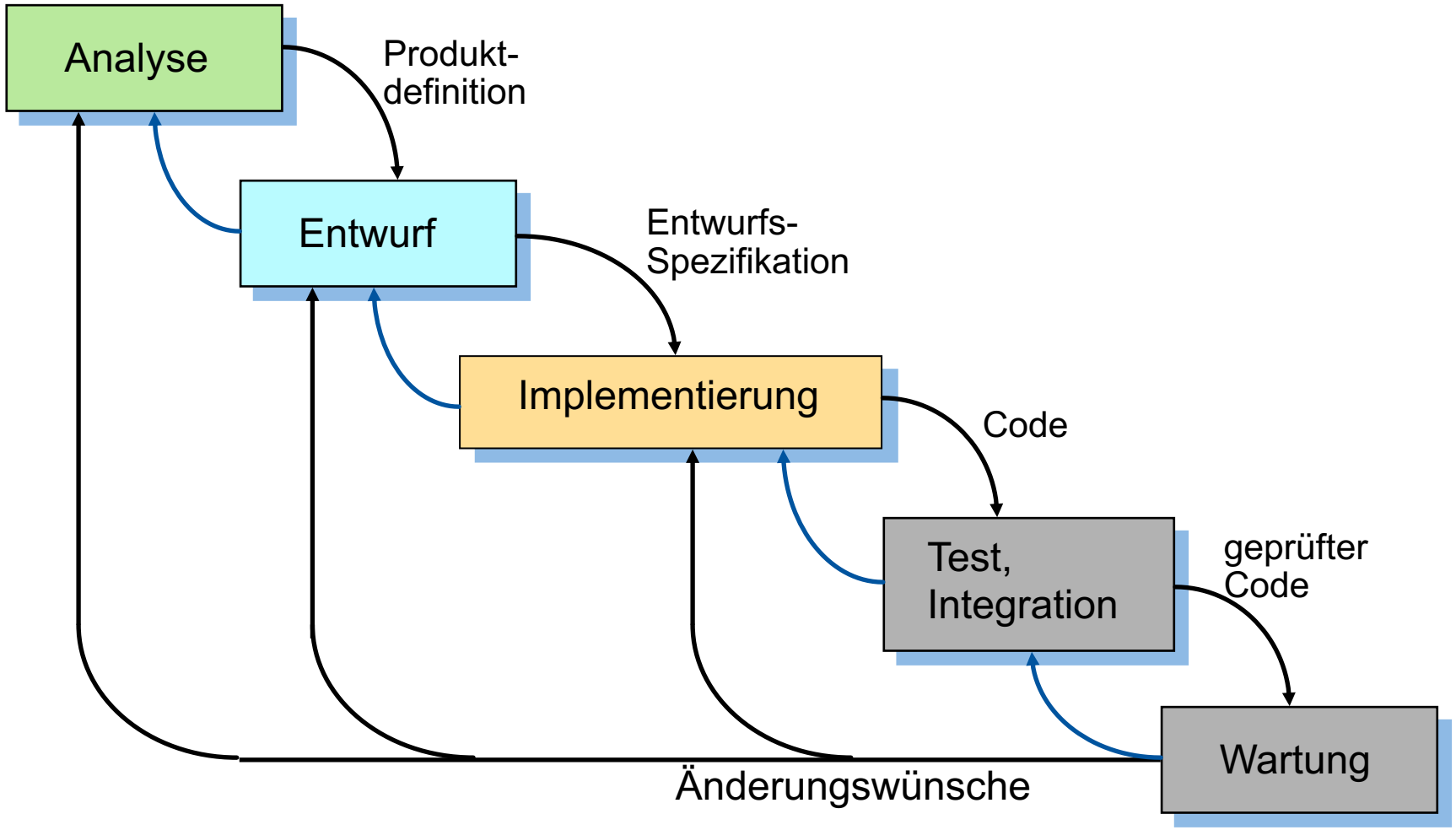


Prof. Bernhard Rumpe
Software Engineering
RWTH Aachen

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

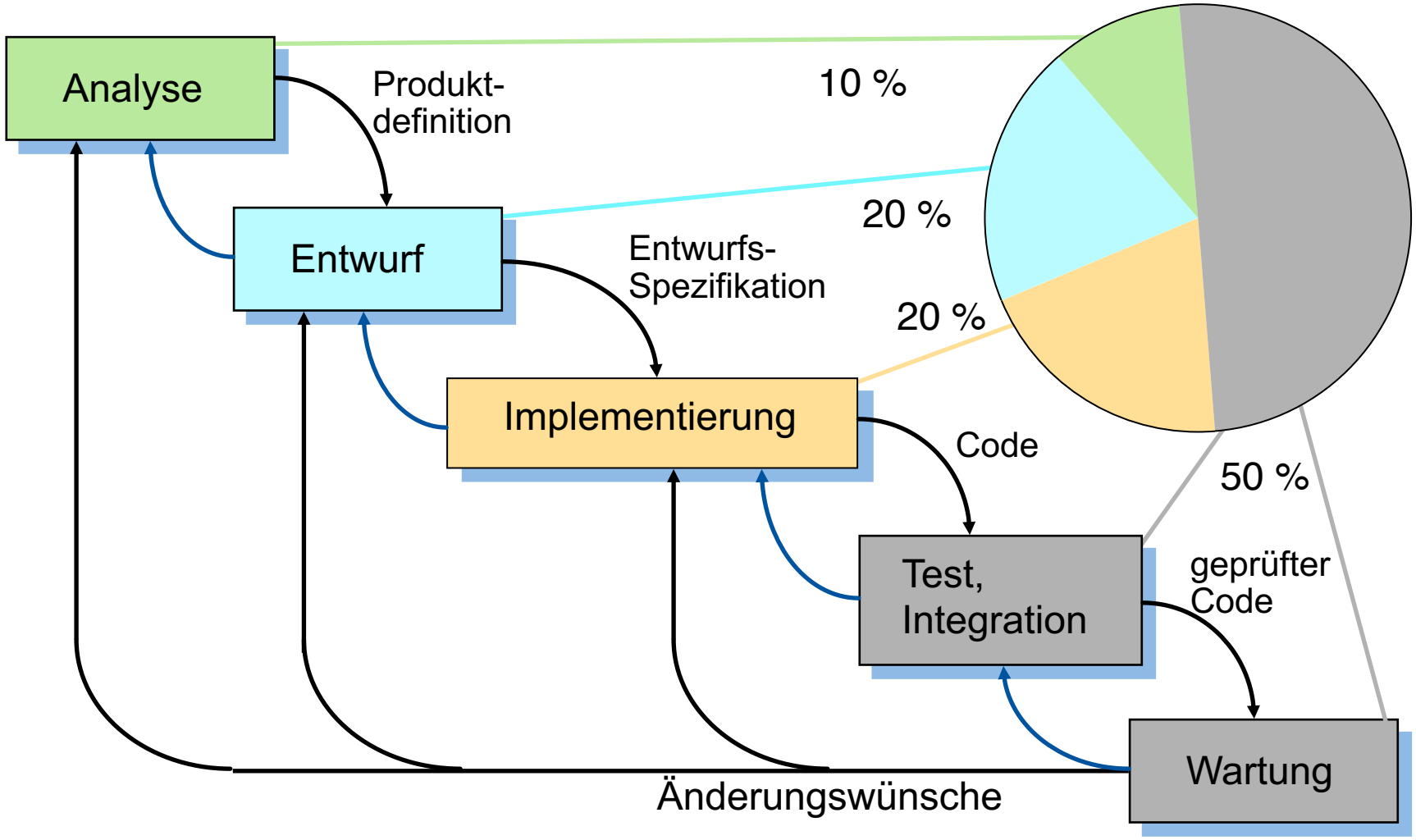
SE Software Engineering | RWTH AACHEN UNIVERSITY

Das klassische Wasserfall-Modell



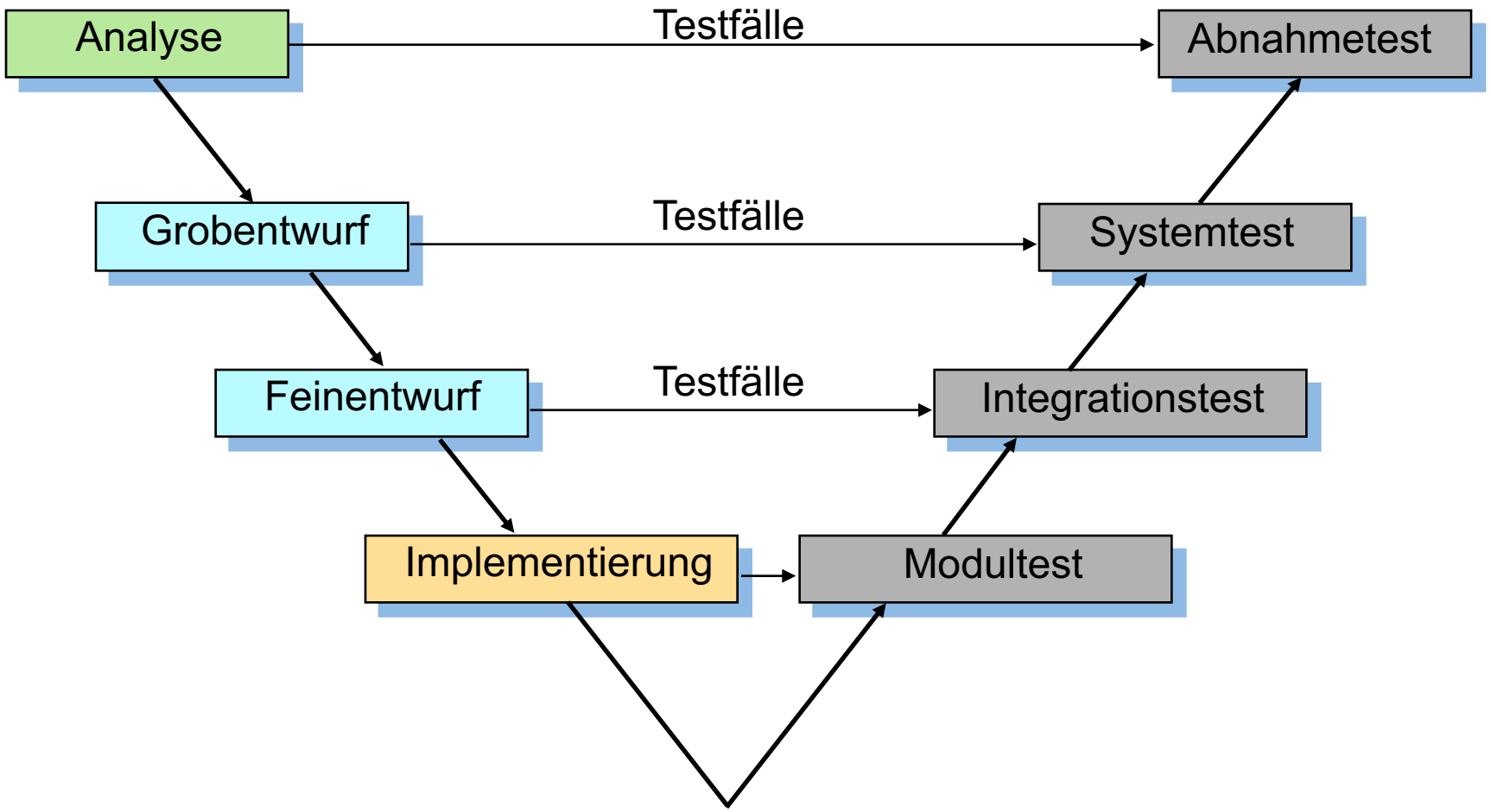
W. Royce (1970)

Ungefähre Verteilung des Arbeitsaufwands



W. Royce (1970)

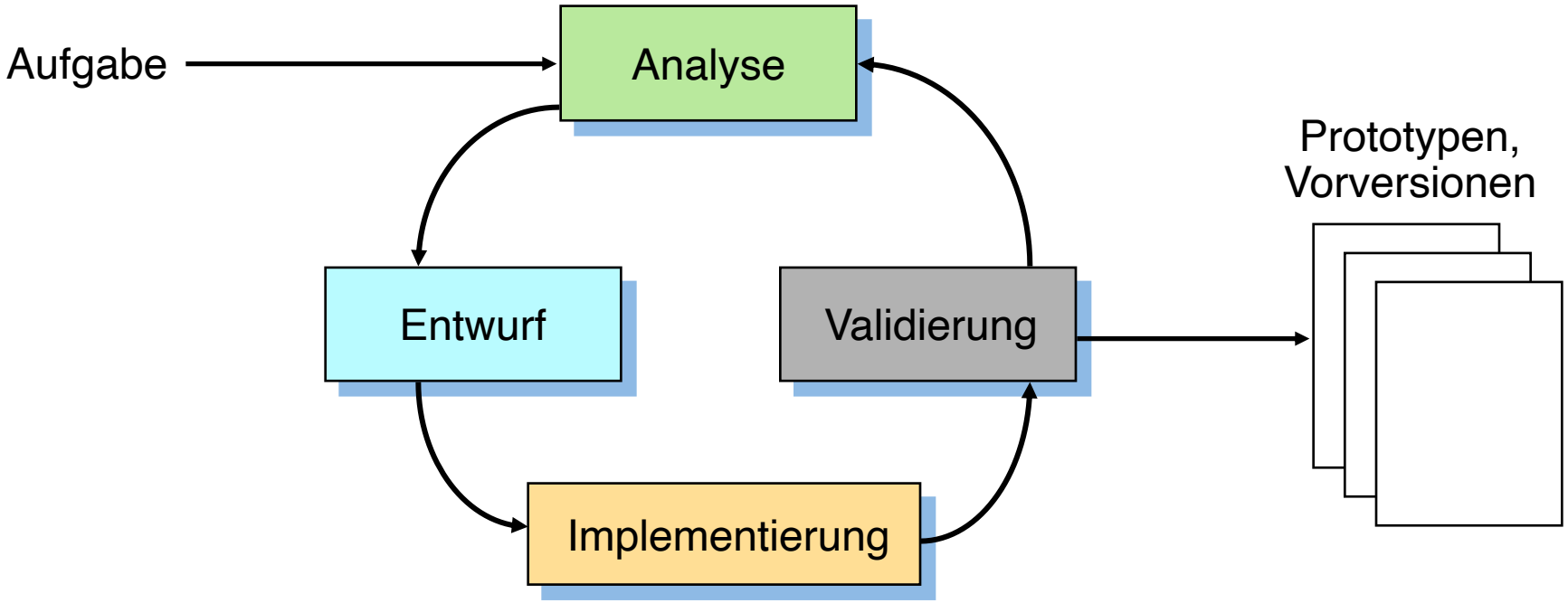
Qualitätssicherung im V-Modell



Boehm 1979 (erstes, altes „V-Modell“)

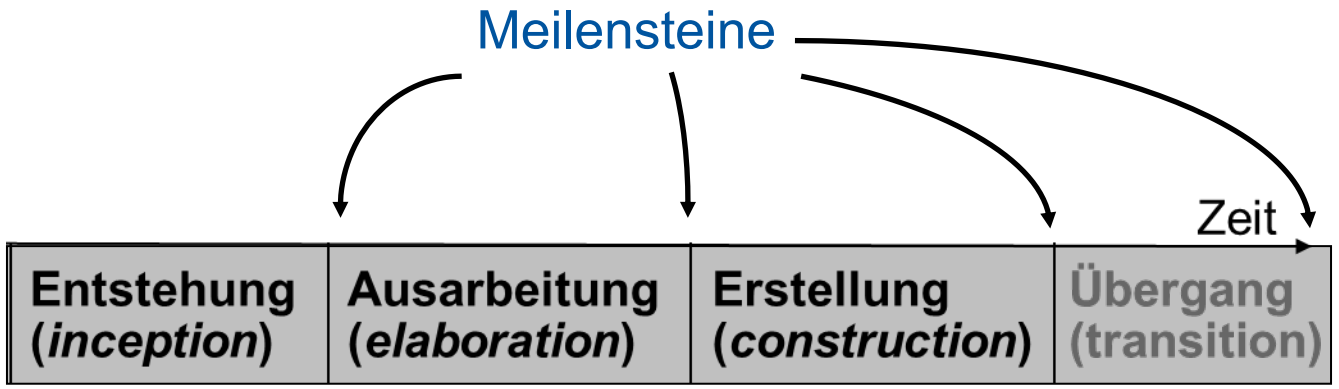
Evolutionäre Entwicklung

- Typisch für kleinere Projekte oder experimentelle Systeme
- Zunehmend auch für größere Projekte angewendet

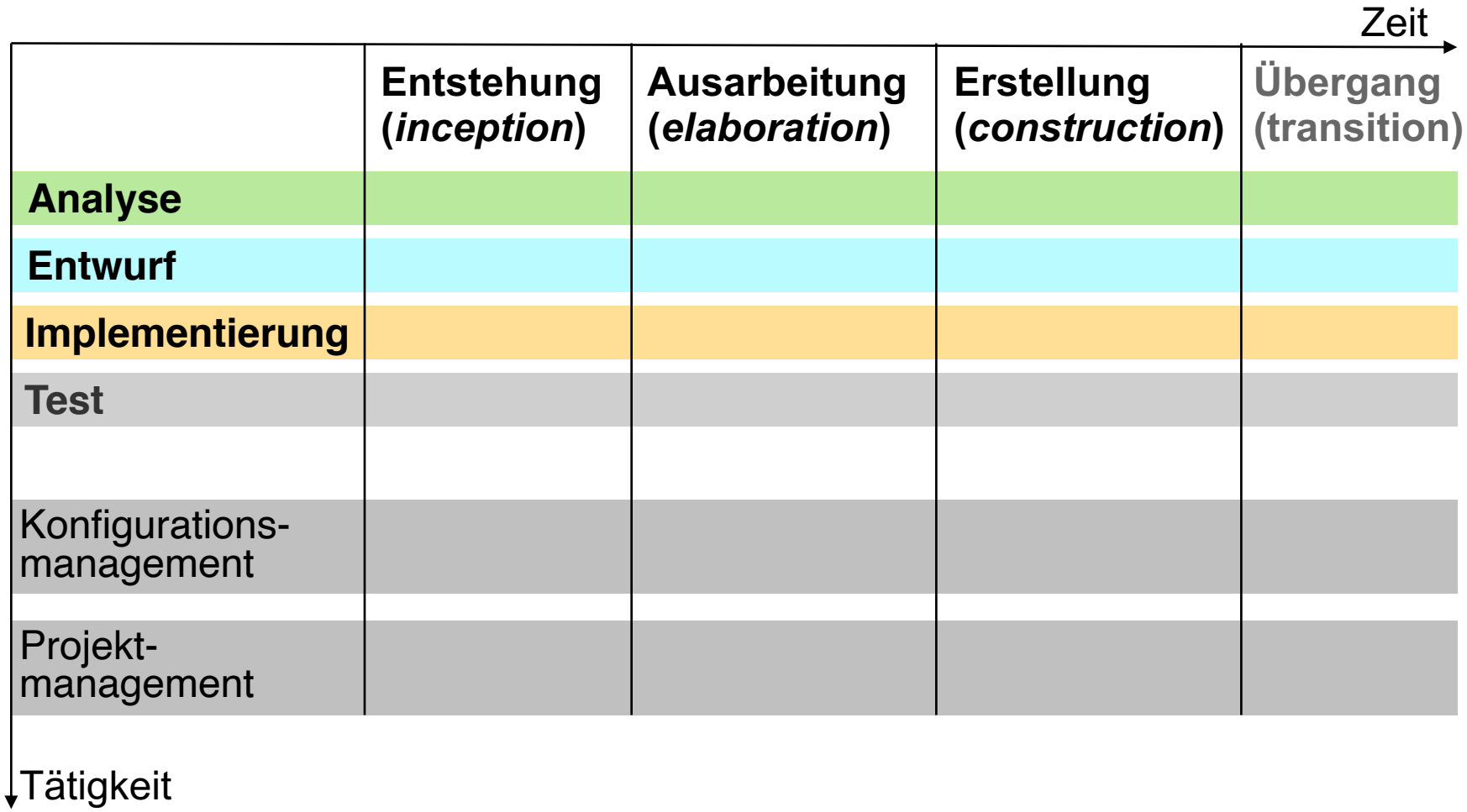


Rational Unified Process (RUP)

- Der Rational Unified Process hat vier Phasen:
  - Entstehung (Inception) – Definiert den Rahmen des Projekts
  - Ausarbeitung (Elaboration) – Planung des Projekts, Spezifikation der Features, Grundlegende Architektur
  - Erstellung (Construction) – Erstellung des Produkts
  - Übergang (Transition) – Einführung des Produkts in der Endbenutzer-Community

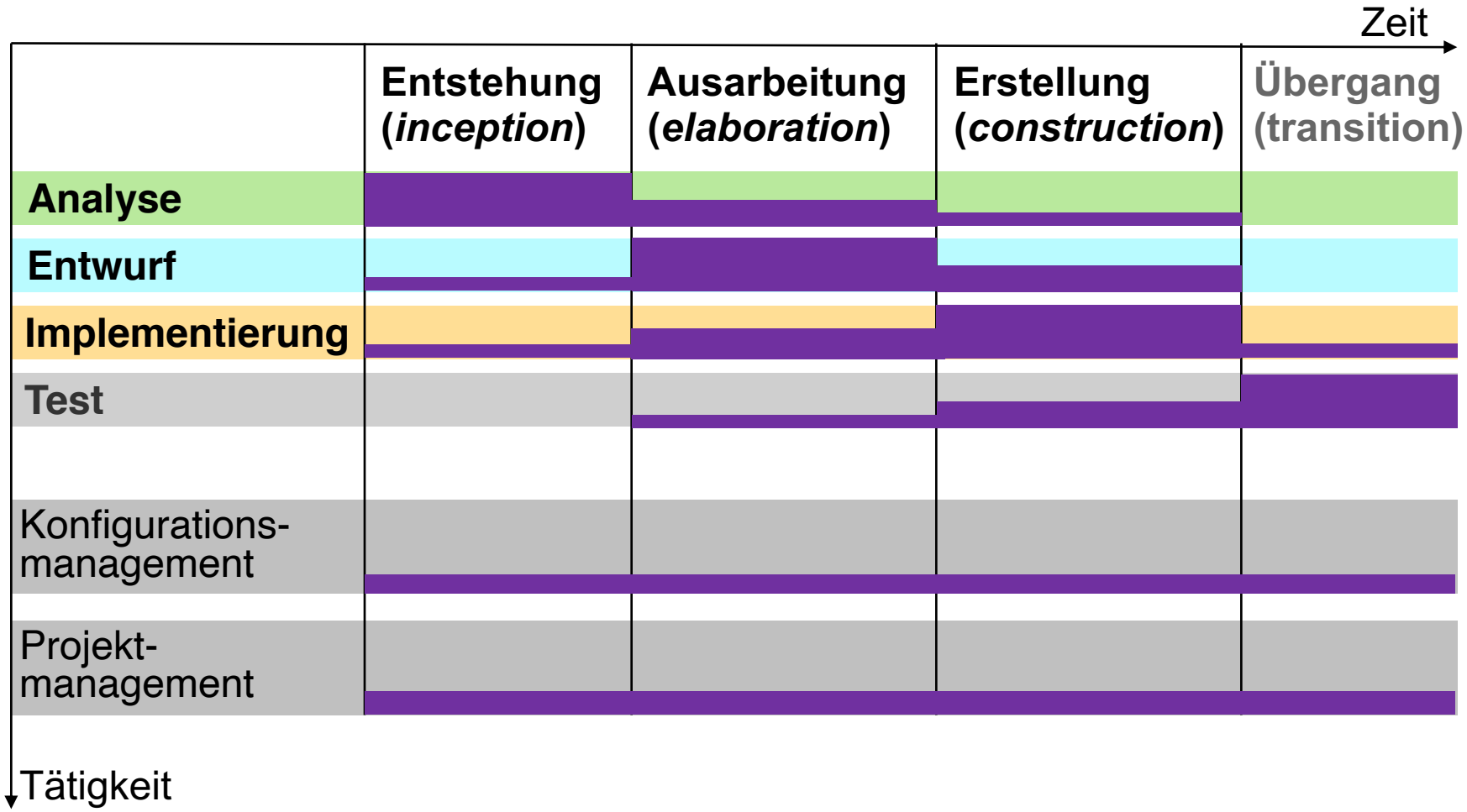


RUP | Zweidimensionales Modell



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

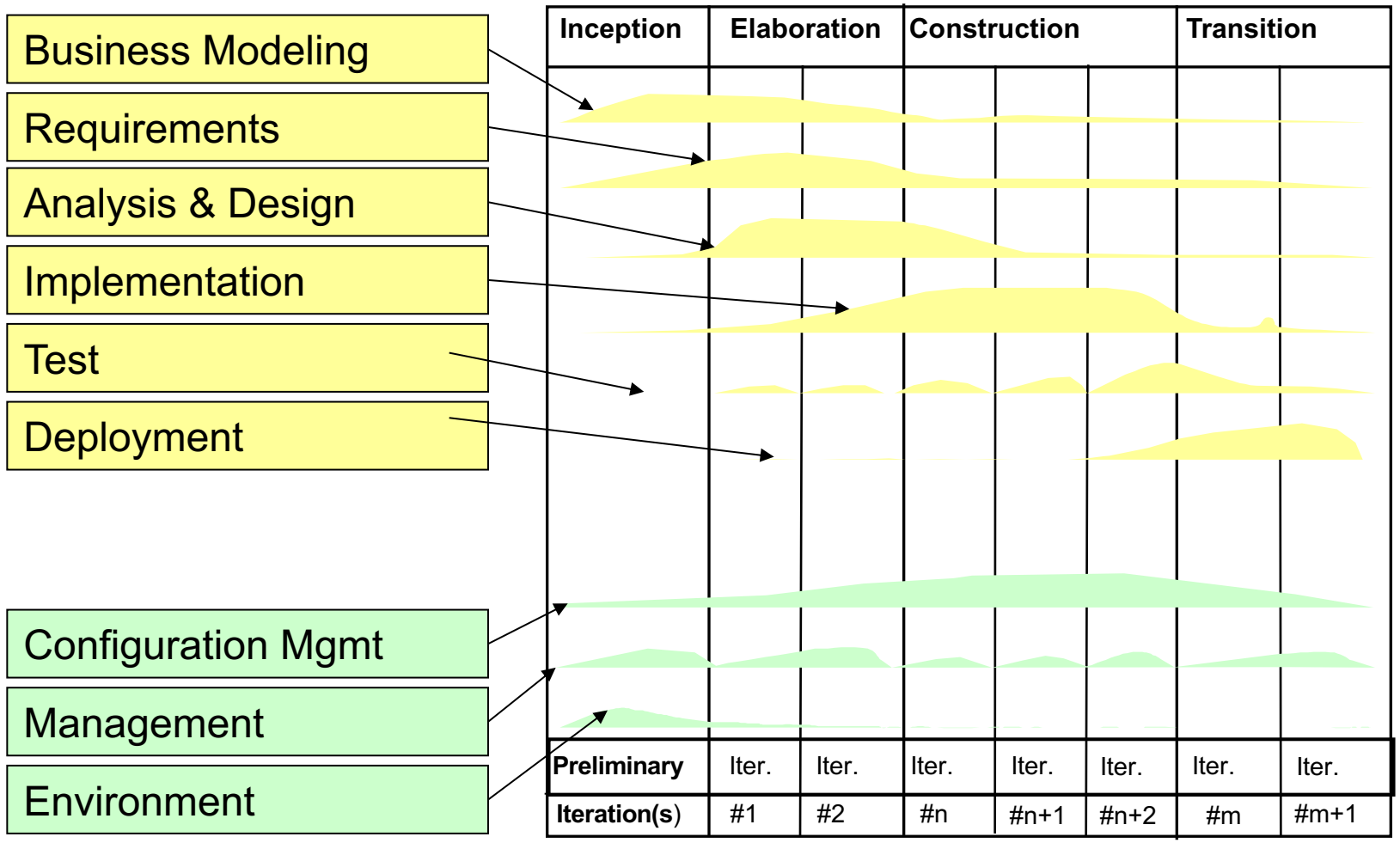
RUP | Aufwandsverteilung und Schwerpunkte



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

Aufwandsverteilung in Iterationen

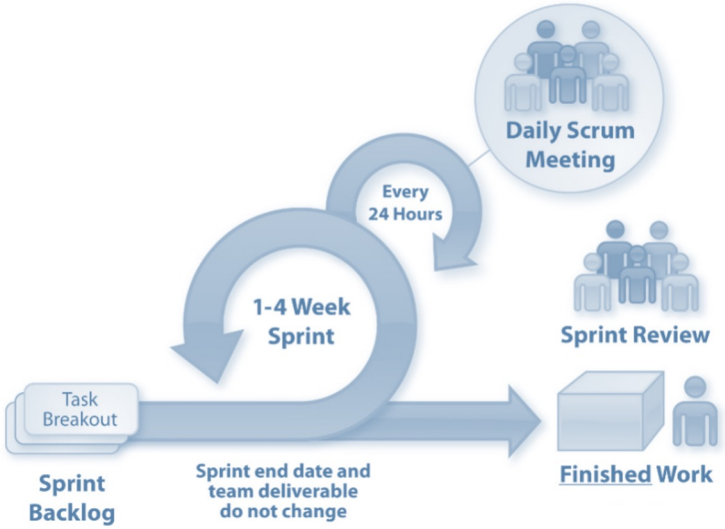
Prozess Workflows



Unterstützende Workflows

Softwaretechnik

2. Vorgehensmodelle
2.3 Agile Vorgehensmodelle



Prof. Bernhard Rumpe
Software Engineering
RWTH Aachen

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

SE Software Engineering | RWTH AACHEN UNIVERSITY

eXtreme Programming (XP): eine agile Methode

- 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

- Konsequenz: nur für kleinere Projekte geeignet

Scrum (engl. „Gedränge“): eine agile Methode

- inkrementell und iterativ
- Komplexität durch drei Prinzipien reduzieren:
  - Transparenz: Fortschritt / Probleme täglich festhalten
  - Überprüfung: iterativ werden Funktionen geliefert und beurteilt.
  - Anpassung: flexibel Anforderungen bewerten & ändern

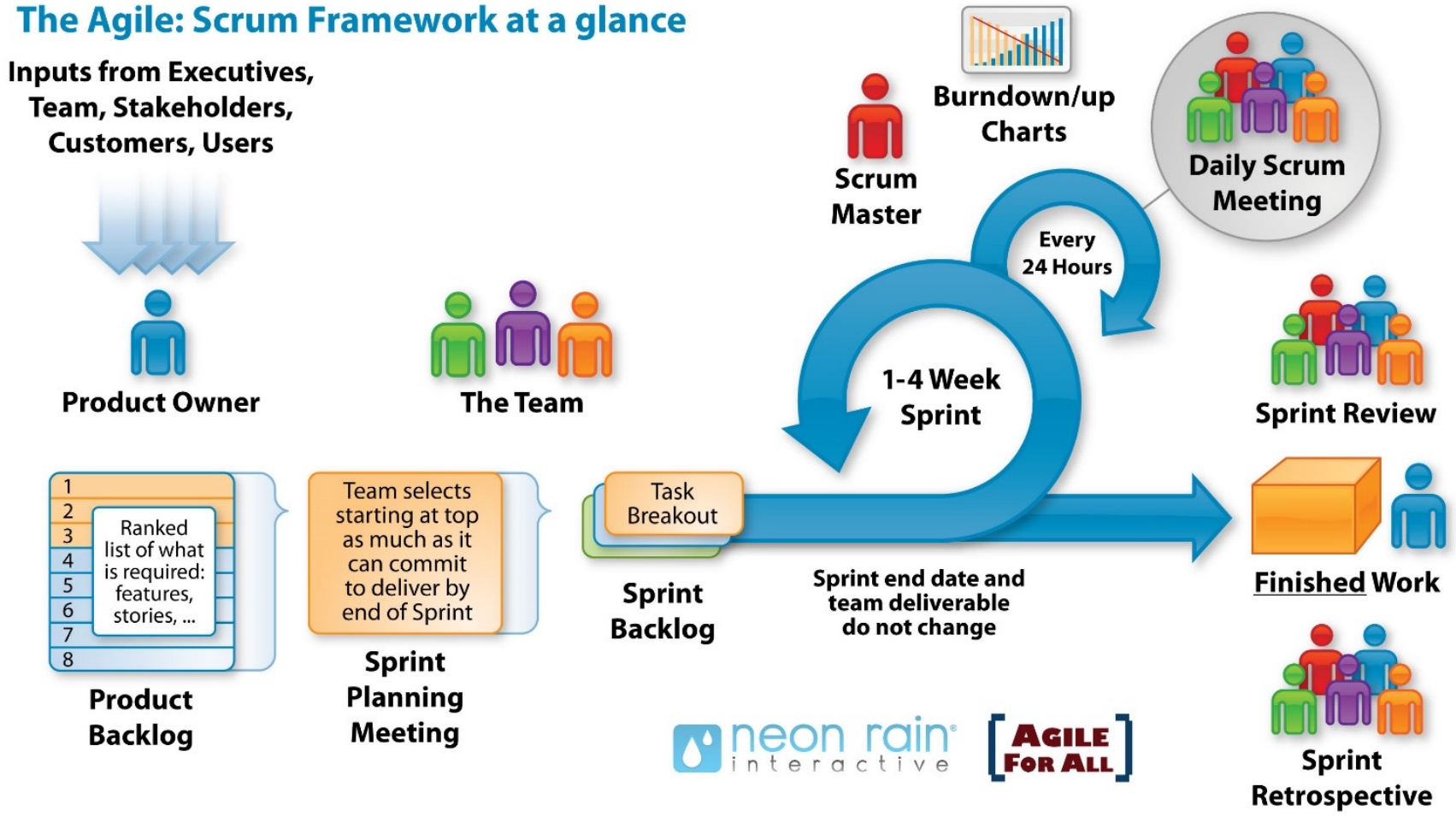
- Drei Rollen
  - Product Owner, Entwicklungsteam, ScrumMaster

- Aktivitäten:
  - Sprint Planning, Sprint Review, Sprint-Retrospektive und Daily Scrum

- Alle Aktivitäten in Scrum sind zeitlich fest beschränkt („timeboxed“), also keine klassischen Meilensteine

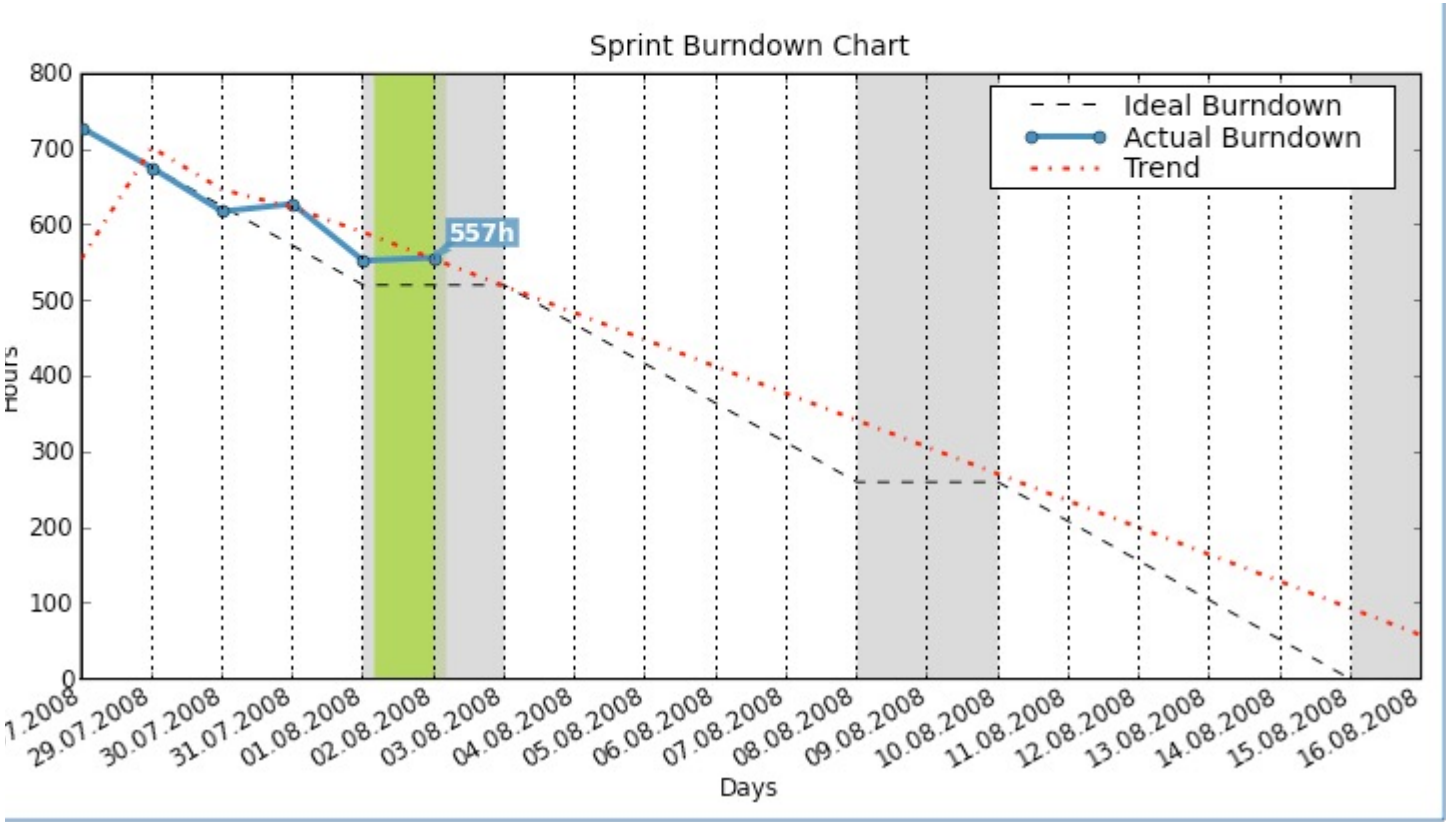
Scrum

- Artefakte
  - Product Backlog
  - Sprint Backlog
  - auslieferbares Produktinkrement
- Sprint
  - zeitlich begrenzte Phase, um das Produkt zu entwickeln
- Sprint Backlog
  - Liste der zu realisierenden User Stories



Quelle: https://www.neonrain.com/agile-scrum-web-development/

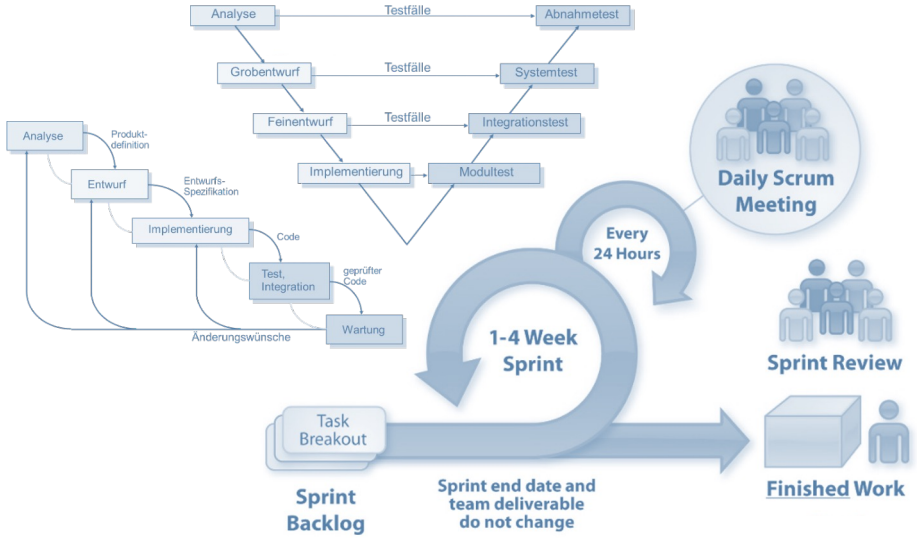
Scrum



- Burndown Chart:
  - Zeigt Ist- und Planung der Zielerreichung an.

Softwaretechnik

2. Vorgehensmodelle
2.4 Vergleich der Vorgehensmodelle

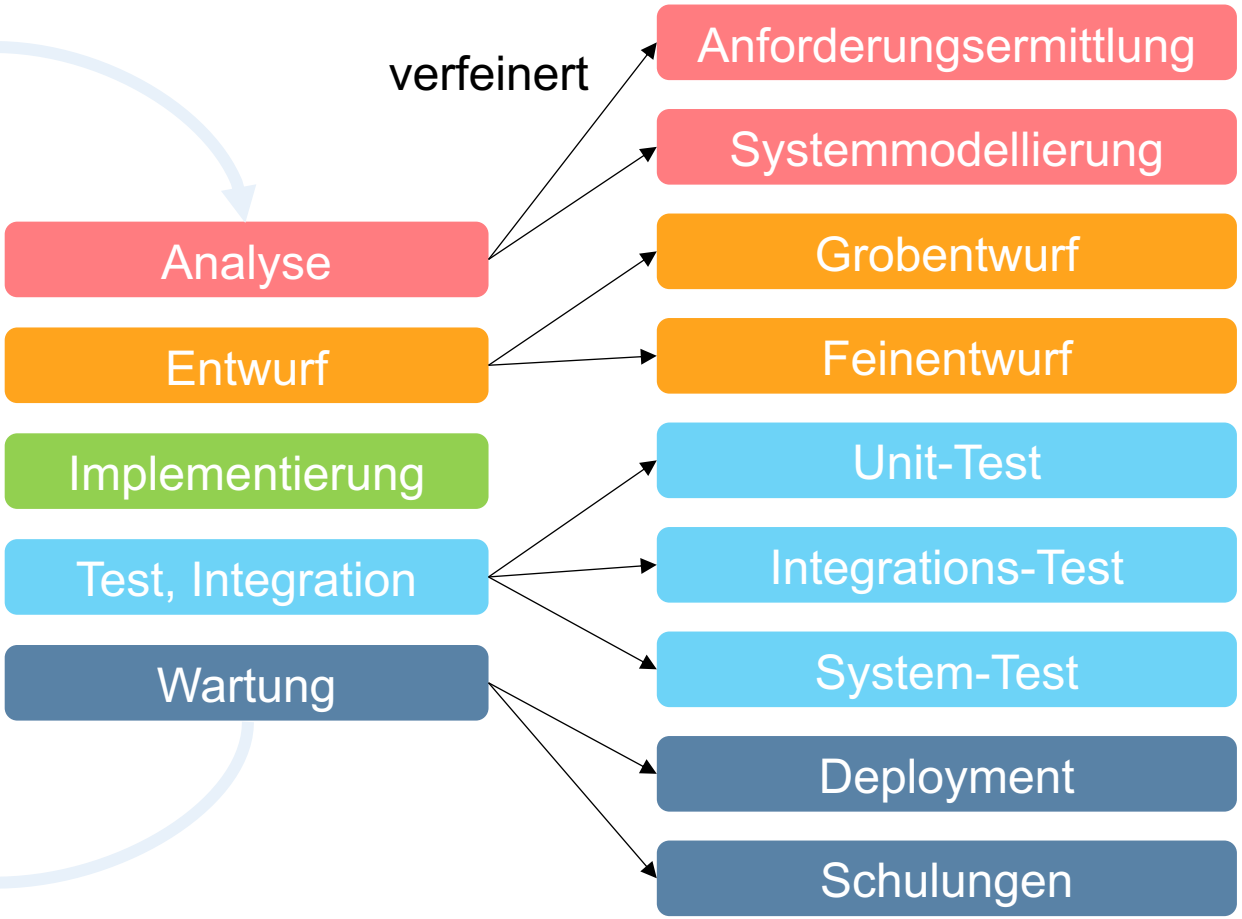


Prof. Bernhard Rumpe
Software Engineering
RWTH Aachen

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

SE Software Engineering | RWTH AACHEN UNIVERSITY

Aktivitäten in der Software Entwicklung



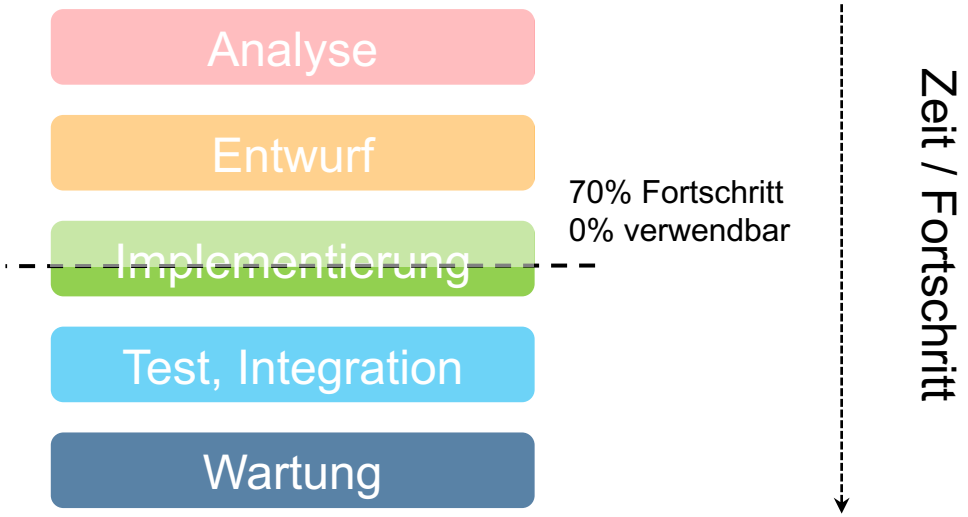
Parallel dazu

- Projekt-Management
- Konfigurations-Management
- Varianten/Versions-Management
- Änderungs-Management
- Tool-Management

Klassische vs. Agile Prozesse

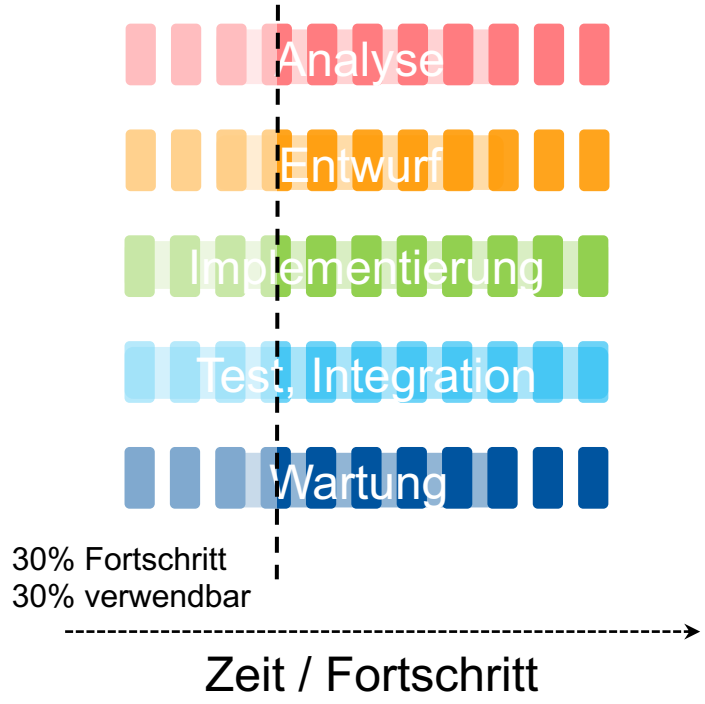
Wasserfall Modell (V-Model, RUP)

- Aktivitäten sind chronologisch in Phasen unterteilt



Agile Prozesse

- Aktivitäten sind kurze Sprints/ Intervalle
- Leben von vielen kleinen Iterationen



Literatur (neben Balzert, Sommerville)

- *Philippe Kruchten:*
  - Rational Unified Process. Addison-Wesley. 2000
- *Kent Beck:*
  - Extreme Programming Explained. Addison-Wesley. 1999
- *Jeff Sutherland, Ken Schwaber:*
  - The Scrum Guide
- *Klaus Leopold, Siegfried Kaltenecker*
  - Kanban in der IT: Eine Kultur der kontinuierlichen Verbesserung schaffen. Carl Hanser Verlag. 2018
- *Testframework JUnit*
  - www.junit.org

Was haben wir heute gelernt?

Vorgehensmodelle

… organisieren einen Entwicklungsprozess in strukturierte Abläufe

… sind organisatorische Hilfsmittel, die an das jeweilige Projekt angepasst werden müssen

Unterschiedliche Arten

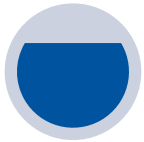
Wasserfall bzw. V-Modell

Rational Unified Process

eXtreme Programming

Scrum

Kanban



Zeitlicher Verlauf

Sequenziell, chronologisch

Agil, iterativ

Anwendung in der Praxis

Meist agil

…aber in unterschiedlichen Ausprägungen und Abwandlungen

Vorlesung Softwaretechnik

3. Anforderungsanalyse

Prof. Bernhard Rumpe
Software Engineering
RWTH Aachen

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

SE Software Engineering | RWTH AACHEN UNIVERSITY

Warum, was, wie und wozu? Anforderungsanalyse

Warum?

Für die Praxis notwendig zu wissen, wie man Anforderungen systematisch erfasst und dokumentiert

Was?

- Analysephase
- Use Case Diagramme für Anwendungsfälle
- Aktivitätsdiagramme für Prozesse
- Prototypen für die Analyse

Wie?

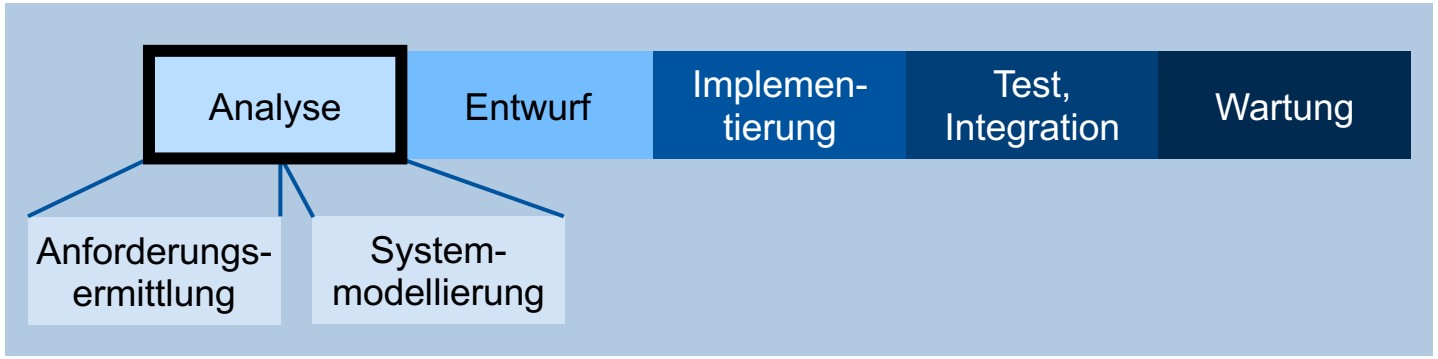
- Konzepte der Modellierungssprachen
- Anwendung der Modellierungstechnik
- Konkrete Beispiele

Wozu?

- Zufriedene AnwenderInnen
- Weniger Risiko in der Produktentwicklung

Softwaretechnik

3. Anforderungsanalyse
3.1. Verwendete Beispiele



Prof. Bernhard Rumpe
Software Engineering
RWTH Aachen

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

SE Software Engineering | RWTH AACHEN UNIVERSITY

Beispiele: Seminarorganisation

- Verwaltung von Seminaren und SeminarteilnehmerInnen
  - Seminare und Termine
  - Teilnehmende und Interessierte
  - Rechnungen
- Zugriff über lokal installierte Software und eine Weboberfläche
  - Teilnahmen buchen und verwalten
  - Abfragen stellen
  - Angebotskatalog erstellen
  - Statistiken erstellen

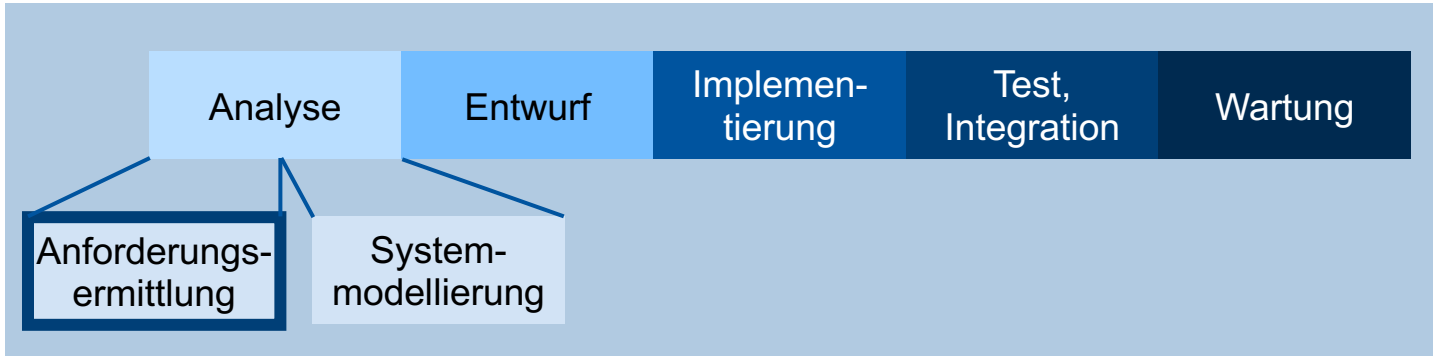
und Auktionssystem

- Auktionsverwaltung und Teilnahme
  - Auktion mit Dauern
  - Angebote mit Beschreibungen und Gebote
  - Anbieter:innen, Bieter:innen, Auktionator:in
- Zugriff über eine Weboberfläche
  - Auktion anlegen und verwalten
  - Nach Auktionen suchen und bei Auktionen mitbieten
  - Verwaltung der Auktionen
  - Abfragen und Statistiken
- Sicherheit?

Was wird notwendig sein, um solche Systeme zu entwickeln?

Softwaretechnik

3. Anforderungsanalyse
3.2. Anforderungsermittlung



Prof. Bernhard Rumpe
Software Engineering
RWTH Aachen

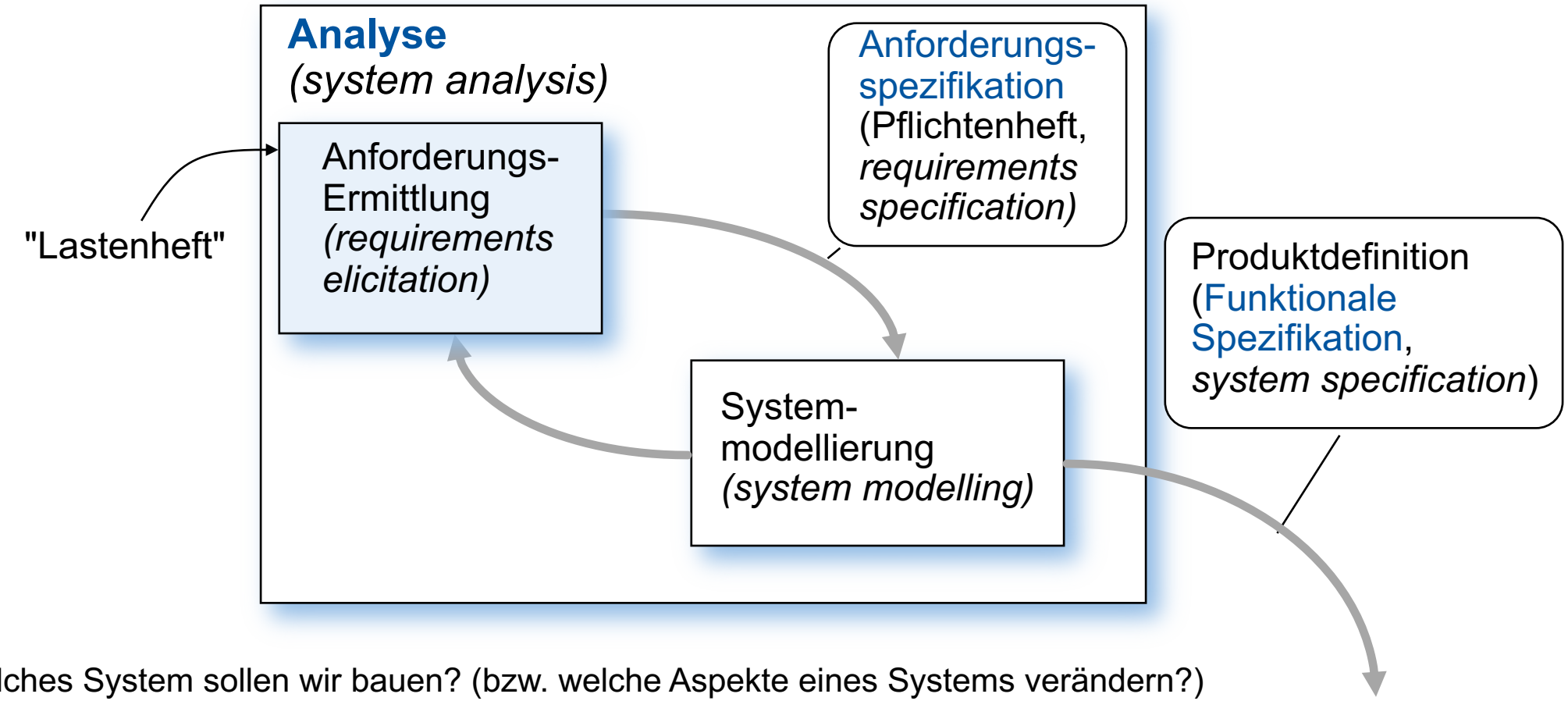
http://www.se-rwth.de/
@SE_RWTH

Literatur:

- Sommerville 5+6
- Balzert Band 1, LE 2-4

SE Software Engineering | RWTH AACHEN UNIVERSITY

Requirements Engineering



- Welches System sollen wir bauen? (bzw. welche Aspekte eines Systems verändern?)
- Was sind die Anforderungen des Kunden?
- Ist es möglich, das geforderte System zu realisieren?

Definitionen

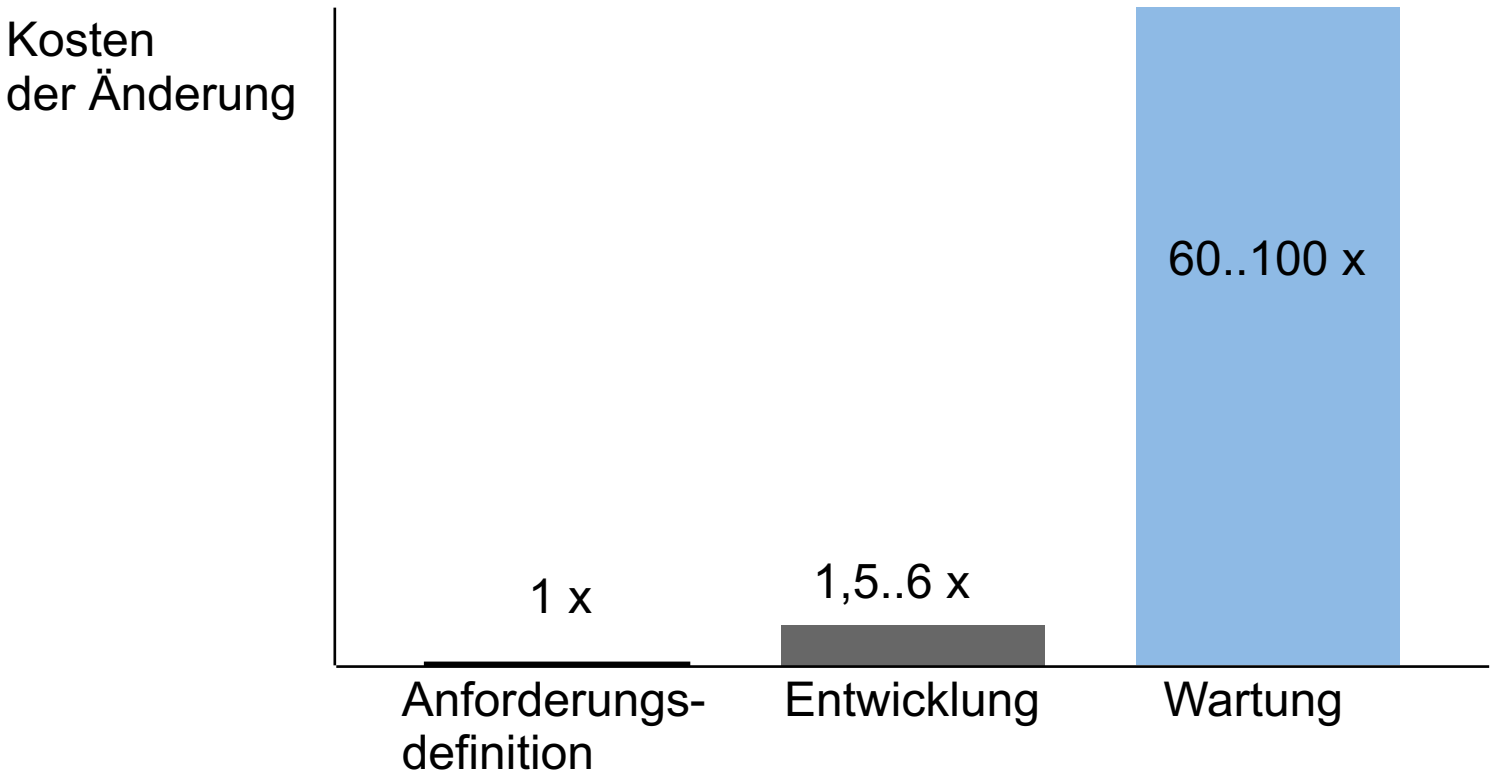
- *Anforderungsermittlung* (requirements elicitation):
  - Tätigkeit: Feststellung der Anforderungen an das geplante System in Zusammenarbeit mit Kunden und potentiellen Benutzern
  - Ergebnis: Anforderungsspezifikation (Pflichtenheft)

- *Systemmodellierung* (system modelling)
  - Tätigkeit: Detaillierte und strukturierte Beschreibung der Anforderungen in einer Form, die als Grundlage für den Systementwurf dienen kann.
  - Ergebnis: Funktionale Spezifikation (Produktdefinition)

- Oberbegriff für beide Tätigkeiten:
  Analyse (engl. requirements analysis, system analysis)

- *Requirements Engineering* ist ein Name für das Teilgebiet des Software-Engineering, das sich mit den frühen Phasen der Entwicklung befasst. (Es gibt keinen wirklich adäquaten deutschen Begriff für requirements engineering.)

Bedeutung der Anforderungsermittlung



- Je später in der Entwicklung ein Fehler gefunden wird, um so aufwändiger ist seine Behebung.
- Jedoch: Moderne Prozesse wie XP versuchen diesen Anstieg zu drücken.

nach [Pressman, S.17, Fig. 1.7]

Funktionale/nicht-funktionale Anforderungen

- Funktionale Anforderung:
  - Beschreibung dessen, **was** das System tun soll
  - Bsp.: "Das System soll einen Mechanismus zur Identifizierung von Benutzern vorsehen."

- Nicht-funktionale Anforderung:
  - Einschränkende Bedingung, **wie** die funktionalen Anforderungen zu realisieren sind
  - Bsp.: "Der Identifizierungsvorgang muss in höchstens 5 Sekunden abgeschlossen sein."

- Mischfälle:
  - Bsp.: "Zur Identifizierung ist das Softwarepaket XY zu verwenden, von dem bekannt ist, dass es den Vorgang in höchstens 5 Sekunden bewältigt.

  → Zerlegbar in funktionale und nicht-funktionale Anforderungen

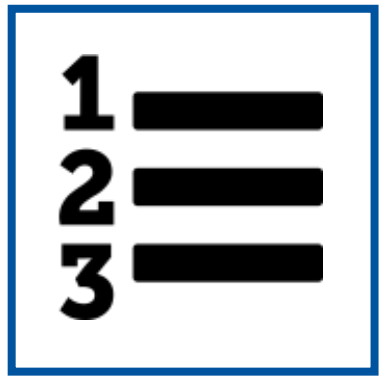
Typen von nicht-funktionalen Anforderungen

- Anforderungen an das Produkt:
  - Effizienzanforderungen: Zeit-, Speicheranforderungen
  - Zuverlässigkeitsanforderungen, „Quality of Service“
  - Betriebsrisiken, Bedrohungen
  - Ergonomische Anforderungen
  - Portabilitätsanforderungen
- Anforderungen an den Entwicklungsprozess:
  - Verwendung von Standards (V-Modell, UML, ...)
  - Anforderungen an die Implementierungstechnik
  - Anforderungen an die Auslieferungsform
- Externe Anforderungen:
  - Interoperabilitätsanforderungen
  - Juristische Anforderungen: Sicherheit, Datenschutz, Zuverlässigkeit
  - Ethische Anforderungen

Inhalte einer Anforderungsspezifikation

- Zielsetzung
- Allgemeine Beschreibung
  - Umgebung, generelle Funktion, Restriktionen, Benutzer
- Spezifische funktionale Anforderungen
  - möglichst quantitativ (z.B. Tabellenform)
  - eindeutig identifizierbar (Nummern)
- Spezifische nicht-funktionale Anforderungen
  - z.B. Antwortzeit, Speicherbedarf, HW/SW-Plattform
  - Entwicklungs- und Produkt-Standards
- Qualitäts-Zielbestimmung
- Zu erwartende Evolution des Systems
  - Grobe Identifikation von Versionen

- Formalia: Abkürzungsverzeichnis, Glossar, Index, Referenzen
  (sehr wirkungsvoll zur Konsistenzsicherung!)



Beispiele funktionaler Anforderungen

Produktfunktionen "Seminarorganisation" (Balzert I, S. 982), Auszug:

Beispiel als textuelle Sammlung (zum Beispiel in einem Word-Dokument):
Man beachte die Form der Nummerierung

Kundenverwaltung

/F10/	Ersterfassung, Änderung und Löschung von Kunden
/F15/	Ersterfassung, Änderung und Löschung von Firmen, die Mitarbeiter zu Seminaren schicken
/F20/	Anmeldung eines Kunden mit Überprüfung
/F30/	- ob er bereits angemeldet ist
/F40/	- ob der angegebene Seminarwunsch möglich ist
/F50/	- ob das Seminar noch frei ist
/F55/	- wie die Zahlungsmoral ist

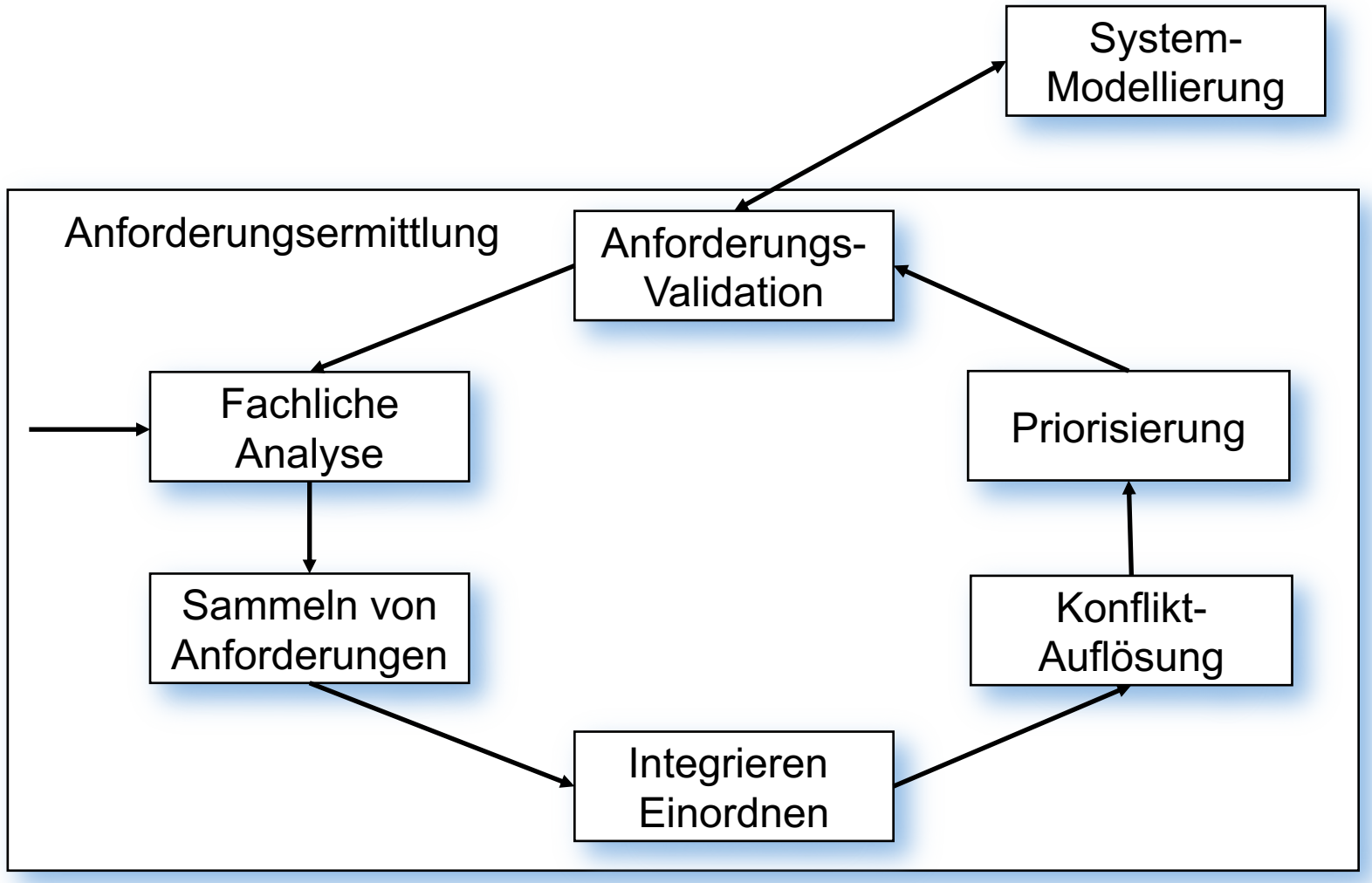
Gliederung nach Versionen (Beispiel)

Beispiel als tabellarische Sammlung (zum Beispiel in einem Excel/Word-Dokument):

Nr.	Anforderung	V1 (6/17)	V2 (1/18)	V3+
	<u>Funktionsgruppe 2</u>			
2.1	Mehrbenutzerfähigkeit	nein	erwünscht	ja
2.2	Max. Anzahl Benutzer	n/a	10	tbd
…	…			

Jargon:
n/a = not applicable
tbd = to be discussed
...

Grobes Vorgehen bei der Anforderungsermittlung



Anforderungs-Management:
- Anforderungs-Datenbanken
- Verfolgung von Querbezügen
- Iteratives Vorgehen

Analysephase als Kommunikationsleistung

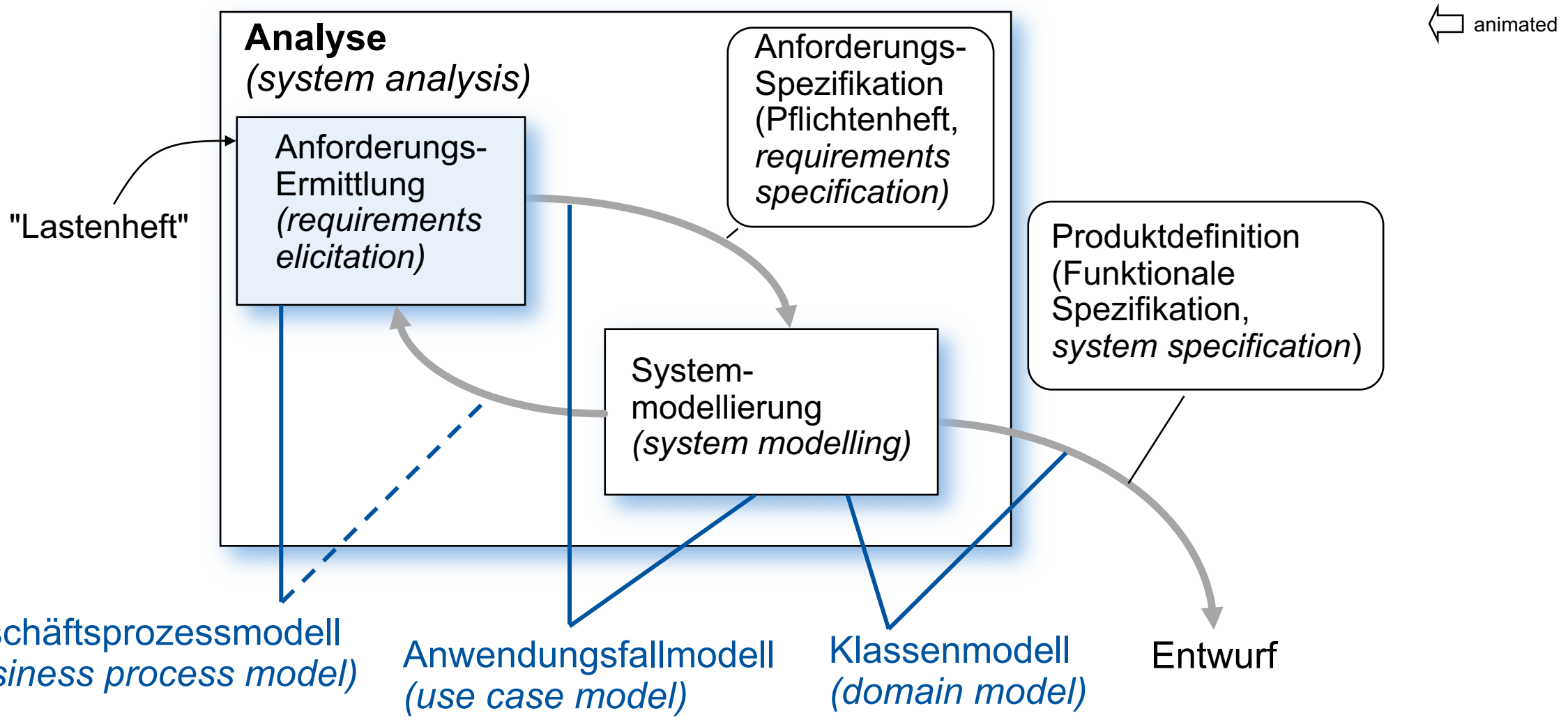
- Die Wirklichkeit der Anforderungsermittlung:
  - Kreative, analytische und kommunikative Leistung
  - Hohe Bedeutung sozialer Kompetenz
    (offene, freundschaftliche, vertrauensfördernde Zusammenarbeit)
  - Weitere Kompetenzen: Urteilsfähigkeit, Kreativität, Erfahrung
- Hilfsmittel des Systemanalytikers:
  - Organisationsprinzipien und -werkzeuge für Informationen
  - Sammlung von bewährten Techniken *(best practices)*
  - Systemmodellierung
- Ein praxisrelevantes Hilfsmittel:
  - Unternehmens- und Geschäftsprozess-Modellierung

> "Attempt to understand before attempting to be understood."
>
> (Alan Davis in IEEE Software, Juli/August 1998)

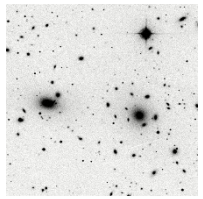
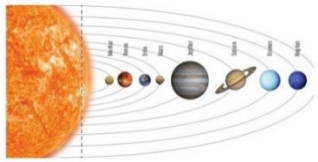
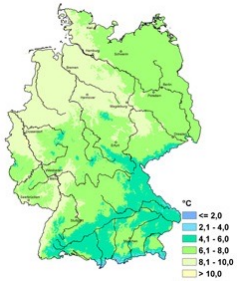
Probleme bei der Anforderungsermittlung

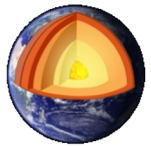
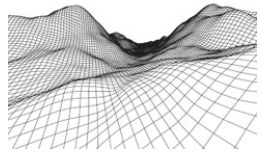
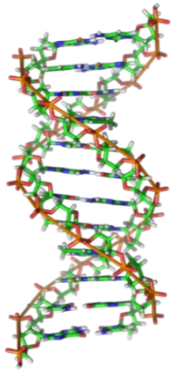
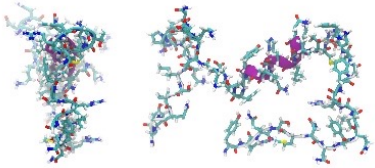
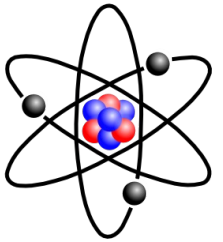
1. Viele Beteiligte in verschiedenen Rollen:
   1. Kunden, Benutzer
   2. Informatik-Spezialisten
   3. Betriebswirtschaft-Spezialisten
   4. Management, Marketing, ...
2. Kunden/Benutzer wissen meist nicht, was sie wirklich wollen.
3. Fachsprachen, nicht allgemein verständliche Begriffe
4. Verschiedene Beteiligte verwenden inkonsistente Begriffe.
5. Verschiedene Beteiligte haben widersprüchliche Ziele.
6. Organisatorische Rahmenbedingungen unklar oder veränderlich
7. Ständig veränderte Anforderungen, auch während Analyse und Entwurf des Systems.
8. Neue Beteiligte während oder nach der Analyse
9. Psychologische und soziale Randbedingungen:
   - z.B.: Mitarbeiter fürchten Rationalisierungseffekt und kooperieren deshalb schlecht.
   - Mitarbeiter und Organisationen sind oft änderungsresistent

Modelle in der Analysephase

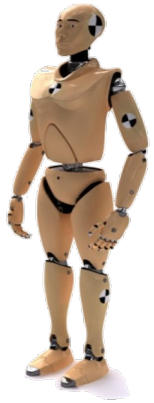
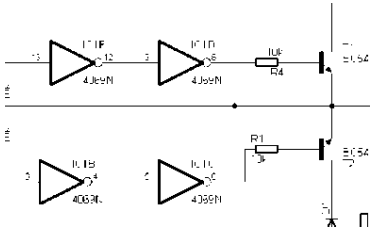
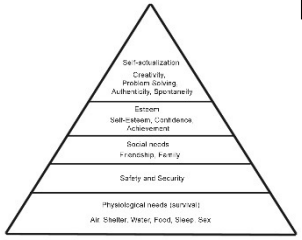


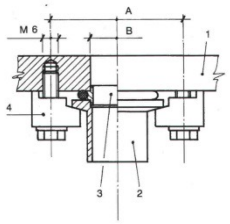
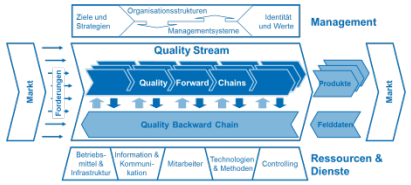
Modelle werden in allen Disziplinen verwendet

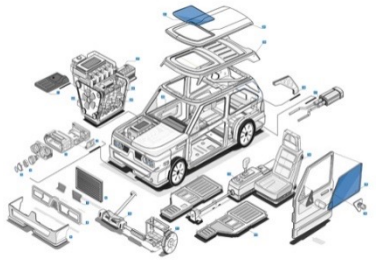




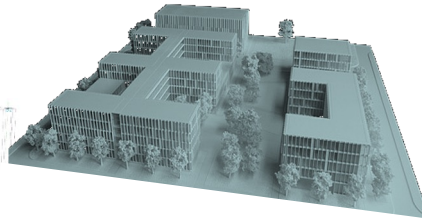







A model is an abstraction of an original made and used for a purpose.





Modellierungssprachen & Standards

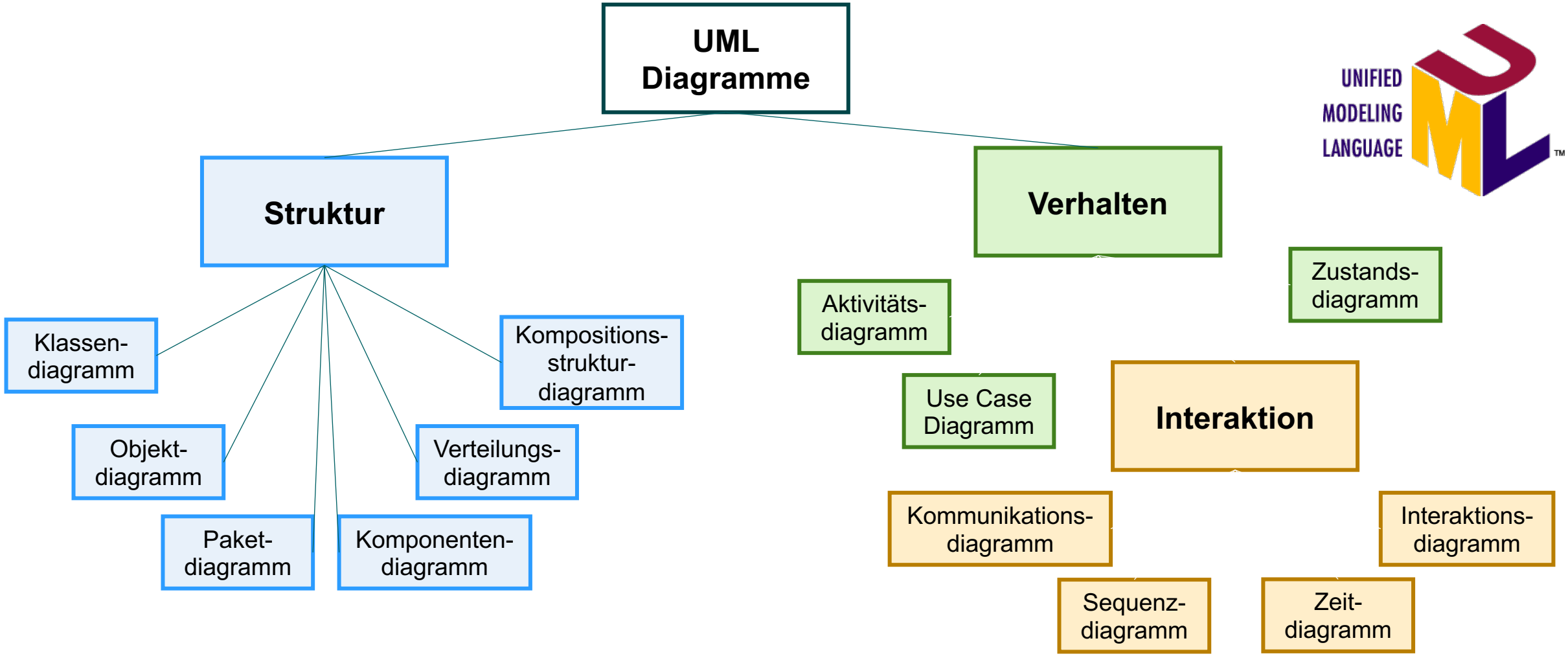
- *General Purpose Languages*
  - Versuchen viele unterschiedliche Zwecke zu vereinen
  - Standardisierungsorganisationen (z.B. Object Management Group, OMG)

  - *Unified Modeling Language, UML*
    - Unified Modeling Language
    - Vielzahl unterschiedlicher Diagrammtypen
    - Aktuelle Version: 2.5.1
    - Varianten: z.B. UML/P für die Programmierung

  - *Systems Modeling Language, SysML*
    - Einsatz: Systems Engineering
    - Modellierung verschiedener komplexer Systeme
    - Diagrammtypen teilweise aus UML 2.0, modifiziert oder ganz neu
    - Aktuell: Version 1.5

- *Domänenspezifische Sprachen*
  - Mehr dazu in Vertiefungen im Master!

Diagrammtypen in der UML



Geschäftsprozess *(Business Process)*

> "Unter einem Geschäftsprozess soll die zeitlich-logische Abfolge von Tätigkeiten zur Erfüllung einer betriebswirtschaftlichen Aufgabe verstanden werden, wobei eine Transformation von Material oder Information stattfindet."
>
> (Allweyer/Scheer 1995)

GP-Modellierung ist eine typische Domäne der *Wirtschaftsinformatik*

- Beschreibung von Geschäftsprozessen
- Analyse und Optimierung von Geschäftsprozessen
- Ableitung einer flexiblen Infrastruktur für die effiziente Durchführung von Geschäftsprozessen

Geschäftsprozess *(Business Process)*

Geschäftsprozessmodellierung in der ***Systemanalyse****:*

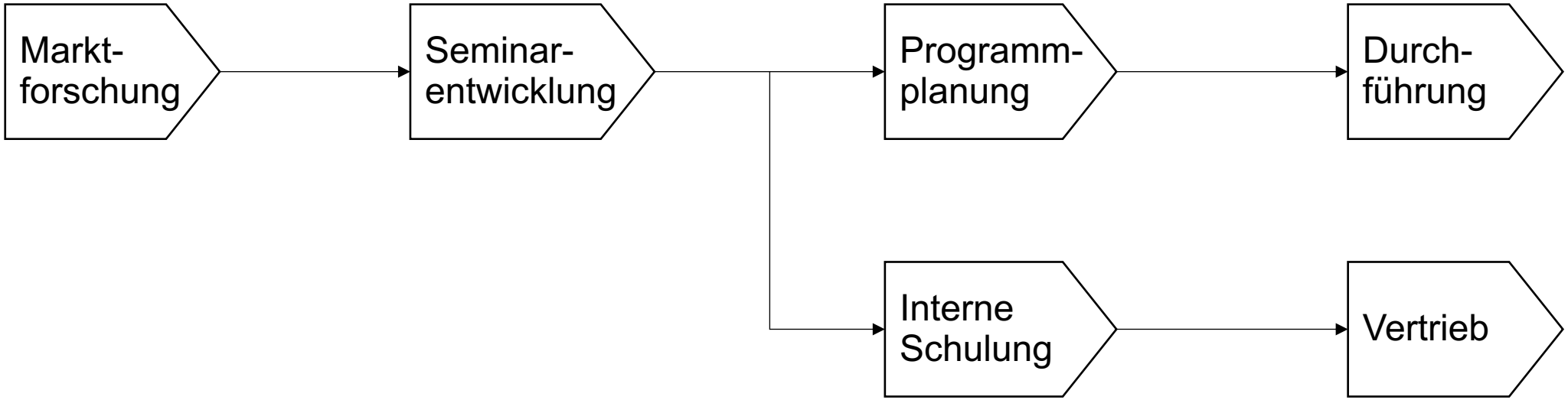
- Zeitlich
  - vor der Erstellung der Anforderungsspezifikation
- Analyse des "IST-Zustands"
  - hilft bei Beurteilung der Ziele und des "SOLL"
- Grobe Geschäftsprozessmodelle
  - Hilfreich beim Finden von Beteiligten und globalen Belangen
- Detaillierte Geschäftsprozessmodelle
  - Hilfreich beim Finden von funktionalen Anforderungen und Anwendungsfällen

Grobes Geschäftsprozessmodell

- Strategisch orientiert
- Ziel: Gesamtverständnis des Unternehmens und der Aufgabe

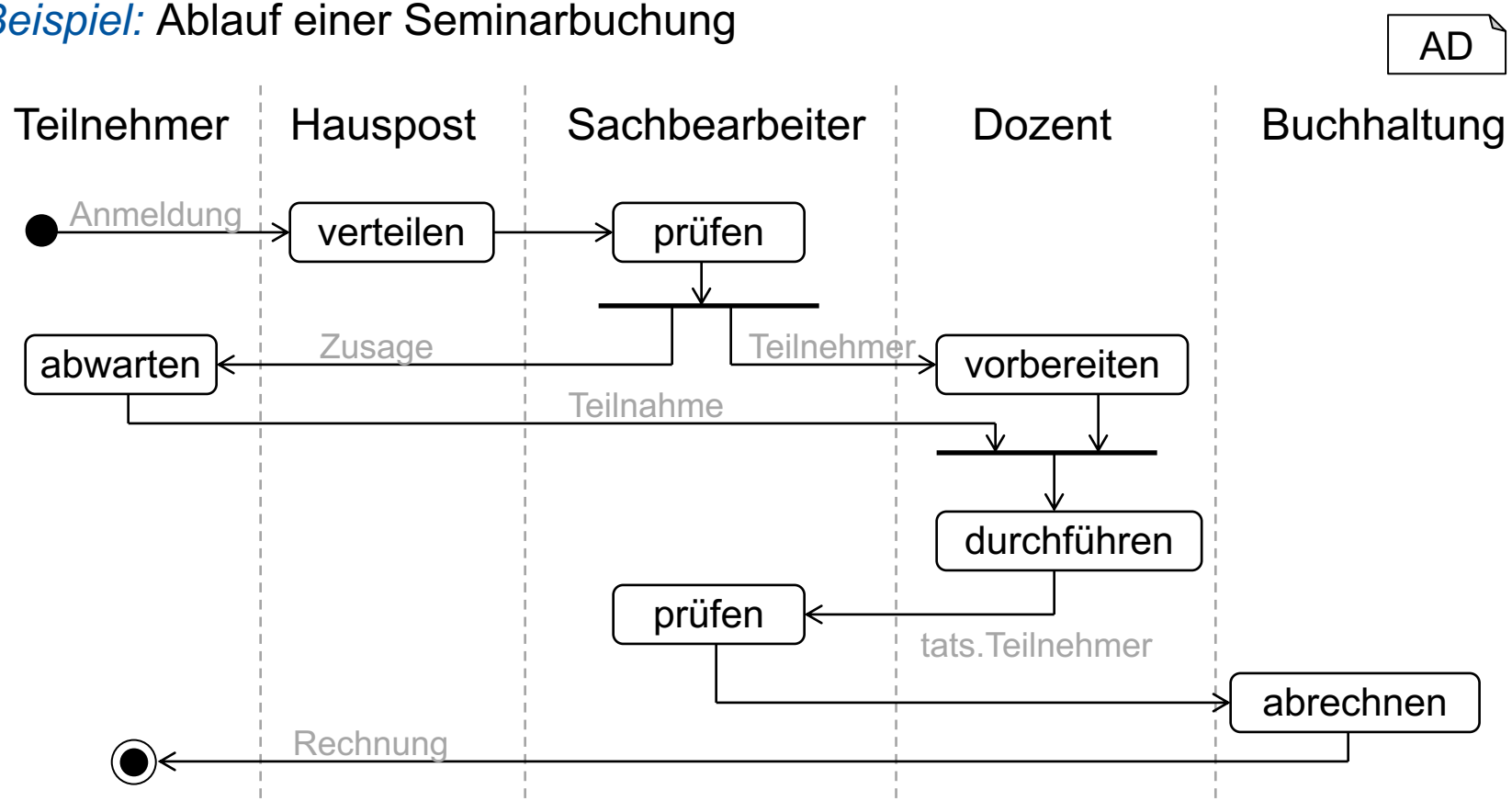
Beispiel

Notation an ARIS von Prof. Scheer angelehnt



Detailliertes Geschäftsprozessmodell

Beispiel: Ablauf einer Seminarbuchung



Gewählte Darstellungsform:

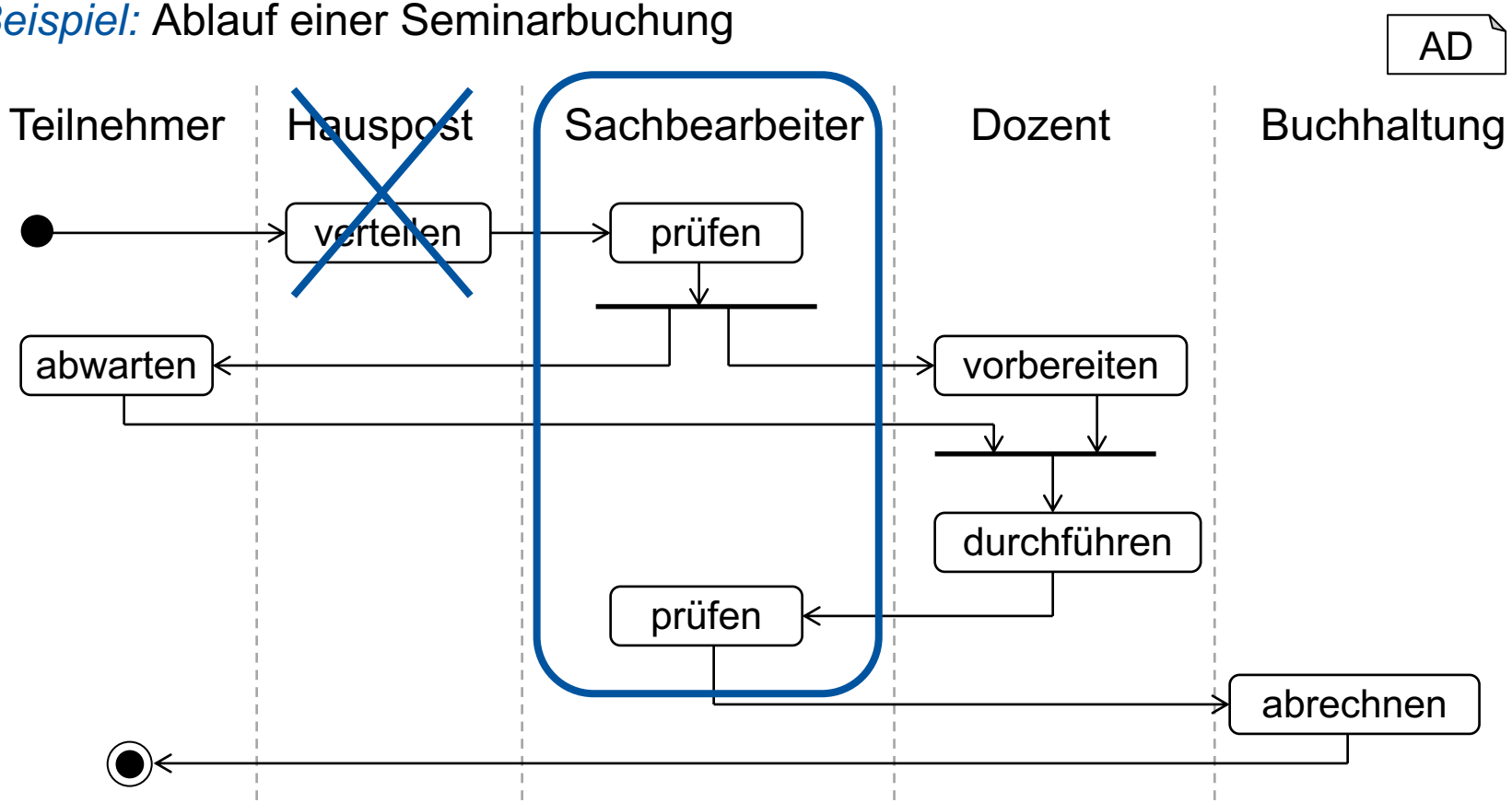
- UML-Aktivitätsdiagramm
- "Swimlanes" zur Betonung von Akteuren

Es gibt auch andere Darstellungsformen:

- Datenflussdiagramme
- Workflow Modeling Languages
- BPMN – Business Process Modelling Notation

Detailliertes Geschäftsprozessmodell

Beispiel: Ablauf einer Seminarbuchung



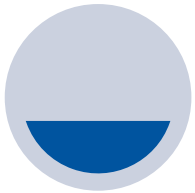
- Externe Kommunikations-beziehungen: sogenannter System-Kontext
- Akteure für das zu entwickelnde System:
  - Akteure innerhalb der Systemgrenze und
  - „benachbarter“ Bediener des Systems

Digitale Transformation

Wenn sie einen Scheißprozess digitalisieren, dann haben sie einen scheiß digitalen Prozess.

Quelle: Thorsten Dirks, CEO von Telefónica Deutschland, am Wirtschaftsgipfel der 'Süddeutschen Zeitung, 2015

Was haben wir gelernt?

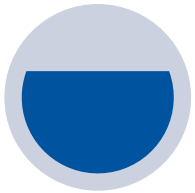


Anforderungsanalyse (System Analysis)

Besteht aus zwei Teilen, der Anforderungsermittlung und der Systemmodellierung

Anforderungsermittlung (Requirements Engineering) zur Feststellung der Anforderungen an das geplante System mit dem Ergebnis Anforderungsspezifikation (Pflichtenheft)

Fehler in der Anforderungsermittlung sind später teuer zu beheben

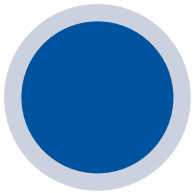


Zwei Arten von Anforderungen

Funktionale Anforderungen beschreiben was das System tun soll

Nicht funktionale Anforderungen beschreiben wie funktionale Anforderungen zu realisieren sind

Glossar ist ein einfaches und wesentliches Hilfsmittel zur Konsistenzsicherung



Modelle in der Analyse Phase

Unterschiedliche Arten für unterschiedliche Zwecke

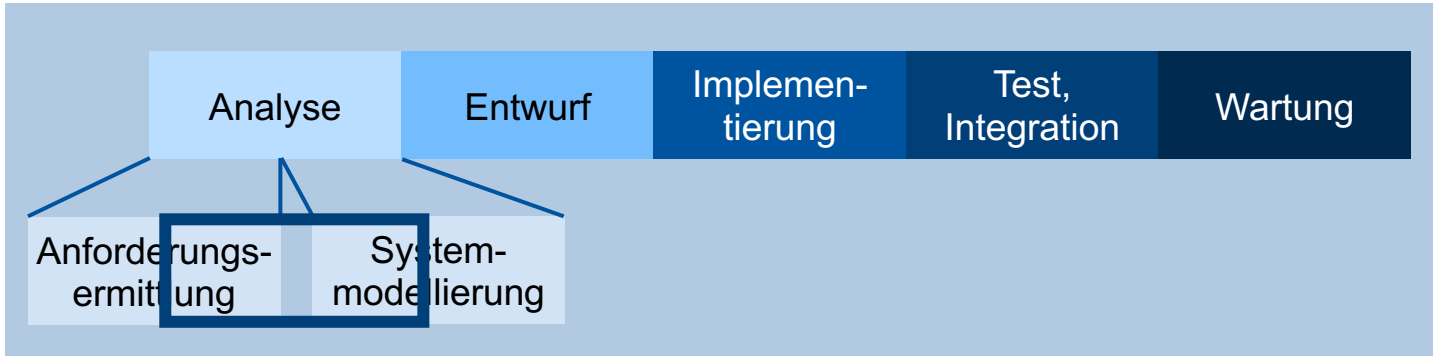
Aktivitätsdiagramme für die Geschäftsprozesse

Use-Cases für Anwendungsfälle

Klassenmodell für die Beschreibung relevanter Konzepte

Softwaretechnik

3. Anforderungsanalyse
3.4. Modellierung von Aktivitäten



Prof. Bernhard Rumpe
Software Engineering
RWTH Aachen

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

SE Software Engineering | RWTH AACHEN UNIVERSITY

Warum, was, wie und wozu? Modellierung von Aktivitäten

Warum?

- Software ist dazu da Verhalten (Funktionen) zu bieten
- Verständnis der Geschäftsprozesse

Was?

- Modellierung von Verhalten konkreter Aktivitäten

Wie?

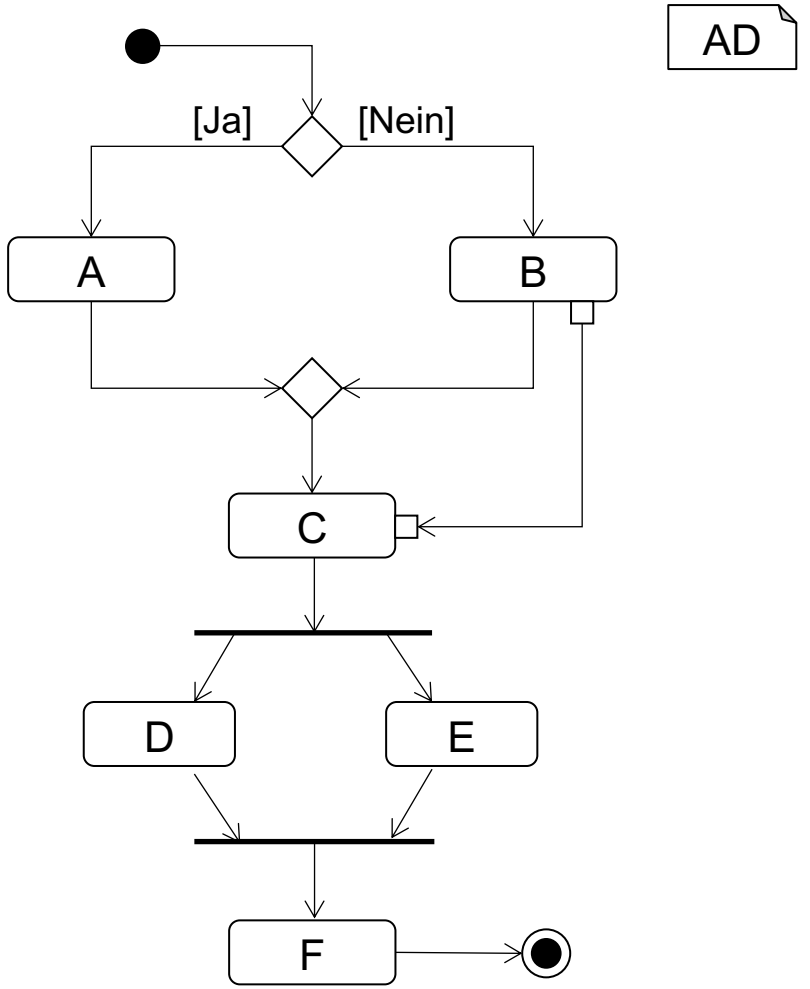
- Erstellung von Aktivitätsdiagrammen
- Verfeinerung von Use Cases oder aus Anforderungen

Wozu?

- Gemeinsames Verständnis für Prozesse und Arbeitsabläufe von KundInnen mit Hilfe von DomänenexpertInnen

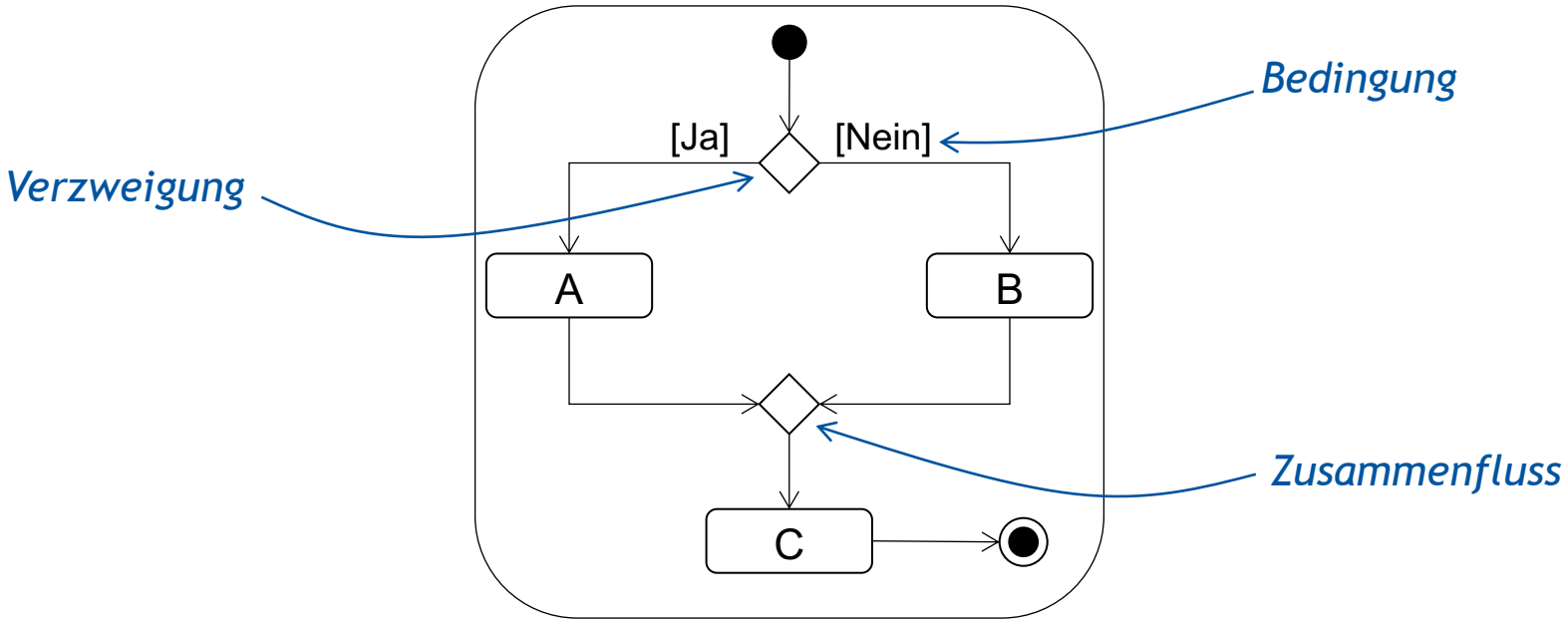
UML Aktivitätsdiagramme

- Beschreibung von Verhalten
- Modellierung von Geschäftsprozessen bzw. Kontrollflüssen zwischen
  - BenutzerInnen und Systemkomponenten
- Beschreibung von
  - parallelen,
  - sequentiell abhängigen, und
  - alternativen
- Prozessen/Aktivitäten
- Werden oft als Verfeinerung von Use-Cases genutzt



Aktivitätsdiagramme (Verzweigung)

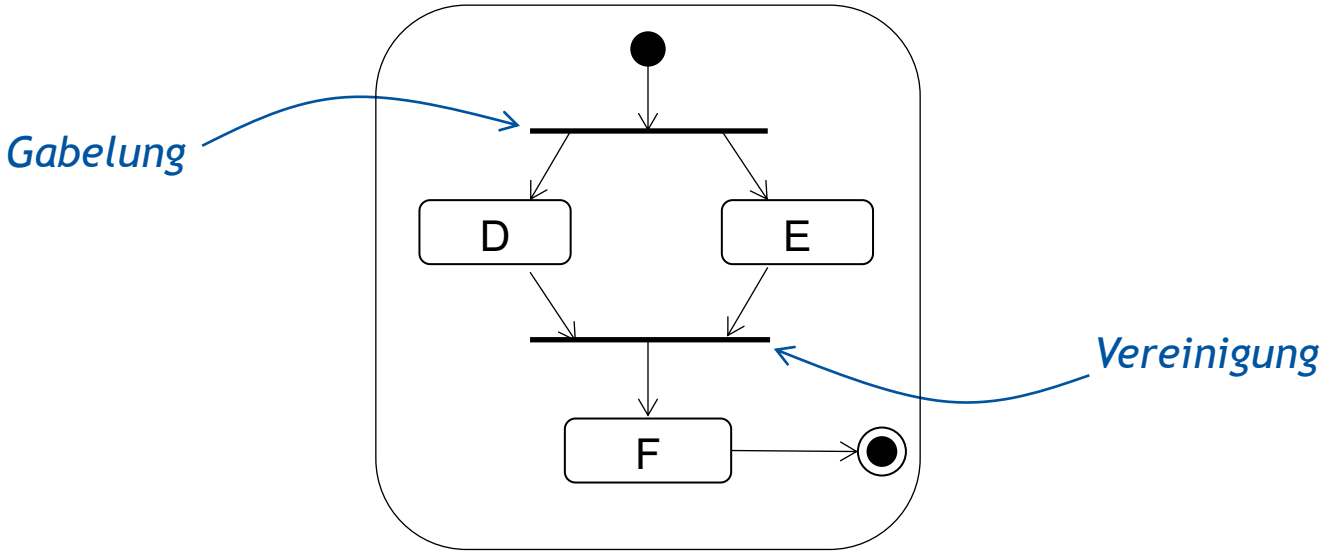
- Verzweigung und Zusammenfluss im Kontrollfluss
- Bedingungen (in eckigen Klammern), die wahr sein müssen, um eine Transition zu nehmen



- Nach Verzweigung kann nur A oder B ausgeführt werden; danach weiter bei C
  - Analog: if-then-else

Aktivitätsdiagramme (Parallelität)

- Gabelung:
  - Verteilung einer eingehenden Transition auf mehrere parallel auszuführende Aktionen
  - An ausgehenden Transitionen dürfen keine Bedingungen stehen
- Vereinigung:
  - Bündelung mehrerer eingehender Transitionen zu einer Transition
  - Wird ausgeführt, sobald alle parallelen Aktivitäten beendet sind



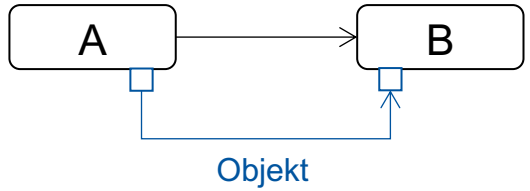
Aktivitätsdiagramme | Konzepte

- Oben stehen (optional) die Akteure (das sind Komponenten oder Klassen)
- Die Aktivitäten jedes Akteurs stehen dann in einer Swimlane.

- Aktivitäten (Rechtecke mit abgerundeten Ecken) = Aktionen
  - Aktionsparameter repräsentieren Ein- und Ausgabe
  - Spezielle Send- und Accept-Message-Aktionen

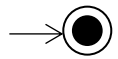
- Transitionen zwischen Aktionen == Kontrollfluss

- Transitionen zwischen Ein- und Ausgaben von Aktionen == Objektfluss

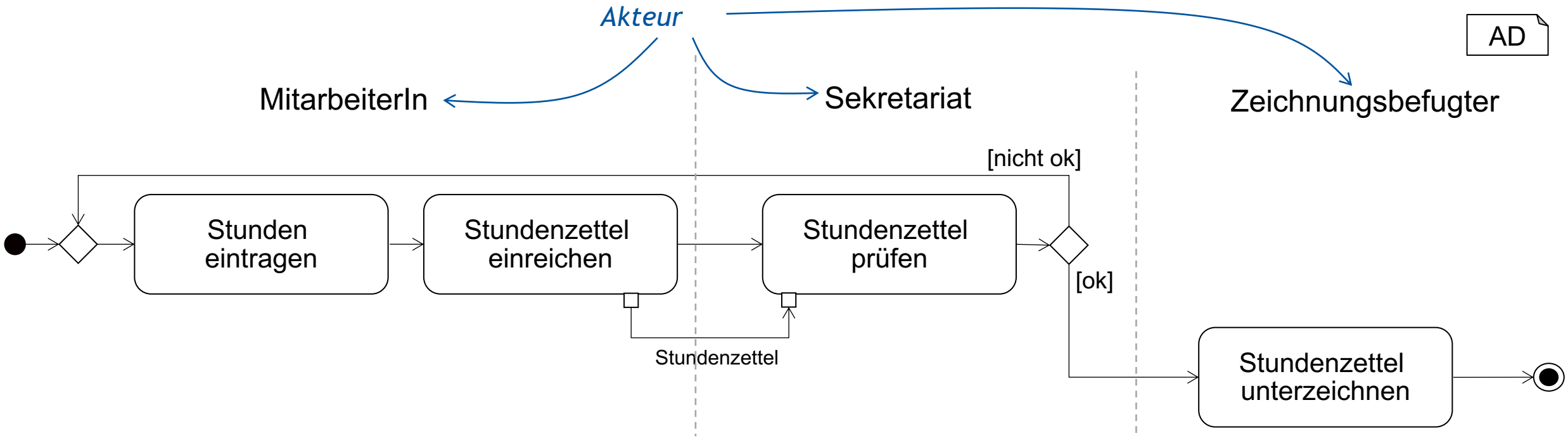


- Gefüllter Kreis = Startknoten

- Bull‘s eye = Endknoten



Stundenzettel archivieren

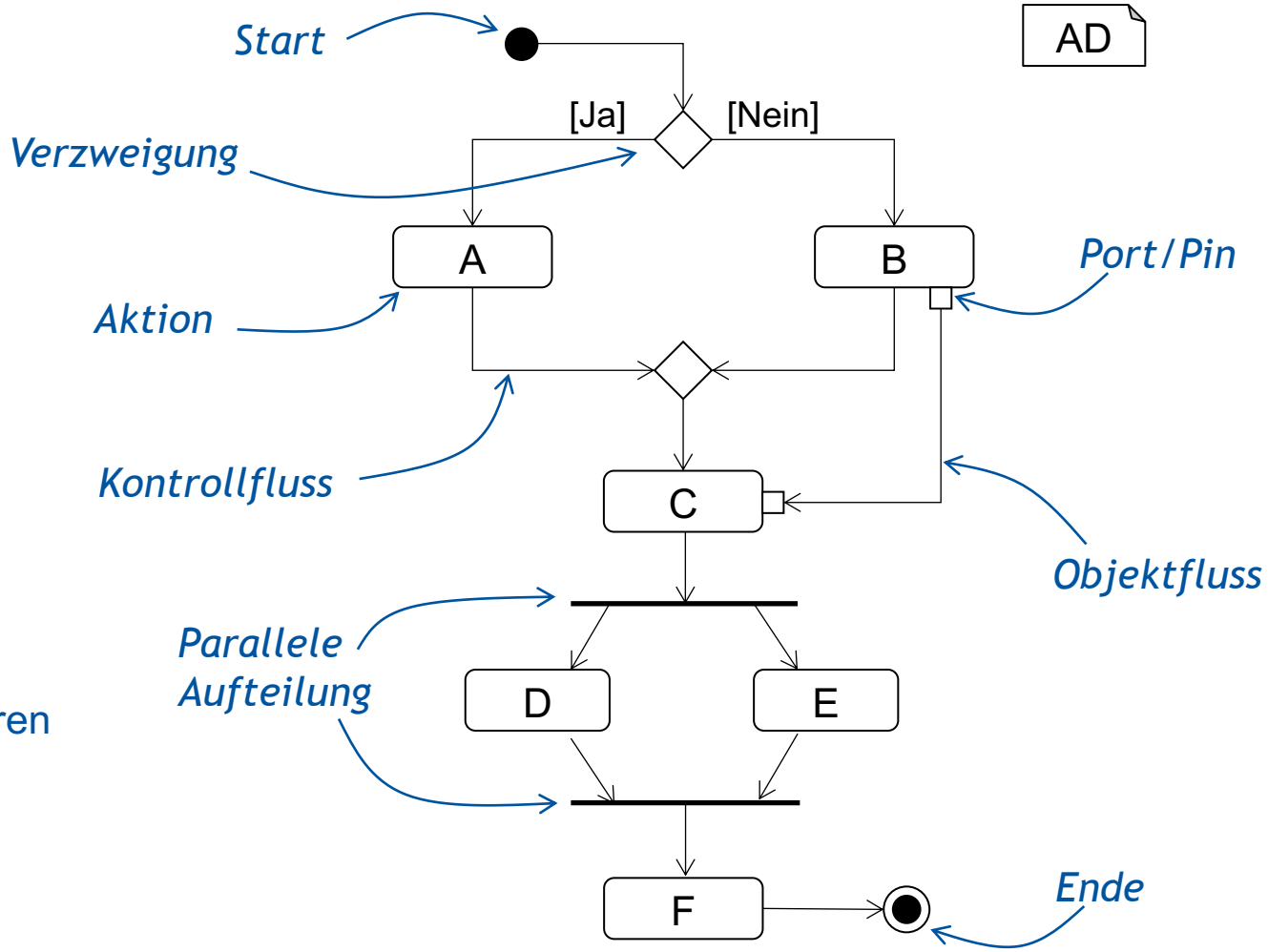


Alle MitarbeiterInnen in Drittmittelprojekten müssen Stundenzettel führen. Dafür trägt ein/e MitarbeiterIn die Stunden in den virtuellen Stundenzettel ein und reicht diesen ein, wenn alle Stunden eines Monats erfasst sind. Das Sekretariat prüft den Stundenzettel z.B. ob an keinem Tag mehr als 10 Stunden gearbeitet wurde. Wenn dieser in Ordnung ist wird er an den Zeichnungsbefugten zur Unterzeichnung weiter geleitet. Wenn die Prüfung fehlschlägt muss die/der MitarbeiterIn die Stunden anpassen und den Stundenzettel wieder einreichen.

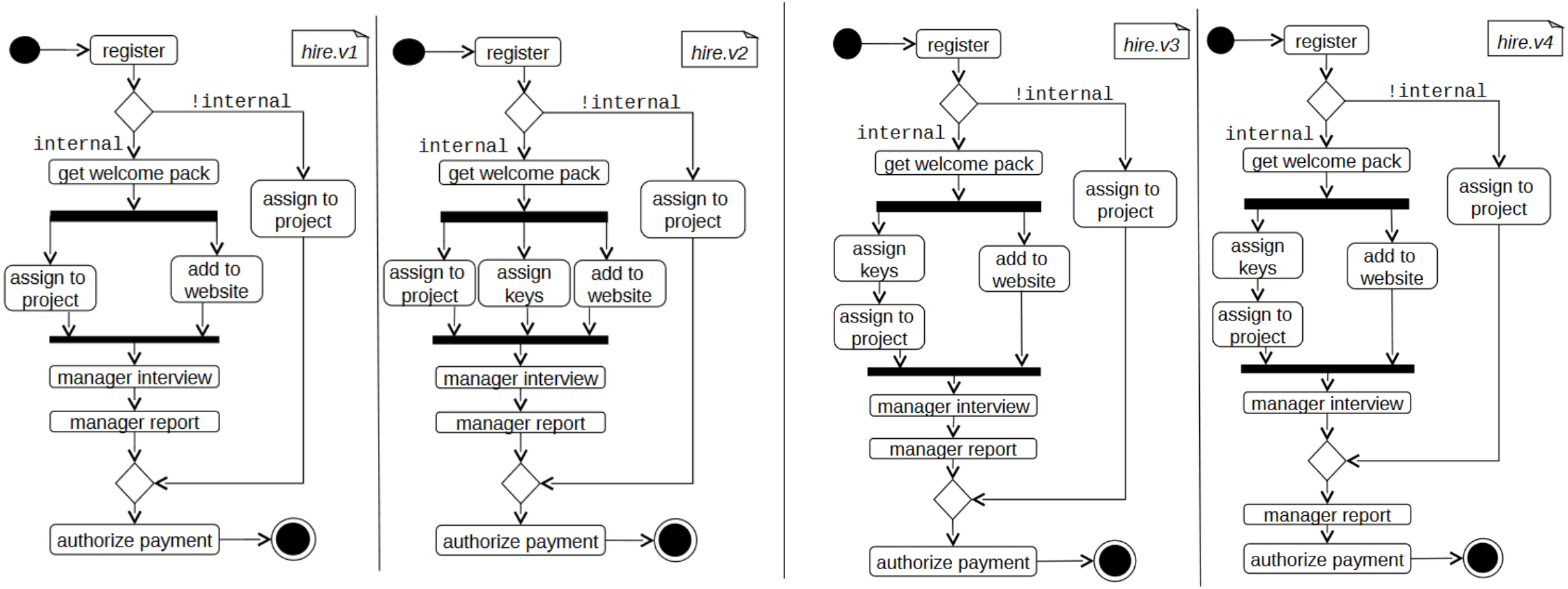
AD im Vorlesungsstandard

UML Aktivitätsdiagramme | Konzepte & Grundregeln

- Abläufe bestehen aus
  - Aktionen
  - Transitionen
    - Kontrollflüsse
    - Objektflüsse
  - Verzweigungen
  - Parallele Aufteilungen
- Optional: Akteure („Swimlanes“)
- Start- und Endpunkt!
  - Wiederholung möglich
- Grundregeln
  - Parallele Aufteilungen immer wieder zusammenführen
  - Verzweigungen immer zusammen- oder rückführen
  - 1 Fluss in Aktion rein sowie 1 Fluss raus
  - Alles zu einem Endpunkt führen
  - Lesefluss: Links nach rechts, oben nach unten

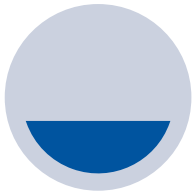


Evolution von Ads über die Projektlaufzeit. Welche Abläufe werden möglich/verboten?



https://www.se-rwth.de/publications/ADDiff-Semantic-Differencing-for-Activity-Diagrams.pdf; AD im Vorlesungsstandard

Was haben wir gelernt?



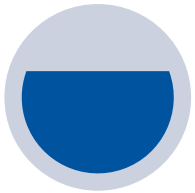
Aktivitätsdiagramme

Beschreiben Verhalten

Geschäftsprozessen bzw. Kontrollflüssen zwischen BenutzerInnen und Systemkomponenten

Parallele, sequentiell abhängige, und alternative Prozessen/Aktivitäten, die über Transitionen verbunden sind

Können sich über die Zeit hinweg ändern und weiter entwickeln



Grundregeln

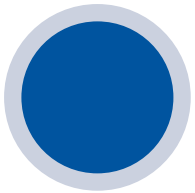
Lesefluss: Links nach rechts, oben nach unten

Parallele Aufteilungen immer wieder zusammenführen

Verzweigungen immer zusammenführen

1 Fluss in Aktion rein sowie 1 Fluss raus

Alles zu einem Endpunkt führen



Erstellung

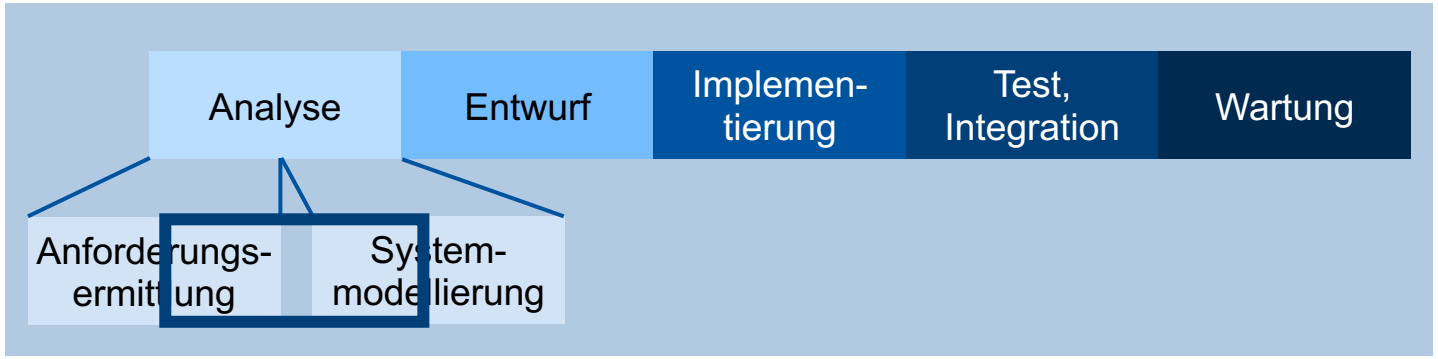
Analyse des Anwendungsfalls

Gespräche mit DomänenexpertInnen und/oder aus einem Lastenheft

Identifikation von AkteurInnen und Aktivitäten

Softwaretechnik

3. Anforderungsanalyse
3.4. Use Case Modellierung



Prof. Bernhard Rumpe
Software Engineering
RWTH Aachen

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

Literatur:

- M. Hitz/G. Kappel, UML@Work, dpunkt 1999

Warum, was, wie und wozu? Modellierung von Anforderungen

Warum?

- Gut erhobene Anforderungen minimieren Fehler im zu erstellenden System
- Text oft nicht ausreichend um darüber mit Kunden zu kommunizieren

Was?

- Modellierung von Anforderungen

Wie?

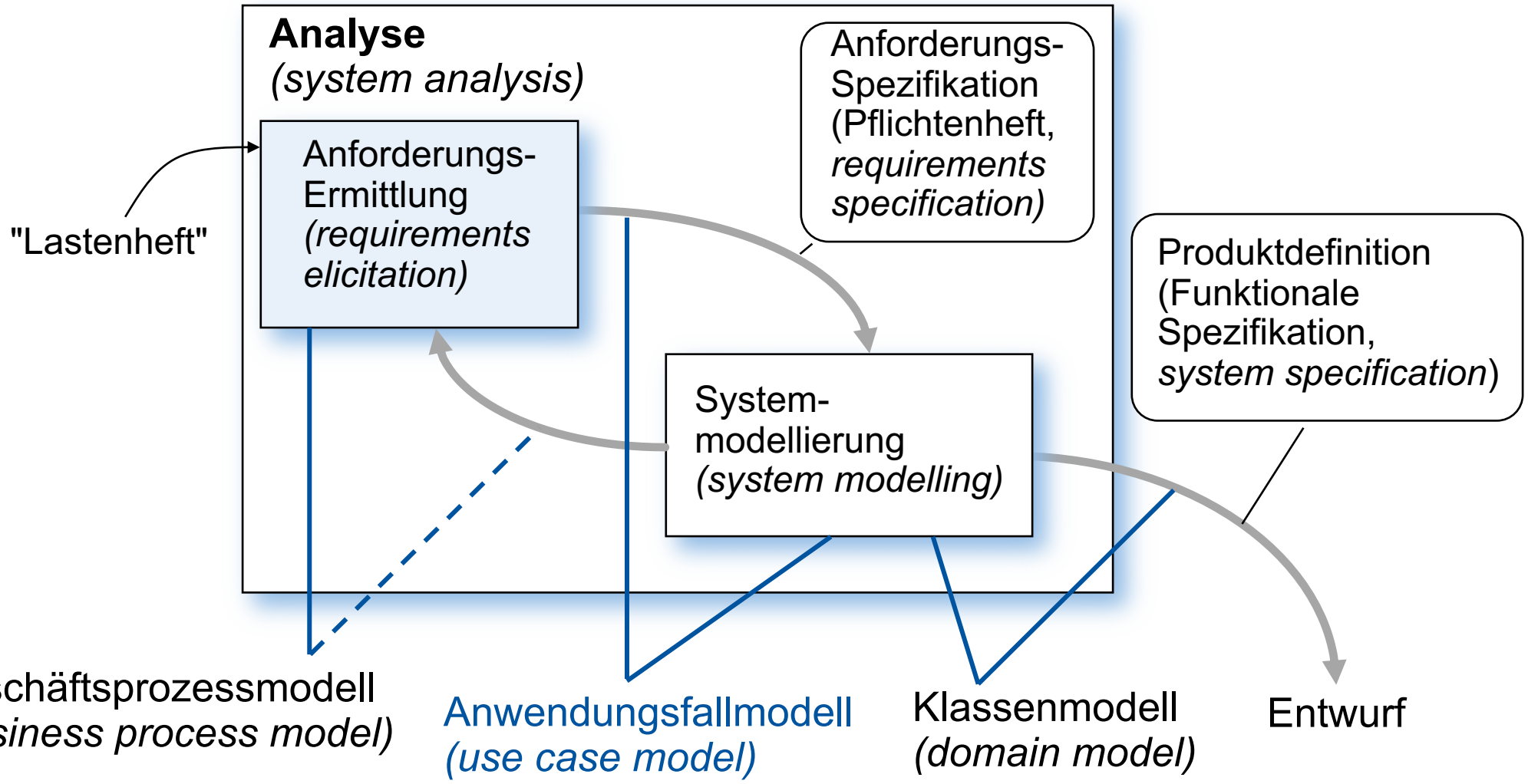
- UML Use Case Diagramme bzw. Anwendungsfallmodelle
- Use Cases aus Geschäftsprozessen und funktionalen Anforderungen ableiten

Wozu?

- Erhöht die Übersichtlichkeit
- Bessere Diskussionsbasis mit Kunden

Modelle in der Analysephase

Wiederholung

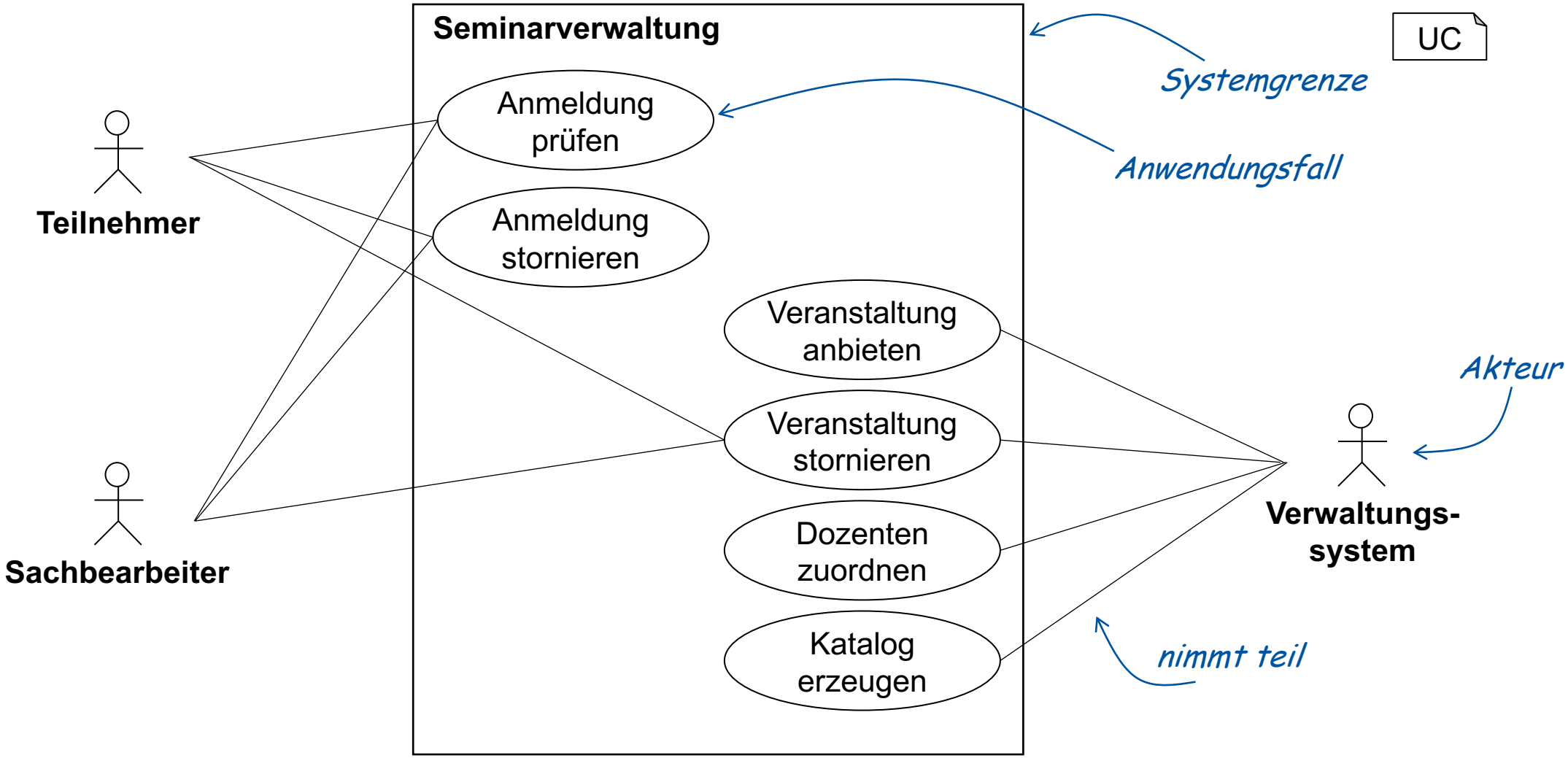


Anwendungsfälle (Use-Cases) | Grundbegriffe

- Ein **Anwendungsfall** (synonym Use-Case) ist die Beschreibung einer Klasse von Aktionsfolgen, die ein System ausführen kann, wenn es mit Akteuren interagiert. ("Systemgestützter Geschäftsprozess")
- Ein **Akteur** ist die Beschreibung einer Rolle, die ein(e) Benutzer(in) oder ein anderes System spielt, wenn er/sie/es mit dem System interagiert.
  - Mensch/Maschine, primär(=Nutznießer)/sekundär(=Mithelfer), aktiv/passiv
- Ein Akteur **nimmt** an einem Anwendungsfall **teil**, wenn er an einer der im Anwendungsfall beschriebenen Aktionen beteiligt ist.

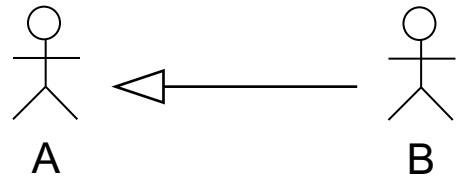
- Beschreibung eines Anwendungsfalls:
  - Liste der Akteure (z.B. "Kundenbetreuer")
  - Name des Anwendungsfalls ("Anmeldungsstornierung")
  - Kurzer Beschreibungstext (oft nicht notwendig)

Use-Case-Diagramm organisiert Anwendungsfälle | Beispiel und Konzepte

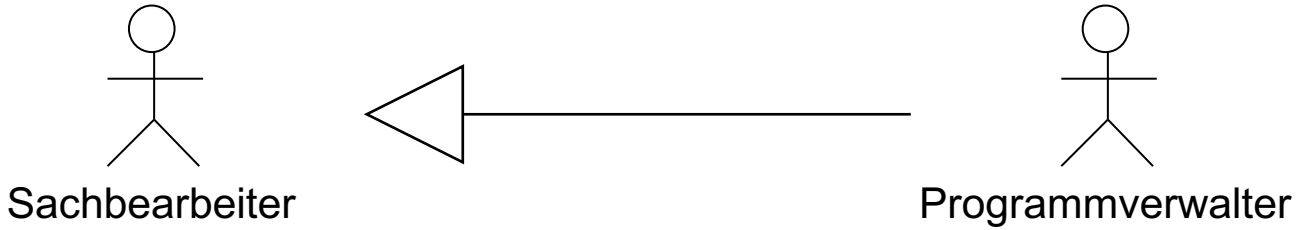


Spezialisierung von Akteuren

- **Spezialisierung** eines Akteurs:
  - Die Rolle von Akteur B enthält die Rolle von Akteur A.
  - Notation: Vererbungsbeziehung
  - Semantik: Akteur B interagiert implizit auch mit allen Anwendungsfällen, die für Akteur A aufgeführt sind.

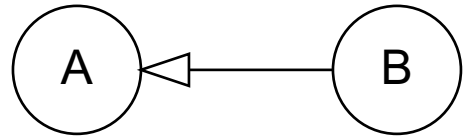


Beispiel:

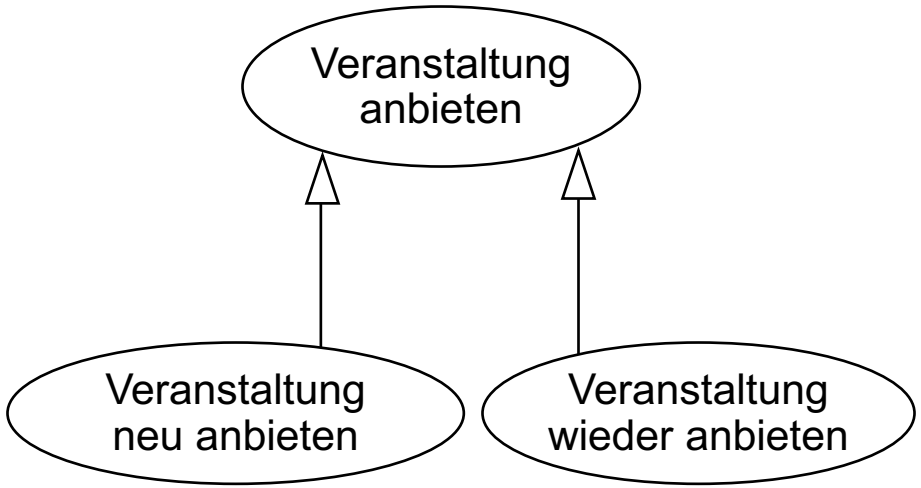


Spezialisierung von Anwendungsfällen

- **Spezialisierung** eines Anwendungsfalles:
  - Anwendungsfall B ist eine spezielle Variante von Anwendungsfall A.
  - Notation und Semantik: Vererbungsbeziehung.



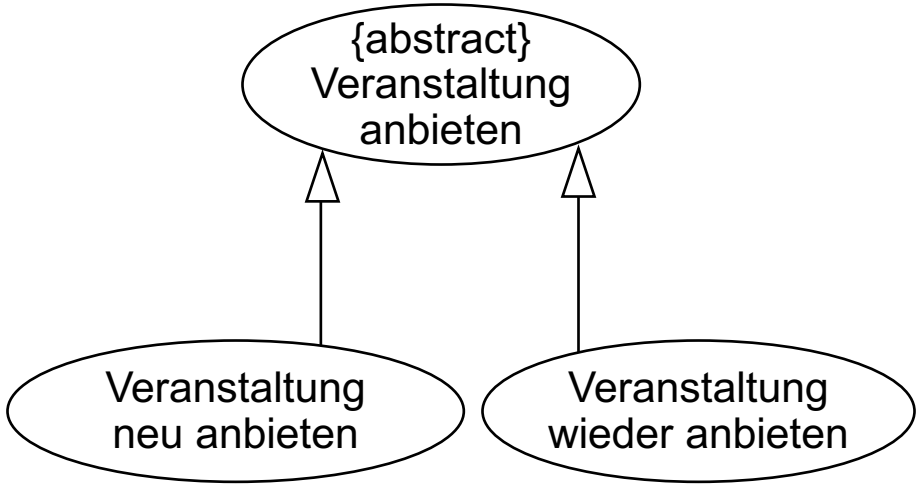
Beispiel:



Abstrakte Anwendungsfälle

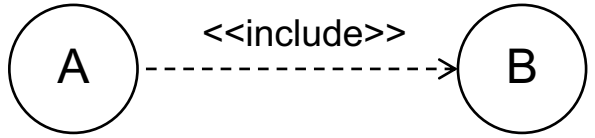
- **Abstrakter** Anwendungsfall:
  - Ein Anwendungsfall, der nicht eigenständig durch Aktionsfolgen realisiert wird, sondern nur Gemeinsamkeiten anderer Anwendungsfälle festhält.
  - Oft beziehen sich abstrakte Anwendungsfälle auf abstrakte Klassen.
- Spezialisierung von Anwendungsfällen ist meist von dieser Art.

Beispiel:

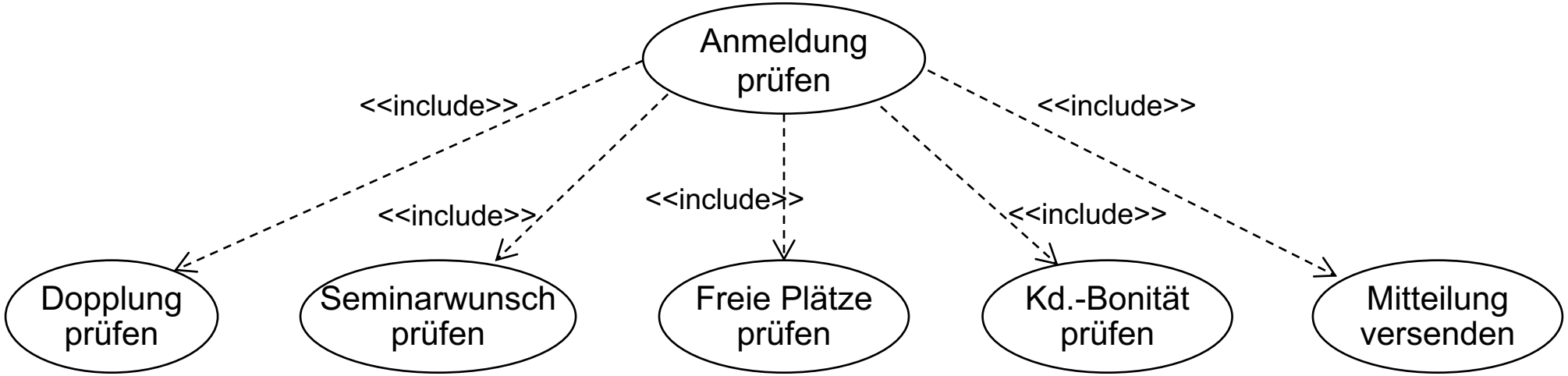


Einbindung von Anwendungsfällen

- **Einbindung** (*include*) eines Anwendungsfalles:
  - Anwendungsfall B wird „aufgerufen“ in Anwendungsfall A.
  - B ist ein unbedingt notwendiger „Unter-Anwendungsfall“ von A.
  - Zweck: Erhöhung der Wiederverwendung; Strukturierung

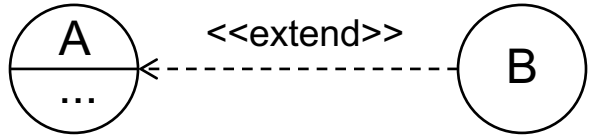


Beispiel:

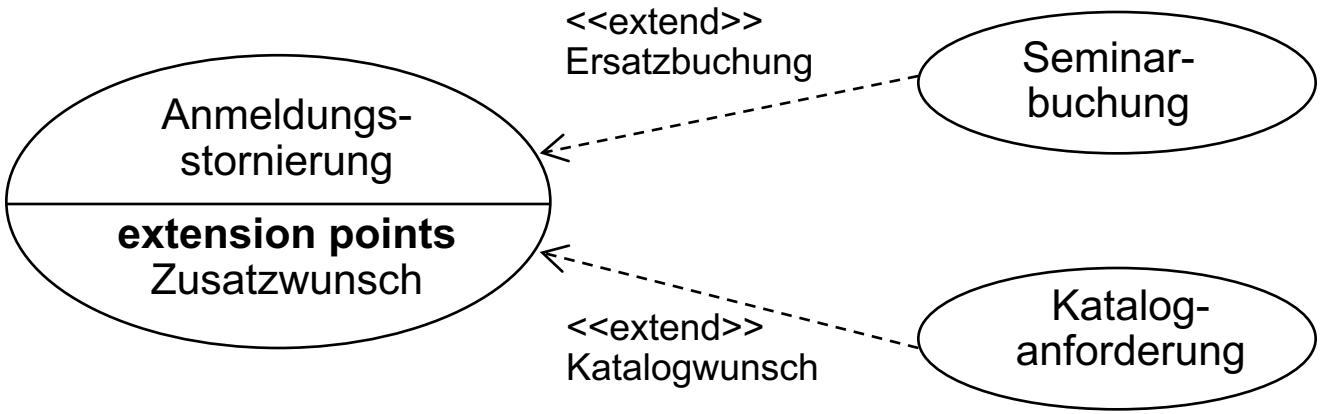


Erweiterung von Anwendungsfällen

- **Erweiterung** (*extend*) eines Anwendungsfalles:
  - Anwendungsfall A kann durch Anwendungsfall B „ergänzt“ werden.
  - Anwendungsfall A muss explizite „Anknüpfungspunkte“ (*extension points*) für B vorsehen
  - Tatsächliche Ausführung von B hängt von speziellen Bedingungen ab.

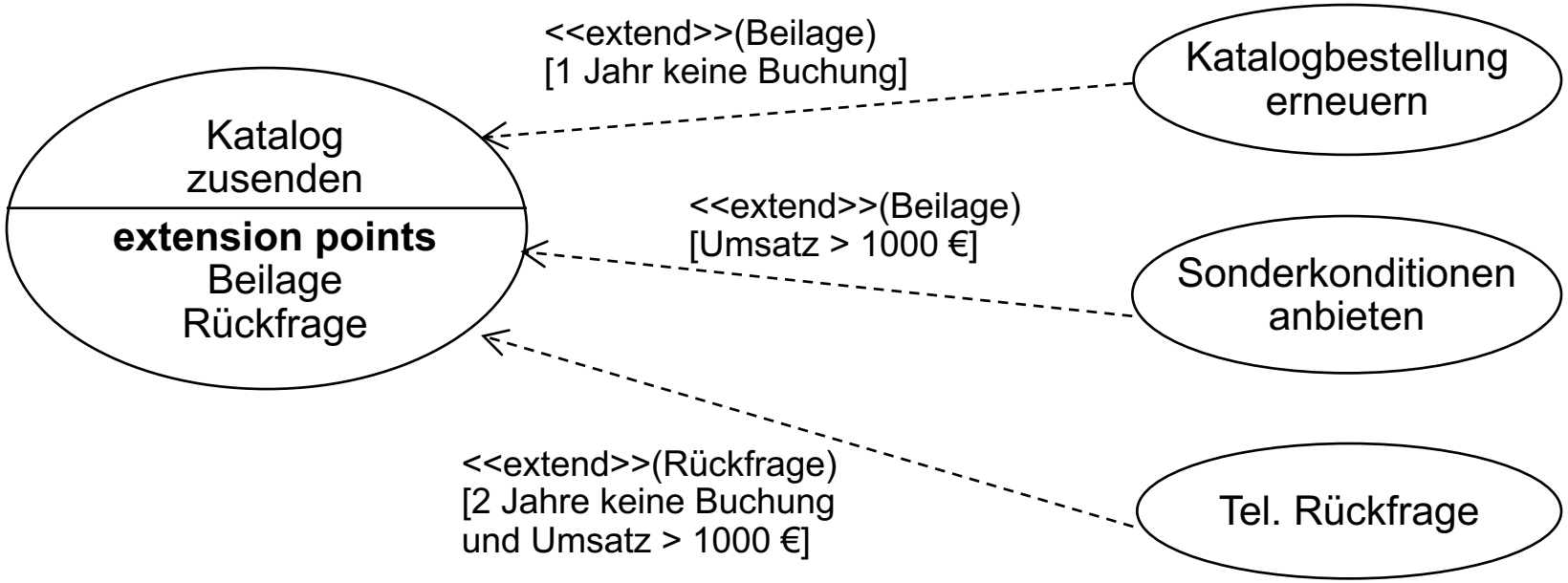


Beispiel:

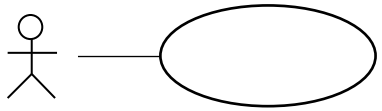
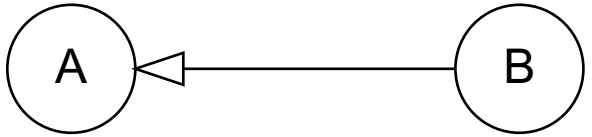
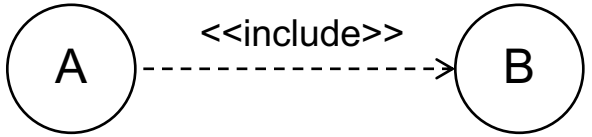
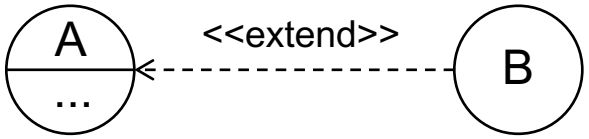


Genaue Spezifikation der Erweiterung

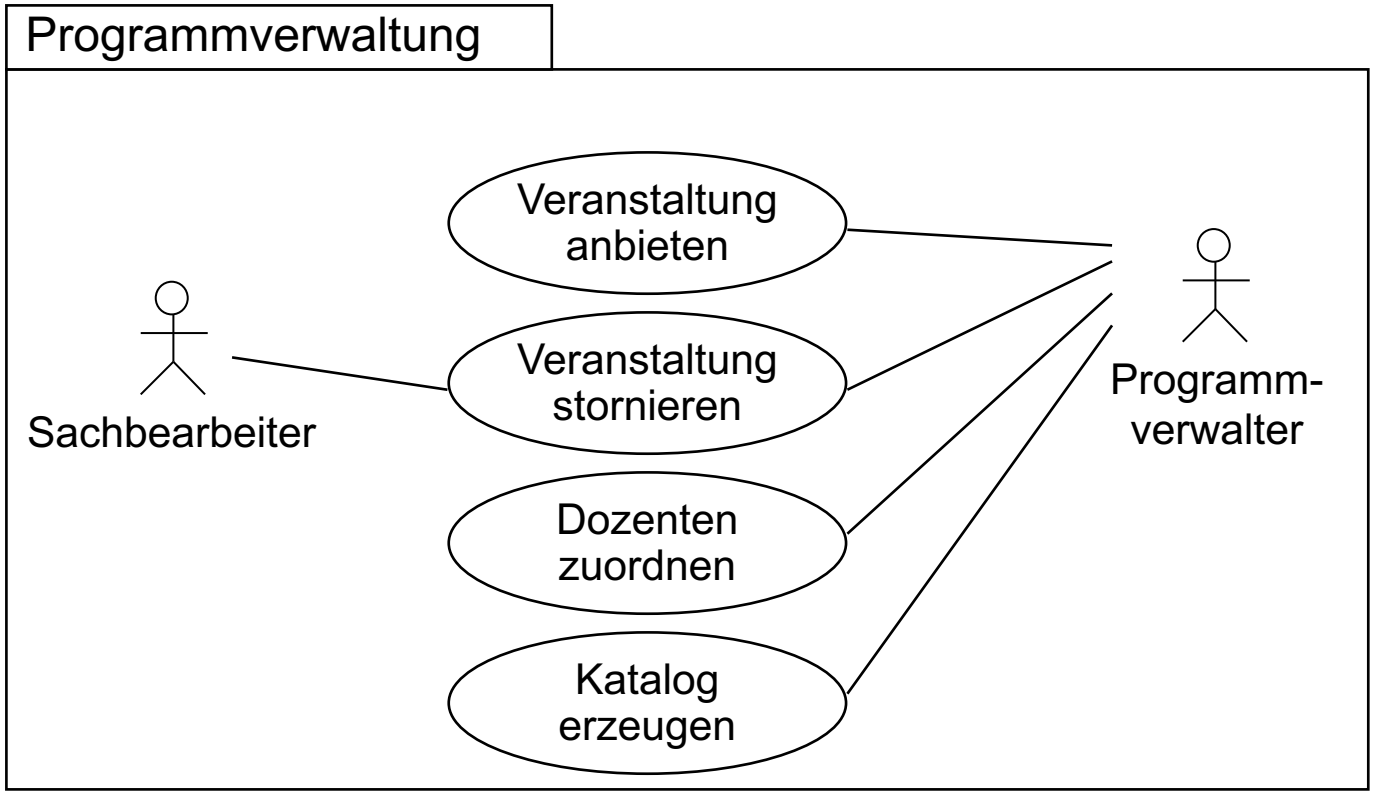
- Wenn mehrere *extension points* angegeben sind, ist es notwendig, den betroffenen *extension point* einer Erweiterung anzugeben (in runden Klammern).
- Es kann darüber hinaus sinnvoll sein, Bedingungen für die Erweiterung anzugeben (in eckigen Klammern).



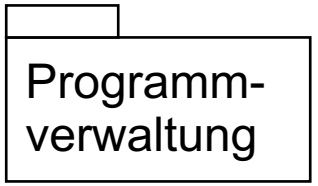
Anwendungsfall-Beziehungen: Vergleich

		Akteur interagiert mit…
	B Spezialfall von A	B (und kennt Funktionalität von A) wenn A abstrakt ist (was meist der Fall ist)
	B notwendiger Teil von A	A (und weiß nichts von B)
	B optionaler Zusatz zu A	A und B (im Allgemeinen)

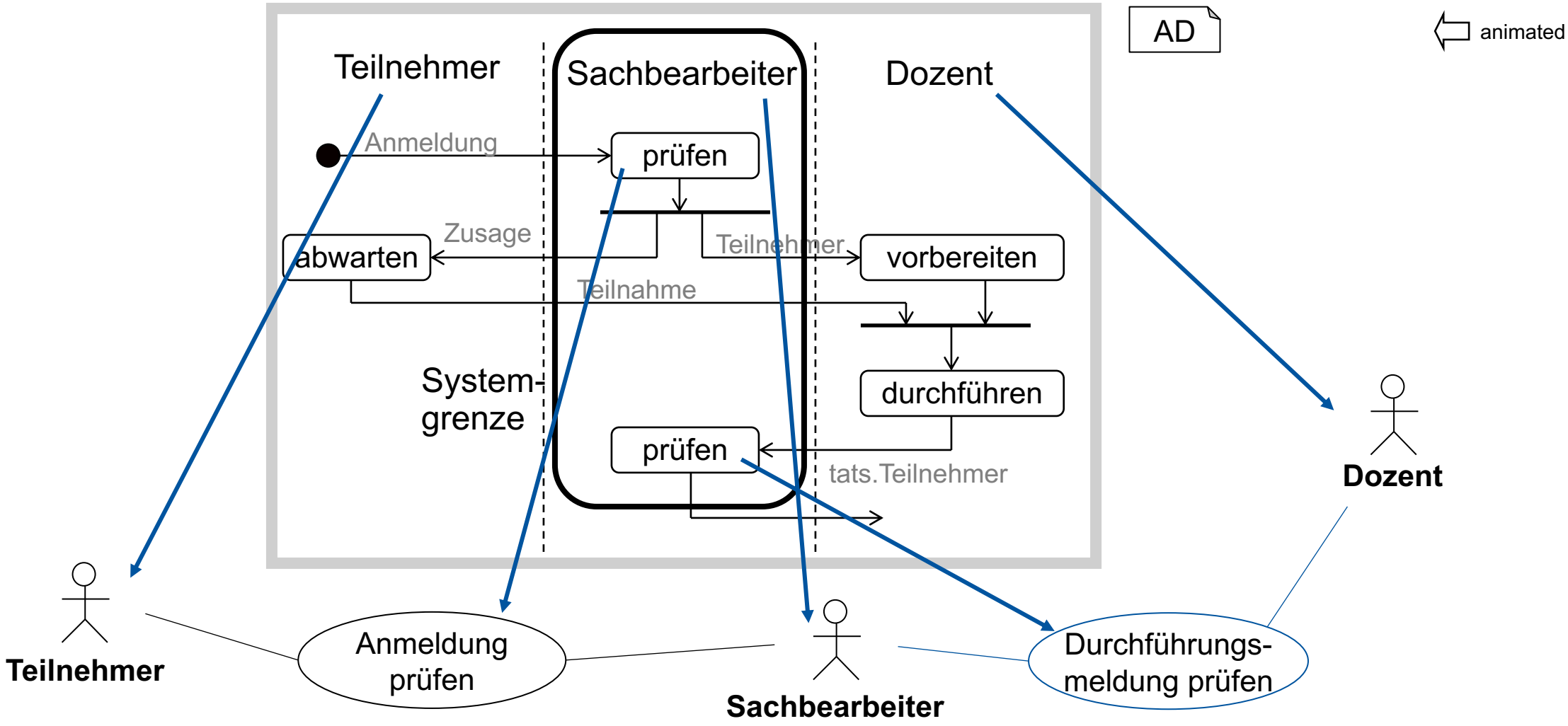
Strukturierung in Pakete



„Ikonifizierte" Ansicht:



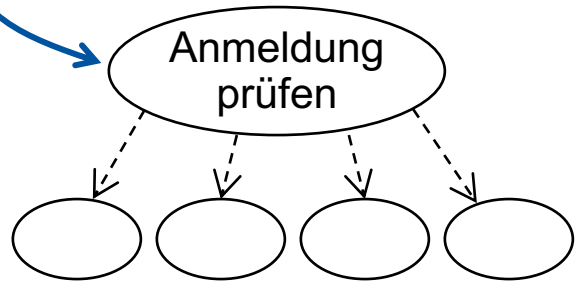
Wie? | Use-Cases aus Geschäftsprozess-Modellen

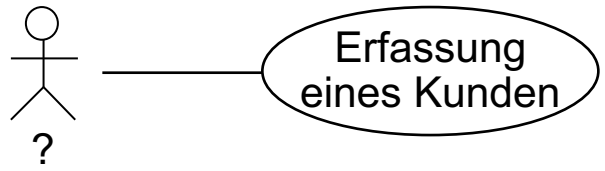


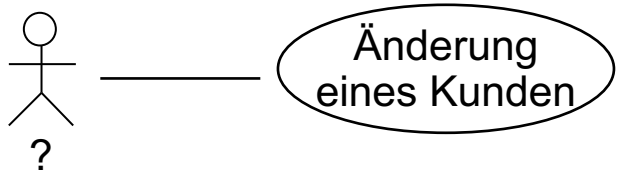
Wie? | Use-Cases aus funktionalen Anforderungen

4.1 Kundenverwaltung

/F10/	Ersterfassung, Änderung und Löschung von Kunden
/F15/	Ersterfassung, Änderung und Löschung von Firmen, die Mitarbeiter zu Seminaren schicken
/F20/	Anmeldung eines Kunden mit Überprüfung
/F30/	- ob er bereits angemeldet ist
/F40/	- ob der angegebene Seminarwunsch möglich ist
/F50/	- ob das Seminar noch frei ist
/F55/	- wie die Zahlungsmoral ist



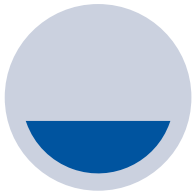




Anwendungsfälle: Detaillierungsgrad

- Was ist ein Anwendungsfall?
  - Menüpunkt für den Anwender
  - Abstrakte Systemfunktion
  - Benutzerziel
- Gefahren:
  - Vielzahl von Anwendungsfällen: Unübersichtlichkeit
  - Detaillierte Szenarien: Niedriger Abstraktionsgrad
  - Anwendungsfälle & Szenarien als Vertrag: Unvollständigkeit
- Erstrebenswerte Ziele:
  - Gute Strukturierung
  - Abstrakte, allgemeine Anwendungsfälle
    - Niemals Anwendungslogik in Anwendungsfällen codieren
    - Ggf. Aktivitätsdiagramm(e) für komplexe Anwendungsfälle
  - Anwendungsfälle & Szenarien hinreichend detailliert
    - geeignet als Testfälle

Was haben wir gelernt?

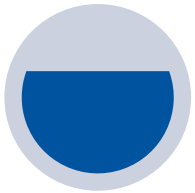


Anwendungsfälle (Use-Cases)

beschreiben die Systemfunktionalität,

die zu bewältigenden Aufgaben und

deren Anwender (Akteure)

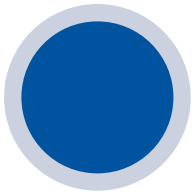


Use Case Diagramme

strukturieren Anwendungsfälle

bestehen aus:

- Akteuren, Anwendungsfällen
- Wer nimmt an welchem Anwendungsfall teil?
- ... Spezialfall von ...
- <<extend>>, <<include>> - Beziehungen



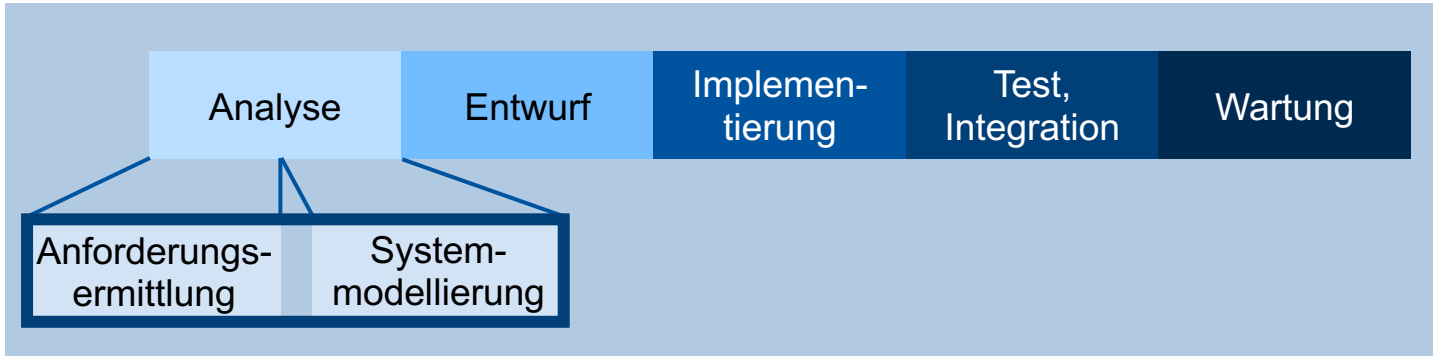
Erstellung

Identifikation in bestehenden Geschäftsprozessen bzw. als Erweiterung von diesen

Identifikation in funktionalen Anforderungen

Softwaretechnik

3. Anforderungsanalyse
3.5. Prototyping



Prof. Bernhard Rumpe
Software Engineering
RWTH Aachen

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

Warum, was, wie und wozu? Prototypen in der Analyse

Warum?

- Realisierbarkeit von Anwendungen und Konzepten überprüfen
- Risiken minimieren (nicht in falsche Richtung entwickeln)

Was?

- Erstellung von explorativen bzw. experimentellen Prototypen für die Analyse

Wie?

- Erstellung des Prototyps aus den Anforderungen
- Feedback von AnwenderInnen

Wozu?

- Bessere Planbarkeit der zu entwickelnden Anwendung

Prototyping und Prototypen

- Prototyp in klassischen Ingenieurdisziplinen:
  - Erstes Exemplar, Modell für ein neues Produkt
  - Voll funktionsfähig
  - Noch nicht in Serie fertigbar
  - Lange Entwicklungszeit, teuer

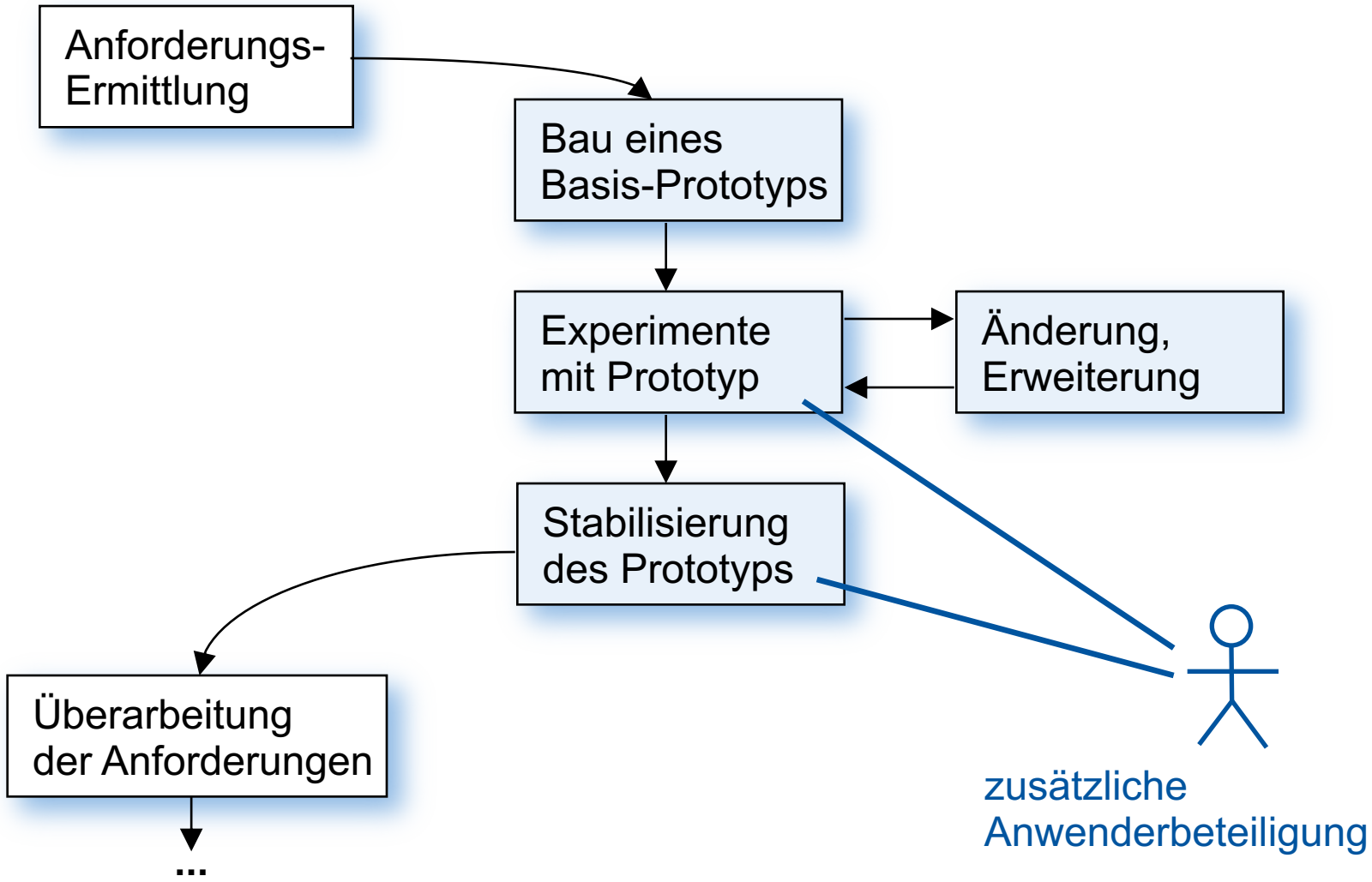
- Prototyp in der Software-Entwicklung:
  - Problemlos in Serie fertigbar
  - Erstes Exemplar, Modell für ein neues Produkt
  - Nicht voll funktionsfähig
  - kurze Entwicklungszeit, preisgünstig
  - “Rapid Prototyping”
  - Zwecke:
    - Frühe Einbindung von AnwenderInnen in die Entwicklung
    - Rechtzeitige Abklärung der Realisierbarkeit und von Risiken

Deshalb: Einsatz bereits in der Analysephase sinnvoll

Klassifikation von Prototyping

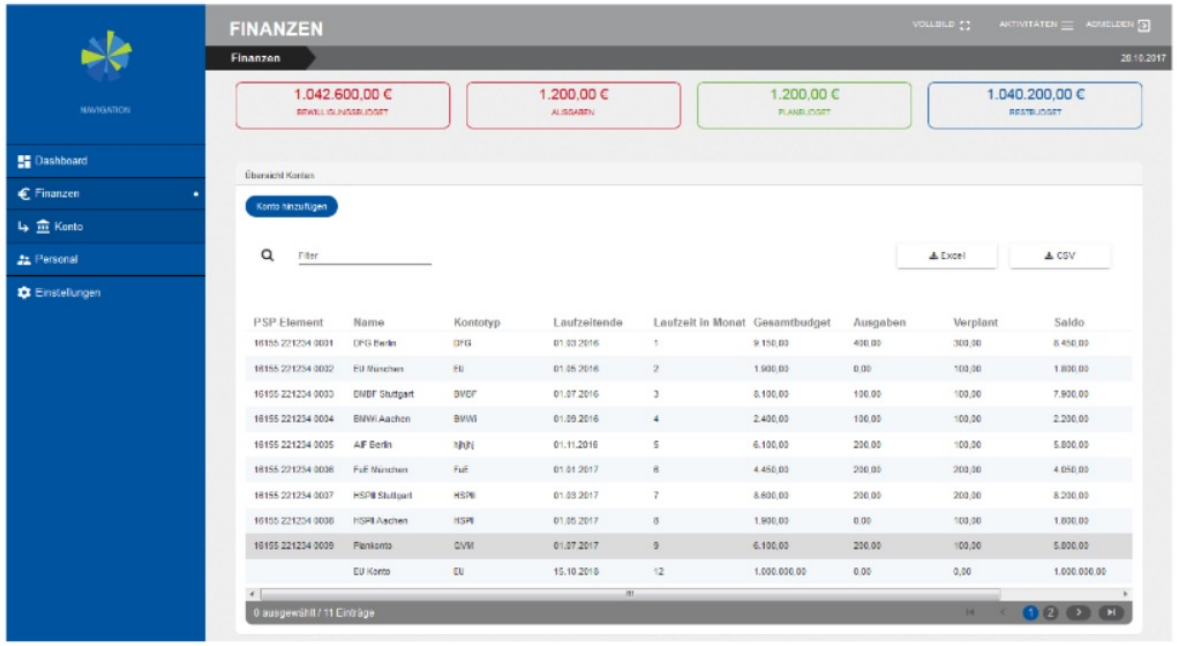
	Weiterverwendung des Prototyps	Phase im Phasenmodell (vorwiegend)	Zielgruppe
explorativ	wegwerfen	Analyse	Anwender, Systemanalytiker
experimentell	wegwerfen	(Analyse,) Entwurf, Implementierung	Entwickler
evolutionär	ausbauen (inkrementelle Entwicklung)	(sinnvoll nur bei evolutionärer Entwicklung)	Entwickler, Anwender

Vorgehensweise beim Explorativen Prototyping



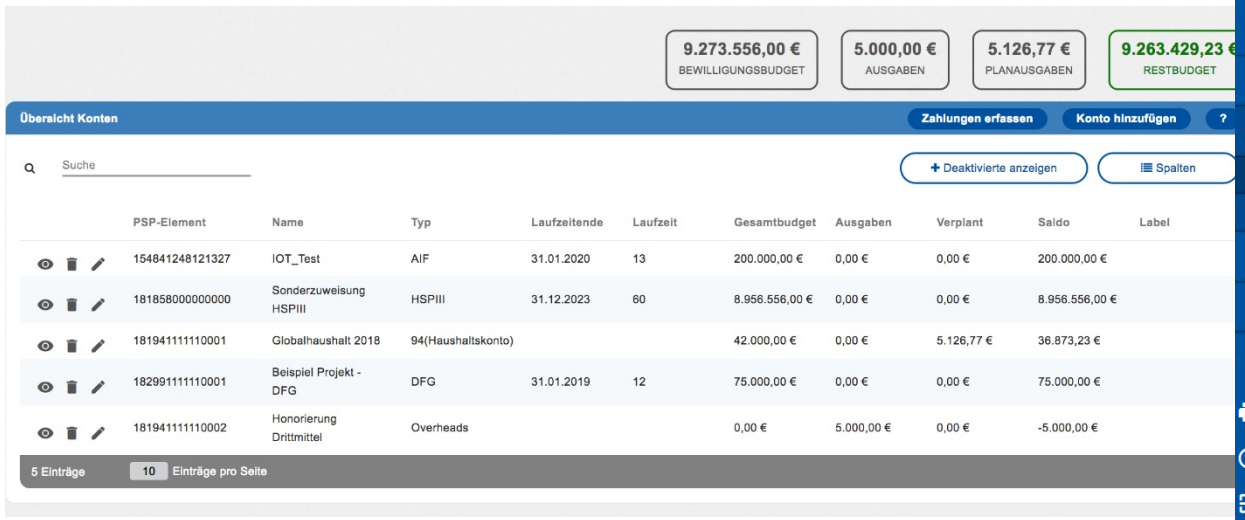
Beispiel Evolutionäres Prototyping

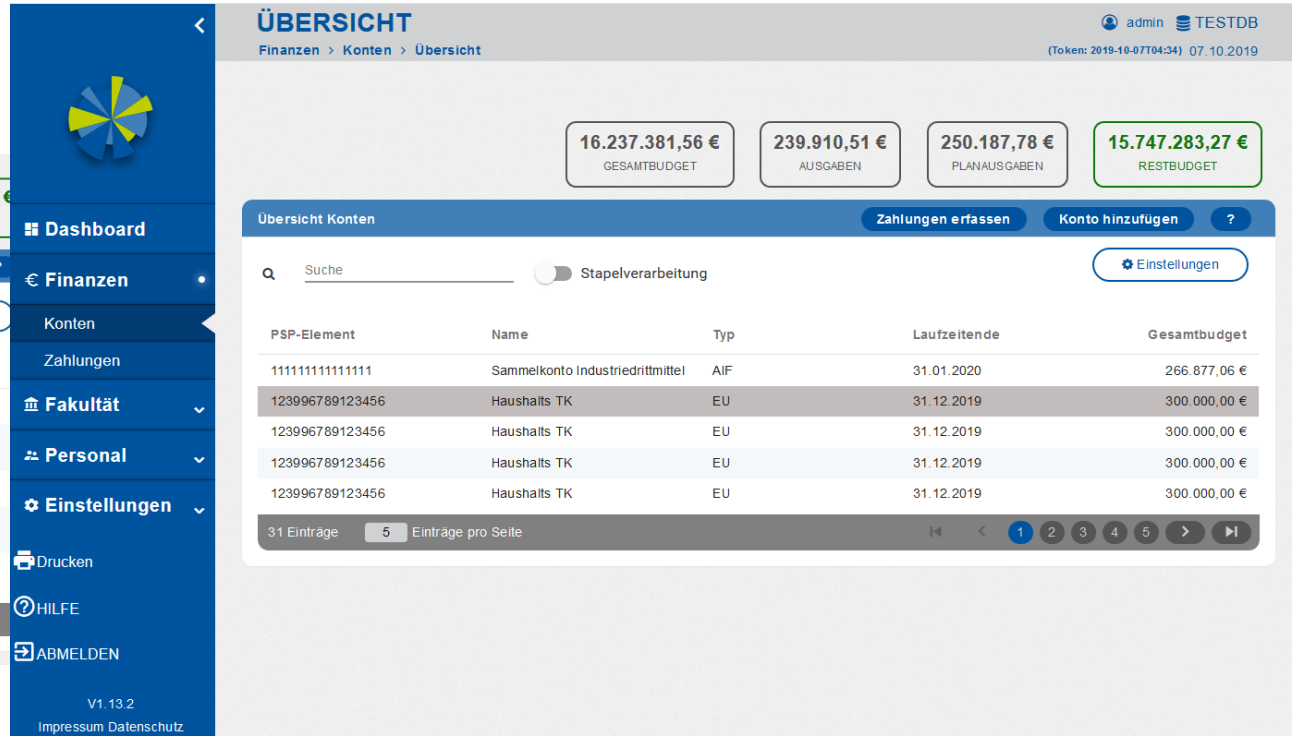
- Web-Plattform im MaCoCo (Management Cockpit for Controlling) Projekt
  - Prototyp: Stand Ende 2017
  - Agile, modell-basierte und generative Entwicklungsmethoden: Inkrementelle Artefakte



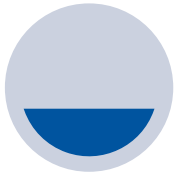
Beispiel Evolutionäres Prototyping

- Web-Plattform im MaCoCo (Management Cockpit for Controlling) Projekt
  - Version 1.1.2 (Januar 2019)
  - Version 1.13.2 (Oktober 2019)





Was haben wir gelernt?

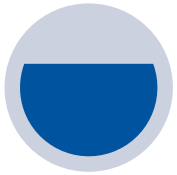


Prototypen

Erstes Exemplar bzw. Modell für ein neues Produkt

Nicht voll funktionsfähig

Kurze Entwicklungszeit, preisgünstig



Arten

Explorativ

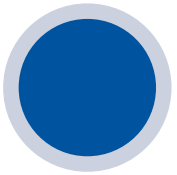
- Für Analyse Phase
- Danach wegwerfen
- Zielgruppe: Anwender und Systemanalytiker

Experimentell

- (Analyse,) Entwurf, Implementierung
- wegwerfen

Evolutionär

- weiterentwickeln



Zweck

Frühe Einbindung von AnwenderInnen in die Entwicklung

Rechtzeitige Abklärung der Realisierbarkeit und von Risiken

Vorlesung Softwaretechnik

4. Systemanalyse und -modellierung

Prof. Bernhard Rumpe
Software Engineering
RWTH Aachen

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

SE Software Engineering | RWTH AACHEN UNIVERSITY

Warum, was, wie und wozu? Systemanalyse und -modellierung

Warum?

- Modellierungssprachen häufig in der Industrie angewandt
- Basiswissen für andere Vorlesungen

Was?

Wie man ein System mit Hilfe einer Reihe von Modellen zur Beschreibung von Struktur und Verhalten entwickeln kann

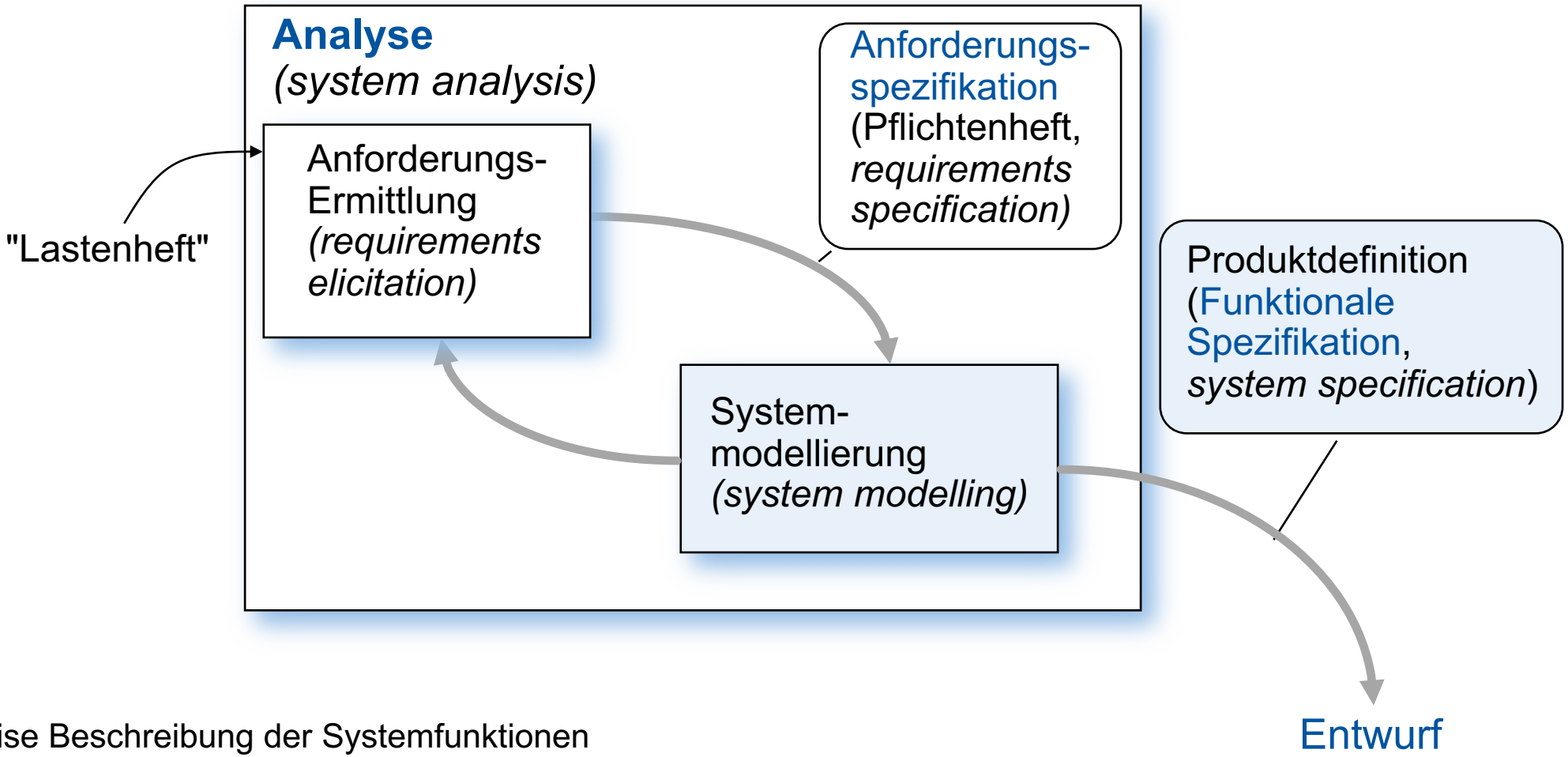
Wie?

- Klassendiagramme
- Objektorientierte Analyse
- Objektdiagramme
- Sequenzdiagramme
- Statecharts

Wozu?

- Verstehen des Aufgabengebiets
- Strukturierung der Lösung
- Schaffung einheitlicher Terminologie
- Auffinden von Grundkonzepten

Systemmodellierung



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

Systemmodellierung

- Abstrakte Modellierung der zu lösenden fachlichen Aufgabe (fachliches Modell, domain model)
- Kleines Team

- Aufgabe:
  - Strukturierung des Aufgabengebiets
  - Schaffung einheitlicher Terminologie
  - Auffinden von Grundkonzepten

- Grundregeln:
  - Zusammenhang mit Anforderungsspezifikation sichern
  - Implementierungsaspekte ausklammern
    - Annahme perfekter Technologie
  - Funktionale Essenz des Systems
    - Datenhaltung, Benutzungsoberfläche im allgemeinen zurückstellen

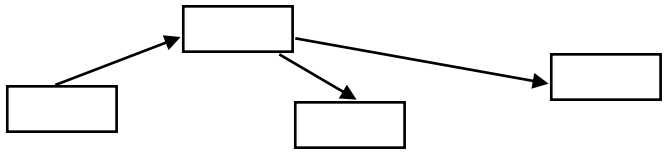
Objektorientierte Analyse (OOA)

- Grundidee: Modellierung der fachlichen Aufgabe durch kooperierende Objekte.

Statisches Modell	Dynamisches Modell
Klassen: Eigenschaften und Aufgaben von Objekten	Nachweis der Bewältigung typischer *Anwendungsfälle*
Beziehungen zwischen Klassen (bzw. Objekten)	*Zustände* und *Verhalten* von Objekten

OOA-Modell

beschreibt



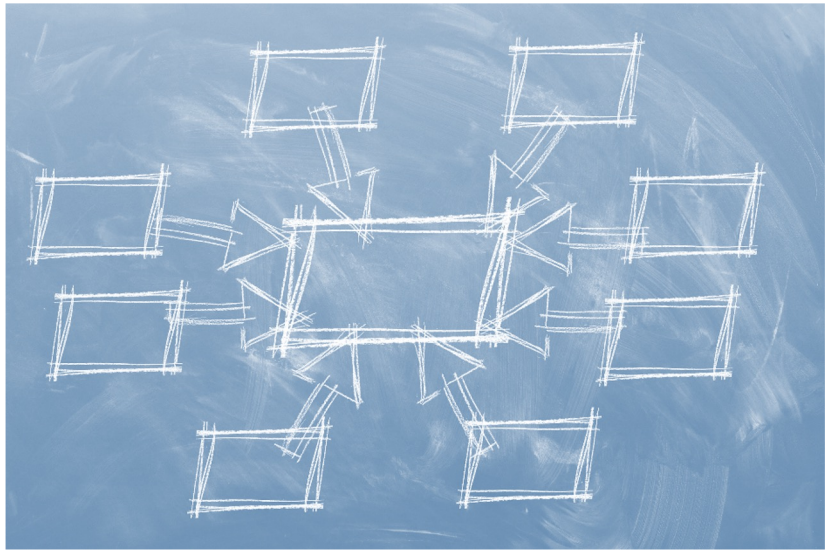
"Gesellschaft" kooperierender Objekte

Modellierung mit UML: Strukturen

- Klassendiagramm
  - Statische Struktur
  - Typen, Vererbung
  - Innerer Aufbau von Klassen
  - mögliche Beziehungen zwischen ihren Objekten

- Objektdiagramm
  - Snapshot einer möglichen Situation
  - Exemplarisch: konkrete Objekte, konkrete Werte

- Später noch (für die Architektur):
  - Composition Structure Diagramm, Deployment Diagramm, …



Modellierung mit UML: Verhalten

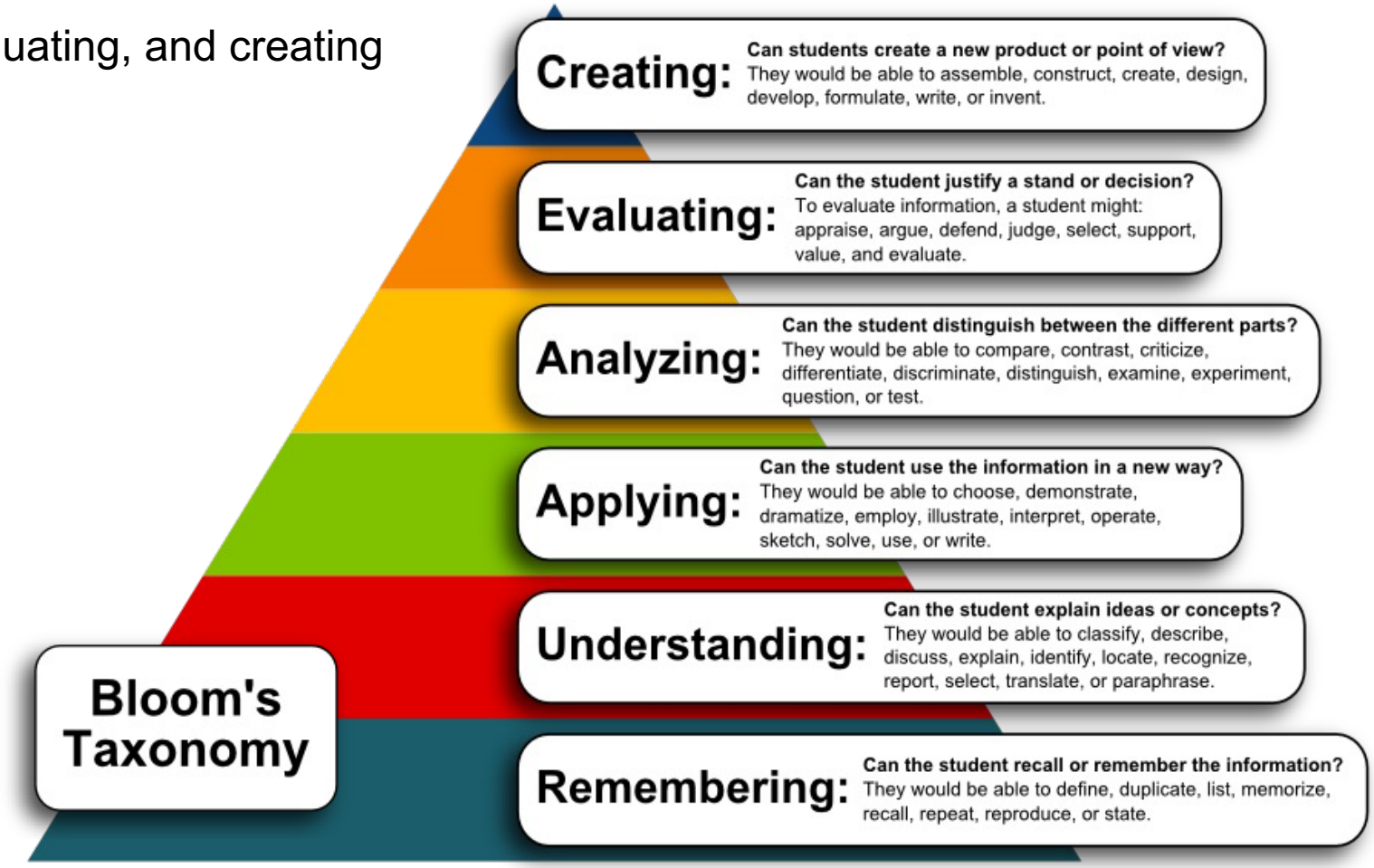
- Sequenzdiagramm
  - Interaktion zwischen Objekten
  - Exemplarische Abläufe, Methodenaufrufe

- Statechart
  - Zustandsautomat + Hierarchie + …
  - Beschreibt mögliche Abläufe und ihre Verbindung mit dem Zustandsraum eines Objekts

- Wir kennen schon:
  - Aktivitätsdiagramm
  - Use case

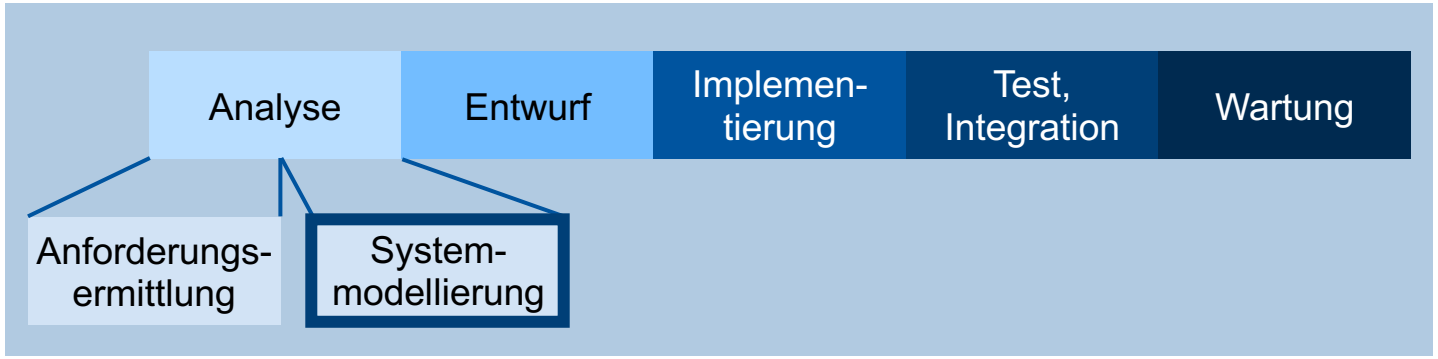
Learning Objectives:

- Understanding, applying, analyzing, evaluating, and creating
  - Models
  - Requirements modeling
  - Data modeling
  - Structure and behavior modeling
- Syntax and semantics of selected modeling languages (mainly UML)



Softwaretechnik

4. Systemanalyse und -modellierung
4.1. Klassendiagramme mit UML/P



Prof. Bernhard Rumpe
Software Engineering
RWTH Aachen

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

Literatur:

- B. Rumpe: Modellierung mit UML, 2011

Vertiefung:

- Vorlesung Modellbasierte Softwareentwicklung
- Software Language Engineering

SE Software Engineering | RWTH AACHEN UNIVERSITY

Warum, was, wie und wozu? Klassendiagramme

Warum?

- Intensiv in der Praxis verwendet
- In der Analyse wichtig um Konzepte und Verbindungen zwischen Konzepten zu verstehen

Was?

- Lesen / Verstehen von Klassendiagrammen
- Erstellung von hochwertigen Klassendiagrammen

Wie?

- Betrachtung der Modellierungssprache & Beispiel
- Klasse mit Attributen und Methoden
- Vererbung, Schnittstellen, Enumerations
- Assoziation, Komposition, Qualifizierte Assoziation

Wozu?

- Gemeinsames Verständnis für Terminologie
- Strukturierung des Aufgabengebiets
- Umbau und/oder Verfeinerung im Design

Grundkonzepte der Objektorientierung

- Ein System besteht aus variabel vielen Objekten. Die Anzahl und Struktur ändert sich dynamisch.

- Ein Objekt hat ein definiertes Verhalten.
  - Menge genau definierter Operationen
  - Operation wird beim Empfang einer Nachricht ausgeführt.

- Ein Objekt hat einen inneren Zustand.
  - Zustand des Objekts ist Privatsache (Kapselungsprinzip).
  - Resultat einer Operation hängt vom aktuellen Zustand ab.

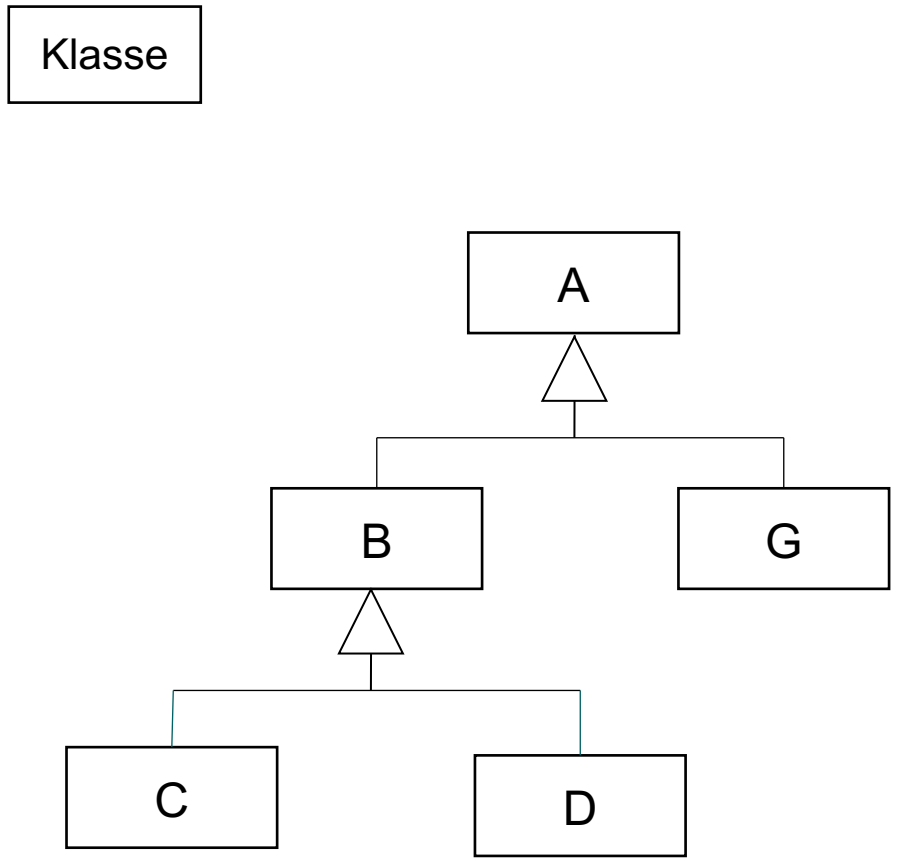
- Ein Objekt hat eine eindeutige Identität.
  - Identität ist fest und unabhängig von anderen Eigenschaften.
  - Es können mehrere verschiedene Objekte mit identischem Verhalten und identischem inneren Zustand im gleichen System existieren.

Konzepte der Objektorientierung

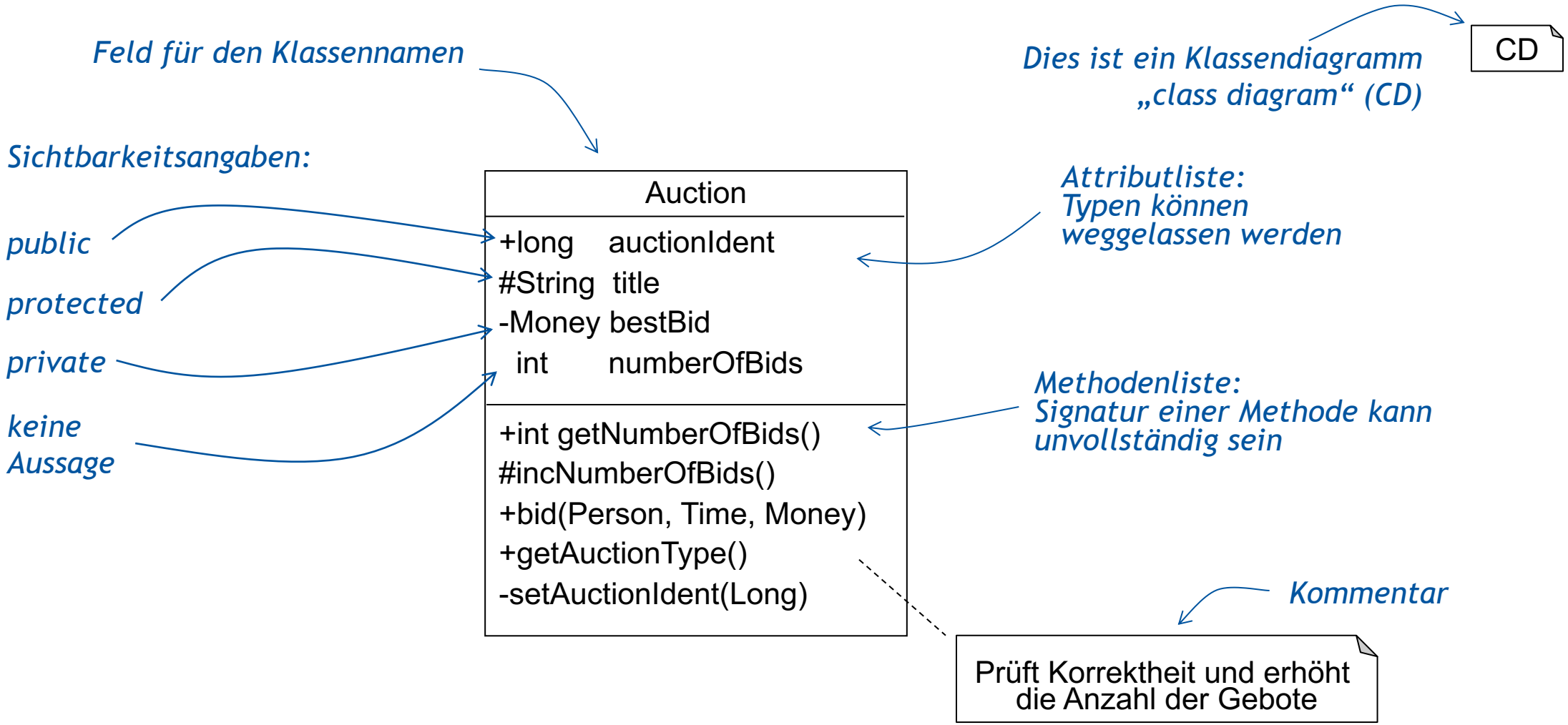
- Ein Objekt gehört zu einer Klasse.
  - Die Klasse schreibt das Verhaltensschema und die innere Struktur ihrer Objekte vor.

- Klassen besitzen einen ‘Stammbaum’, in der Verhaltensschema und innere Struktur durch Vererbung weitergegeben werden.
  - Vererbung bedeutet Generalisierung einer Klasse zu einer Oberklasse.

- Polymorphie**:** Eine Nachricht kann verschiedene Reaktionen auslösen, je nachdem zu welcher Unterklasse einer Oberklasse das empfangende Objekt gehört.
  - Polymorphe Methode: ist in verschiedenen Klassen unterschiedlich implementiert

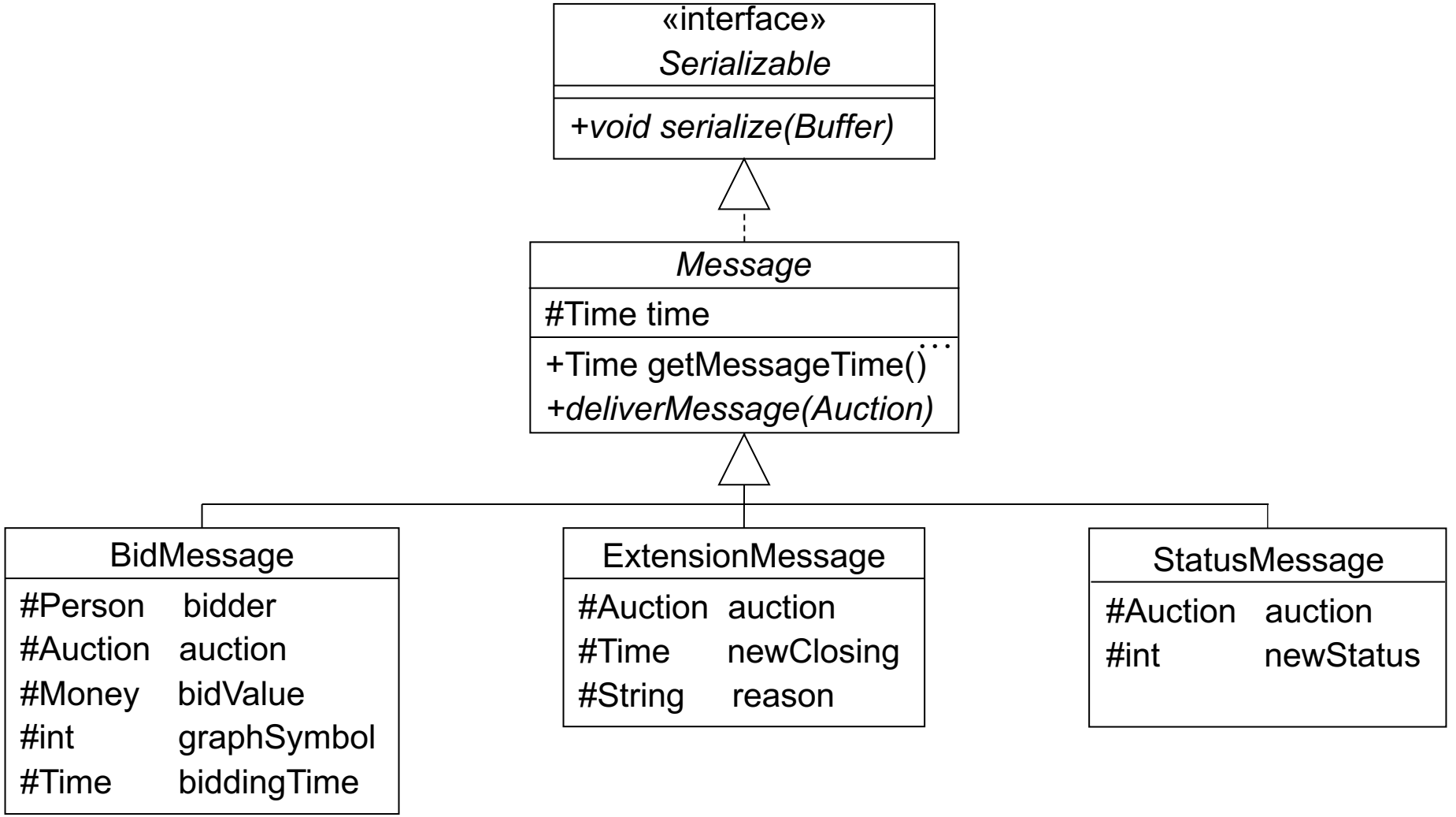


Klasse mit Attributen und Methoden



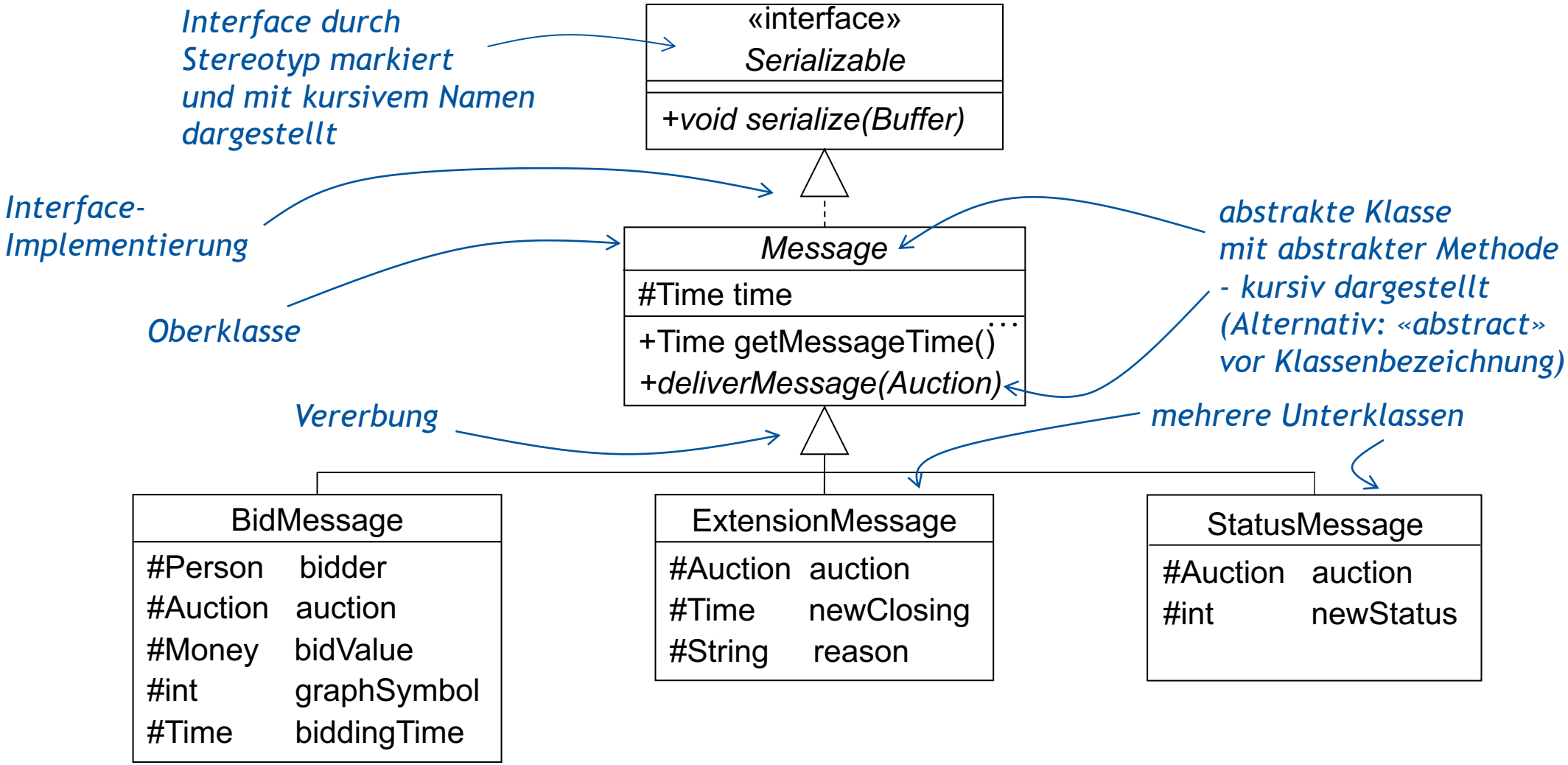
Vererbung, Schnittstellen-Implementierung

⇨ analyze



CD

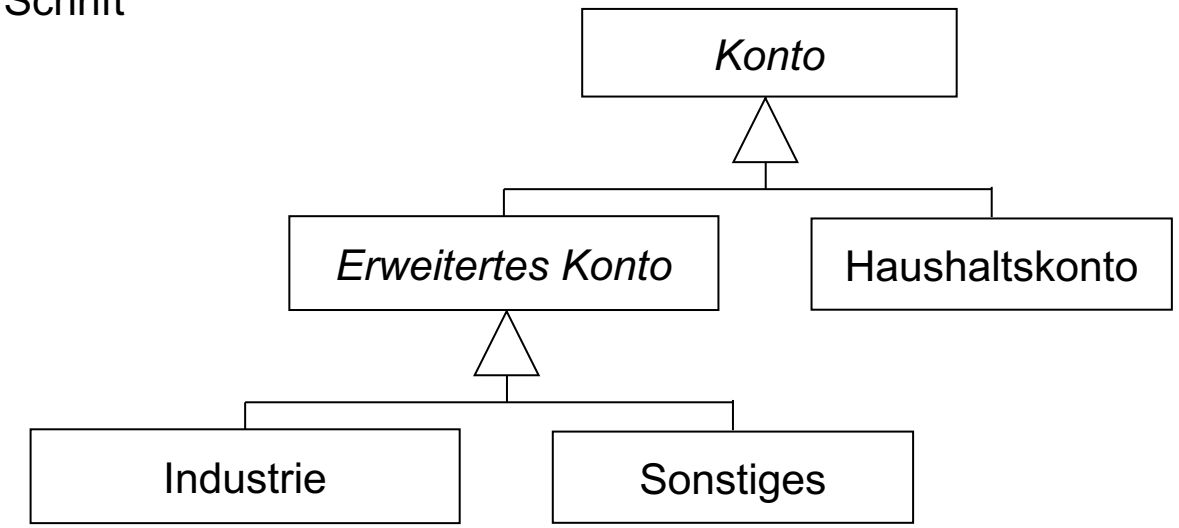
Vererbung, Schnittstellen-Implementierung



Abstrakte Klassen

- Klasse kann nicht instanziiert werden
- primär in Generalisierungshierarchien sinnvoll
- Dient zum "Herausheben" gemeinsamer Merkmale der Unterklassen

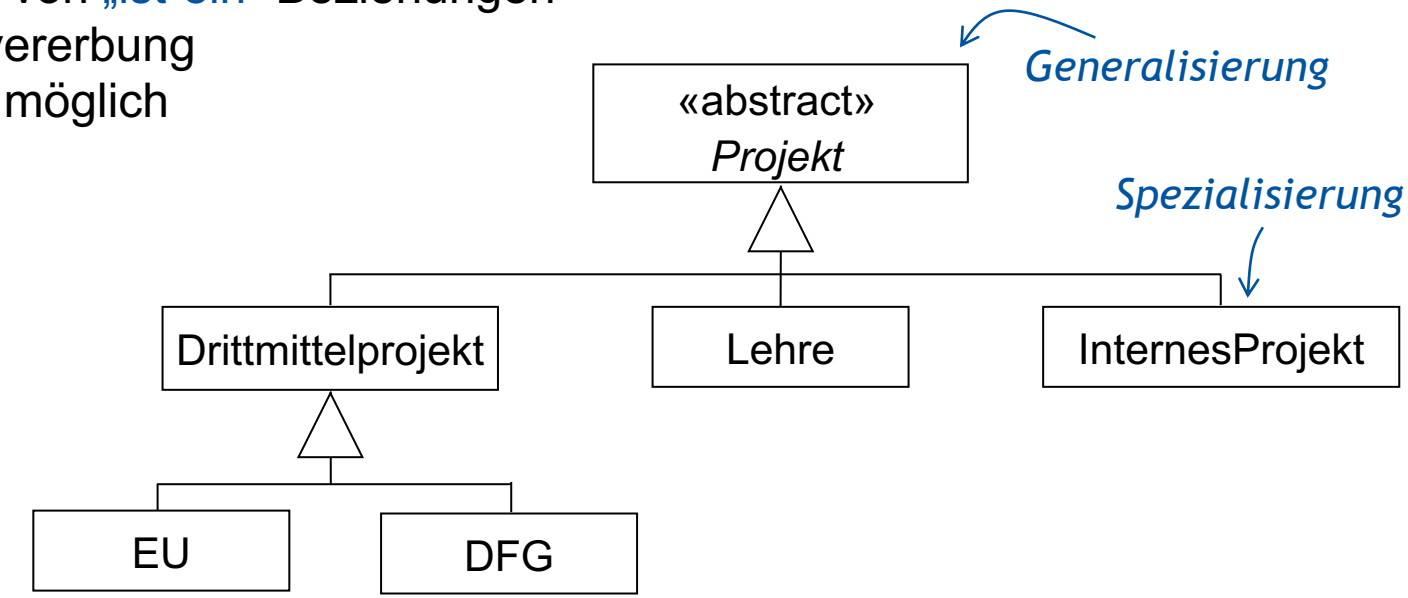
- Notation:
  - Schlüsselwort «abstract» oder Klassenname in kursiver Schrift

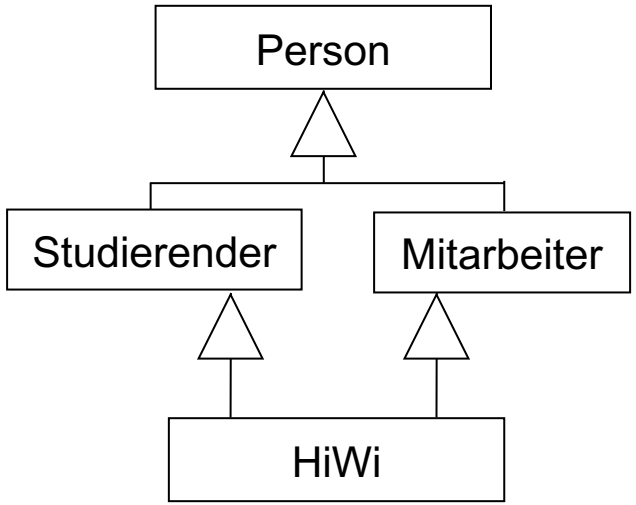


Vererbung (Generalisierung/Spezialisierung)

- Beziehung zwischen einer spezialisierten Klasse und einer allgemeineren Klasse
  - Die spezialisierte Klasse erbt die Eigenschaften der allgemeineren Klasse
  - Kann weitere Eigenschaften (in Form von Attributen und Methoden) hinzufügen
  - Eine Instanz der Unterklasse kann überall dort verwendet werden, wo eine Instanz der Oberklasse erlaubt ist (zumindest laut Signatur)

- Hierarchie von „ist-ein“ Beziehungen
- Mehrfachvererbung im Prinzip möglich

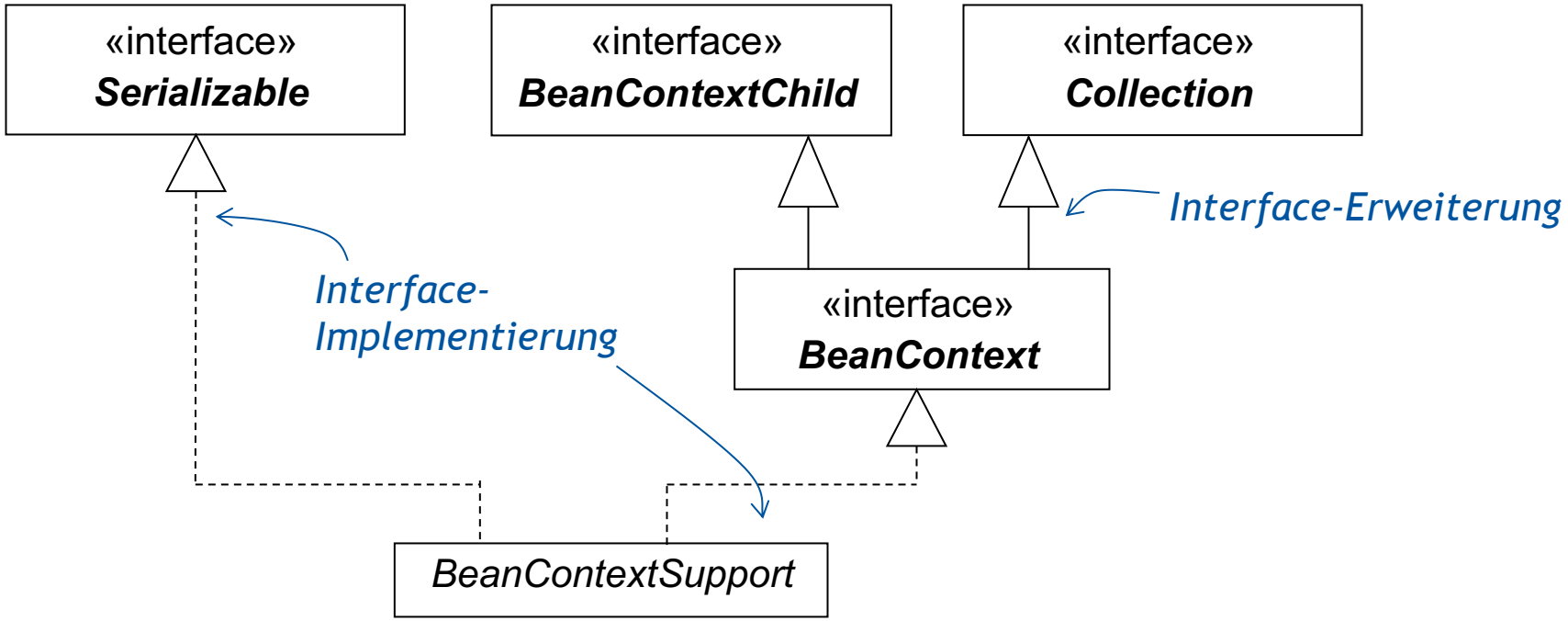




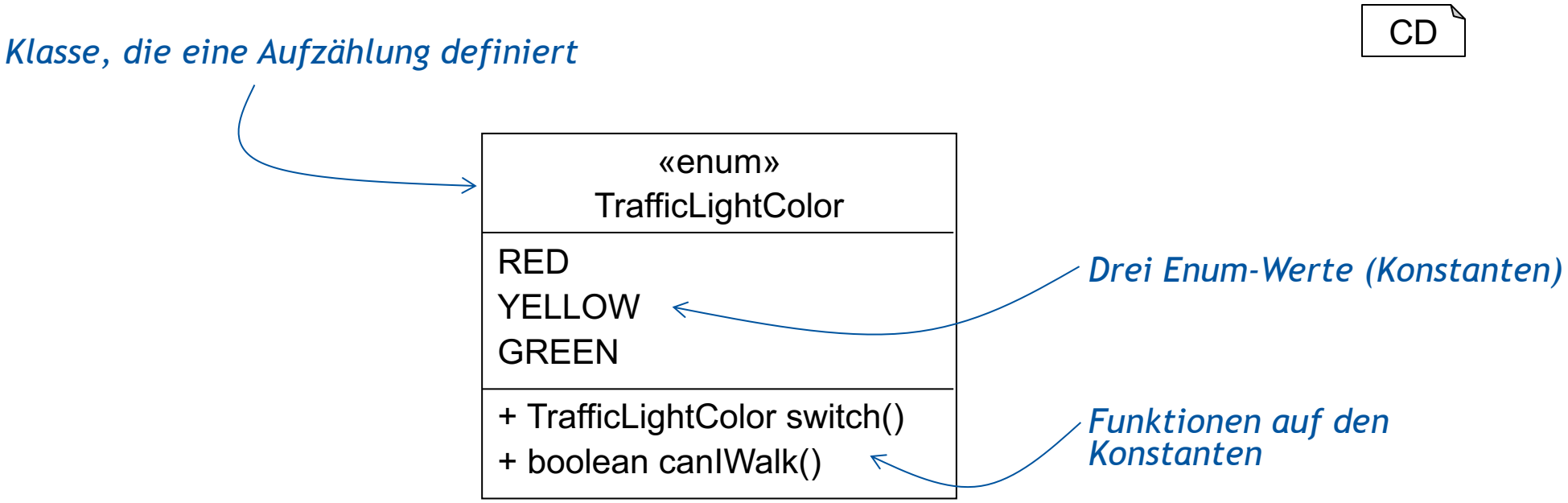
Interface Implementierung

- Ein Interface kann viele andere Interfaces erweitern
- Eine Klasse kann viele Interfaces implementieren
- UML: Eine Klasse kann von vielen Klassen erben
  - Java: von nur einer Klasse erben *(extends);* C++: Mehrfachvererbung möglich

CD



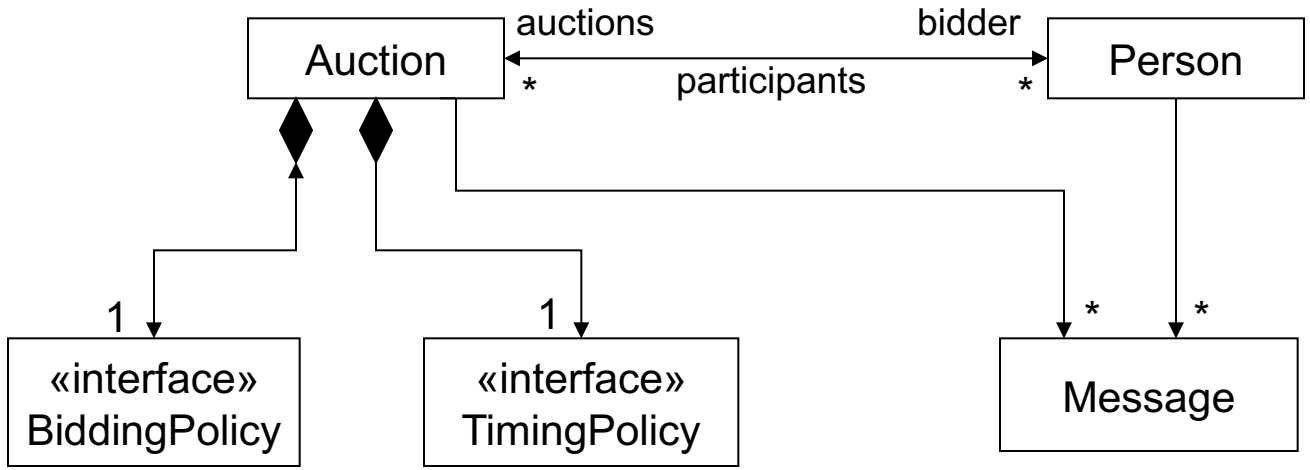
Enumeration-Klasse



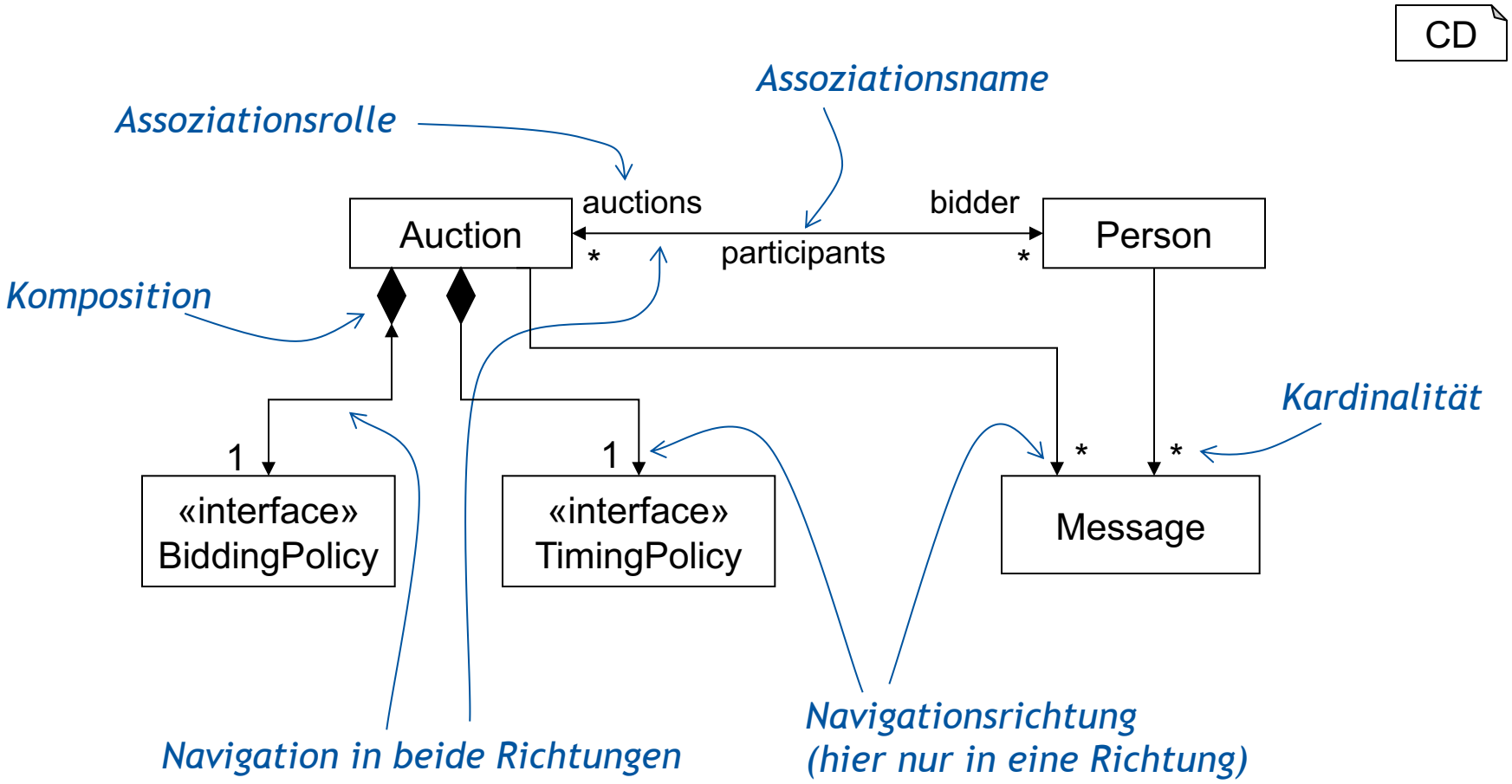
Assoziationen

⇨ analyze

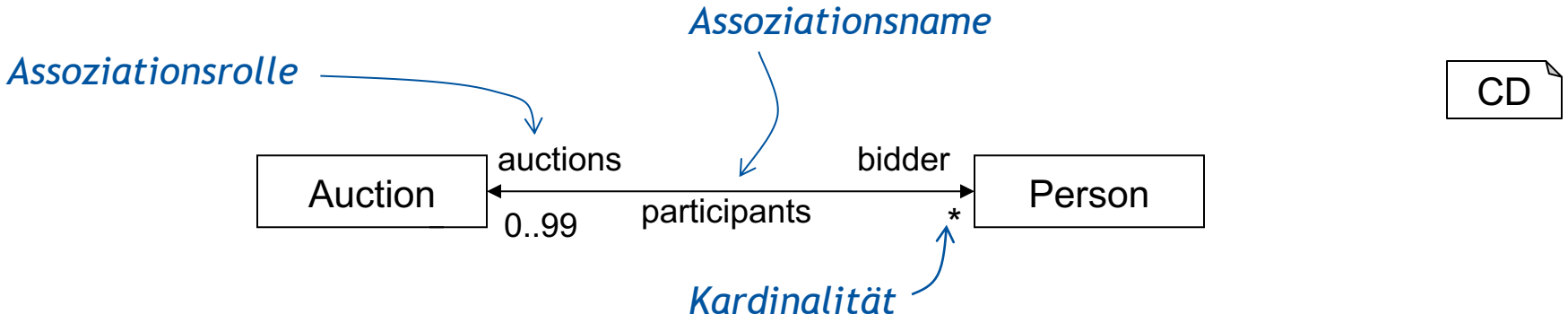
CD



Assoziationen

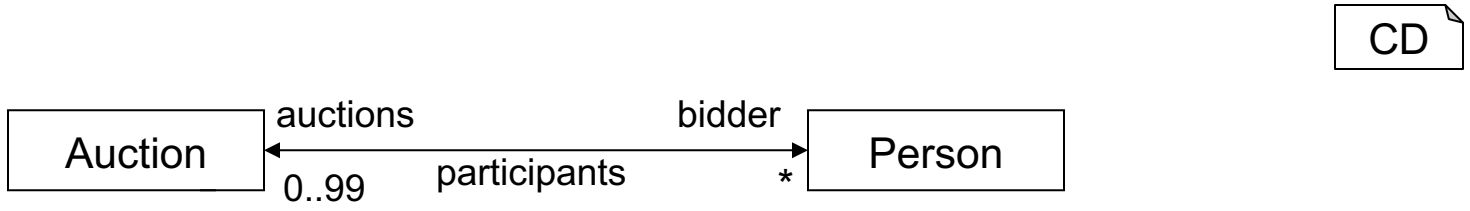


Assoziationen

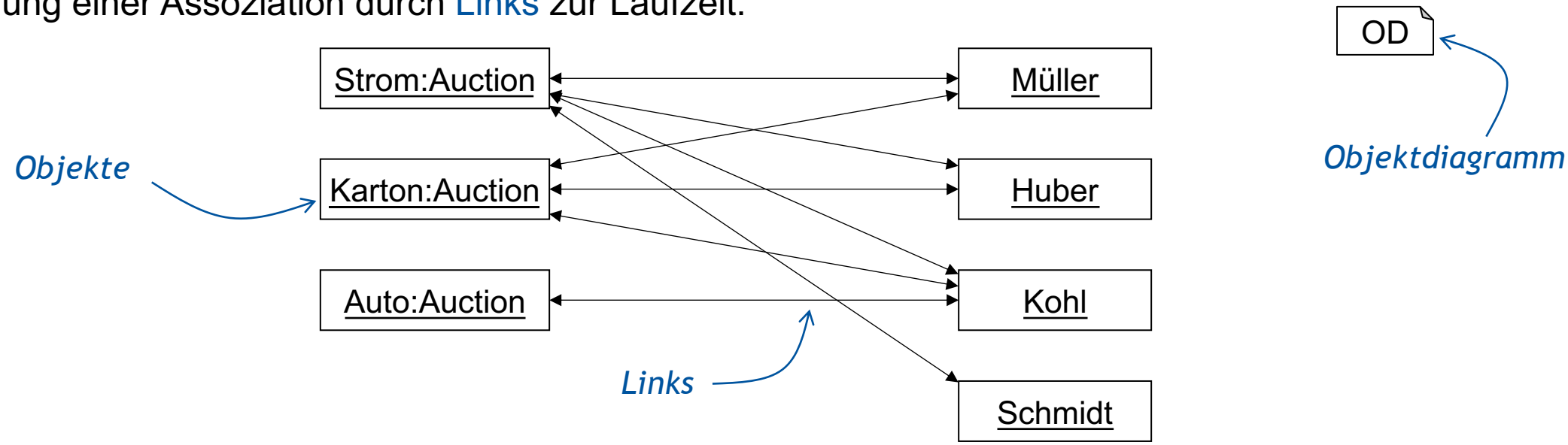


- Assoziation als binäre Beziehung zwischen Klassen
  - „Person participates in Auction“
  - Aus Sicht der Person: Navigation zu den Auktionen, deshalb
    - Assoziationsrolle „auctions“ links
    - Eine Person kann an bis zu 99 Auktionen teilnehmen (0..99)
- Kardinalitäten:
  - genau-eines: 1
  - optional: 0..1
  - beliebig: *
  - nicht-null: 1..* (oder +)
  - feste Intervalle: 3..9, 41 (aber nur selten verwendet)

Assoziationen und Links



- Ausprägung einer Assoziation durch Links zur Laufzeit:

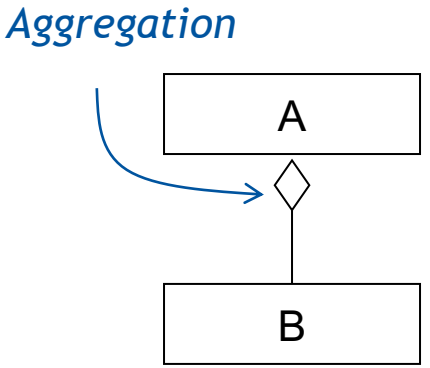


Aggregation

- Aggregation = spezielle Form der Assoziation

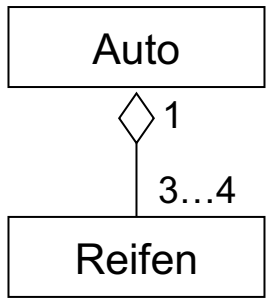
- Bedeutung
  - Schwache Zusammengehörigkeit,
  - Lesbar als „A hat ein B“
  - Jedes der Teile hat einen eigenen Lebenszyklus
  - Die Teile sind austauschbar

- Probleme
  - Schwer unterscheidbar von „normaler“ Assoziation
  - In Programmcode gleich umgesetzt wie Assoziation
  - Eher selten verwendet
  - Daher: in Vorlesung nicht weiter behandelt



A „hat“ B

Beispiel

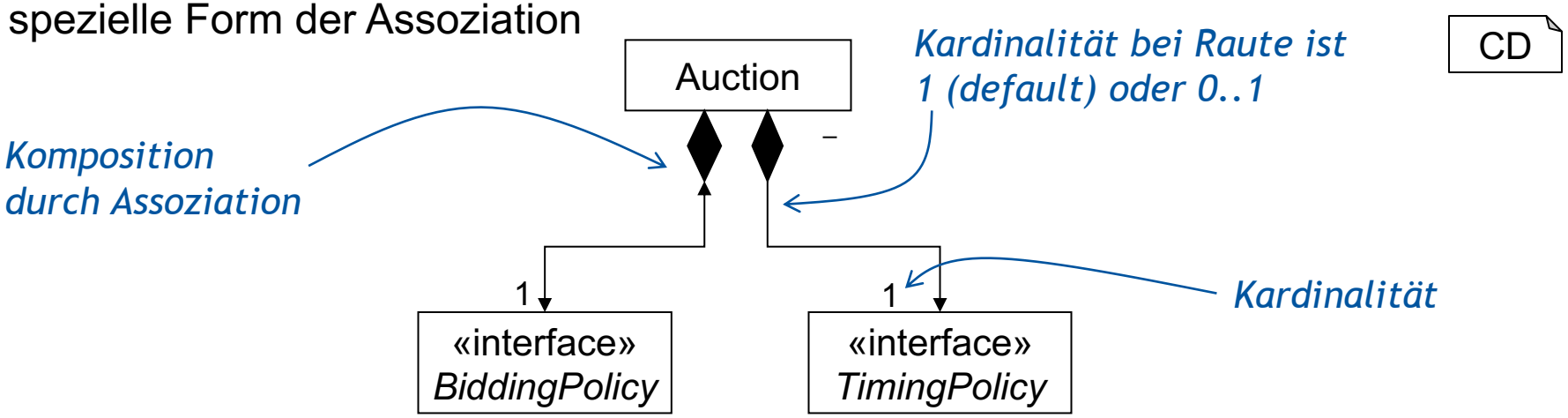


CD

Auto „hat“ 3 bis 4 Reifen

Komposition

- Komposition = spezielle Form der Assoziation

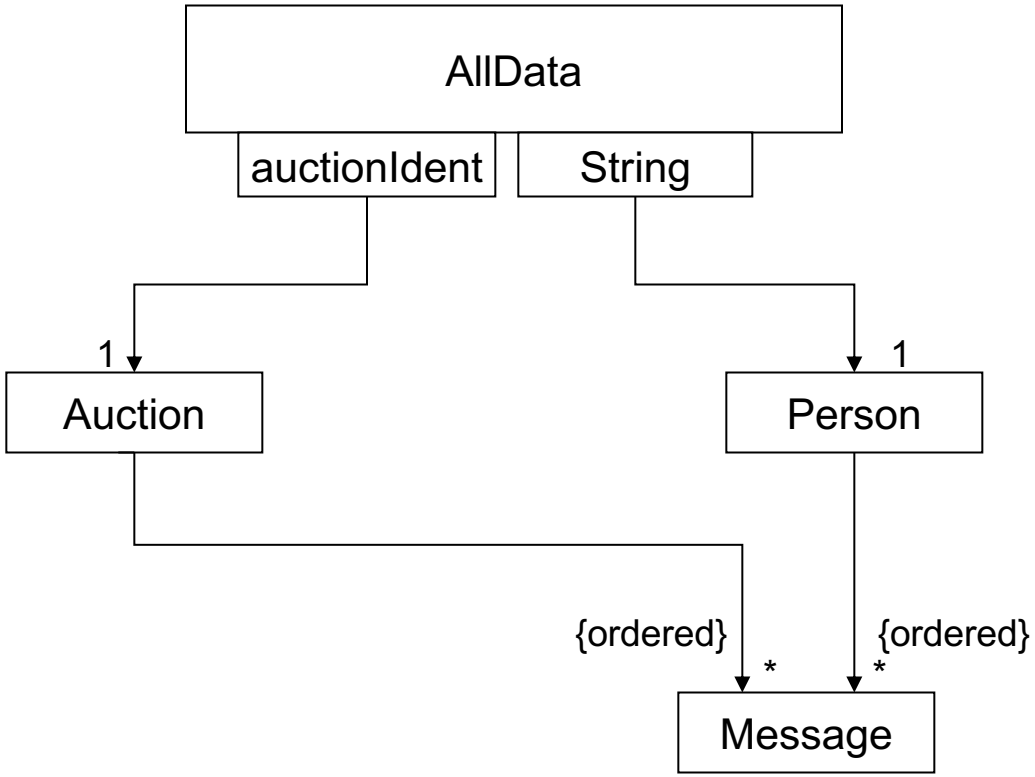


- Bedeutung:
  - Kompositum aus Teilen zusammengesetzt
  - Objekte bilden eine zusammengehörende Einheit
  - Teile vom Kompositum abhängig
    - Lebenszyklus kombiniert
    - Austauschbarkeit nicht gegeben
  - Allerdings: Interpretationsunterschiede bei Werkzeugen
    - Daher: Projektspezifisch ggf. präzisieren!

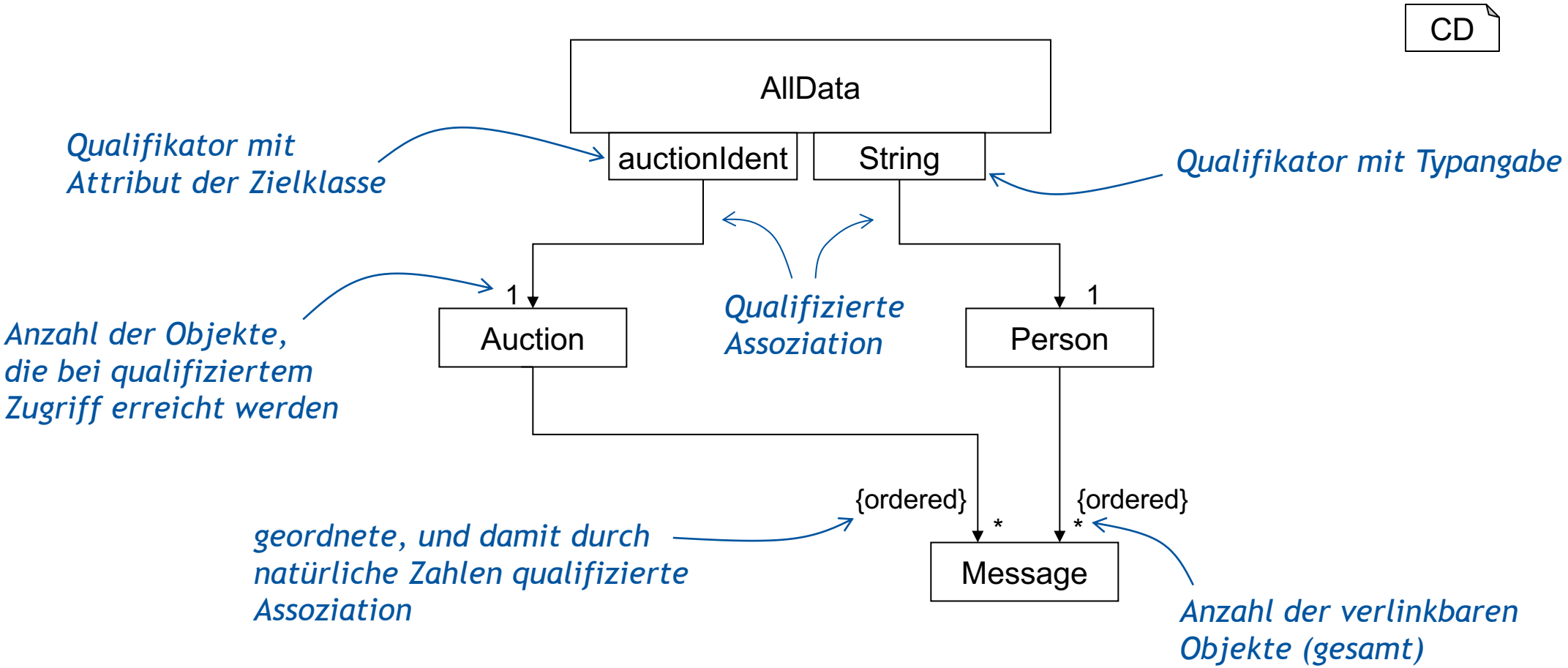
Qualifizierte Assoziation

⇨ analyze

CD

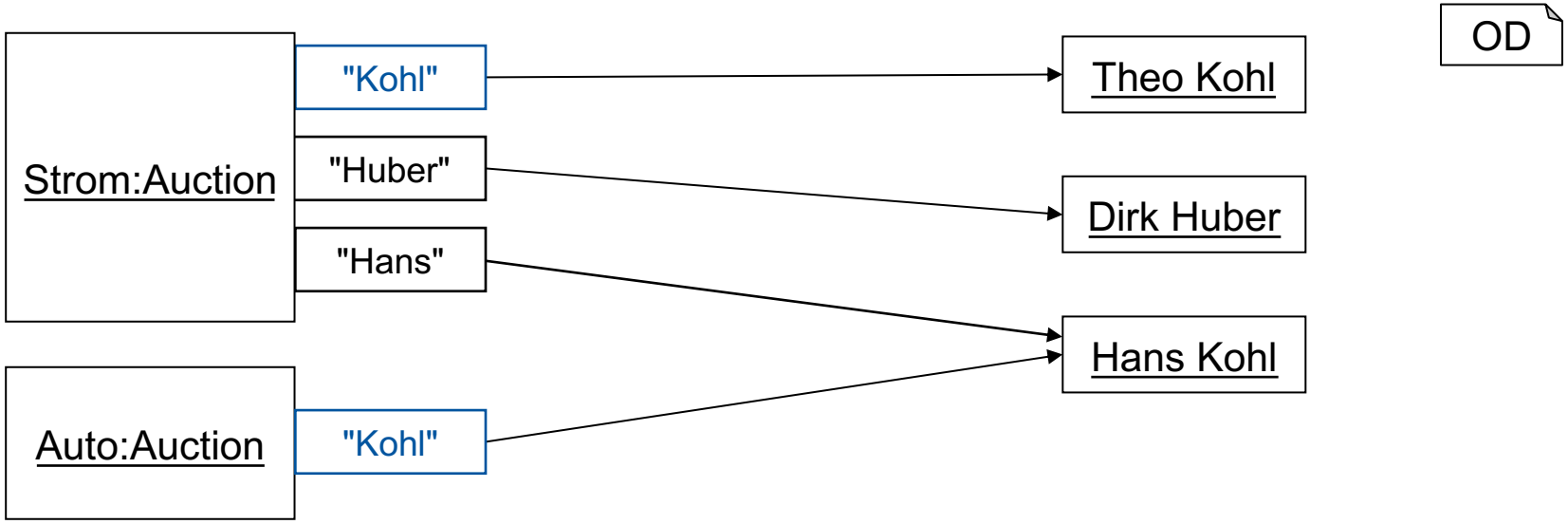


Qualifizierte Assoziation



Qualifizierte Assoziation und Links

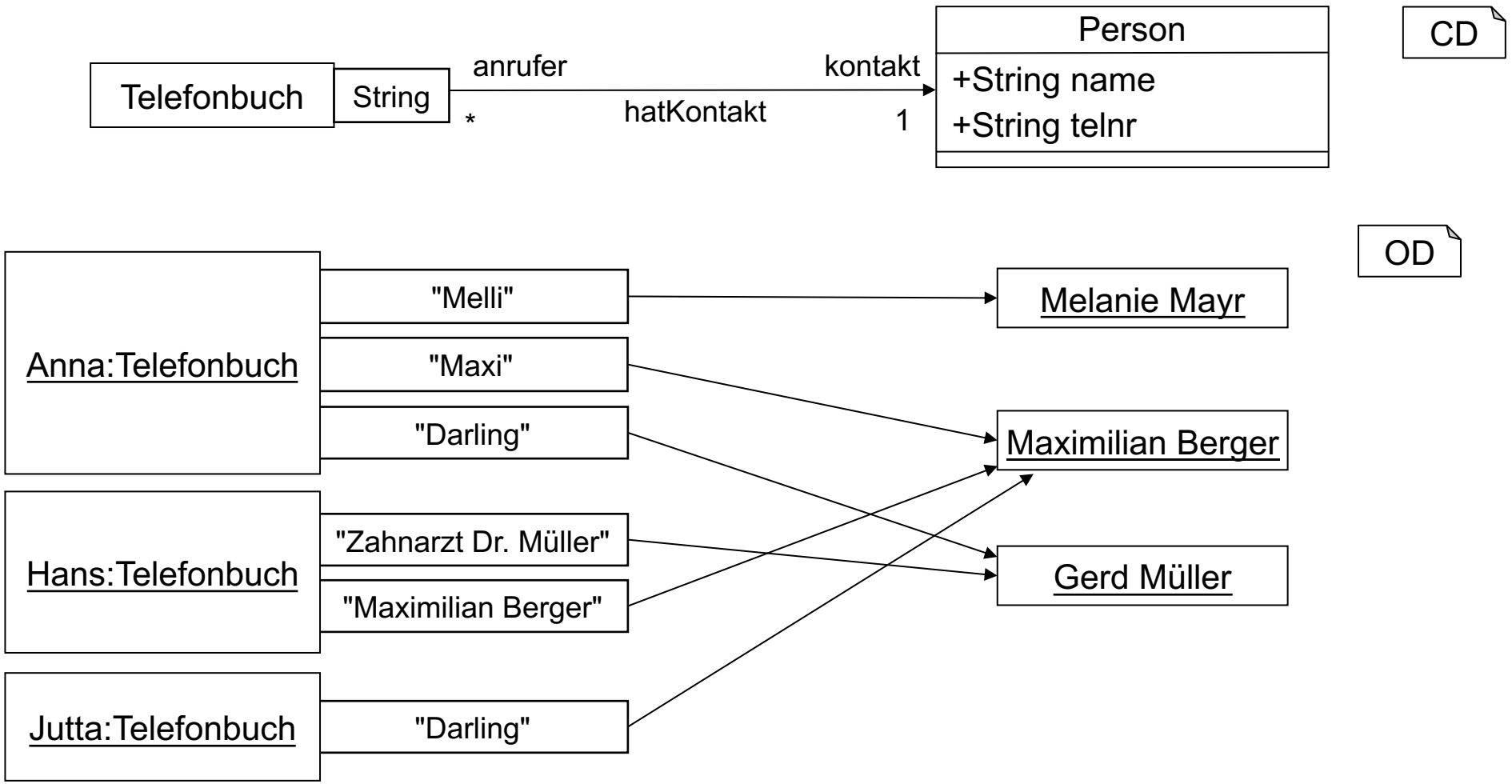
- Qualifikator „login“ erlaubt einzelne Objekte zu selektieren
- Dasselbe Objekt kann in unterschiedlichen Auktionen verschieden qualifiziert sein (unterschiedliche Logins)



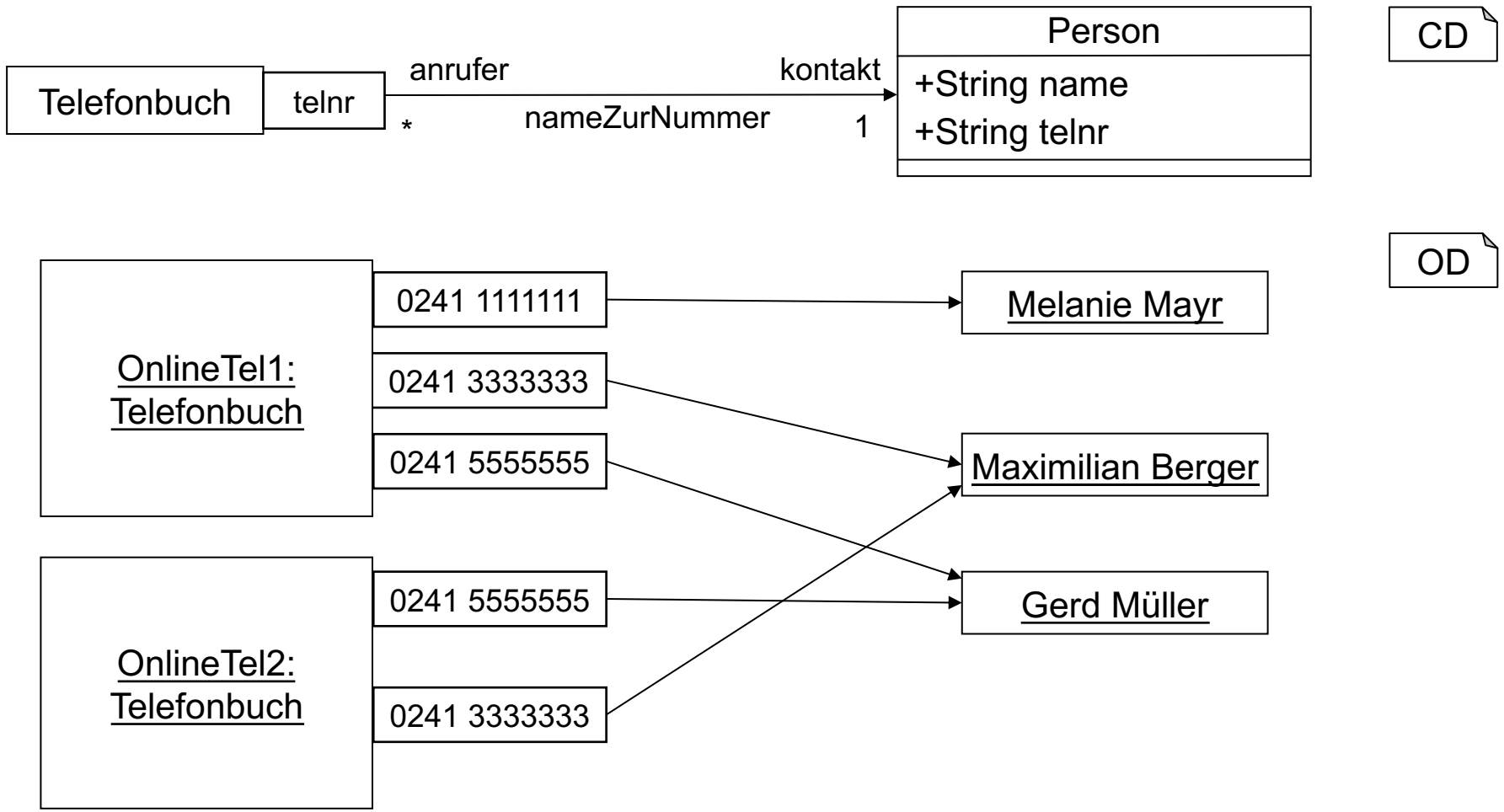
Qualifizierte Assoziation

- Qualifizierte Assoziationen erlauben Selektion einzelner Objekte aus einer Menge mit Qualifikator
- Qualifikatoren können sein:
  - Zahlen-Intervall (0-..), das Reihenfolge anzeigt ({ordered})
  - Expliziter Identifikator (Attribut) des Zielobjekts (auctionIdent)
  - Frei wählbarer Wert eines vorgebenen Typs
- Auch Komposition kann qualifiziert sein
- Qualifikation an beiden Enden möglich (aber selten)
- Qualifizierte Assoziation benötigt erweiterte Mechanismen zur Bearbeitung
  - Selektiver Zugriff, selektive Änderung

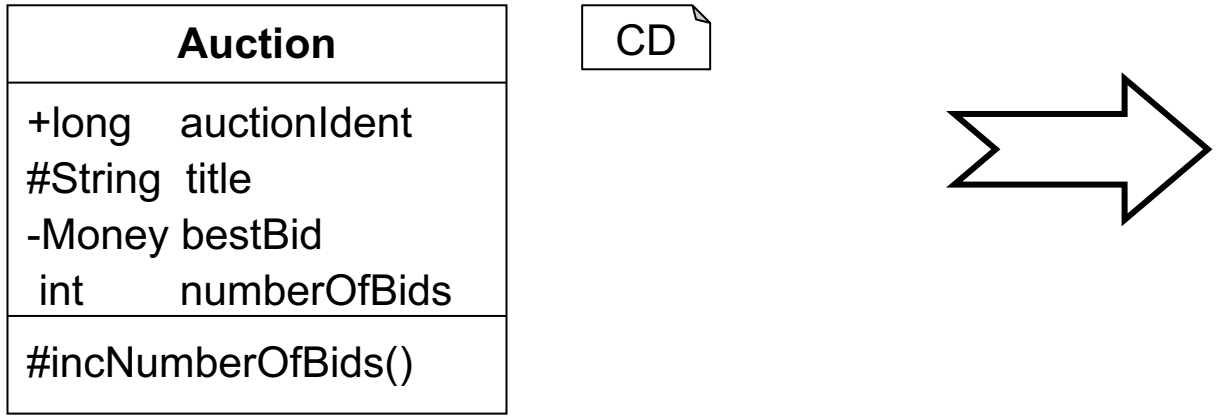
Beispiel: Qualifizierte Assoziation und Links mit Typ als Qualifier



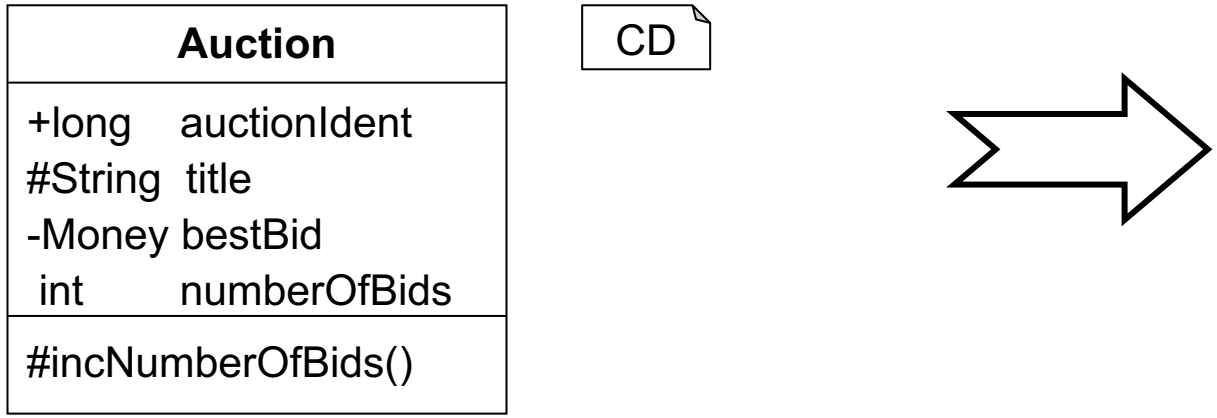
Beispiel: Qualifizierte Assoziation und Links mit Attribut als Qualifier



Code aus einer Klasse synthetisieren?



Code aus einer Klasse synthetisieren?




```java
class Auction {
  public long       auctionIdent;
  protected String  title;
  private Money     bestBid;
  public int        numberOfBids;

  protected void incNumberOfBids() { ... }
}
```
Java

```cpp
class Auction {
  public:
    long          auctionIdent;
    int           numberOfBids;
  protected:
    std::string   title;
  private:
    Money         bestBid;
  protected:
    void incNumberOfBids() { ... }
};
```
C++

Alternative Code-Synthesen?

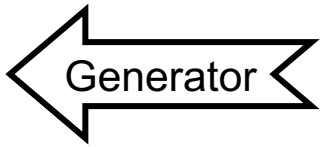
- Mögliche zusätzliche Anforderungen:
  - get/set-Methoden für Attribute
  - Serialisierbarkeit des Objekts
  - Speichern von Objekten in einer Datenbank-Tabelle
    - evtl. sogar die Erzeugung der Tabelle als SQL-Statement
  - Attributzugriff wird durch Security-Manager gesichert
  - Plattformabhängigkeit des Codes

- Unterschiedliche Anforderungen führen zu unterschiedlichen Realisierungen:
  - Das Modell als abstrakte (!) und kompakte Darstellung
  - Technologieunabhängig!

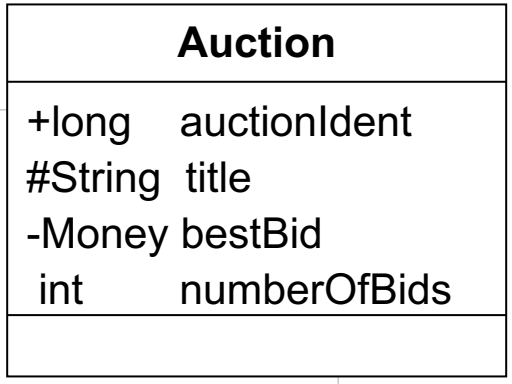
Beispiel: Code-Generierung mit get/set-Methoden

JAVA

```
1  class Auction {
2    private long   _auctionIdent;
3    private String _title;
4    private Money  _bestBid;
5    private int    _numberOfBids;
6
7    synchronized public long getAuctionIdent() { return _auctionIdent; }
8    synchronized protected String getTitle()   { return _title; }
9    synchronized private Money getBestBid()    { return _bestBid; }
10   synchronized public int getNumberOfBids()  { return _numberOfBids; }
11
12   synchronized public    void setAuctionIdent(long x) { _auctionIdent=x;}
13   synchronized protected void setTitle(String x)      { _title =x; }
14   synchronized private   void setBestBid(Money x)     { _bestBid =x; }
15   synchronized public    void setNumberOfBids(int x)  { _numberOfBids =x;
16   }
17 }
```



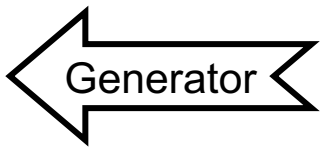
CD



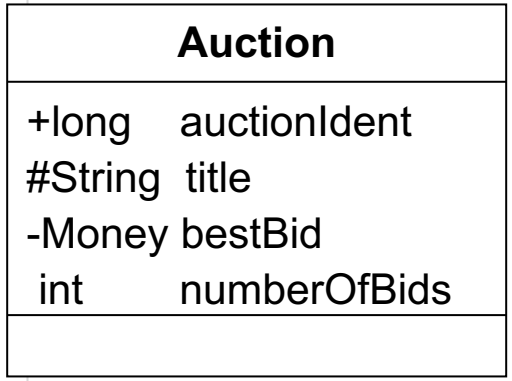
Beispiel: Code-Generierung mit get/set-Methoden

C++

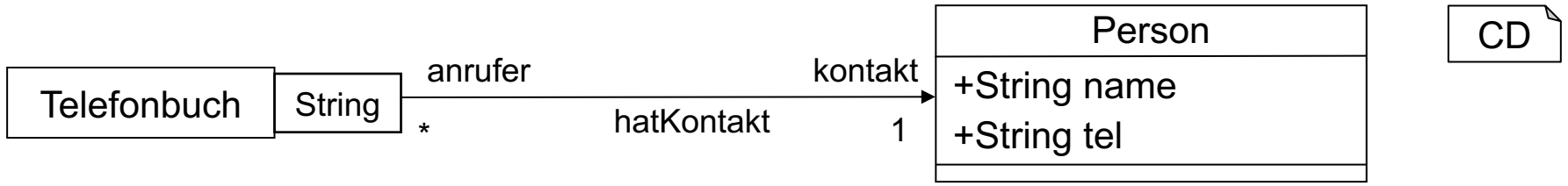
```
1  class Auction {
2    private:
3      long   _auctionIdent;
4      std::string _title;
5      Money  _bestBid;
6      int    _numberOfBids;
7
8    public:
9      long getAuctionIdent() const { return _auctionIdent; }
10     void setAuctionIdent(long x) { _auctionIdent=x;}
11     int getNumberOfBids() const { return _numberOfBids; }
12     void setNumberOfBids(int x)  { _numberOfBids =x;}
13   protected:
14     std::string getTitle() const { return _title; }
15     void setTitle(std::string x)       { _title =x; }
16   private:
17     Money getBestBid() const { return _bestBid; }
18     void setBestBid(Money x)     { _bestBid =x; }
19 };
```

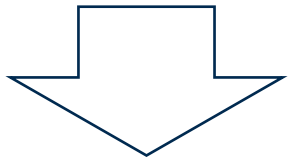


CD

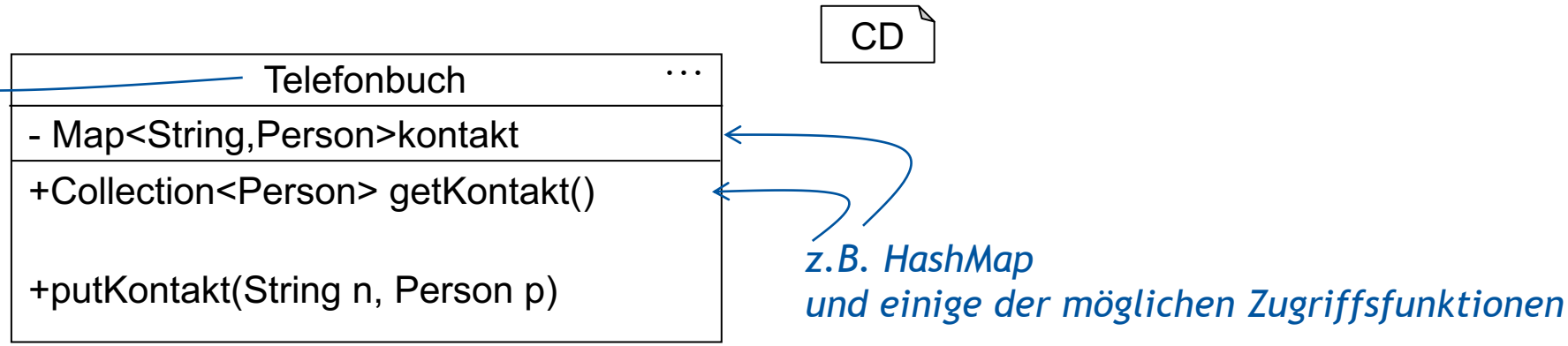


Code für Qualifizierte Assoziation und Links



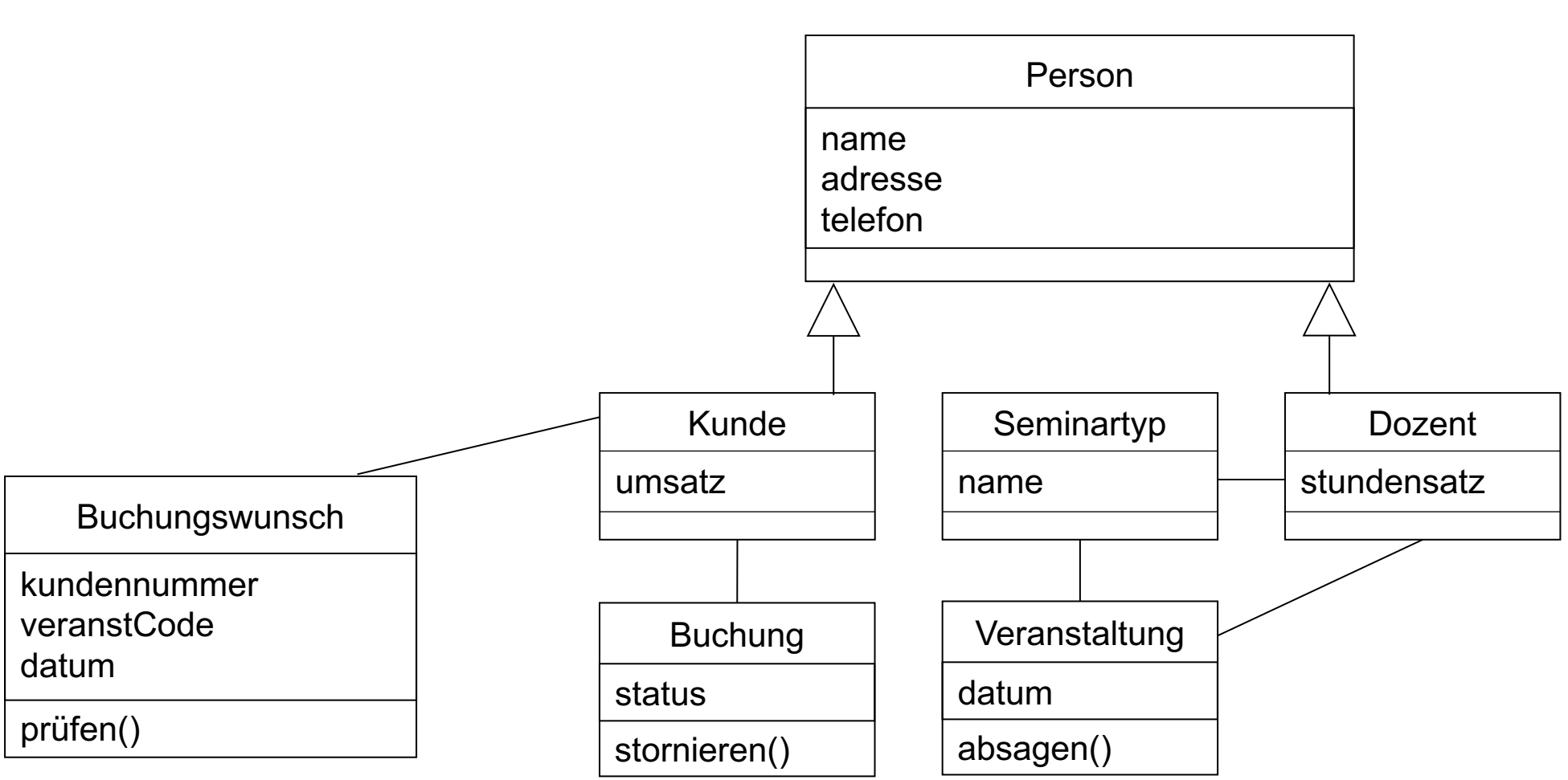


Auszug



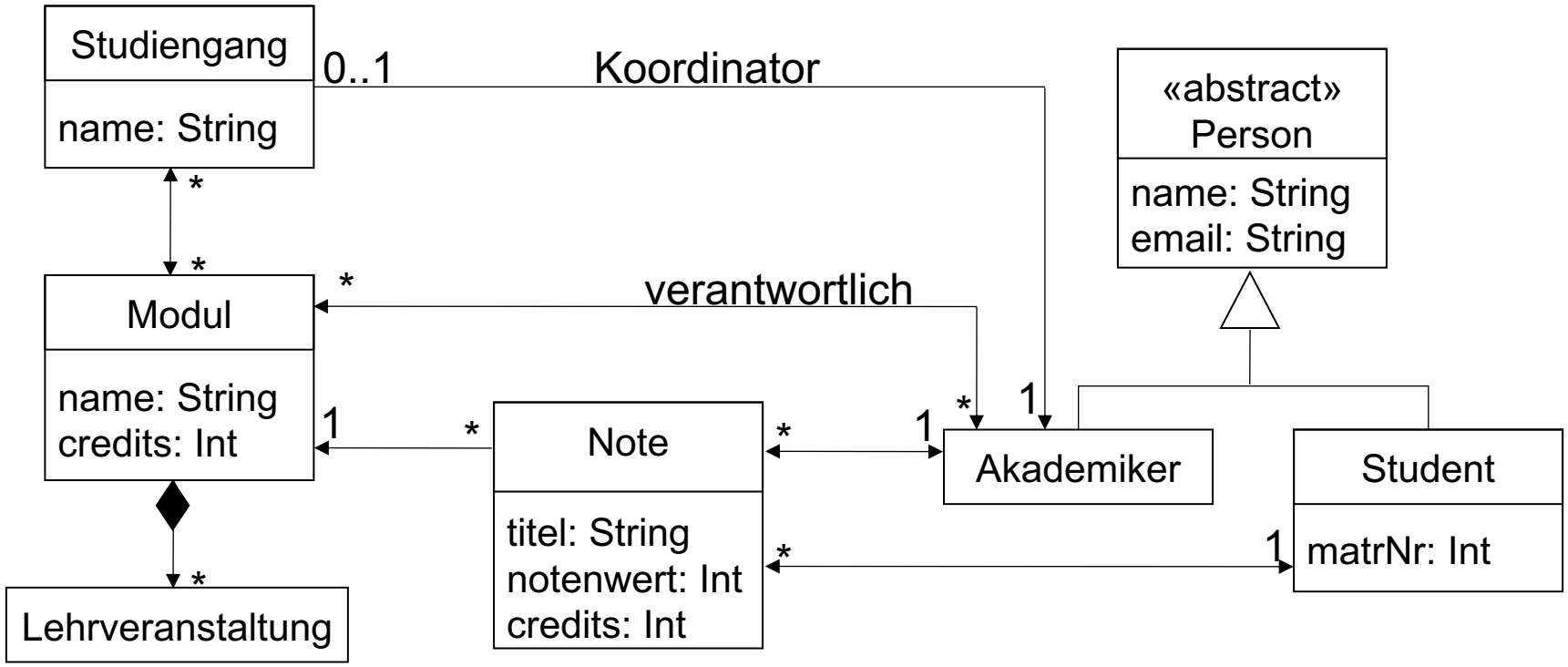
Beispiel-Klassendiagramm aus der Seminarverwaltung

⇨ analyze



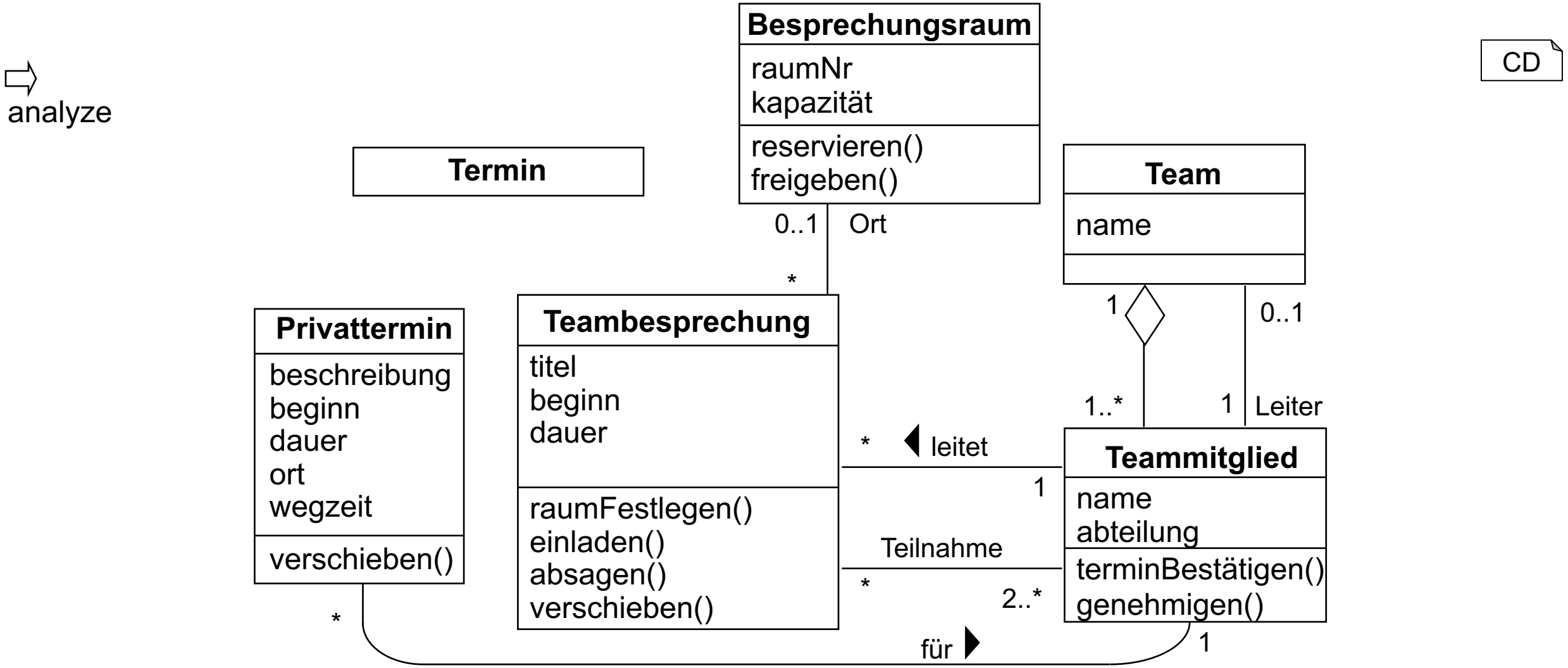
Beispiel: Campus-Management

⇨ analyze



Beispiel: Kalender

⇨ analyze



Eigeninitiative: Klassendiagramm für ...

- Auktion bei Ebay
- Autoverleih
- E-Scooter Verleih

discuss

Was haben wir gelernt?

Klassen-diagramme in der Analyse

…sind notwendig um die wichtigsten Begriffe und Zusammenhänge zu verstehen

… werden im weiteren Verlauf (Entwurf) ev. umgebaut bzw. verfeinert

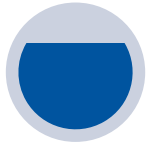
Klassen

…haben Attribute und Methoden

… können Abstrakte Klassen, Interfaces, Enumerations sein

… können von anderen Klassen erben

…können Interfaces implementieren



Assoziationen

… haben Namen, Rollen, Kardinalitäten, Navigations-richtungen

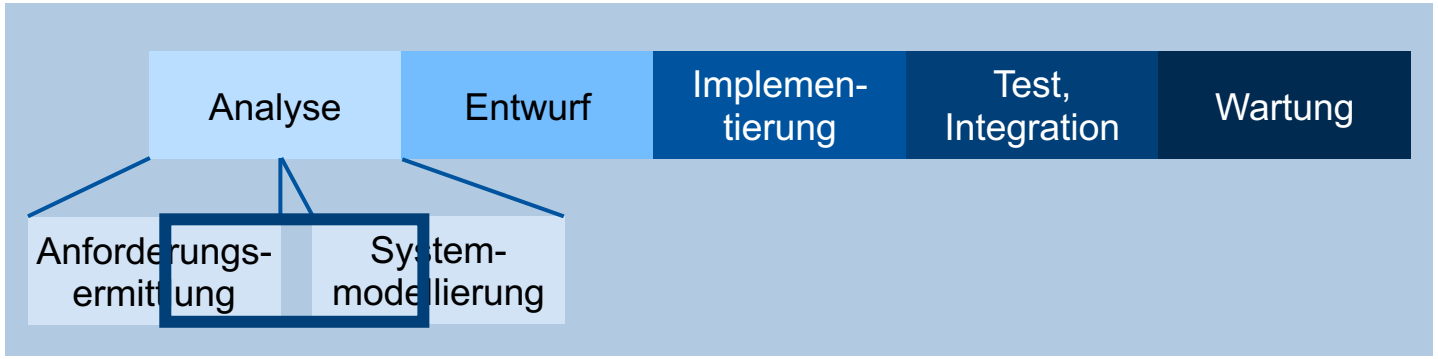
…gibt es in unterschiedlichen Varianten: Assoziation, Komposition, Qualifizierte Assoziation

Gute Modelle erstellen

…benötigt viel Übung und Können

Softwaretechnik

4. Systemanalyse und -modellierung
4.2. Methodik der
Objektorientierten Analyse



Prof. Bernhard Rumpe
Software Engineering
RWTH Aachen

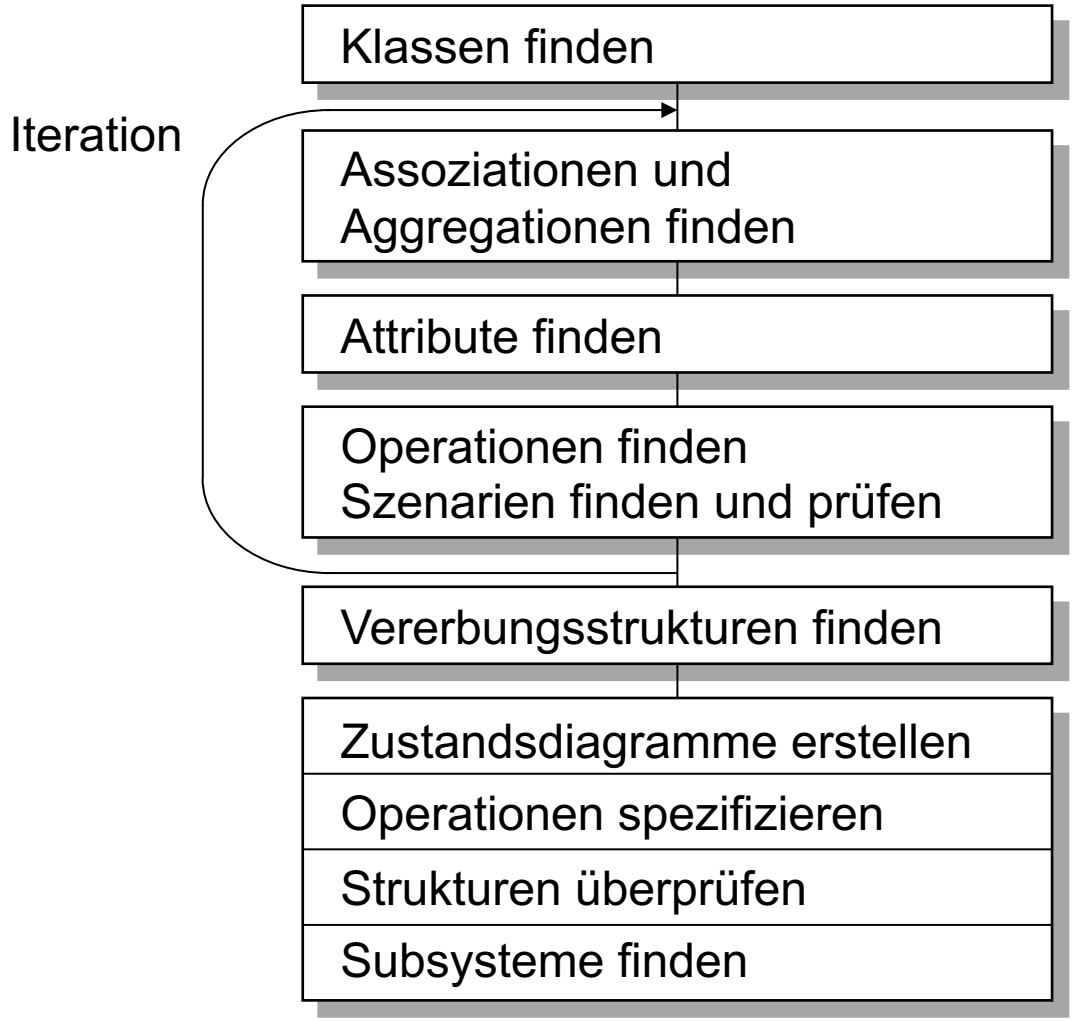
http://www.se-rwth.de/
@SE_RWTH

SE Software Engineering | RWTH AACHEN UNIVERSITY

Methodik der Objektorientierten Analyse

Ziel:
Von den Anforderungen zu einem Modell der fachlichen Aufgabe

Diese Variante ist datenorientiert

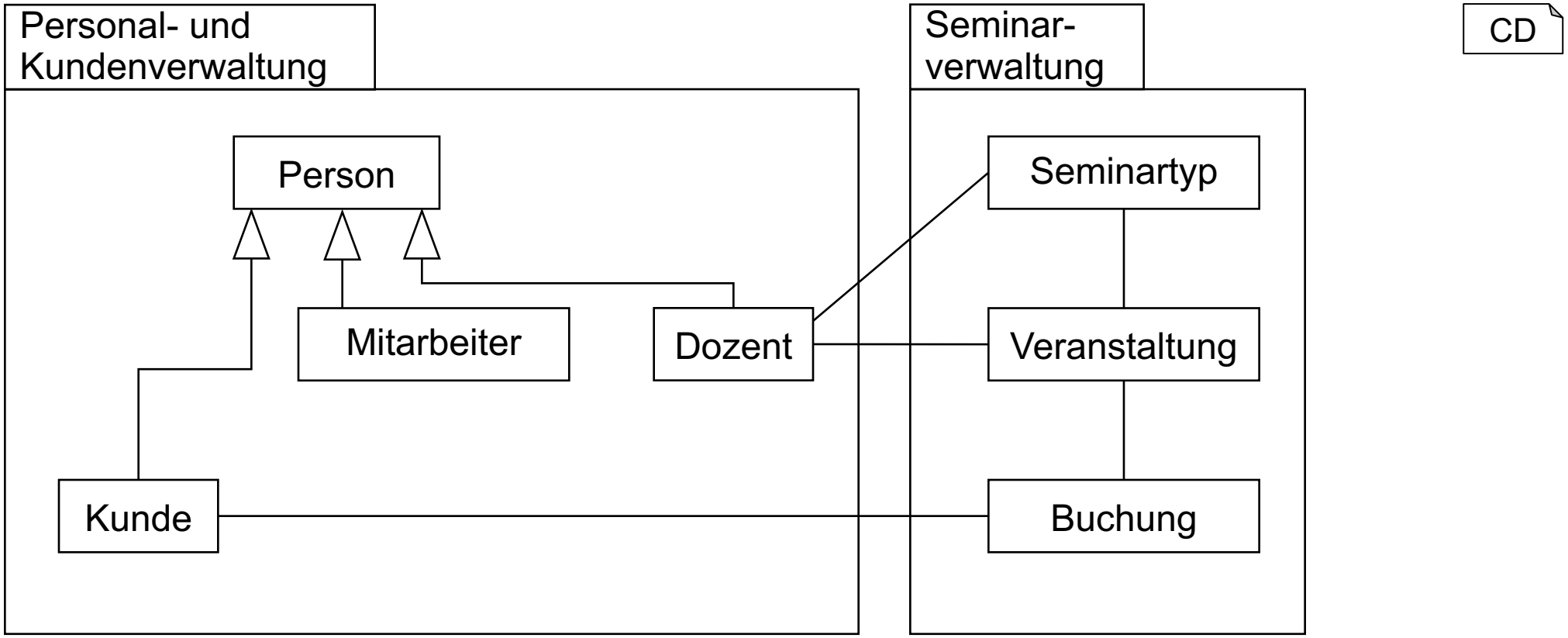


animated

Klassen in der Analyse

- Konzentration auf relevante Klassen:
  - Das sind meist Datenklassen
  - Oft nicht:
    - Technische Klassen (GUI, DB, etc.)
    - Eher keine Manager-/ Verwalterklassen
- Ggf. eigene Klassendiagramme für
  - reines Datenmodell
  - Datenmodell + Manager
  - … + technische Klassen
- Oft ist ein Umbau der (Analyse-)Datenklassen bei der Architekturdefinition sinnvoll!
- Viele kleine Methodiken lösen Detailprobleme …

Pakete: Beispiel



Optimierbar?

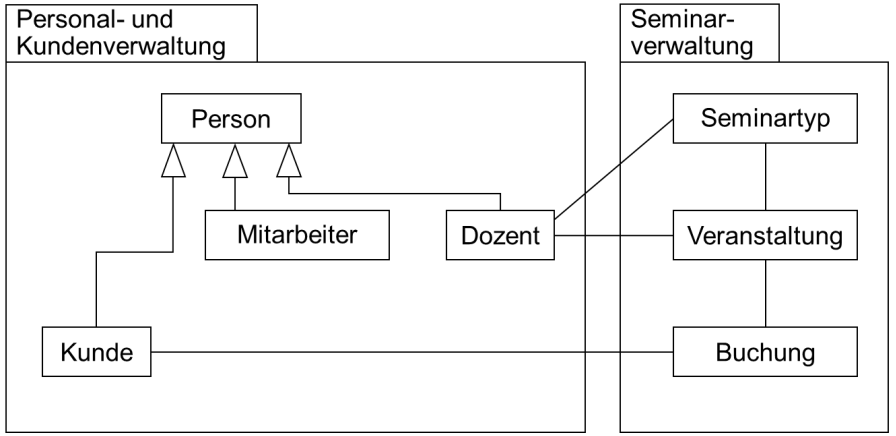
Pakete finden

- Ein Paket ist eine Gruppe von Klassen
  - UML: "Subsystem" als spezielles Paket (Architektureinheit)

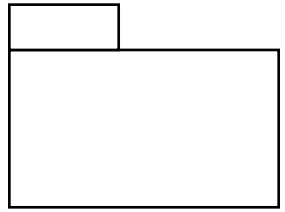
- Ein Paket:
  - ist für sich allein verständlich
  - hat eine wohldefinierte Schnittstelle zur Umgebung
  - ermöglicht Betrachtung des Systems aus einer abstrakteren Sicht

- Ziel: Starke Bindung innerhalb des Pakets
  - Einheitlicher Themenbereich
  - Aggregation und Vererbung soweit wie möglich nur innerhalb des Pakets
- Ziel: Schwache Kopplung zwischen Paketen
  - Möglichst wenig Assoziationen über Paketgrenzen hinweg

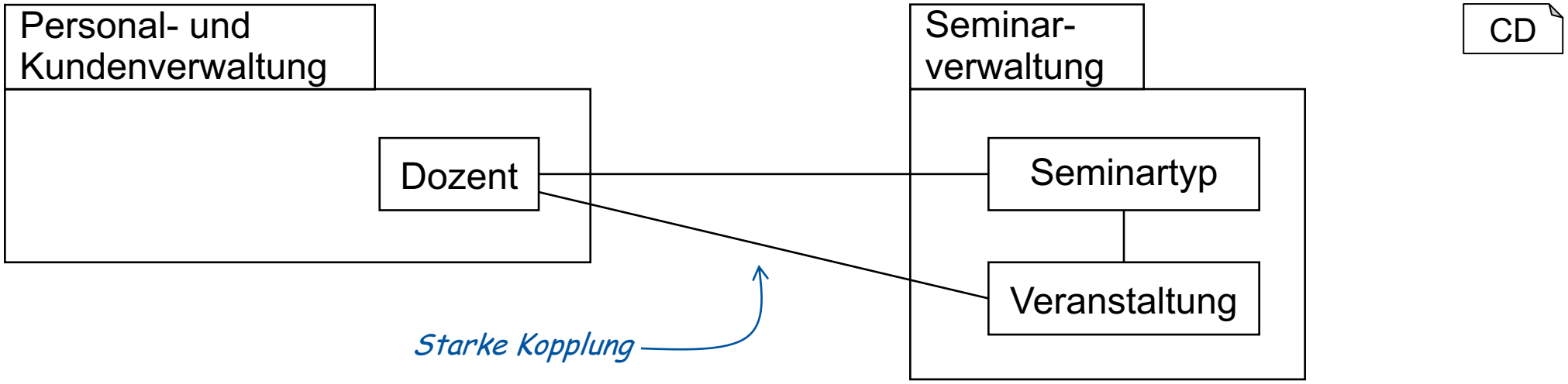
- Faustregeln für ein sinnvolles Paket:
  - 10-15 Klassen
  - 1 DIN A4 Seite für ein Diagramm



UML:



Methode: Entkopplung von Paketen



- Wichtigste Möglichkeit zur Entkopplung: Fachliche Zusammengehörigkeit sicherstellen

- Weitere Möglichkeiten zur Entkopplung (schon entwurfsnah):
  - Dozentenklasse zerlegen in Personal- und seminarbezogene Information
  - "Stellvertreter"-Klassen einführen
    (siehe auch Entwurfsmuster Proxy, Half-Object-plus-Protocol)
  - Schnittstellen-Objekte einführen (z.B. Personalverwaltungs-API)
    (Entwurfsmuster Facade)

Methode: Zuordnung von Operationen

- Prinzipien:
  - Möglichst intensive Ausnutzung der Datenzuordnung:
    - da zuordnen, wo am besten von lokaler Information Gebrauch gemacht werden kann
  - Notwendigkeit zur Objektinteraktion möglichst minimieren
  - Möglichst Gebrauch von vorhandenen Operationen machen bzw. Operationen in verschiedenen Kontexten wiederverwenden.

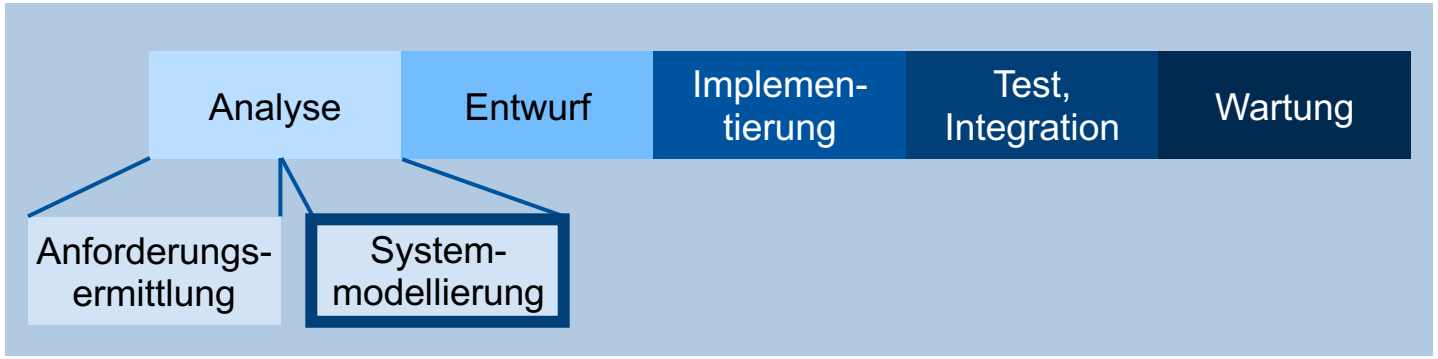
- Es gibt auch Zweifelsfälle
  - Insbesondere ist Nutzungsverhalten oft schwer abschätzbar

Zusammenfassung | Objektorientierte Analyse

… also nicht übertreiben!

Softwaretechnik

4. Systemanalyse und -modellierung
4.3. Objektdiagramme



Prof. Bernhard Rumpe
Software Engineering
RWTH Aachen

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

SE Software Engineering | RWTH AACHEN UNIVERSITY

Warum, was, wie und wozu? Objektdiagramme

Warum?

- Exemplarische Beschreibung konkreter Situationen
- Details hilfreich z.B. beim Testen

Was?

- Modellierung von Objekten
- Lesen von Objektdiagrammen

Wie?

- Objekte mit Attributen, Typen und Werte
- Links der Assoziationen
- Komplexere Objektstrukturen

Wozu?

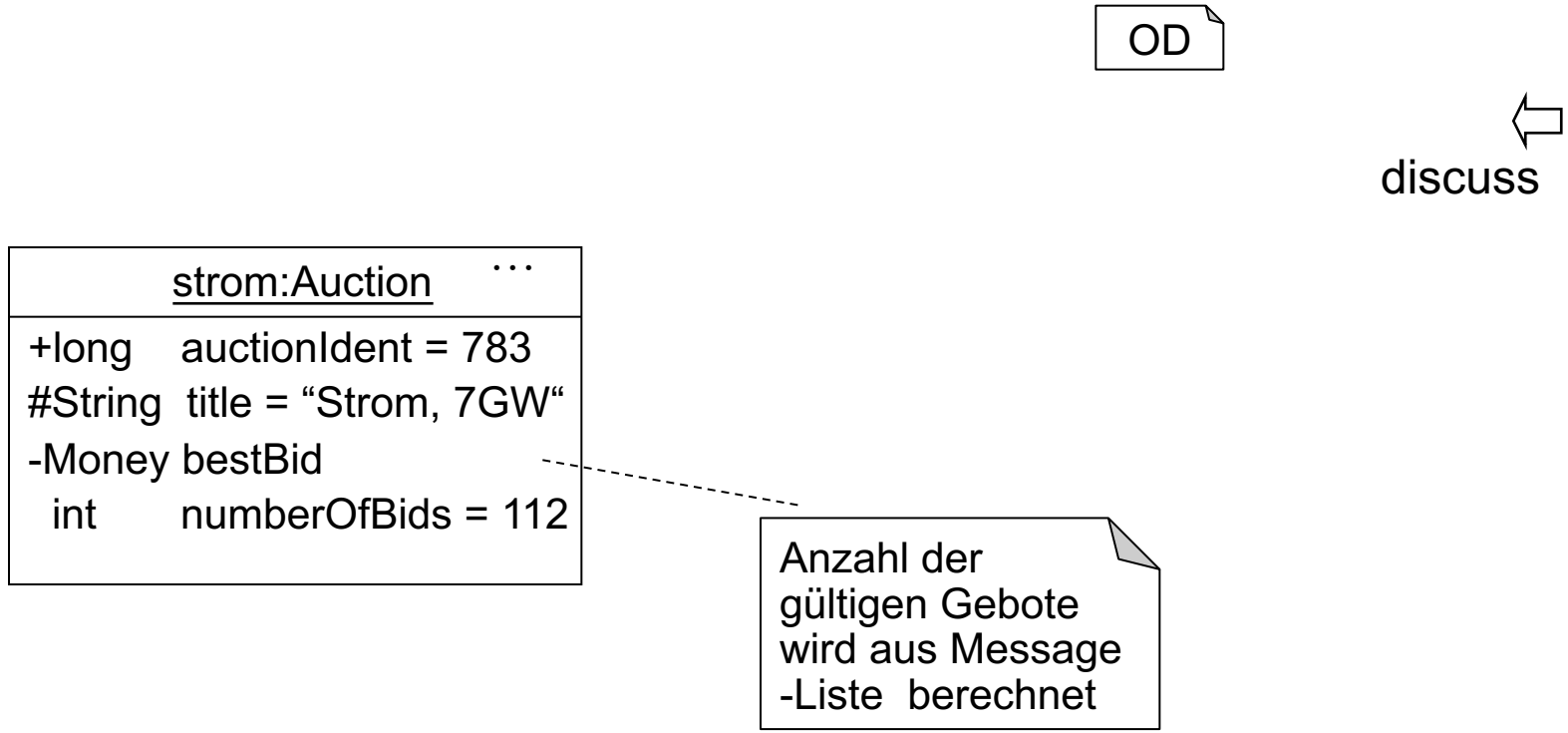
- Modellierung spezieller Situationen bzw. Randfälle
- Kommunikation mit den Kunden
- Verwendung in Tests

Objektdiagramme

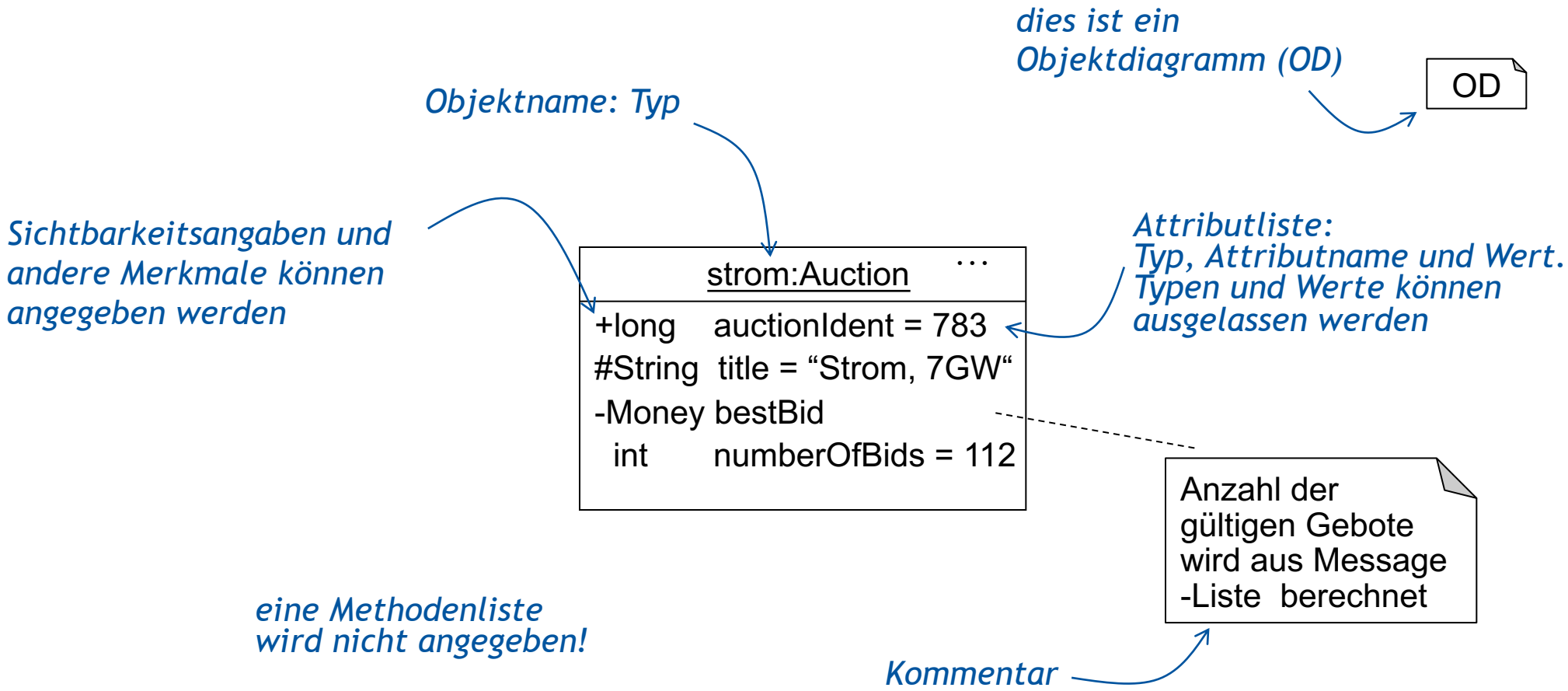
- Modellierung von Strukturen auf exemplarischer Basis
  - statisch unveränderliche Strukturen in dynamischen objektorientierten Systemen
  - spezielle Situationen, die z.B. als Vor- oder Nachbedingung bei Tests verwendet werden

- Sammlung von Objekten und deren Beziehungen zu einem Zeitpunkt im Leben eines Systems
  - Instanzen
  - Exemplarisch
  - Mehrere Objektdiagramme können unterschiedliche Situationen zu verschiedenen Zeitpunkten innerhalb desselben Systemteils beschreiben

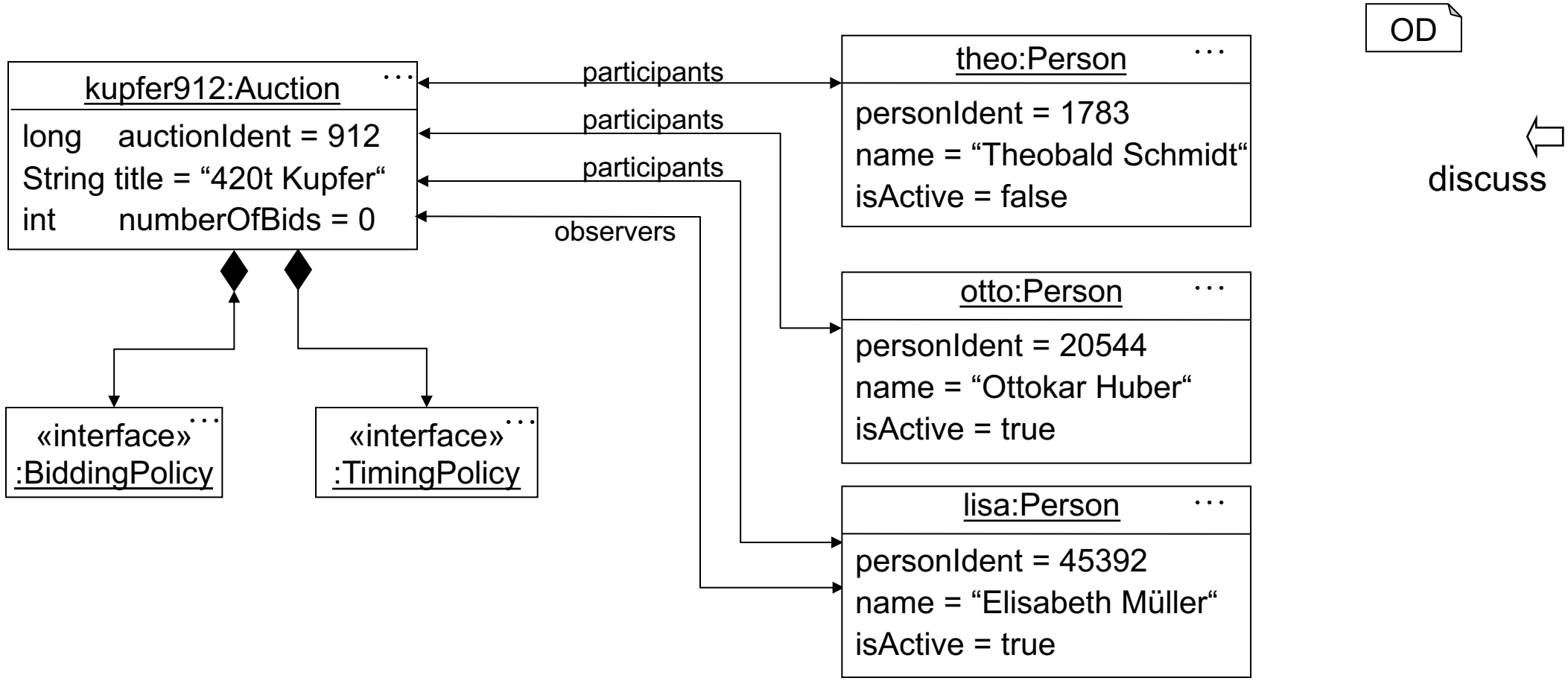
Beispiel: Einzelnes Objekt



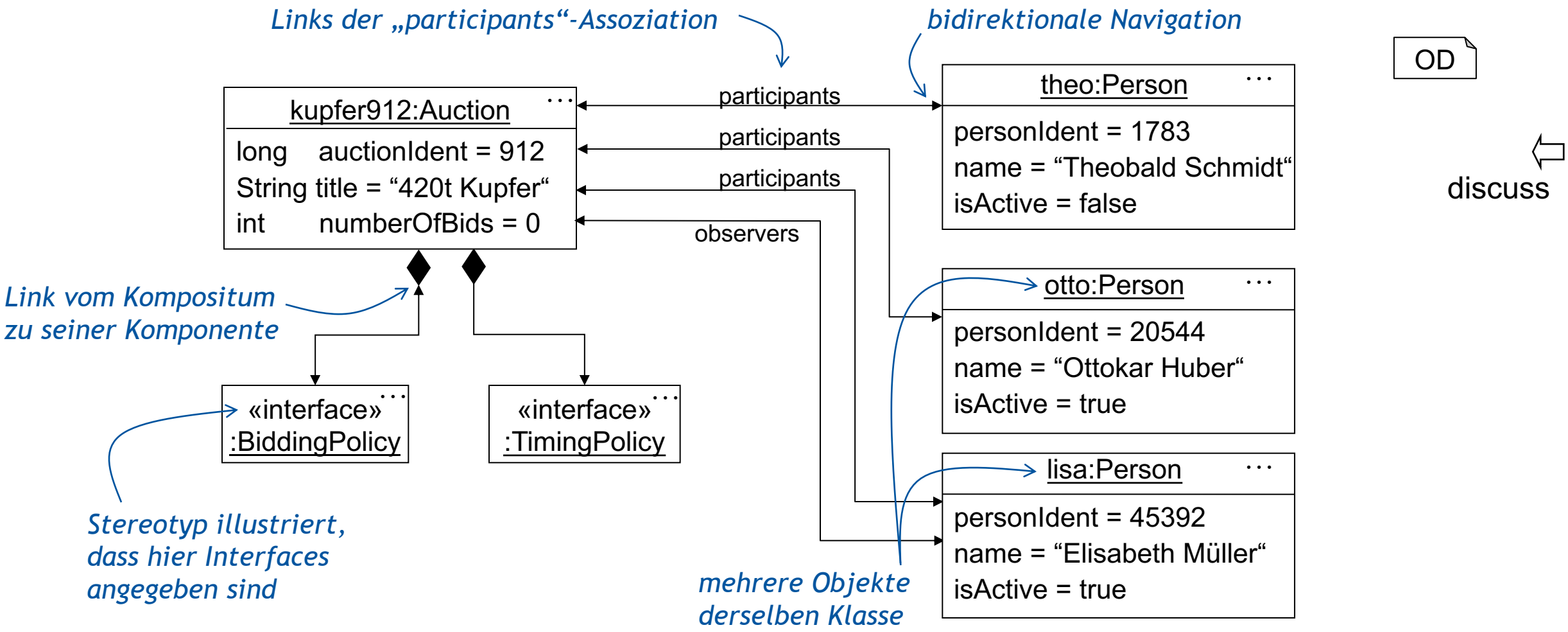
Beispiel: Einzelnes Objekt



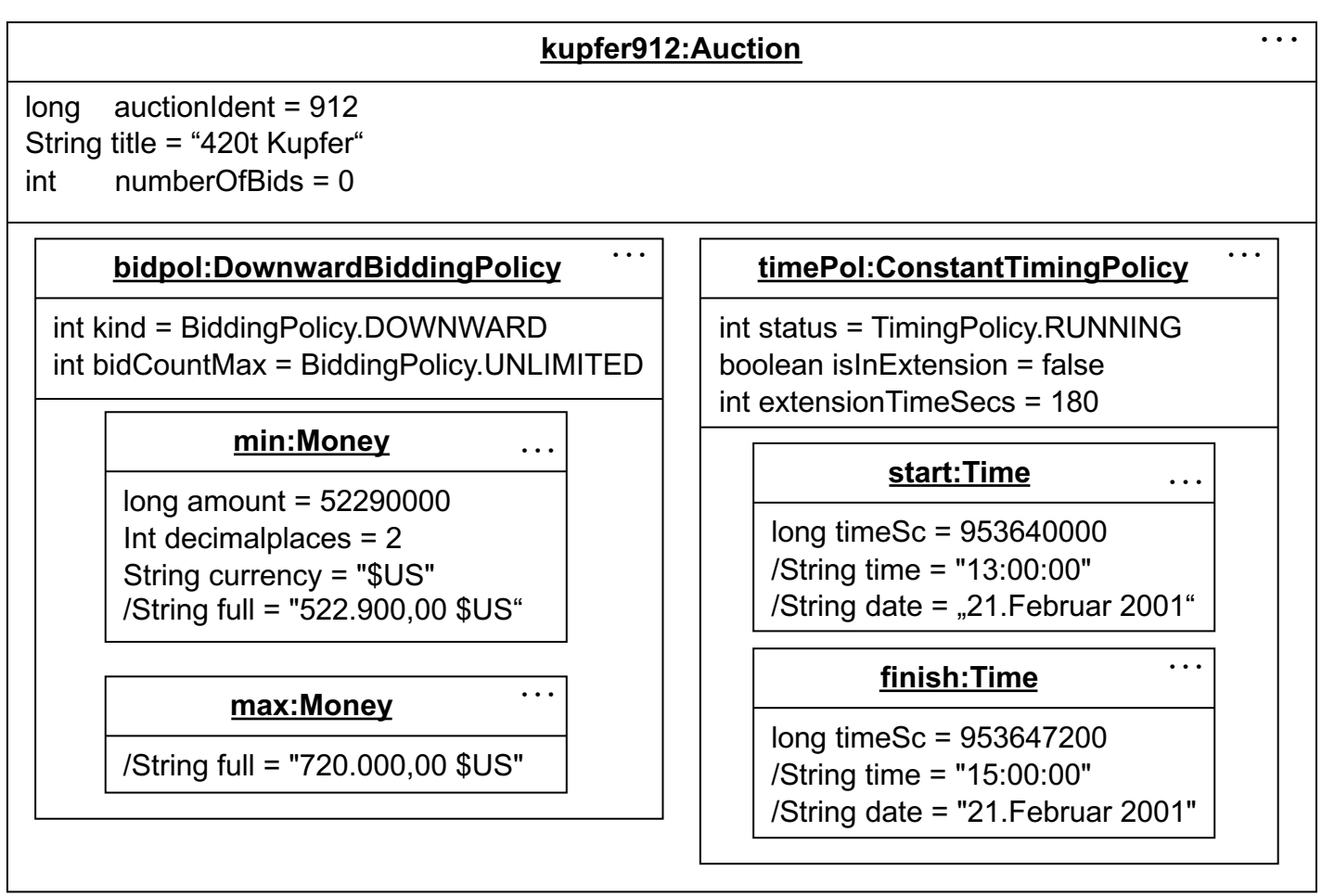
Beispiel: Linkstruktur



Beispiel: Linkstruktur



Objektstruktur mit Einbettung, um Komposition zwischen Objekten darzustellen

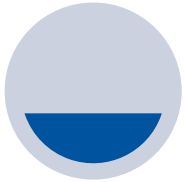


OD

animated

Siehe: http://mbse.se-rwth.de/book1/index.php?c=chapter4-1#x1-9100412

Was haben wir gelernt?

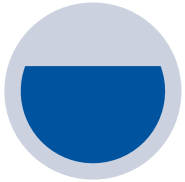


OO-Analyse Methode

Ziel: Von den Anforderungen zu einem Modell der fachlichen Aufgabe

Datenorientiert (vs. Verhaltensorientiert)

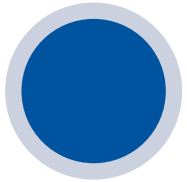
Identifikation von Klassen, Assoziationen, Attribute, Operationen, Szenarien, Vererbungsstrukturen,…



Objektdiagramme

Modellierung von Strukturen auf exemplarischer Basis

… können für Tests verwendet werden z.B. Randfälle und Spezialfälle



Objekte

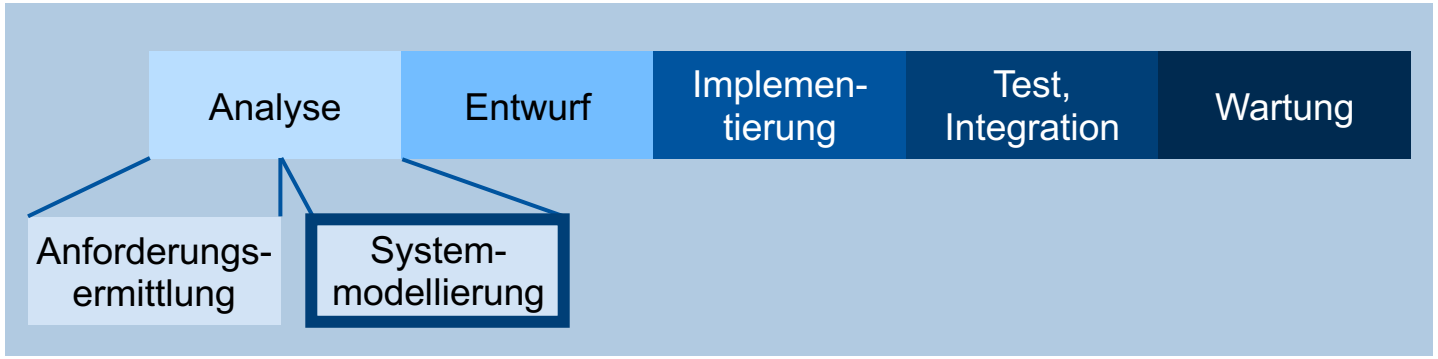
… haben Attribute mit konkreten Werten aber beschreiben KEINE eigenen Methoden,

… können Links zu anderen Objekten haben und

… können komponiert sein

Softwaretechnik

4. Systemanalyse und -modellierung
4.4. Modellierung von Szenarien: Sequenzdiagramme



Prof. Bernhard Rumpe
Software Engineering
RWTH Aachen

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

SE Software Engineering | RWTH AACHEN UNIVERSITY

Warum, was, wie und wozu? Sequenzdiagramme

Warum?

- Benötigen Verständnis zu Interaktionen der Systemkomponenten
- Normalfälle beschreiben und Fehlerfälle untersuchen

Was?

- Modellierung von Verhaltenssequenzen, d.h. beispielhafte Folgen von Interaktionen
- Szenarien für Normalfälle, Ausnahmefälle, Tests

Wie?

- Erstellung von Sequenzdiagrammen für einzelne Szenarien
- Kommunikation zwischen Objekten

Wozu?

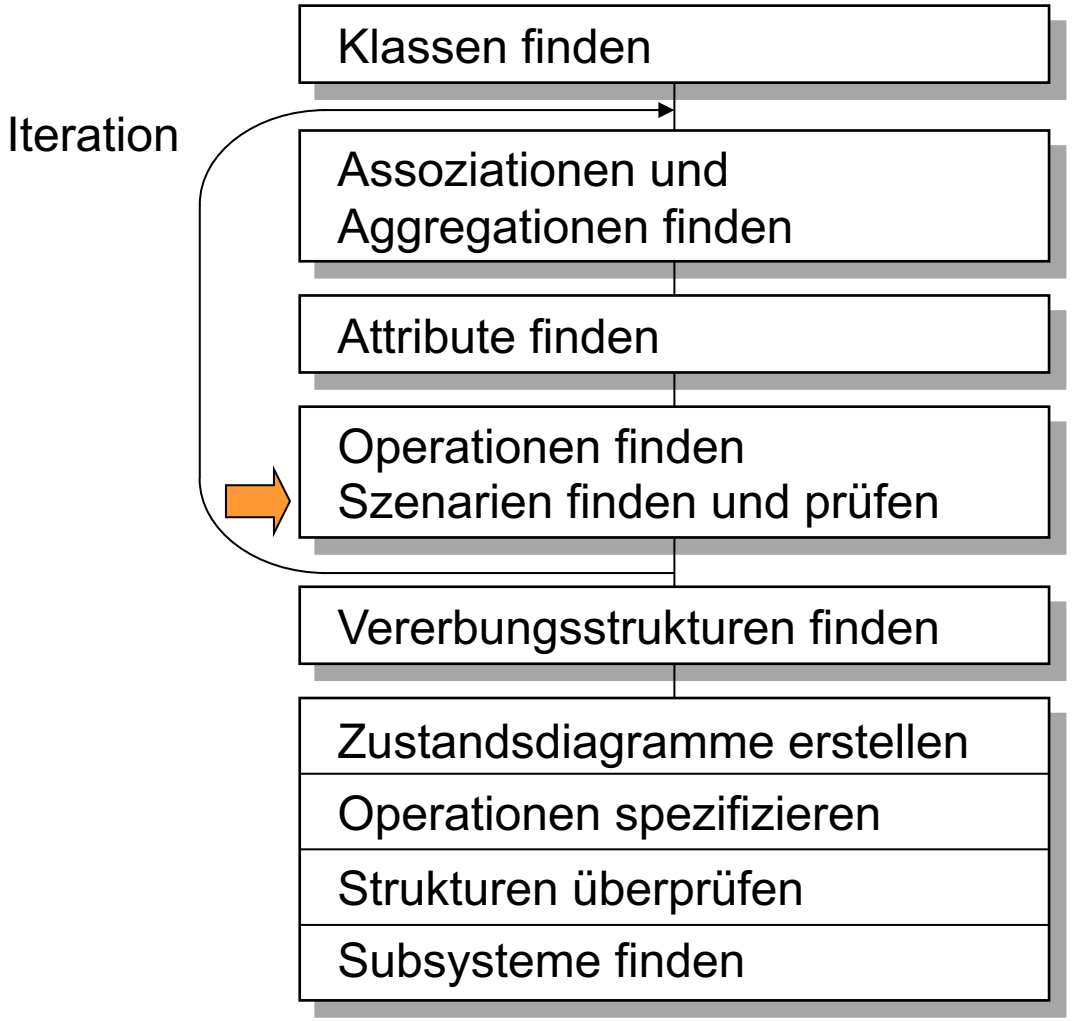
- Besseres Verständnis von Szenarien und Fehlerfällen
- Modellierung von Testszenarien

Methodik der Objektorientierten Analyse

Wiederholung

Ziel:
Von den Anforderungen zu einem Modell der fachlichen Aufgabe

Diese Variante ist datenorientiert



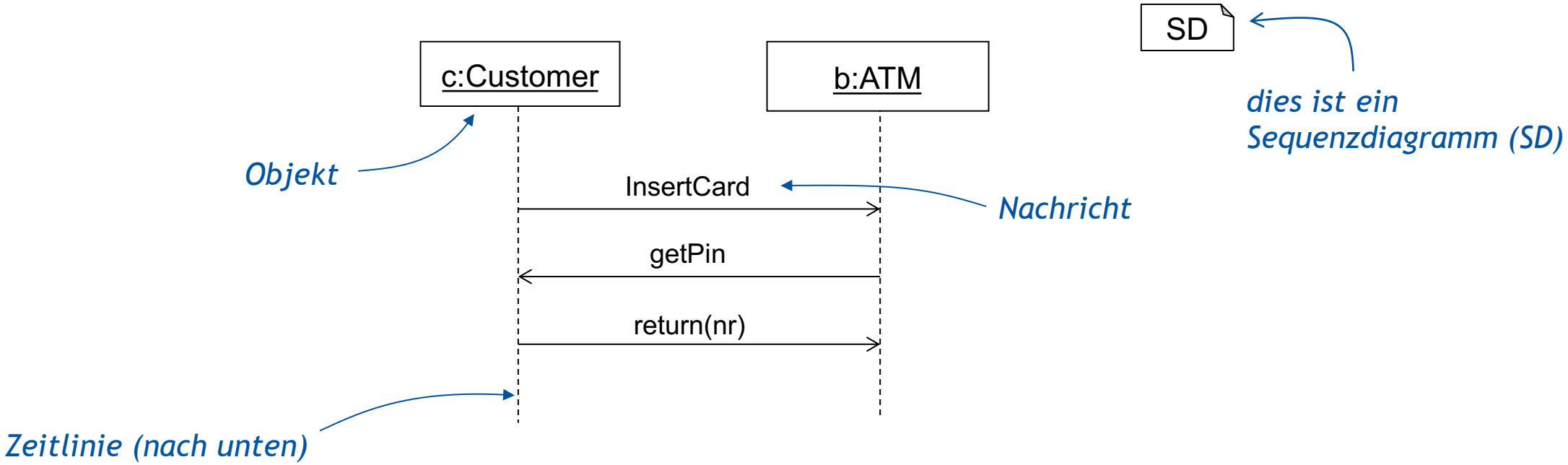
Szenarien

- Definition: Ein ***Szenario*** ist eine Beschreibung einer beispielhaften Folge von Interaktionen von Akteuren mit dem System zur Beschreibung eines Anwendungsfalls.
- Es gibt Szenarien für Normalfälle ('gut-Fälle') und Ausnahmefälle.

- Anwendungsgebiete:
  - Normalfall-Szenarien zur Diskussion mit Anwendern
  - Ausnahmefall-Szenarien zur Erkennung abzufangender Fehlerquellen
  - Testszenarien um festzulegen, welche Tests auszuprobieren sind
  - In Abläufen erzeugte Szenarien, um Debugging durchzuführen

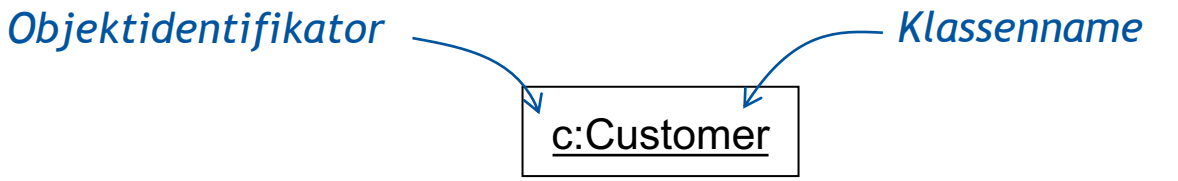
Einfaches Sequenzdiagramm

- Objekte werden oben angeordnet
- Die Zeitlinie schreitet für alle Objekte gleich voran
- Ein Sequenzdiagramm zeigt den Nachrichten- bzw. Ereignisfluss zwischen Objekten
  - z.B. Versenden von Nachrichten, Datenaustausch, Methodenaufruf

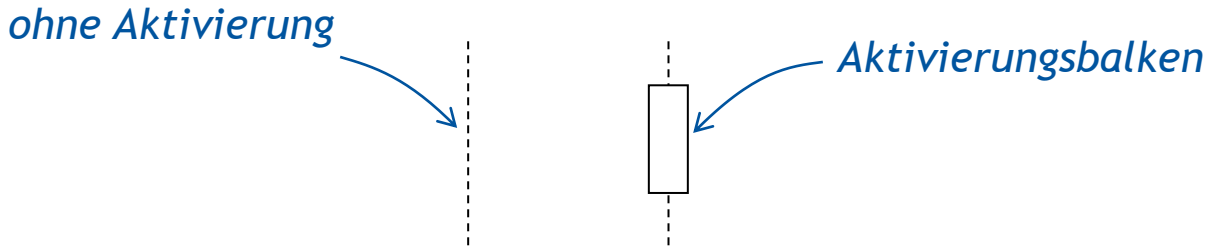


Grundlegende SD Elemente

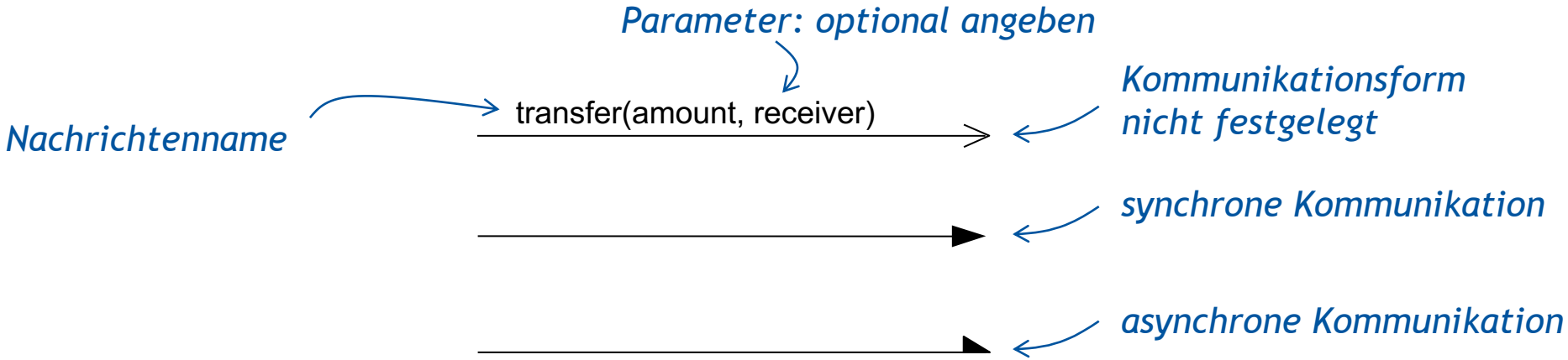
- Objektsymbol:



- Zeitlinie:

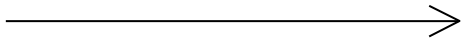


- Pfeile:

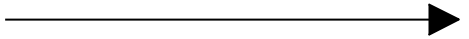


Pfeilarten

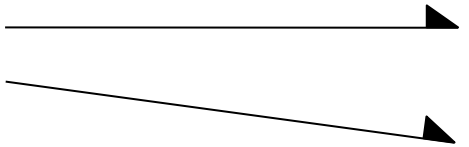
- Neutrale Pfeile:
  - legen den Kommunikationsmechanismus nicht fest
  - erlauben diese Entscheidung später zu treffen



- Synchrone Pfeile:
  - Interaktion ist ein gemeinsames Ereignis zwischen Sender und Empfänger
  - keine Verzögerung, z.B. wie ein Telefongespräch
  - Beispiele: Methodenaufruf

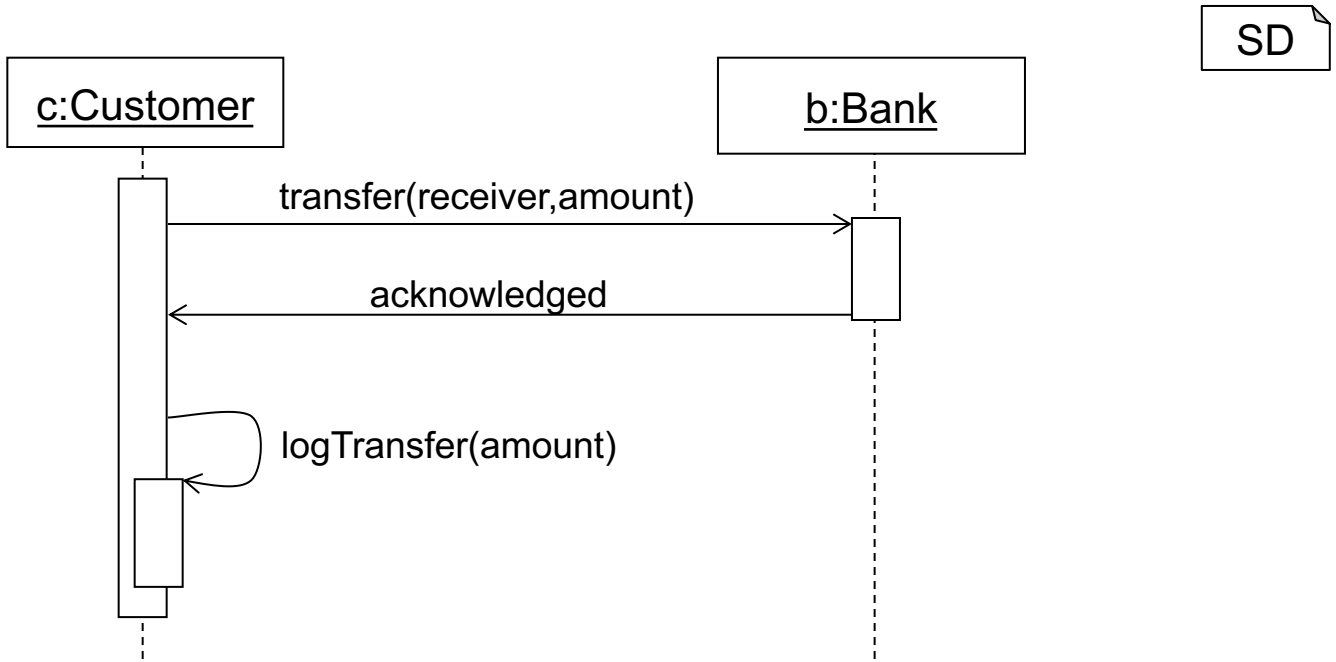


- Asynchrone Pfeile:
  - Senden und Empfang einer Nachricht sind unterschiedliche Ereignisse
  - Normalerweise ist Verzögerung im Spiel, wie bei SMS-Senden
  - Empfänger muss nicht sofort bereit sein, die Nachricht zu empfangen



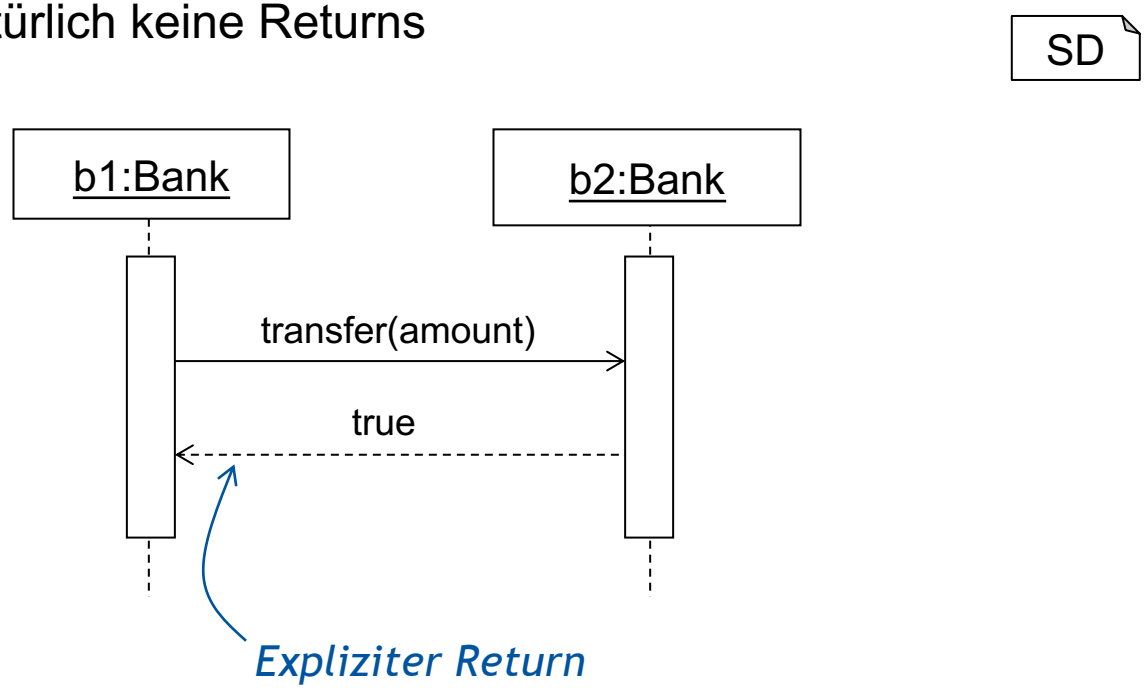
Aktivierungsbalken (Aktivierung)

- Aktivierungsbalken
  - erlauben anzuzeigen, wenn ein Objekt aktiv ist
  - können den Kontrollfluss im System darstellen
  - können verschachtelt werden (Objektrekursion)



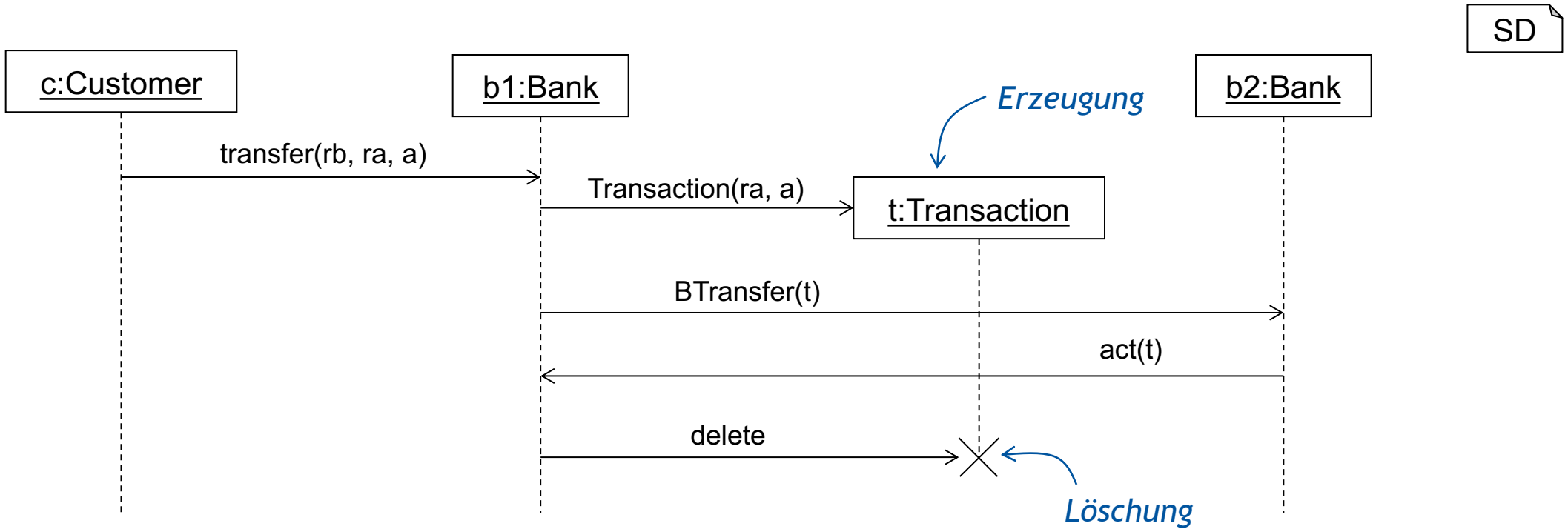
Returns

- Return zeigt an,
  - wenn der Kontrollfluss zum Aufrufer zurück geht
  - welches Ergebnis dabei übertragen wird
- Spezielle Pfeile zeigen Returns an (diese sind optional)
- Asynchrone Nachrichten haben natürlich keine Returns

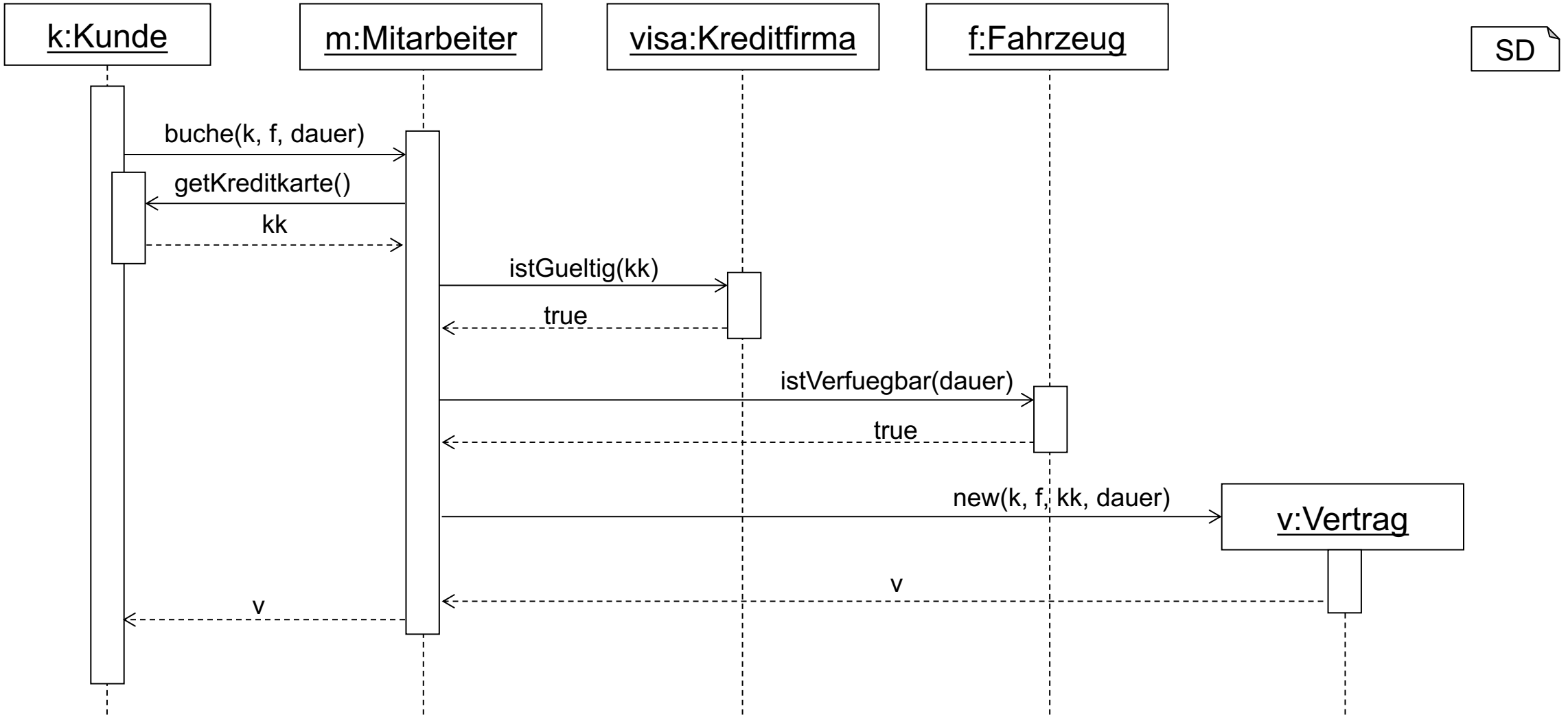


Objekterzeugung und -löschung

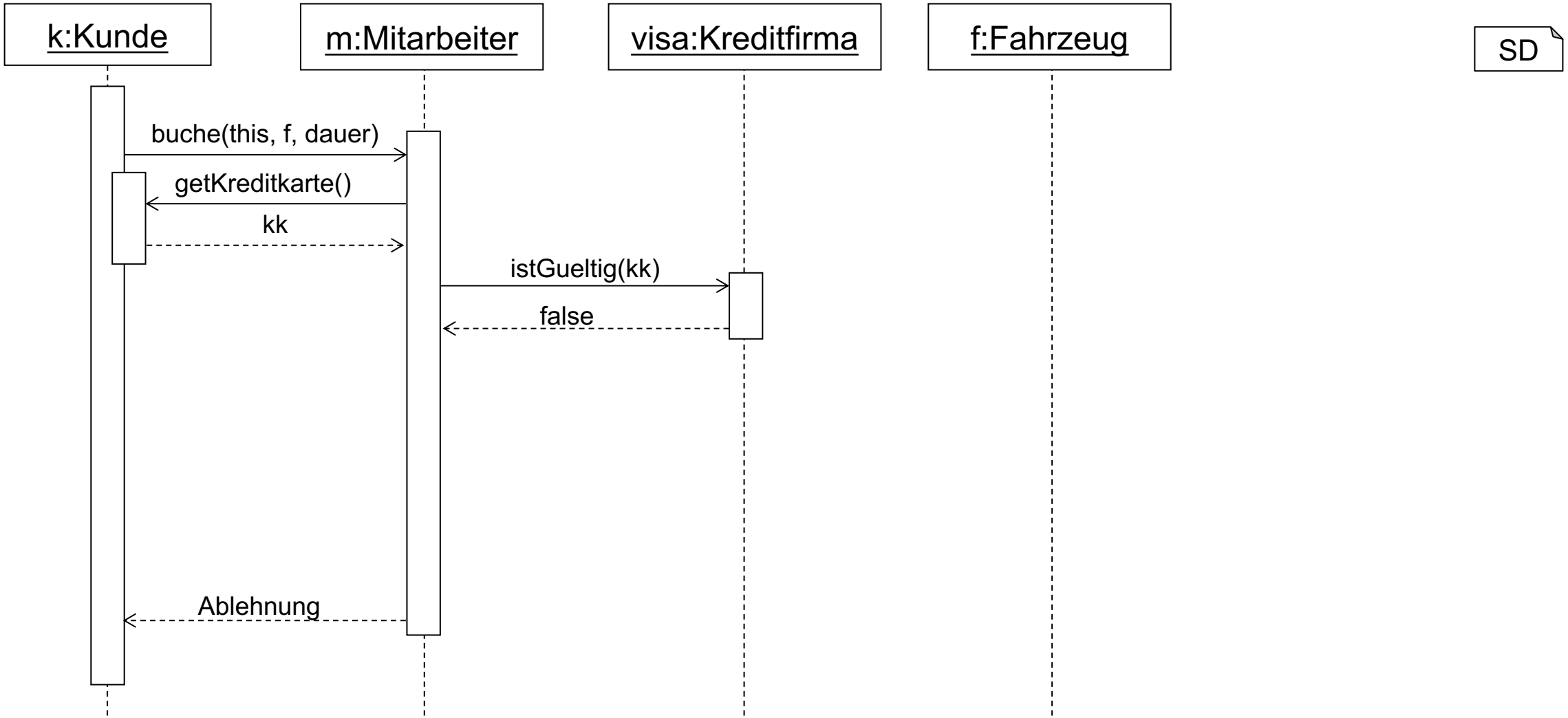
- Objekte die erzeugt werden, werden an der Erzeugungsstelle dargestellt
- Eine create-Nachricht zeigt direkt auf das Objekt
- Objektlöschung wird durch ein Kreuz am Ende der Zeitlinie angezeigt
  - (Java kennt keine übrigens explizite Objektlöschung)



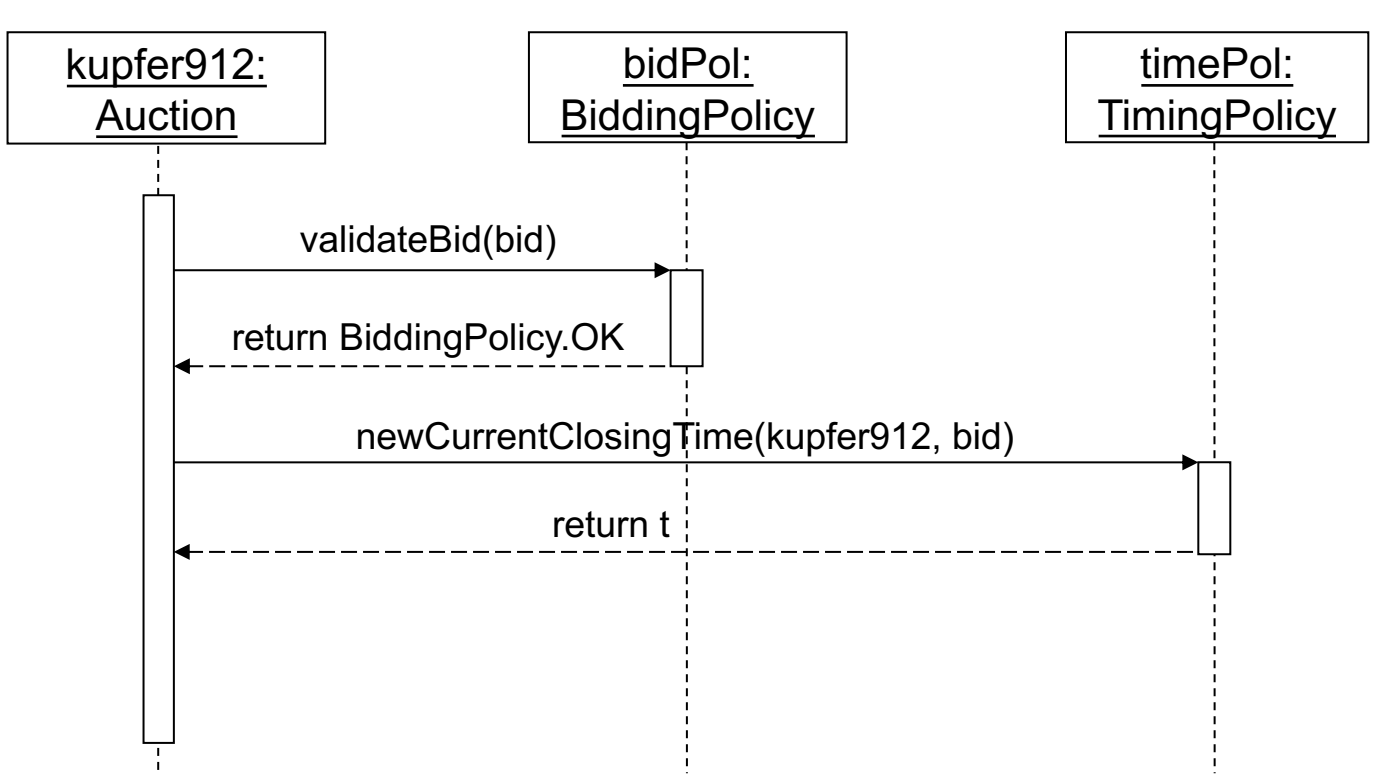
Beispiel: Autovermietung



Beispiel: Autovermietung



Beispiel: Auktionssystem



SD

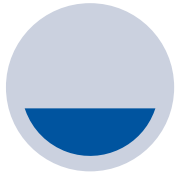
⇦ Analyse

Eigeninitiative: Sequenzdiagramm für …

- Telefonanruf über Vermittlung
- oder: Auktionsverlauf bei Ebay, oder ...

⇦
discuss

Was haben wir gelernt?

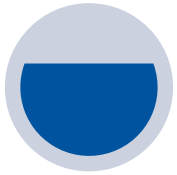


Szenarien

… beschreiben eine exemplarische Folge von Interaktionen von Akteuren mit dem System

…werden durch Sequenzdiagramme dargestellt

Normalfälle ('gut-Fälle'), Ausnahmefälle, Testfälle



Sequenz-diagramme

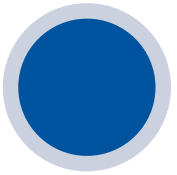
…sind exemplarische Verhaltens-beschreibungen

bestehen aus…

… einer horizontal angeordneten Menge von Objekten

… nach unten voranschreitenden Zeitlinien und

… synchronen oder asynchronen Interaktionen zwischen den Objekten



Beschrieben werden kann…

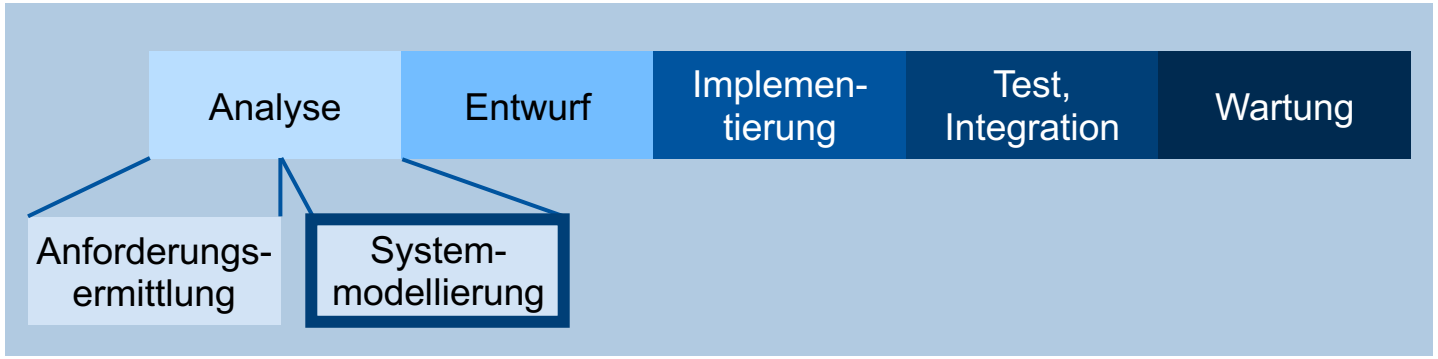
Systeminterne Kommunikation

Kommunikation zwischen System und Anwender

Kommunikation zwischen Anwendern/Menschen

Softwaretechnik

4. Systemanalyse und -modellierung
4.5. Dynamik Modellierung mit Statecharts



Prof. Bernhard Rumpe
Software Engineering
RWTH Aachen

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

SE Software Engineering | RWTH AACHEN UNIVERSITY

Warum, was, wie und wozu? Statecharts

Warum?

- Verhalten ist die Essenz von Software
- Verhalten abhängig von internen Zuständen

Was?

- Beschreibung des Objektverhaltens
- Modellierung von Objektzuständen und Zustandsübergängen

Wie?

- Erstellung von Statechart Modellen

Wozu?

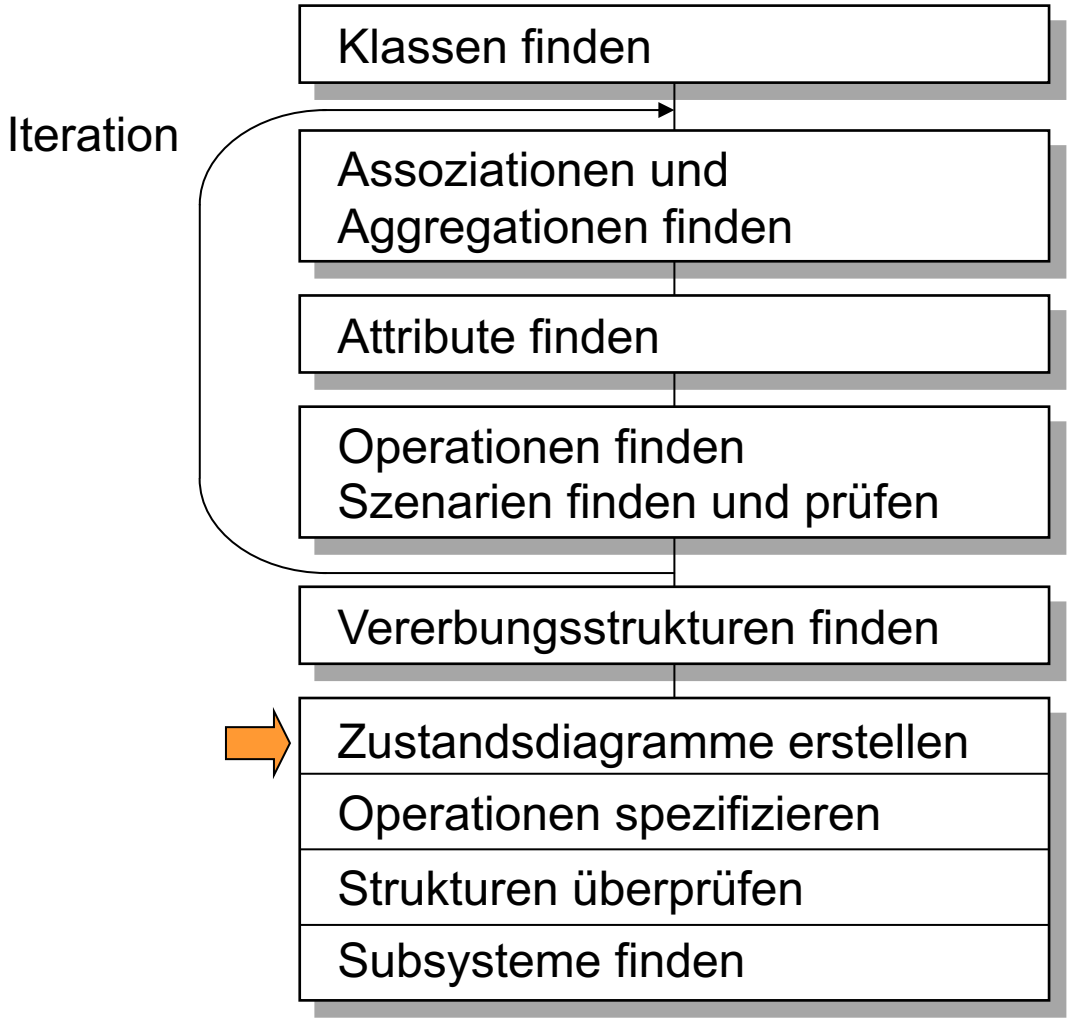
- Definition zulässiger zustandsabhängiger Ereignisfolgen, Protokolle, Steuerungen

Methodik der Objektorientierten Analyse

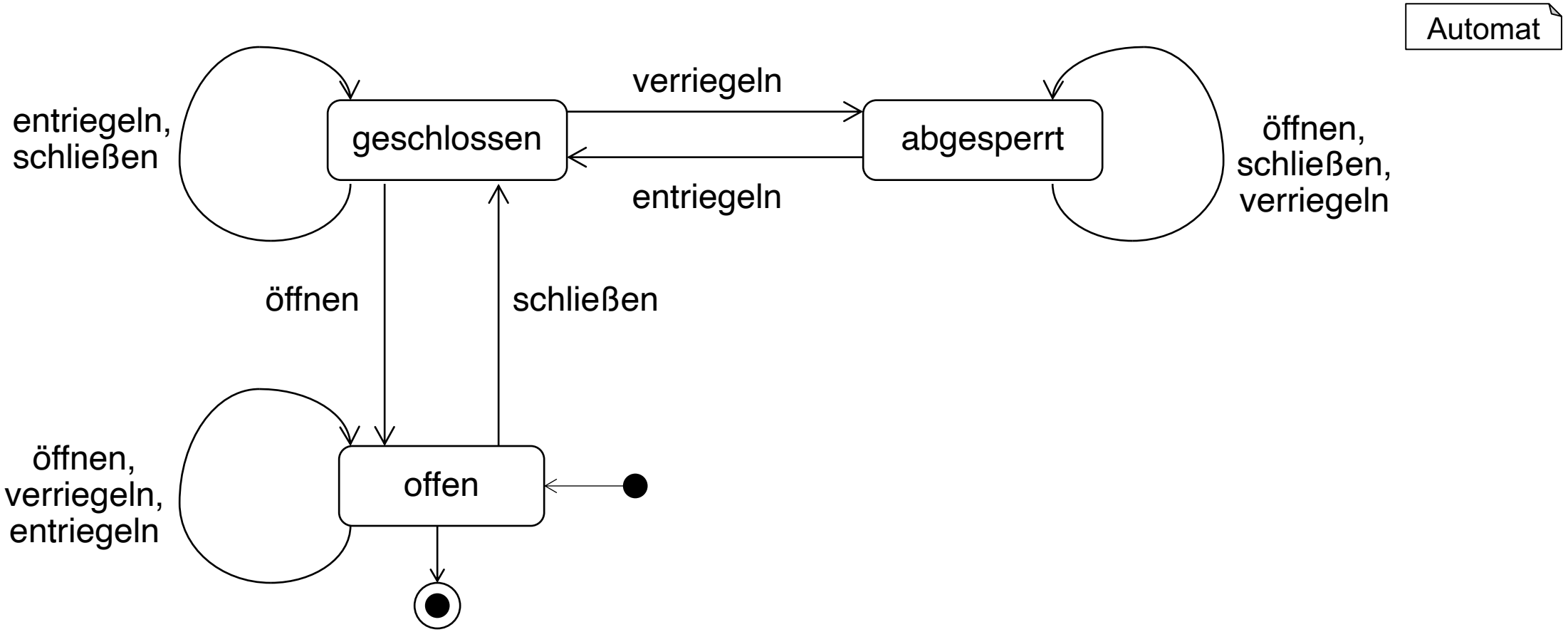
Wiederholung

Ziel:
Von den Anforderungen zu einem Modell der fachlichen Aufgabe

Diese Variante ist datenorientiert



Beispiel: Zustandsmodell einer Tür

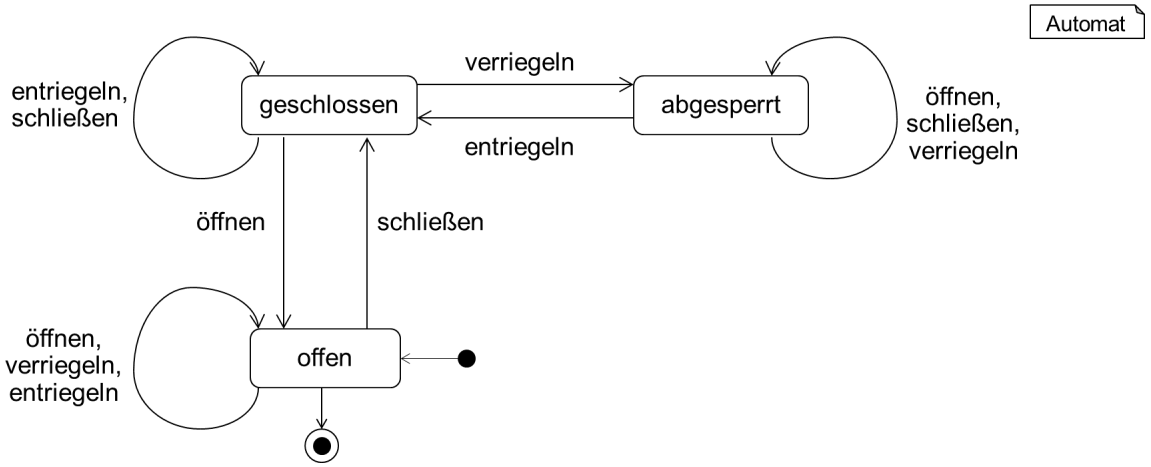


Statecharts

- Ziel ist die Beschreibung von Objektverhalten
- Annahmen:
  - Objekte haben im Zustand einen Daten- und einen Kontrollanteil
  - Kontrollanteil beschrieben durch endlichen Zustandsraum
  - Objektveränderungen sind Transitionen

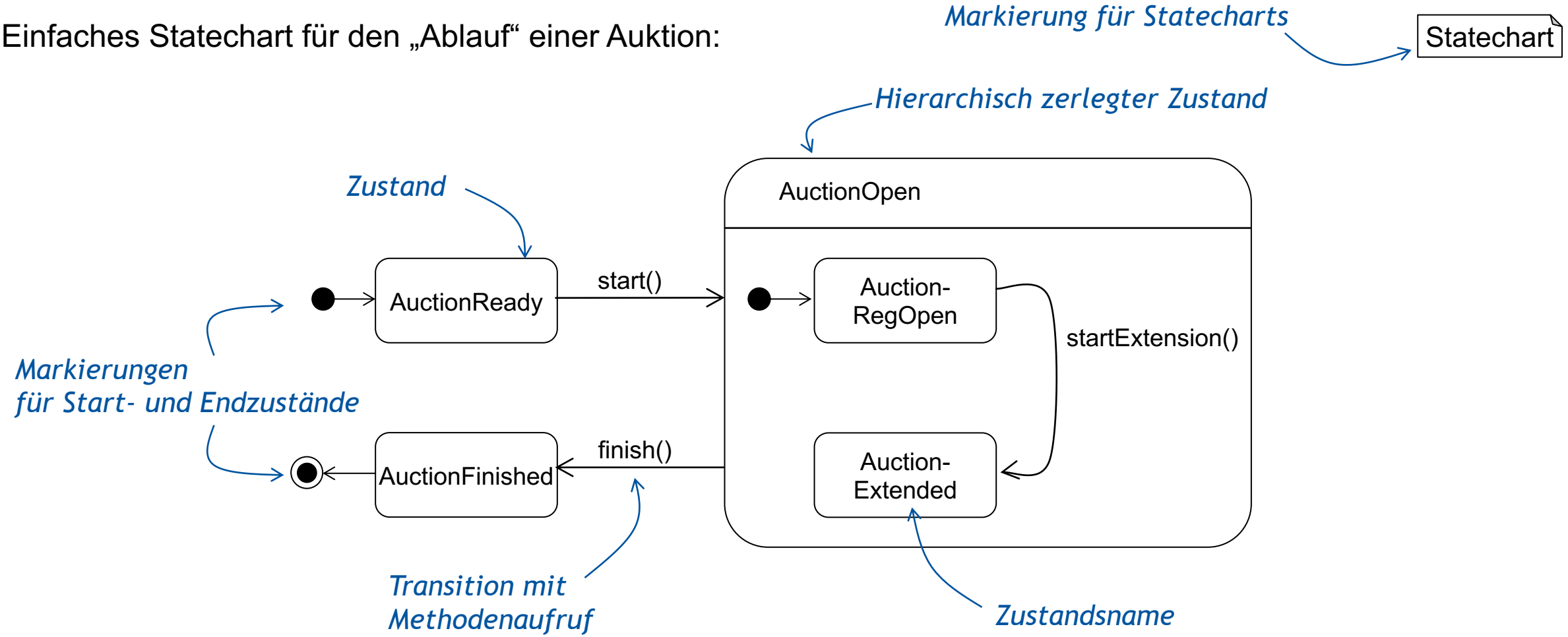
- Statecharts erweitern Automatentheorie:
  - Hierarchische Zustände,
  - Aktionen in Transitionen und Zuständen, ...

- Historie:
  - Statecharts von David Harel, 1987 eingeführt
  - in viele Modellierungssprachen übernommen
  - viele Varianten entwickelt
  - von Beginn an Teil der UML



Beispiel-Statechart

- Einfaches Statechart für den „Ablauf“ einer Auktion:



Semantik eines Zustandsmodells

- Die Semantik eines Zustandsmodells ist definiert als Menge von Sequenzen:
  - in der Theoretischen Informatik: Menge von "akzeptierten Wörtern" (über Grundalphabet von Ereignissen)
  - in der Softwaretechnik: Menge von zulässigen *Ereignisfolgen*

- Wichtige Verallgemeinerung: "Automaten mit Ausgabe"
  - Mealy-Automaten: Ausgabe bei Übergang
    - Softwaretechnik: *Aktion* bei Übergang
  - Moore-Automaten: Ausgabe bei Erreichen eines Zustands
    - Softwaretechnik: *Aktion bei Erreichen eines Zustands (entry action)*

Arten von Ereignissen

- „Übliche“ Arten von Ereignissen:
  - Empfang einer Nachricht von außen
  - Ablaufen einer Zeitbedingung *(time-out)*
  - Veränderung einer (überwachten) Bedingung *(change event)*

- Spezielle Arten von Ereignissen für einzelne Objekte:
  - Eintreffen einer Nachricht bei einem Objekt als
    - Aufruf einer Operation (Methode)
    - Signal (ohne „empfangende“ Methode zu nennen)
  - Erzeugen oder Löschen des Objekts

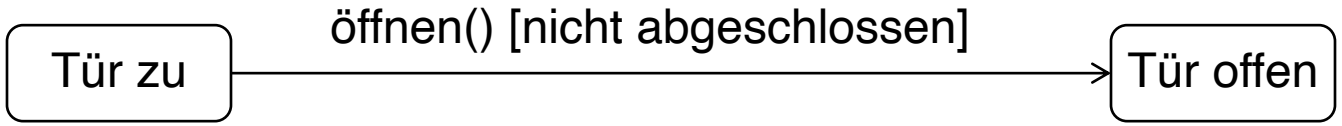
- Detaillierungsgrad in der Analysephase nutzt anwendungsbezogene, keine technischen Ereignisse.

Bedingungen

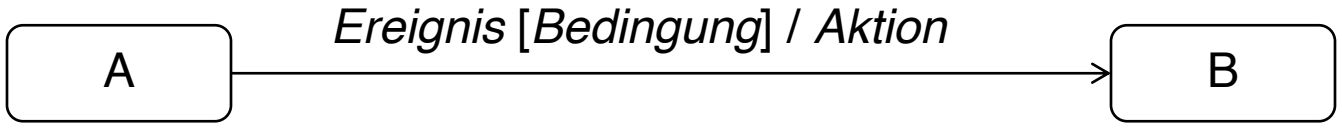
- Definition: Eine ***Bedingung*** *(guard)* ist eine Boolesche Bedingung, die zusätzlich bei Auftreten des Ereignisses erfüllt sein muss, damit der beschriebene Übergang eintritt.

- Notation: Eine Bedingung wird in der Analysephase meist noch textuell beschrieben. In formalerer Beschreibung (v.a. im Entwurf) kann eine Bedingung folgende Informationen verwenden:
  - Parameterwerte des Ereignisses
  - Attributwerte und Assoziationsinstanzen (Links) der Objekte
  - ggf. Navigation über Links zu anderen Objekten

- *Beispiel:*



Aktionen



- Definition: Eine ***Aktion*** ist die Beschreibung einer ausführbaren Anweisung, wobei angenommen wird, dass die Dauer der Ausführung vernachlässigbar ist. Aktionen sind nicht unterbrechbar.
  Eine Aktion kann auch eine Folge von Einzelaktionen sein.

- Typische Arten von Aktionen:
  - Lokale Änderung eines Attributwerts
  - Versenden einer Nachricht an ein anderes Objekt
  - Erzeugen oder Löschen eines Objekts

- Eine Aktion wird in der Analysephase meist textuell beschrieben. Als „Aktionssprache“ kann aber auch Java verwendet werden.

Verwendung von UML-Zustandsmodellen

- zur Steuerung:
  - Für steuernde Systeme, eingebettete Systeme etc.
  - Ereignisse sind Signale der Umgebung oder anderer Systemteile
  - Reaktion in gegebenem Zustand auf ein bestimmtes Signal:
    - neuer Zustand
    - ausgelöste Aktion (wie im Zustandsmodell spezifiziert)
  - Ähnlich zum Konzept von Mealy-Automaten
  - Zustandsmodelle definieren die *Reaktion* auf mögliche Ereignisse

- als Protokolle (oder Objekt-Lebenszyklen):
  - Für Informationssysteme, Datenbankanwendungen etc.
  - Ereignisse sind Operationsaufrufe
  - Reaktion in gegebenem Zustand auf bestimmten Aufruf:
    - durch Methodenrumpf gegeben (komplex)
    - keine Aktionen im Zustandsmodell
  - Zustandsmodelle definieren zulässige *Reihenfolgen* von Aufrufen

Beispiel: Zustandsmodell zur Steuerung (1)

discuss

Kaffeeautomat

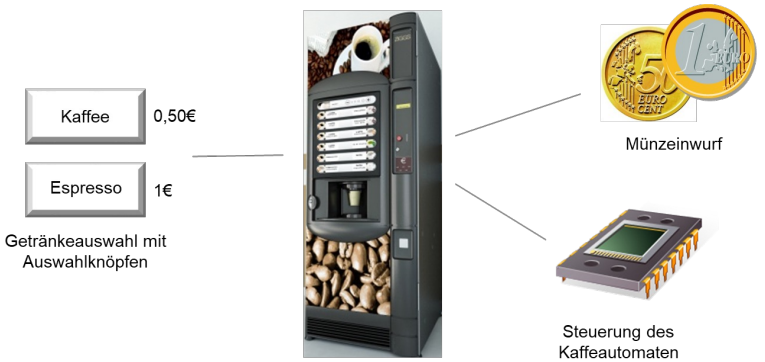
Aufgabe: Zustandsmodell entwickeln, das Zustände des Kaffeeautomaten und Interaktionen mit dem Benutzer beschreibt.

Beispiel: Zustandsmodell für den Gesamtautomat (2)

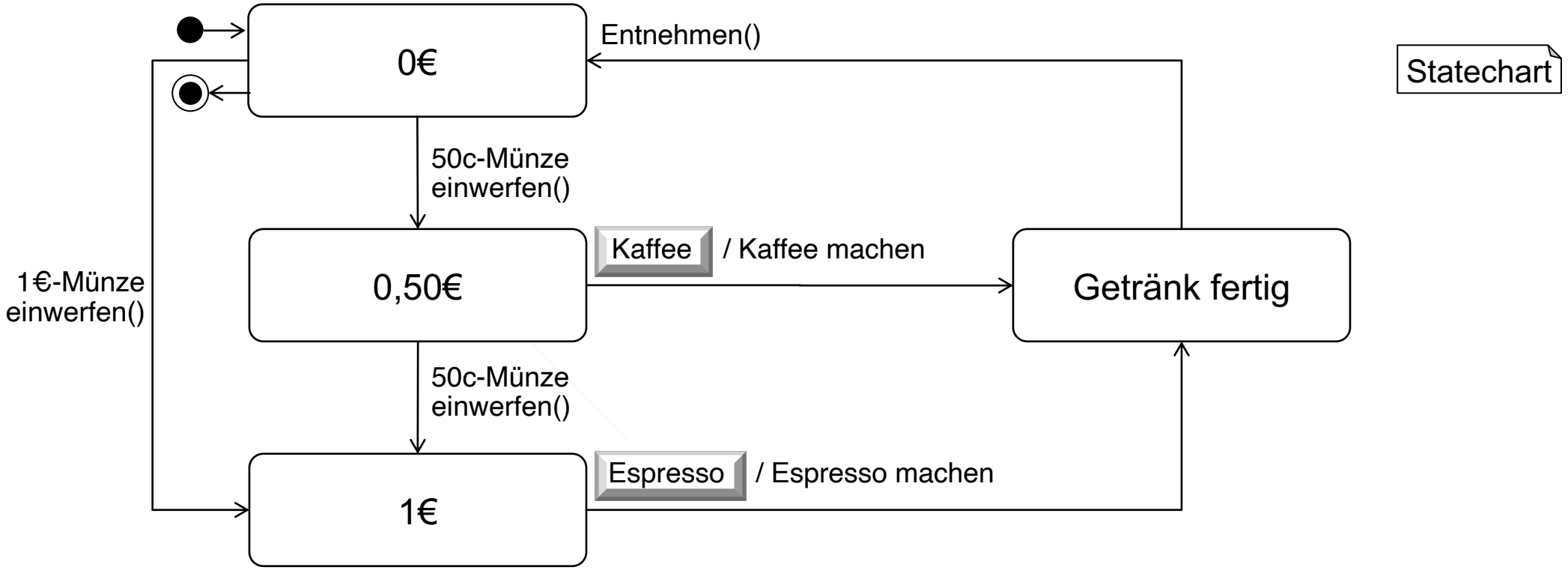
Statechart

⇦

discuss



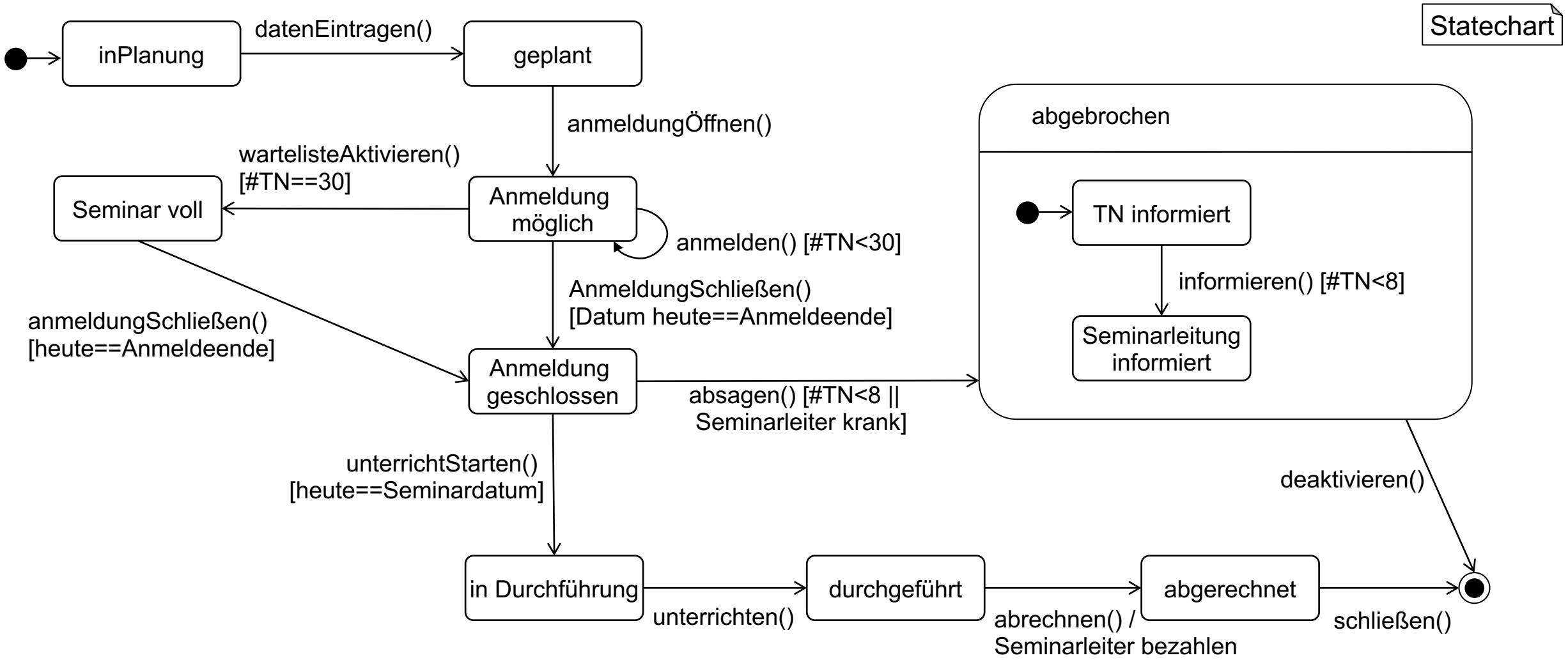
Beispiel: Zustandsmodell für den Gesamtautomat (2)



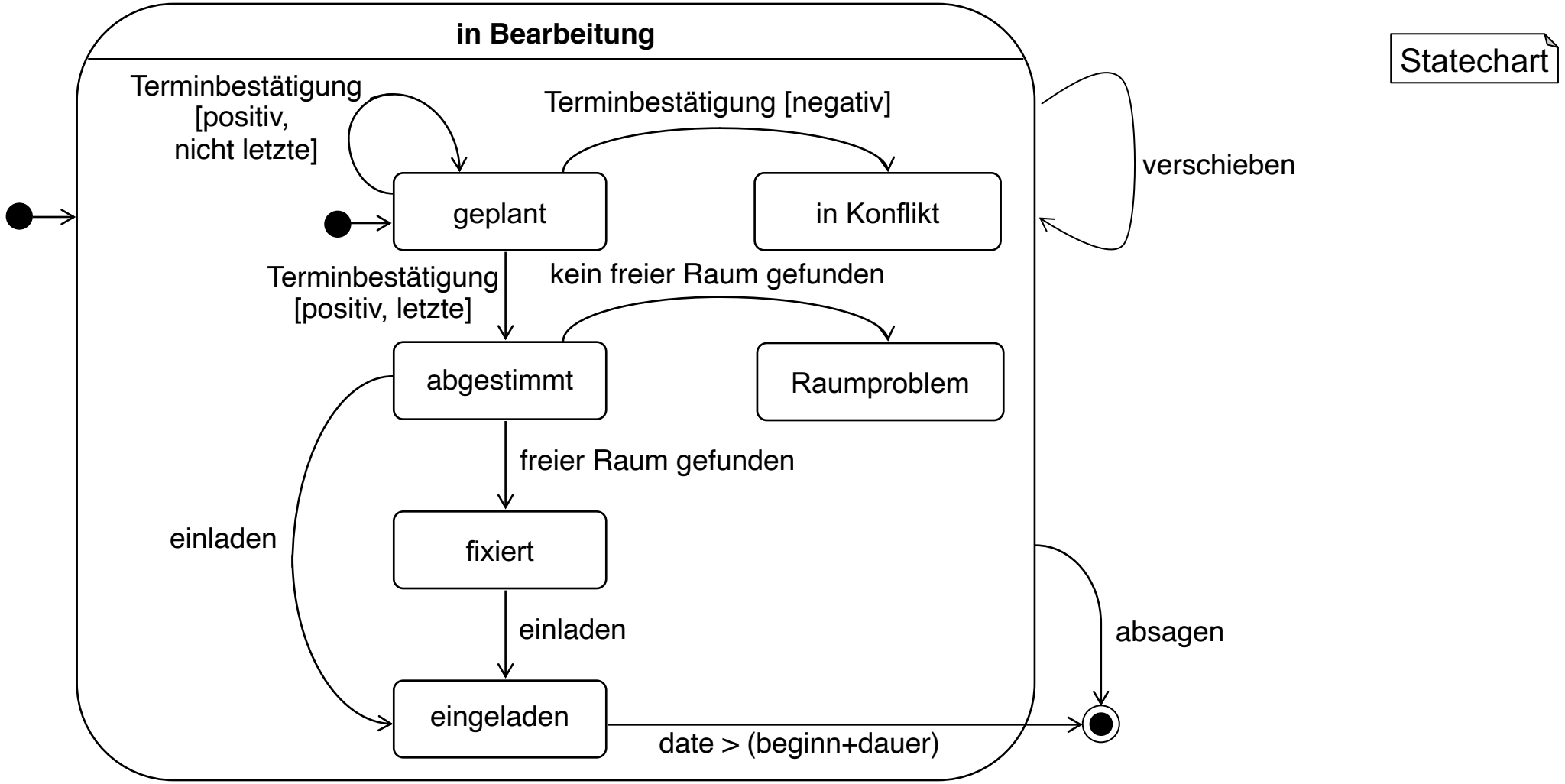
Was passiert, wenn:

- Im Zustand **0,50€** eine 1€-Münze eingeworfen wird?
- Im Zustand **0€** der Kaffee-Knopf gedrückt wird?
- Im Zustand **1€** der Kaffee-Knopf gedrückt wird?

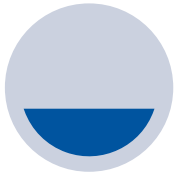
Beispiel Seminarplanung: Zustände eines Seminars



Objektlebenszyklus der Klasse "Teambesprechung"



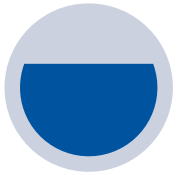
Was haben wir gelernt?



Objekte…

…können diverse Zustände haben

Systemverhalten ist abhängig von Zuständen

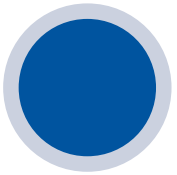


Statecharts

Konzepte: Zustände und Zustandsübergänge (Transitionen)

Zustandshierarchie, Ein-/Ausgabe

Vollständige Verhaltensbeschreibung (im Gegenteil zu Sequenzdiagrammen)



Verwendung

Modell einer Steuerung

Protokollbeschreibung

Objekt-Lebenszyklus

Vorlesung Softwaretechnik

5. Muster in der Softwareentwicklung

Prof. Bernhard Rumpe
Software Engineering
RWTH Aachen

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

SE Software Engineering | RWTH AACHEN UNIVERSITY

Warum, was, wie und wozu? Muster in der Analyse

Warum?

- Bessere Softwarestrukturen erstellen
- Effizienz, Wiederverwendung

Was?

- Was ist ein Muster?
- Muster in der Analyse
- Betrachten 7 Muster genauer

Wie?

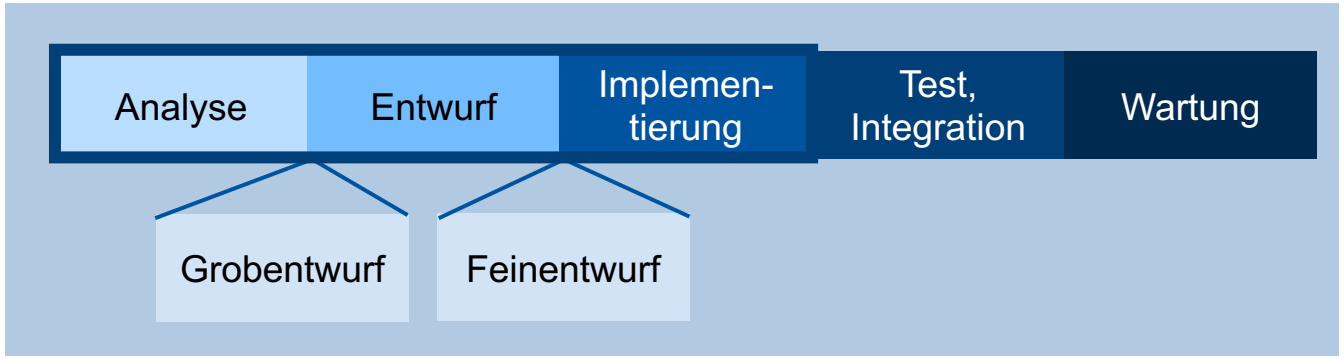
- Problem & Lösung
- Abstraktes Prinzip & Beispiel

Wozu?

- Struktur in die Software bekommen
- Bessere Basis fürs Design

Softwaretechnik

5. Muster
5.1. Muster-Begriff



Prof. Bernhard Rumpe
Software Engineering
RWTH Aachen

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

Literatur:

- Gamma/Helm/Johnson/Vlissides: Design Patterns, Addison-Wesley 1994 *(= „Gang of Four“, „GoF“)*
- Buschmann/Meunier/Rohnert/Sommerlad/Stal: A System of Patterns, Wiley 1996

Muster für besseren Entwurf / Analyse / Architektur

- Ein ***Muster*** ist eine schematische Lösung für eine Klasse verwandter Probleme. Muster beschreiben Erfahrungswissen.

- Darstellung eines Musters (nach *GoF*)
  - Name, evtl. Synonyme
  - Problem
    - Motivation, Anwendungsbereich
  - Lösung(en)
    - Struktur (Klassendiagramm)
    - Bestandteile (schematische Klassen- und Objektnamen)
    - Beschreibung, evtl. Abläufe, z.B. Sequenzdiagramm
  - Diskussion
    - Vor- und Nachteile, Abhängigkeiten, Einschränkungen
  - Beispielanwendungen
  - Verwandte Muster (Ähnlichkeiten)

Beispiel: Text zu „Adapter“ (1)

Ausschnitt aus „Design Patterns“:

ADAPTER

Also known as: Wrapper

Motivation:

Sometimes a toolkit class that's designed for *reuse* isn't reusable only because its *interface* doesn't match the domain-specific interface an *application* requires.
Consider for example a drawing editor that lets users draw and arrange graphical elements (lines, polygons, text etc) into pictures and diagrams.
... (eine Seite Text & Beispiel)

Applicability:

Use the Adapter pattern when

- you want to use an existing class, and its interface does not match the one you need.
- you want to create a reusable class that cooperates with unrelated or unforeseen classes ...

Beispiel: Text zu „Adapter“ (2)

Ausschnitt aus „Design Patterns“:

ADAPTER (Fortsetzung)

Structure: (Zwei Klassendiagramme + weitere Information)

Consequences: (1 Seite Text)

Implementation: (2 Seiten Text mit Klassendiagrammen)

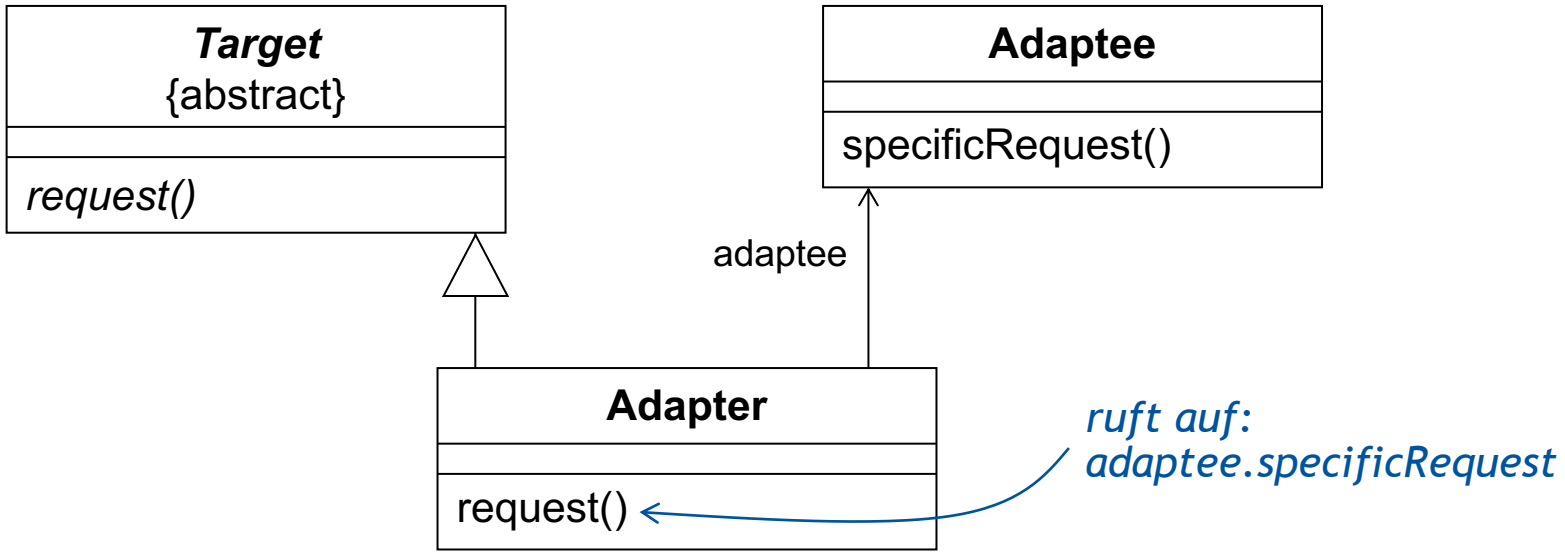
Sample Code: (3 Seiten Text mit C++-Beispielen)

Known Uses: (1 Seite Text; Beispiele)

Related Patterns: (1/4 Seite Text; konkrete Patterns & Unterschiede)

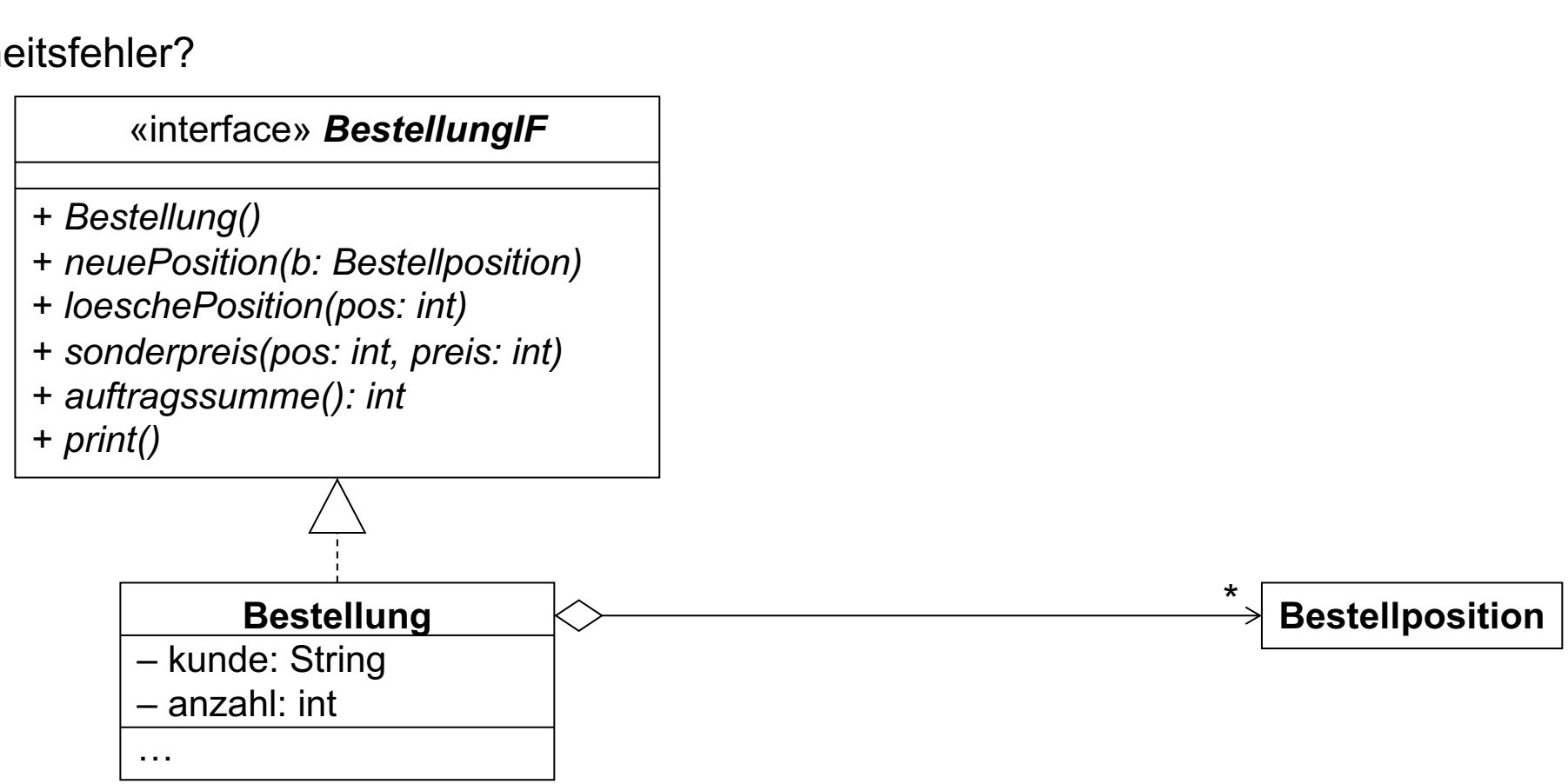
Strukturmuster Adapter | Variante 1: Objektadapter

- Name: **Adapter** (auch: **Wrapper**)
- *Problem*:
  - Anpassung der Schnittstelle eines vorgegebenen Objekts (adaptee) auf eine gewünschte Schnittstelle (target)
- *Lösung*:



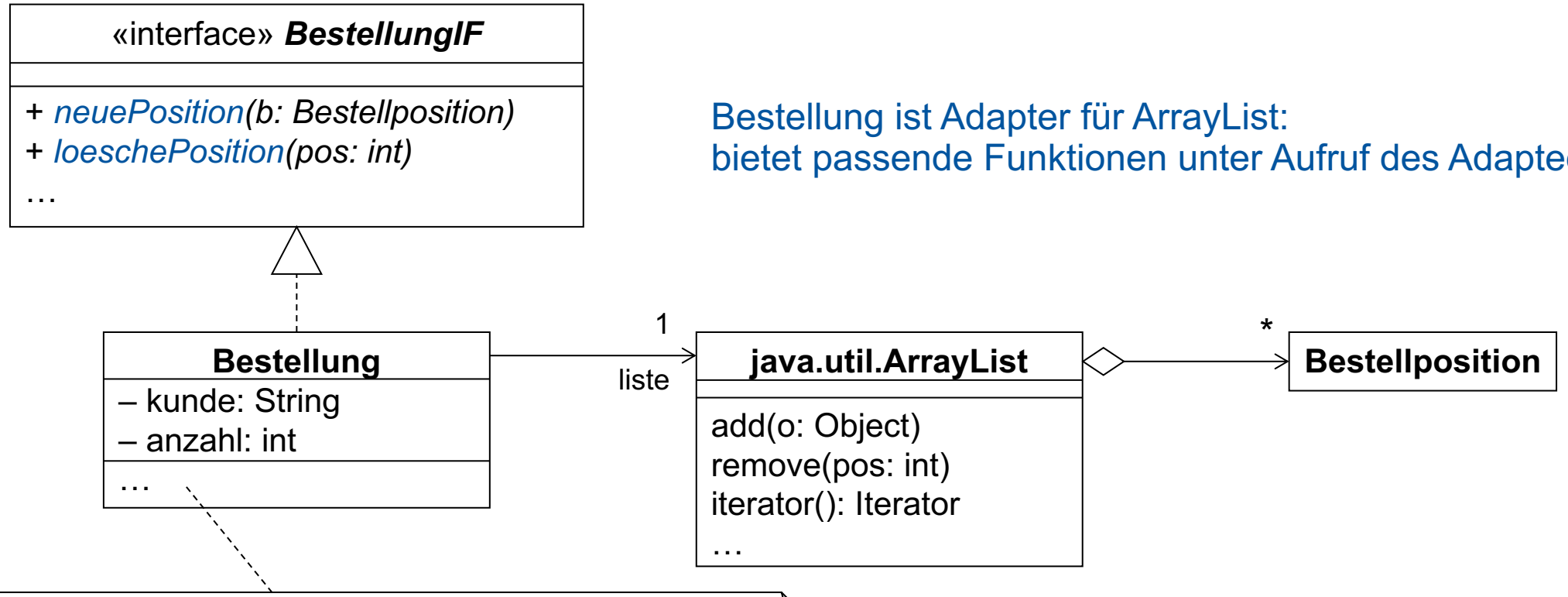
Objektadapter-Beispiel (1)

- Schönheitsfehler?



⇦ discuss

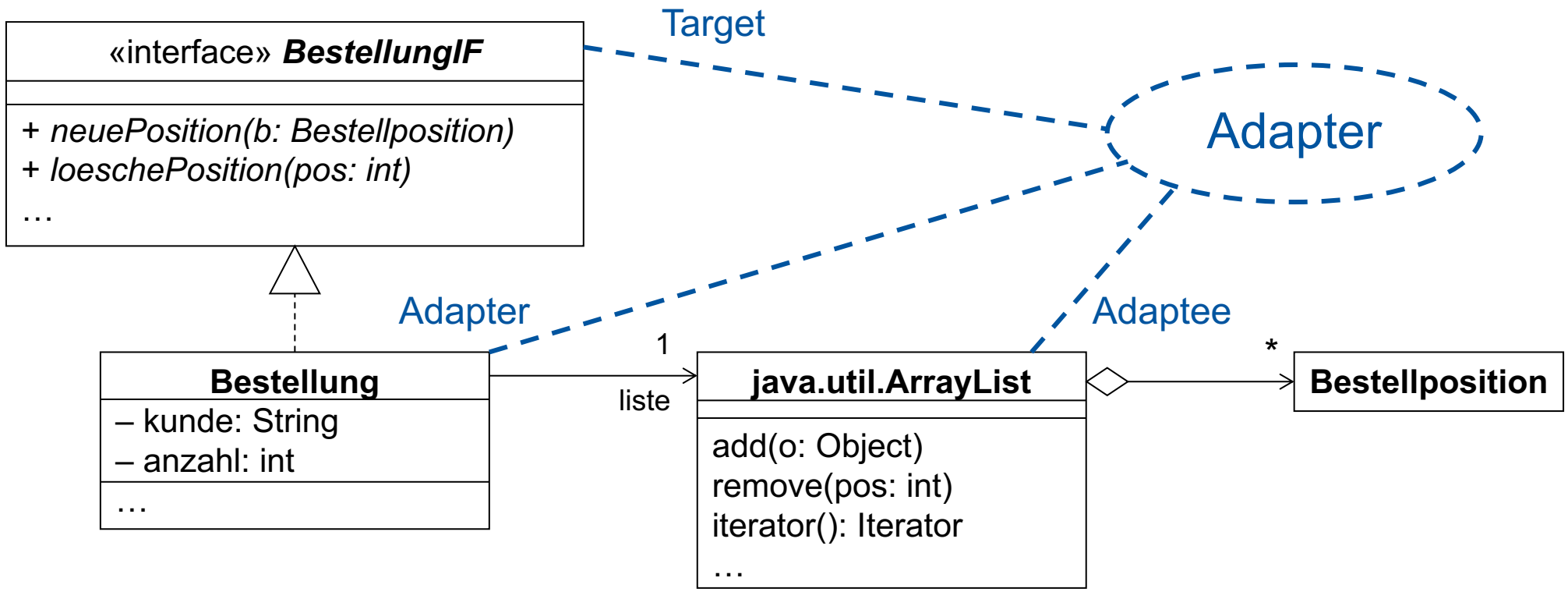
Objektadapter Beispiel



Bestellung ist Adapter für ArrayList:
bietet passende Funktionen unter Aufruf des Adaptee

```
public void neuePosition (Bestellposition b) {
    liste.add(b);
}
public void loeschePosition (int pos) {
    liste.remove(pos);
}
```

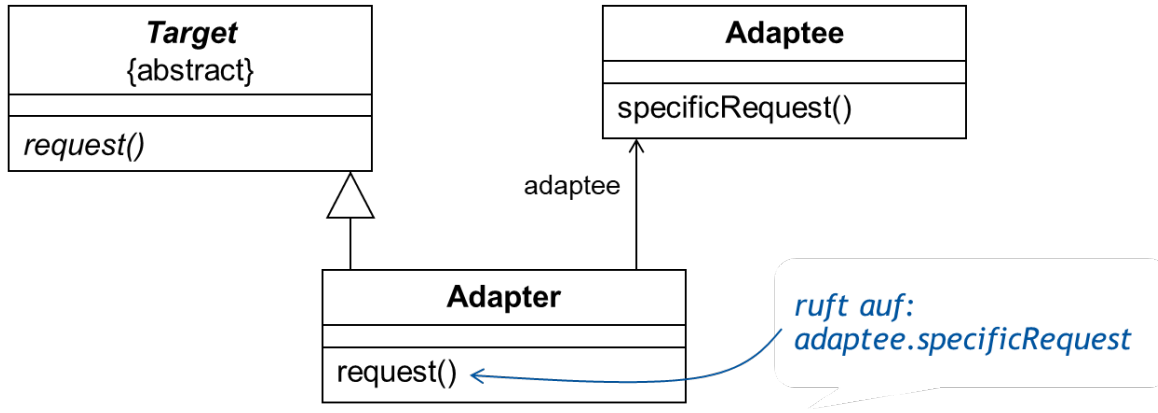
Objektadapter-Beispiel in UML (3)



Anwendung eines Musters

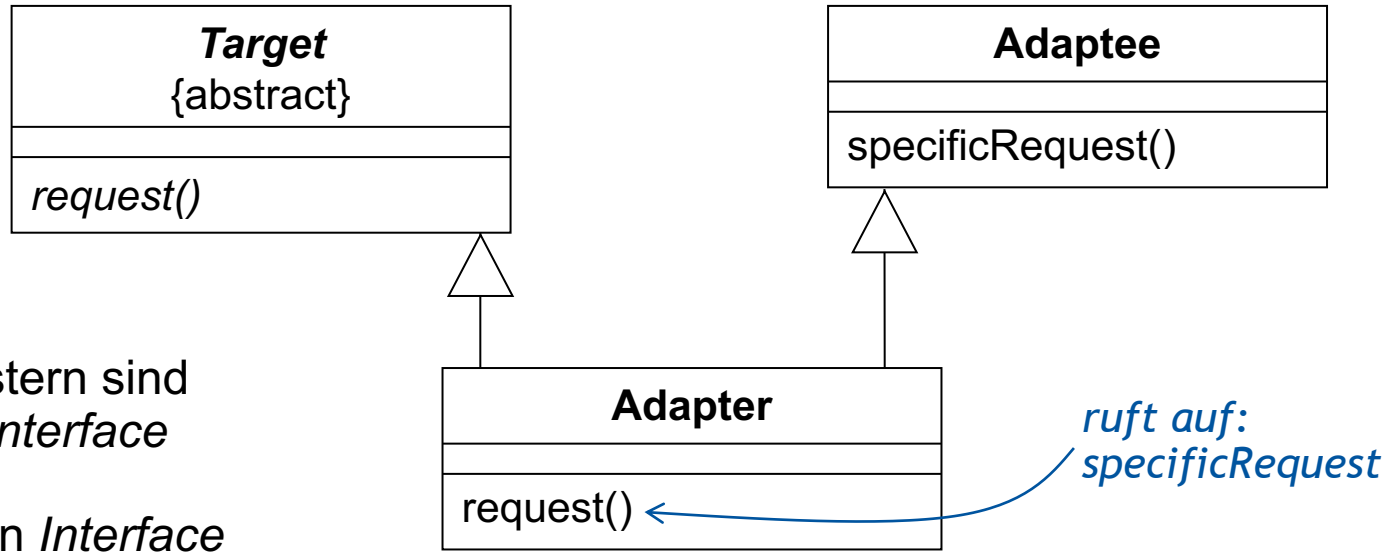
- Kein mechanisches „Pattern Matching“!
  - eher Übertragung der Idee des Musters
- Grundstruktur des Musters sollte sich wiederfinden lassen
  - ggf. vorliegenden Entwurf etwas anpassen bzw. anders darstellen
- Auch Verhaltensschema muss im Code ähnlich zur Musteridee auftreten:
  - *Muster:*
    Adapter.request() ruft Adaptee.specificRequest() auf

    (und tut nicht wesentlich mehr)
  - *Konkreter Fall:*
    Bestellung.neuePosition() ruft ArrayList.add() auf
    Bestellung.loeschePosition() ruft ArrayList.remove() auf
    ...
- Intension/Zweck des Musters ist zu beachten



Strukturmuster Adapter | Variante 2: Klassenadapter

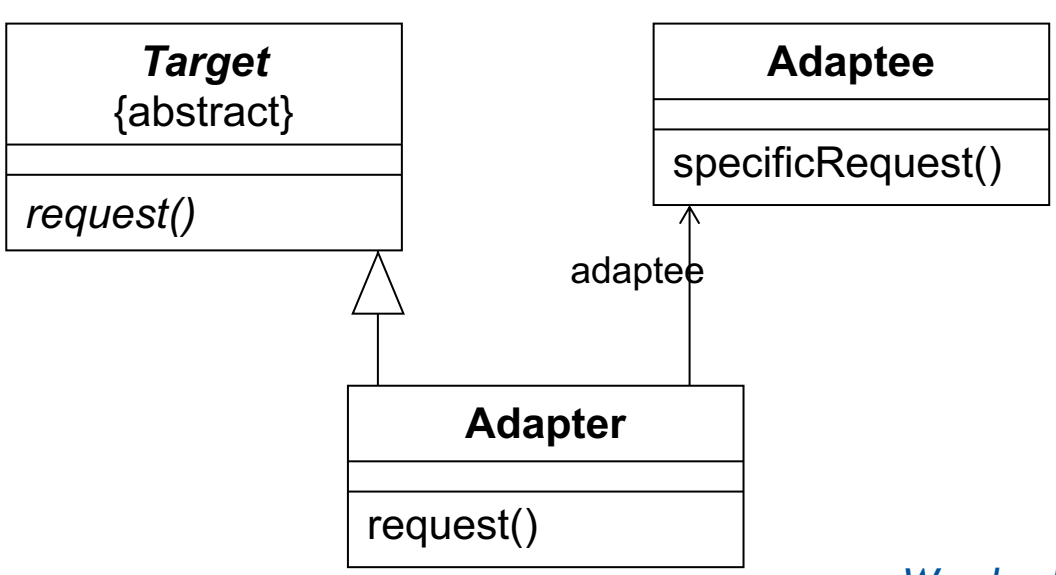
- Name: **Adapter** (auch: **Wrapper**)
- *Problem*:
  - Anpassung der Schnittstelle eines vorgegebenen Objekts (adaptee) auf eine gewünschte Schnittstelle (target)
  - Viele Operationen sind identisch in Adaptee und Target haben aber unterschiedliche Namen oder Argumentreihenfolgen
- *Lösung*:



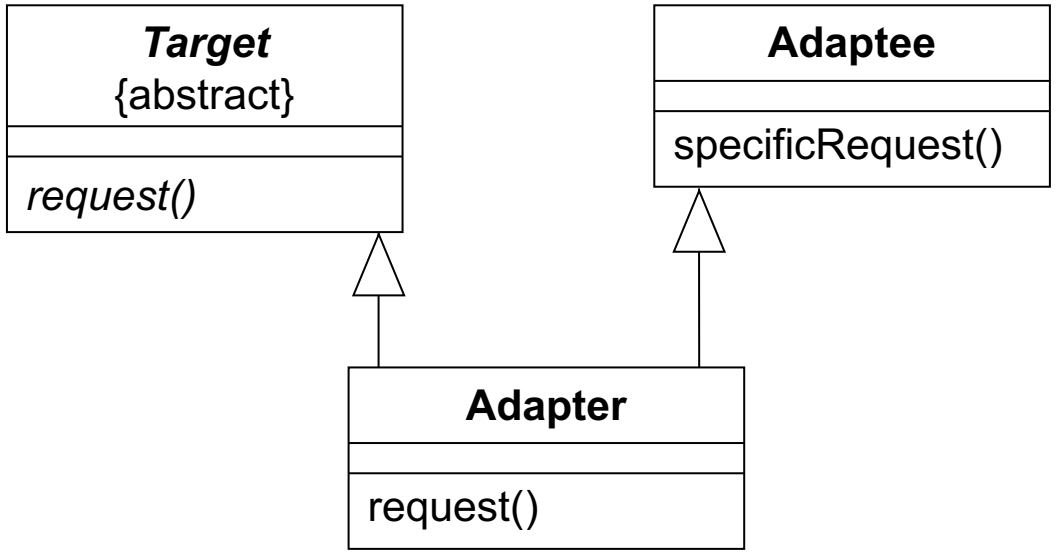
Beachte: In vielen Mustern sind *abstrakte Klasse und Interface* synonym.
in Java muss *Target* ein *Interface* sein (wg. Mehrfachvererbung).

Strukturmuster Adapter | Vergleich beider Varianten

Variante 1:
Objektadapter

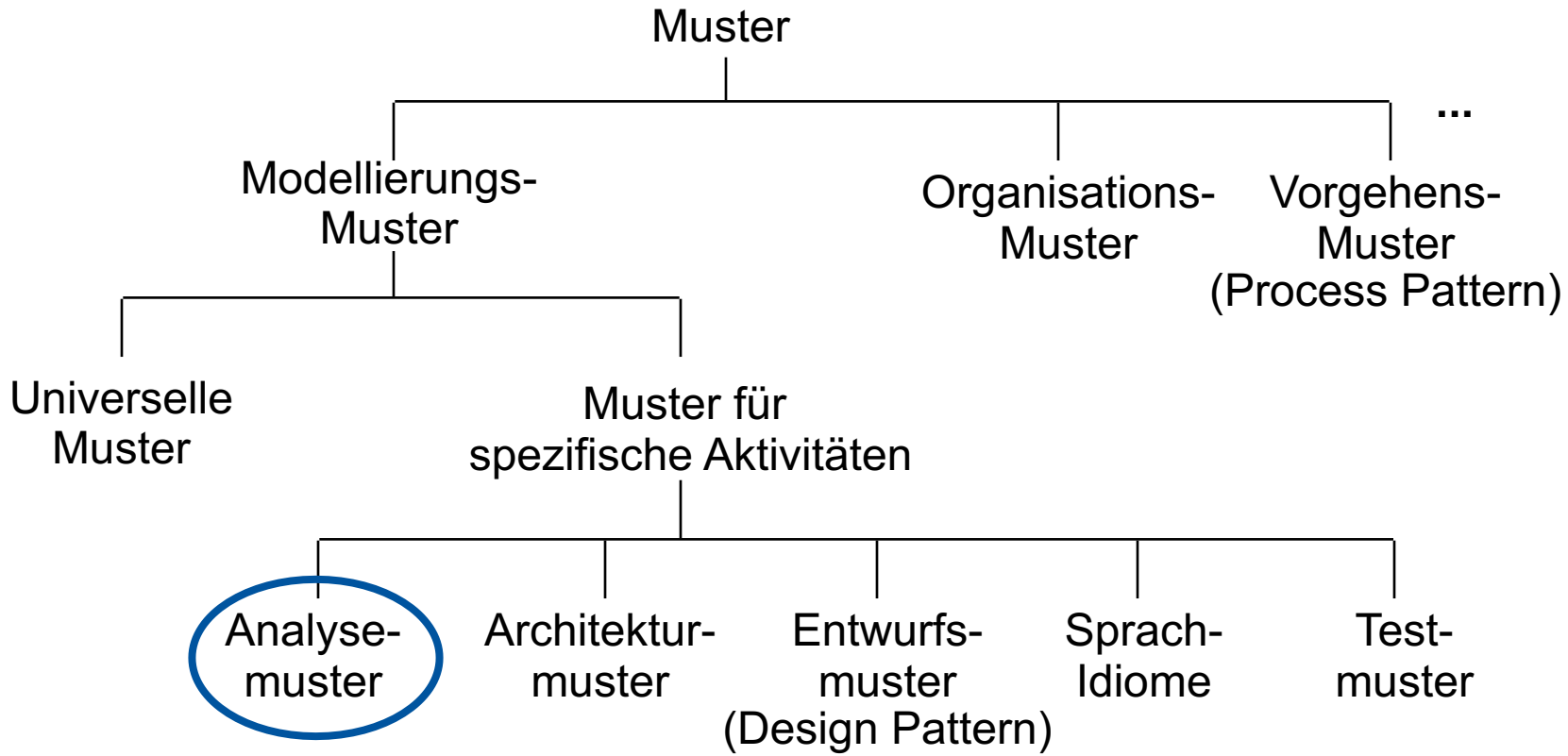


Variante 2:
Klassenadapter



Was bedeutet der Unterschied?
Auswirkungen?

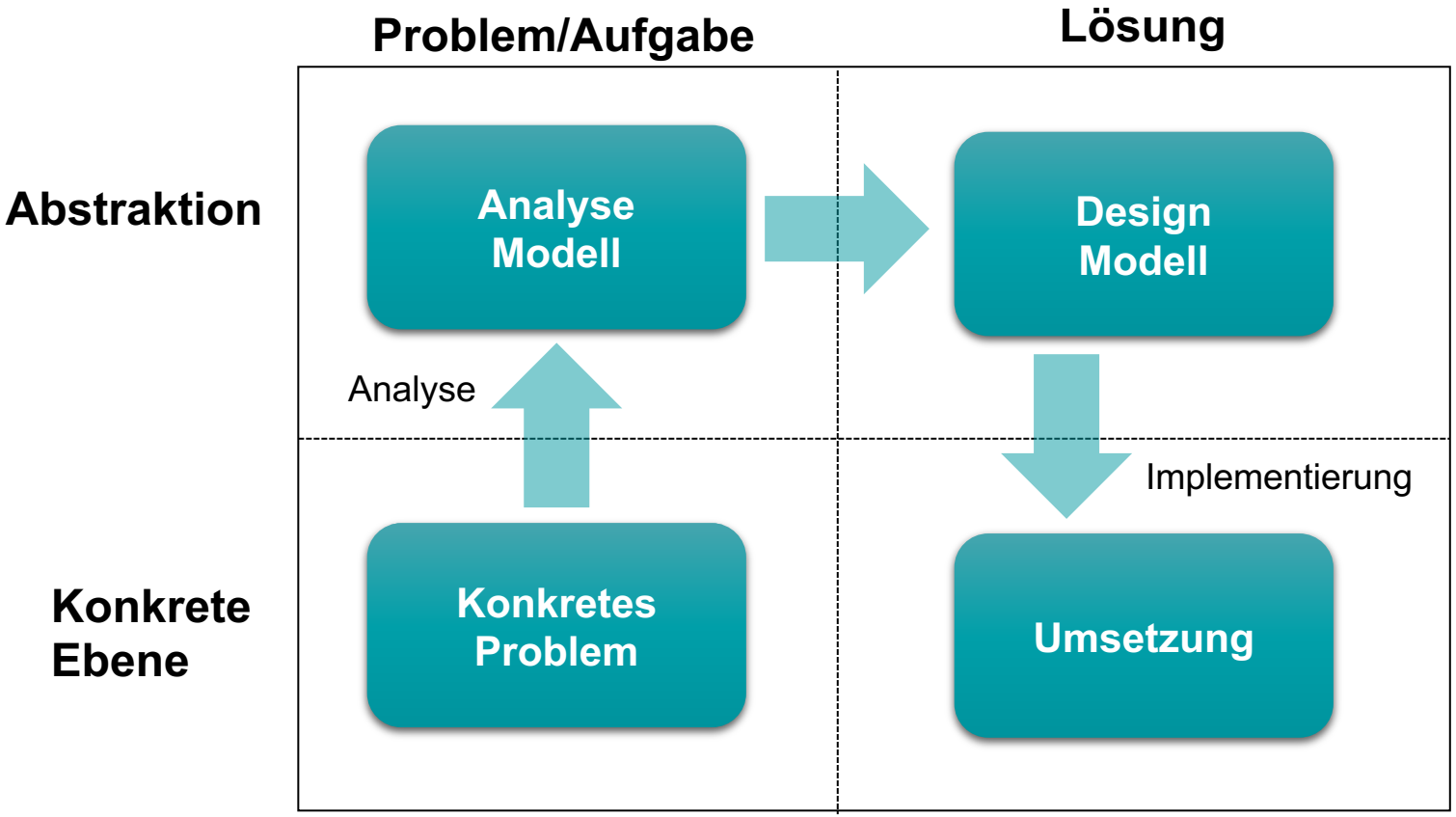
Arten von Mustern



- Weitere Unterteilung:
  - Allgemein anwendbare Muster (z.B. Komposition)
  - Domänenspezifische Muster (z.B. Kontostruktur im Finanzbereich)

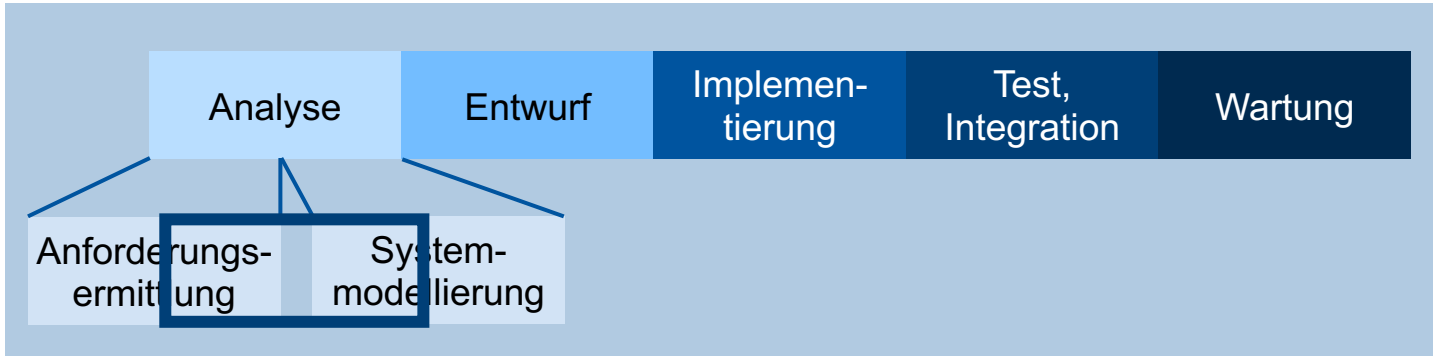
Vom Problem zur Lösung

- Analyse Patterns
  - helfen eine Struktur in die vorliegenden Informationen zu bringen
- Design Patterns
  - Hilfestellung für konkrete Implementierung



Softwaretechnik

5. Muster
5.2. Muster in der Objektorientierten Analyse



Prof. Bernhard Rumpe
Software Engineering
RWTH Aachen

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

Literatur:

- Balzert Band I, LE 13
- Martin Fowler: Analysis Patterns 1997
- Scott Ambler: Building Object-Oriented Applications That Work, SIGS Books 1998 (Kap. 4)

SE Software Engineering | RWTH AACHEN UNIVERSITY

Analyse Muster

- Modellierung häufiger vorkommender Strukturen
- Beschreibung einer Gruppe von Klassen oder Objekten mit
  - Festen Verantwortlichkeiten
  - Definierten Beziehungen
  - Definierten Interaktionen
- Beschreiben die Lösung für eine typische Teilaufgabe bei der Erstellung des fachlichen Modells
- Beantworten die Frage:
  - Wie modelliere ich diese Klassenbeziehung?

Einige Muster

Universelle/ Allgemeine Muster

- *Heide Balzert 99, Balzert 96*
  - Liste
  - Exemplartyp
  - Baugruppe
  - *Stückliste (Composite)*
  - *Koordinator*
  - *Rollen*
  - *Wechselnde Rollen*
  - Historie
  - Gruppe
  - *Gruppenhistorie*
- *Coad et al. 95*
  - *Item-Item Description (Exemplartyp)*

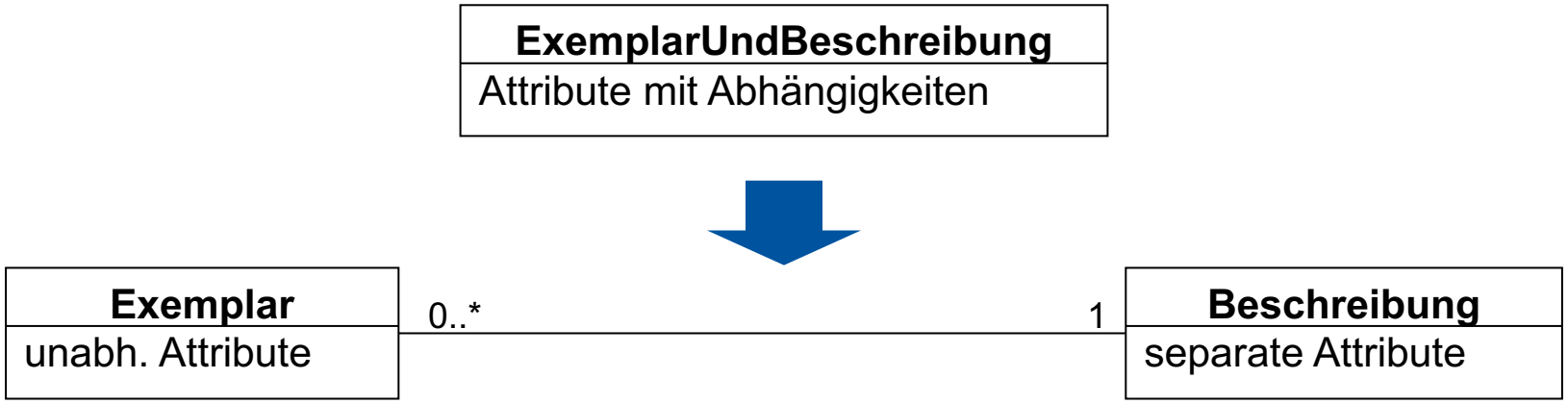
Spezifische Analysemuster/ Anwendungsspezifische Muster

- *Fowler 97*
  - *Party (Juristische Person)*
  - Account
  - Transaction
  - Contract
  - Portfolio
  - Money
  - …

Muster: Exemplar und Beschreibung

Universelles Muster (nach Coad et al. 95)

- **Name**: Item-Item Description (Exemplartyp)
- **Problem**: Manche Attribute nehmen bei vielen Instanzen immer wieder die gleichen Kombinationen von Werten an (Attribut-Abhängigkeit).
- **Lösung**: Einführung einer neuen Klasse und einer Aggregation.



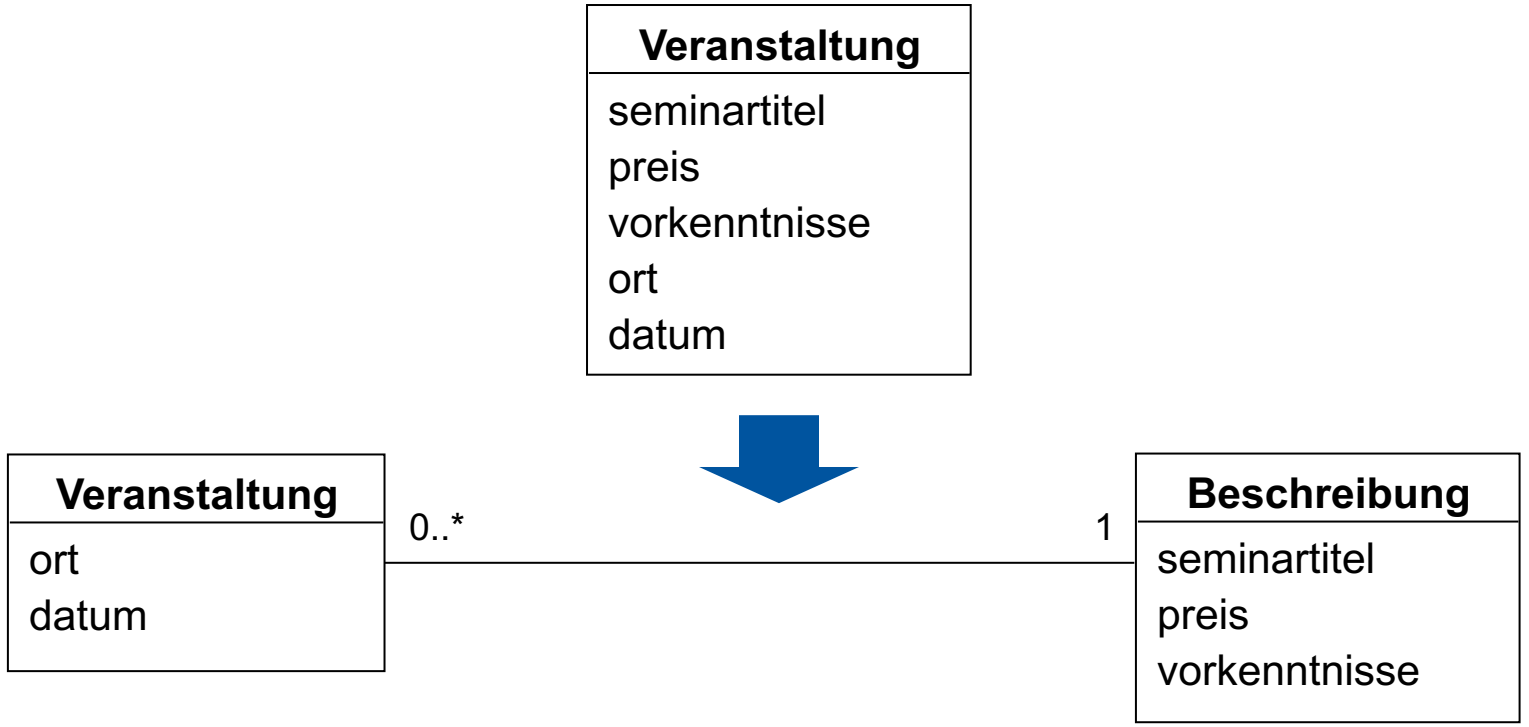
- vgl. dieses Muster mit *Normalisierung von relationalen DB-Modellen*

Beispiel: Transformation mit Mustern

- Typische Anwendungsform für Muster („Refactoring“):
  - Finden verbesserungswürdiger Strukturen in Modellen
  - Ersetzen durch besser strukturierte Fassung

animated

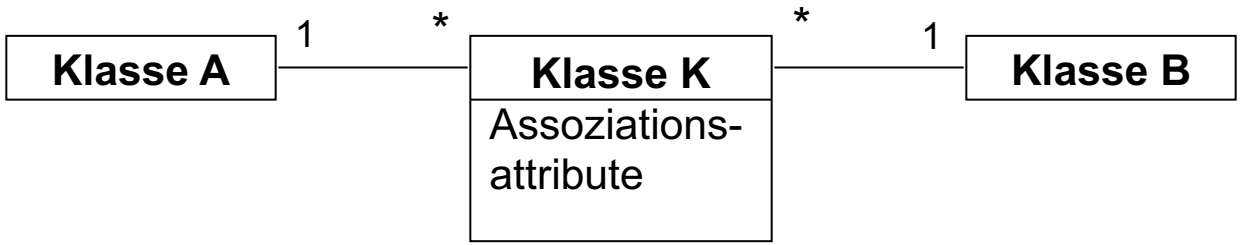
discuss



Muster: Koordinator

Universelles Muster (nach Heide Balzert 99, Balzert 96)

- **Problem:** Eine Assoziation besitzt Attribute, die zu keiner der beteiligten Klassen gehören.
- **Lösung:** Einführung einer eigenen Koordinator-Klasse



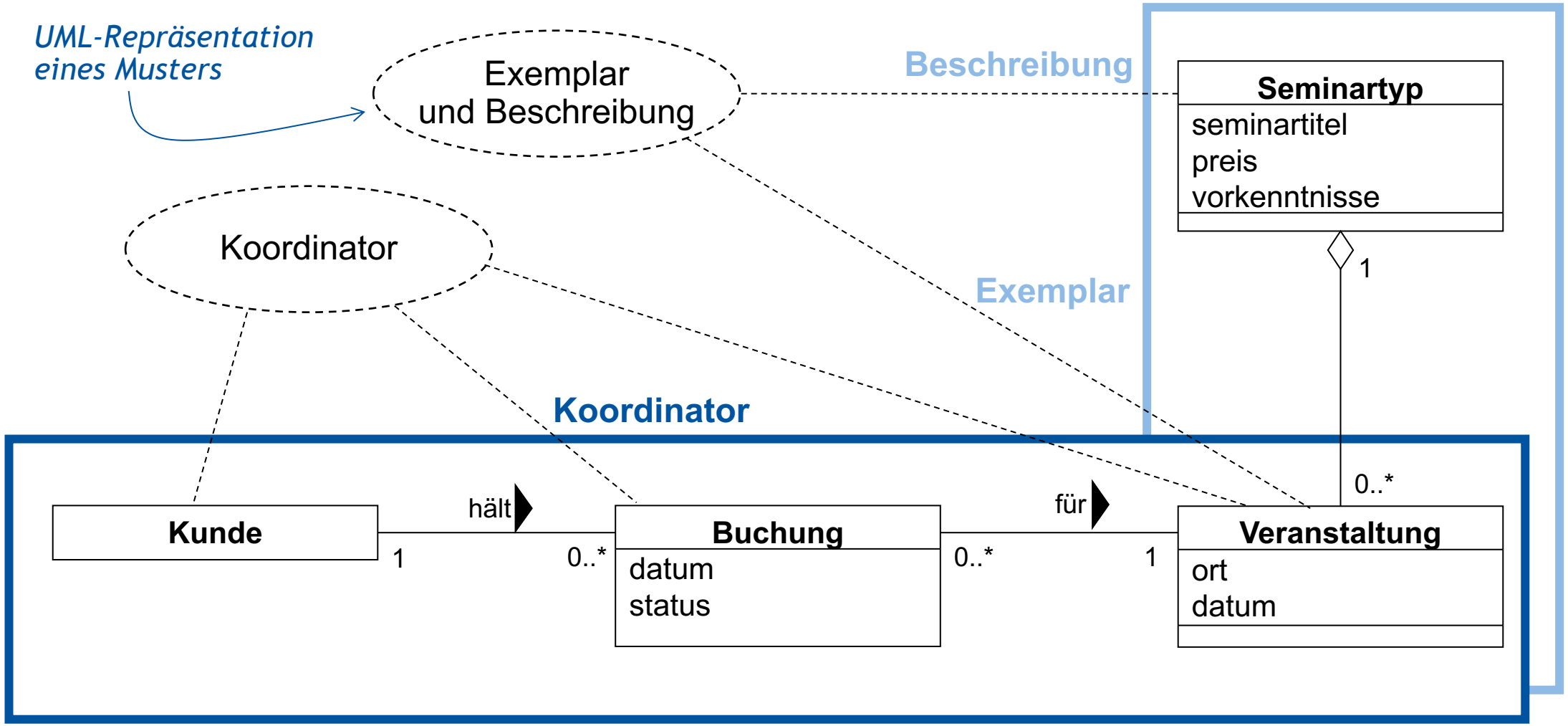
Beispiel: Koordinator

Buchungsdatum ?
Buchungsstatus ?

⇦ discuss

- Ein wesentliches Merkmal dieses Musters ist die beidseitige Multiplizität „0..*“, die es verhindert Attribute der einen oder anderen Klasse zuzuordnen.
- Typisch für Koordinatorklassen ist, dass sie selbst nur wenige Attribute und Operationen besitzen, und mit mehreren Klassen in Verbindung stehen.

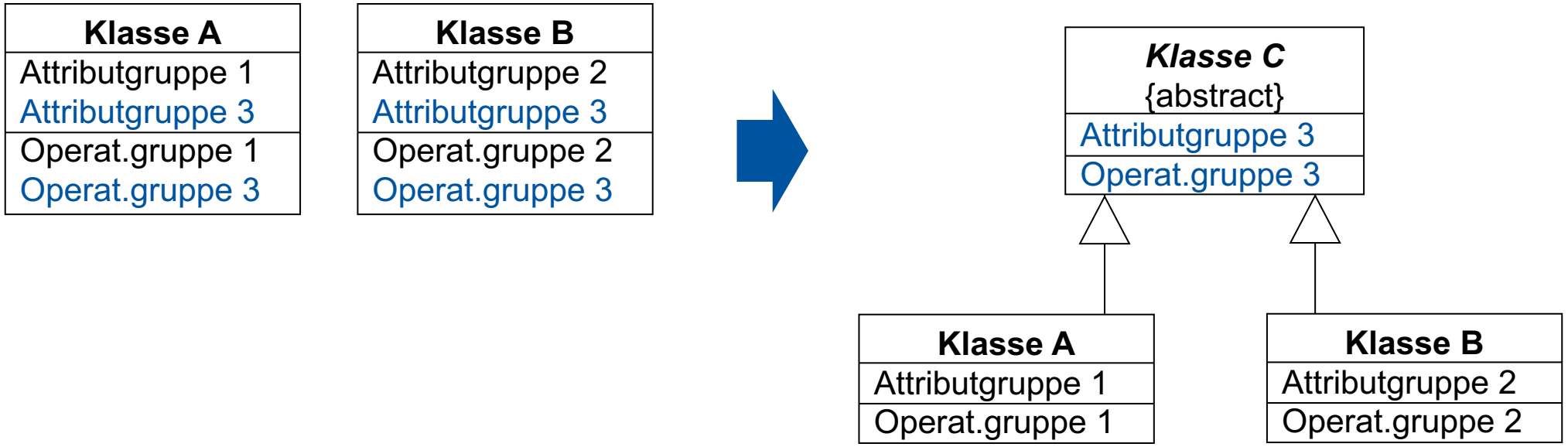
Kombination von Musteranwendungen



Muster: Abstrakte Oberklasse

Universelles Muster (nach Balzert 96)

- **Problem**: Klassen enthalten Gruppen identischer Attribute und Operationen.
- **Lösung**: Separieren der identischen Bestandteile in einer abstrakten Oberklasse.



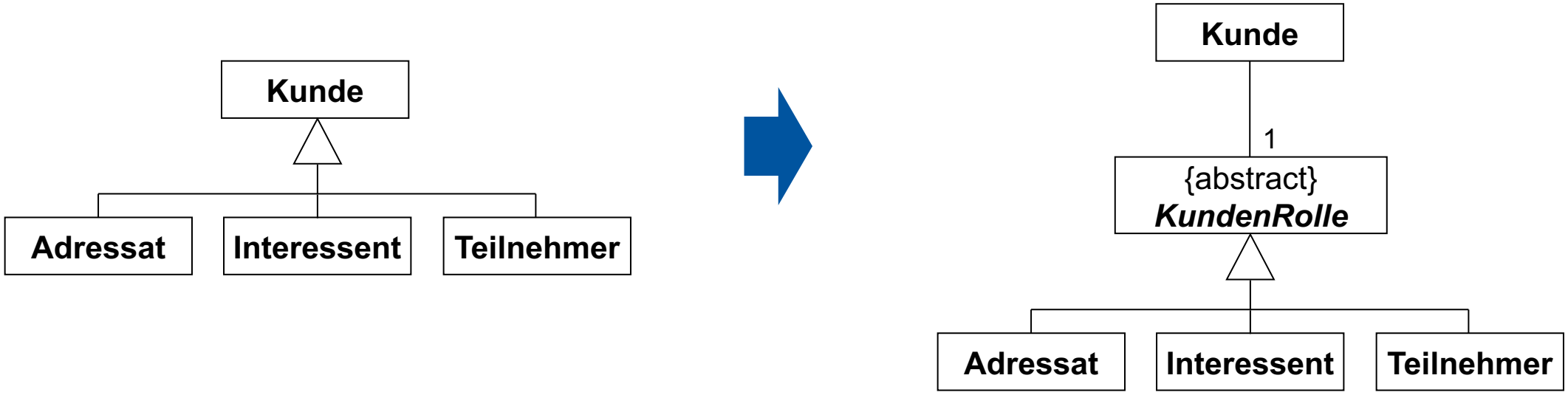
Muster: Wechselnde Rolle

Universelles Muster (nach Heide Balzert 99)

- **Problem**: Ein Objekt einer Unterklasse A1 kann in eine Unterklasse A2 wechseln.
- **Lösung**: Einführung einer abstrakten Rollenklasse.
- Variante des Musters „Rolle“ mit dynamischem Auswechseln des Rollen-Objekts

Beispiel: Wechselnde Rolle

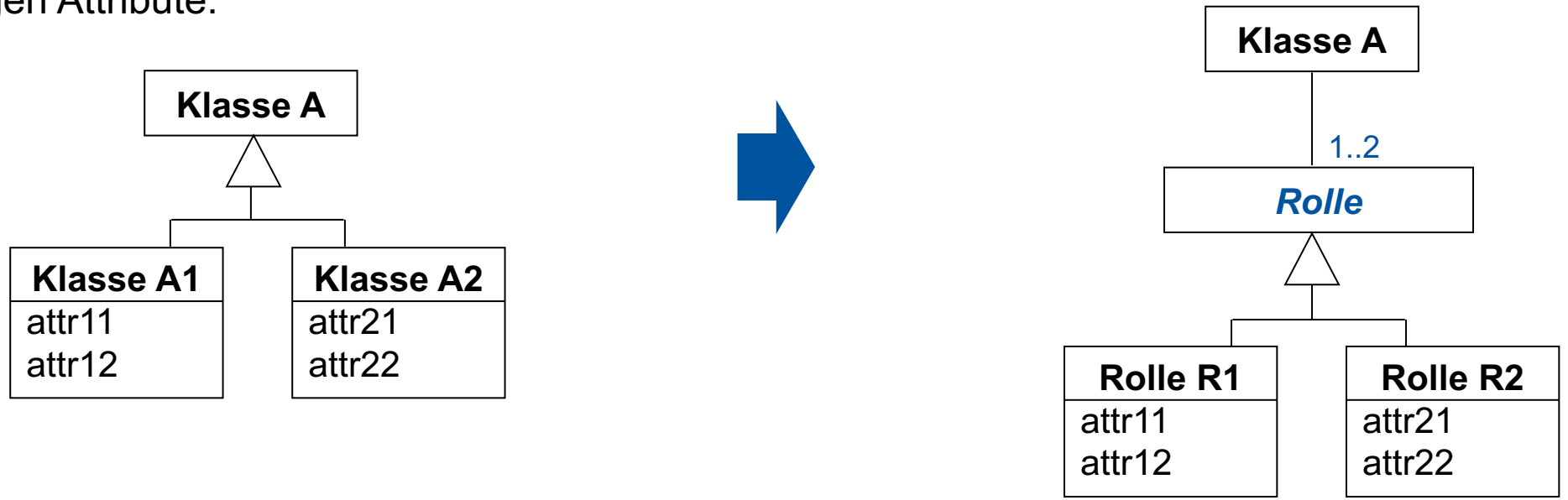
- **Lösung**: Einführung einer abstrakten Rollenklasse.



Muster: Rollen

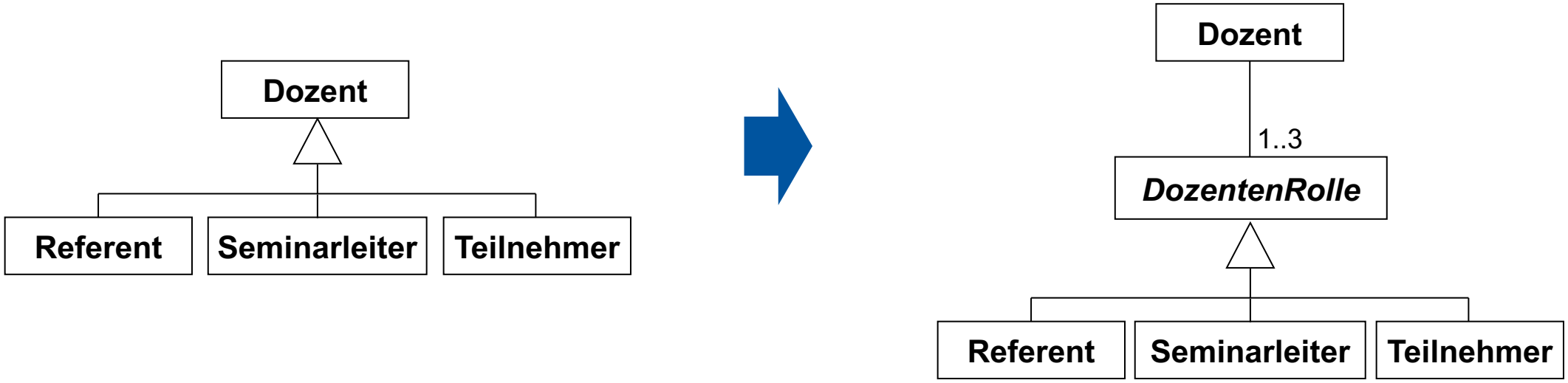
Universelles Muster (angelehnt an Heide Balzert 99)

- **Problem**: Es gibt mehrere Einsatzstellen/-arten einer Klasse. Unterklassenbildung würde „überlappende“ Vererbung und damit Mehrfachzugehörigkeit von Objekten in Klassen erfordern.
- **Lösung**: Einführung einer „Rollen“-Klasse mit Multiplizität entsprechend der Einsatzarten. Rollen enthalten die notwendigen Attribute.



Beispiel: Rollen

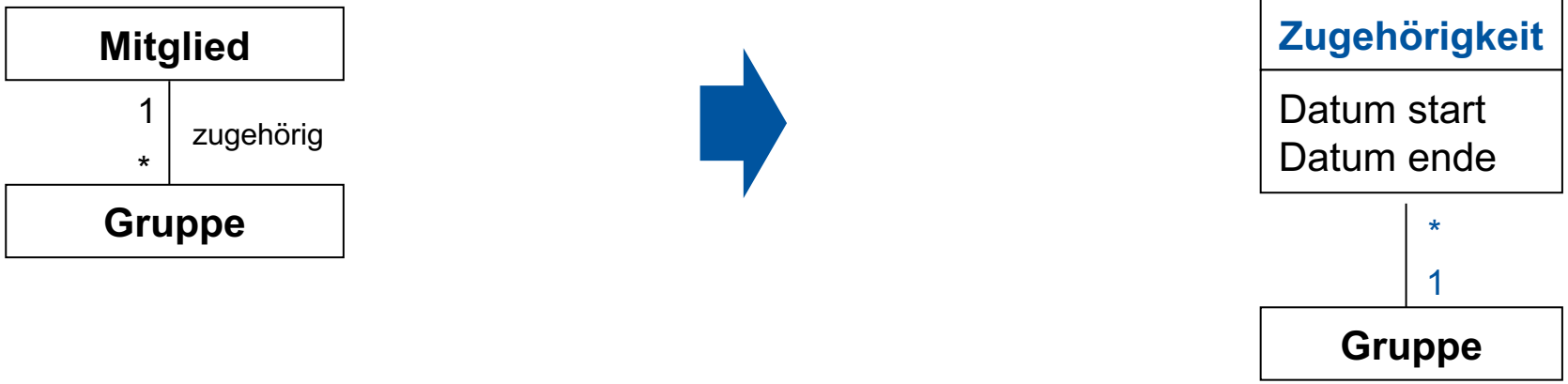
- **Lösung**: Einführung einer „Rollen“-Klasse mit Multiplizität entsprechend der Einsatzarten. Rollen enthalten die notwendigen Attribute.



Muster: Gruppenhistorie

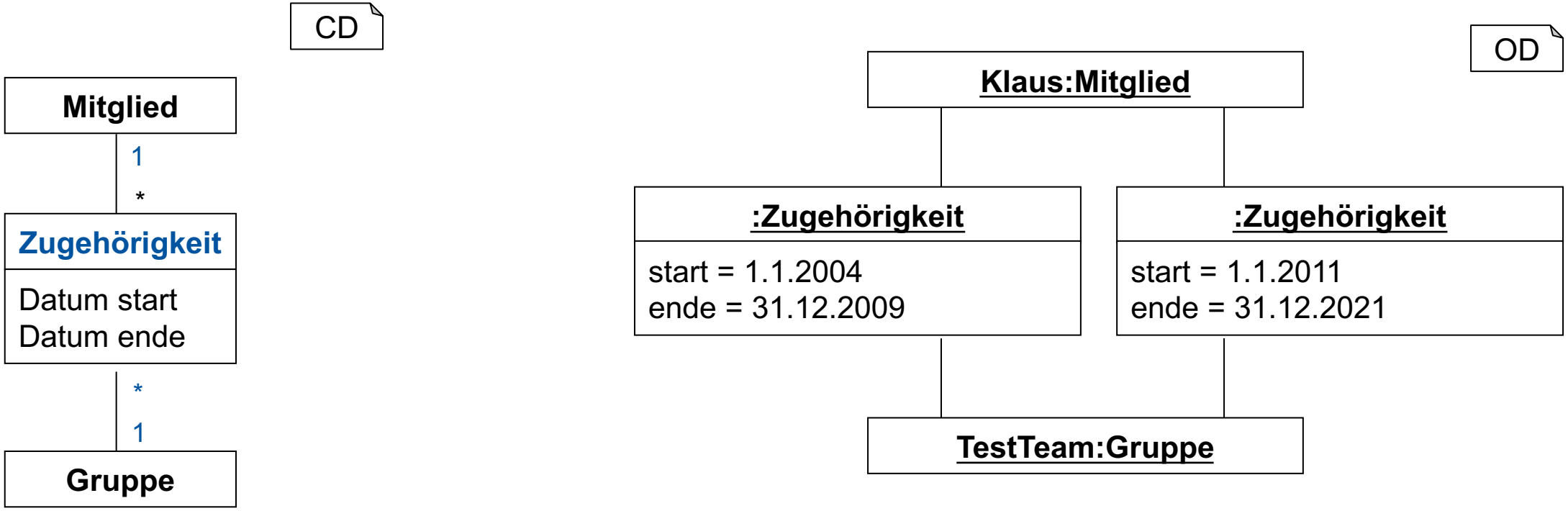
Universelles Muster (angelehnt an Heide Balzert 99)

- **Andere Namen**: *Historic Mapping* (engl.), (Fowler 97)
- **Problem**: Ein Objekt kann im Laufe der Zeit zu verschiedenen Gruppen gehören und die historische Entwicklung muss festgehalten werden.
- **Lösung**: Eine Assoziation, die Zugehörigkeit, wird durch eine Zwischenklasse oder eine Assoziationsklasse ersetzt. Die Dauer wird durch Attribute passend realisiert.



Muster: Gruppenhistorie

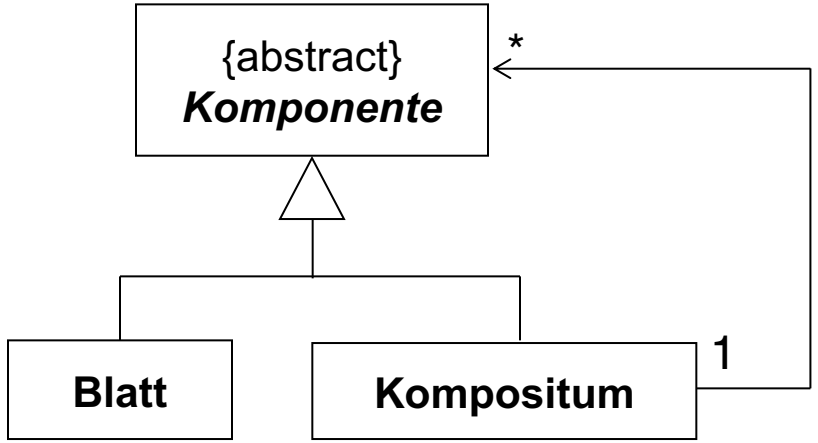
- **Lösung**: Eine Assoziation, die Zugehörigkeit, wird durch eine Zwischenklasse oder eine Assoziationsklasse ersetzt. Die Dauer wird durch Attribute passend realisiert.



Muster: Komposition

Universelles Muster (nach Gamma et al. 95)

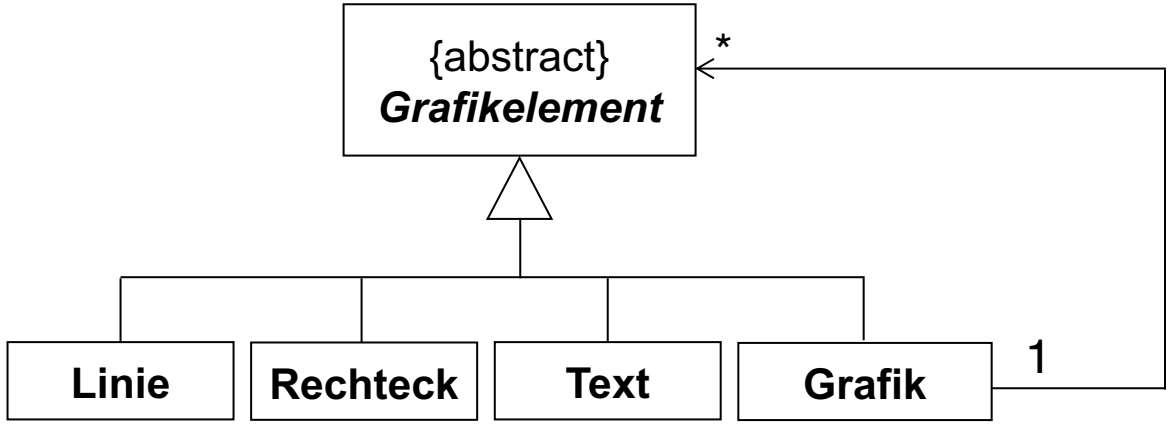
- **Andere Namen**: *Composite* (engl.), *Stückliste* (Heide Balzert)
- **Problem**: Tiefe und evtl. veränderliche Hierarchie von Aggregationen
- **Lösung**: Abstrakte Oberklasse mit verschiedenen Blattklassen sowie Kompositum-Klassen als Unterklassen.



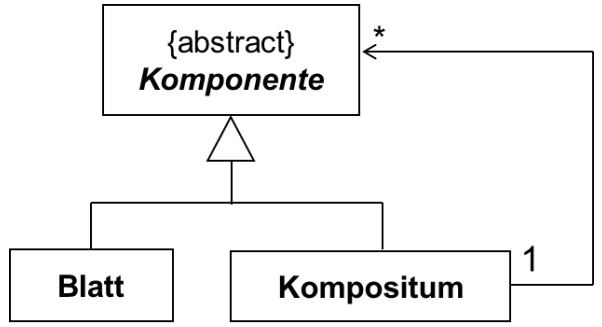
Komposition ist ein sehr häufig angewandtes Muster:
Bäume, Teile-Ganzes-Hierarchien etc. sind damit realisierbar.

Beispiele: Komposition

- **Lösung**: Abstrakte Oberklasse mit verschiedenen Blattklassen sowie Kompositum-Klassen als Unterklassen.



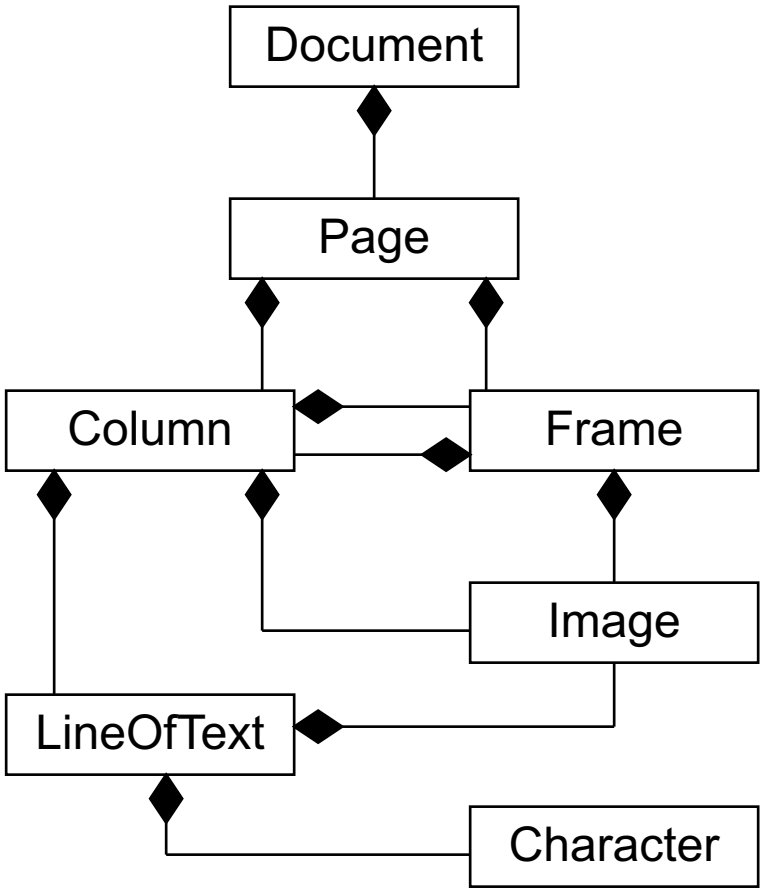
Konkrete Anwendung



Allgemeines Muster

Komposition - Beispiel

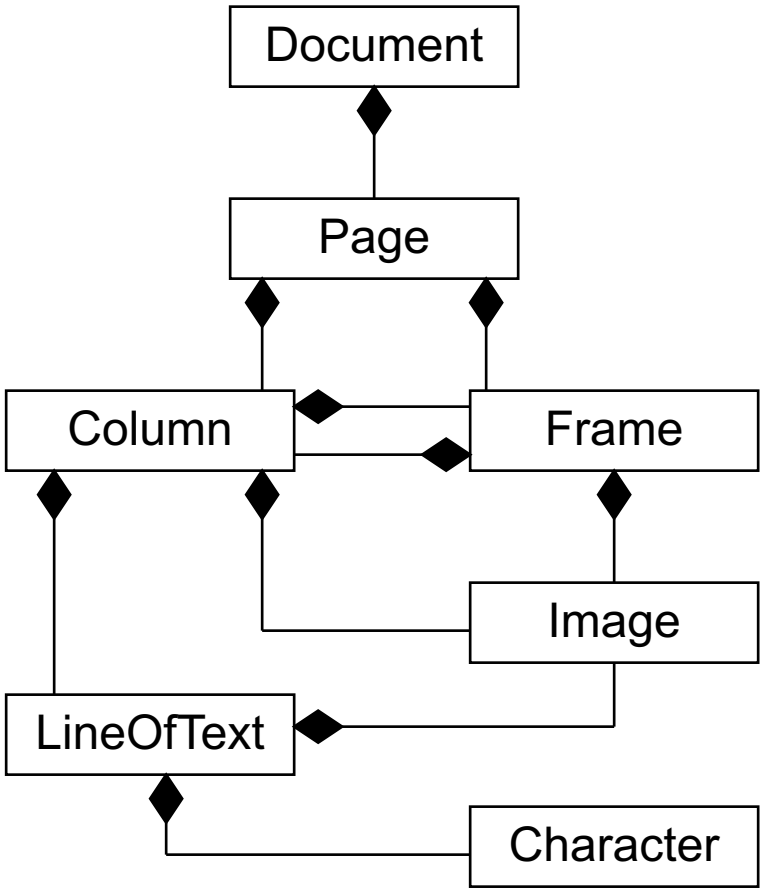
Aus der Analyse



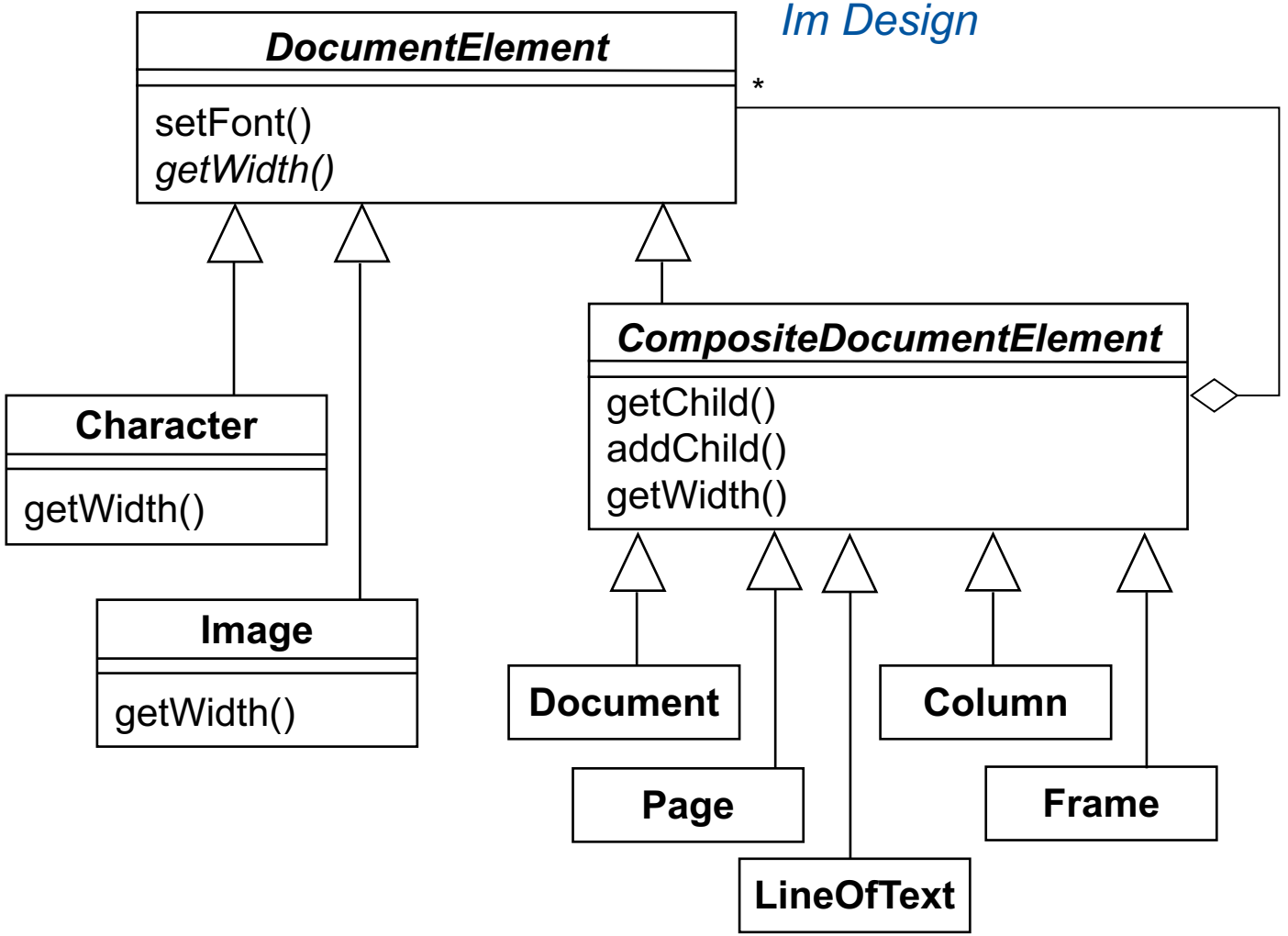
- *Aufgabe*: Dokument-Struktur und -Formatierung
- Erstes Klassendiagramm (aus Analyse)
- muss im Design umgebaut werden,
  um Verwaltung zu erleichtern

Anwendung des Composite-Musters

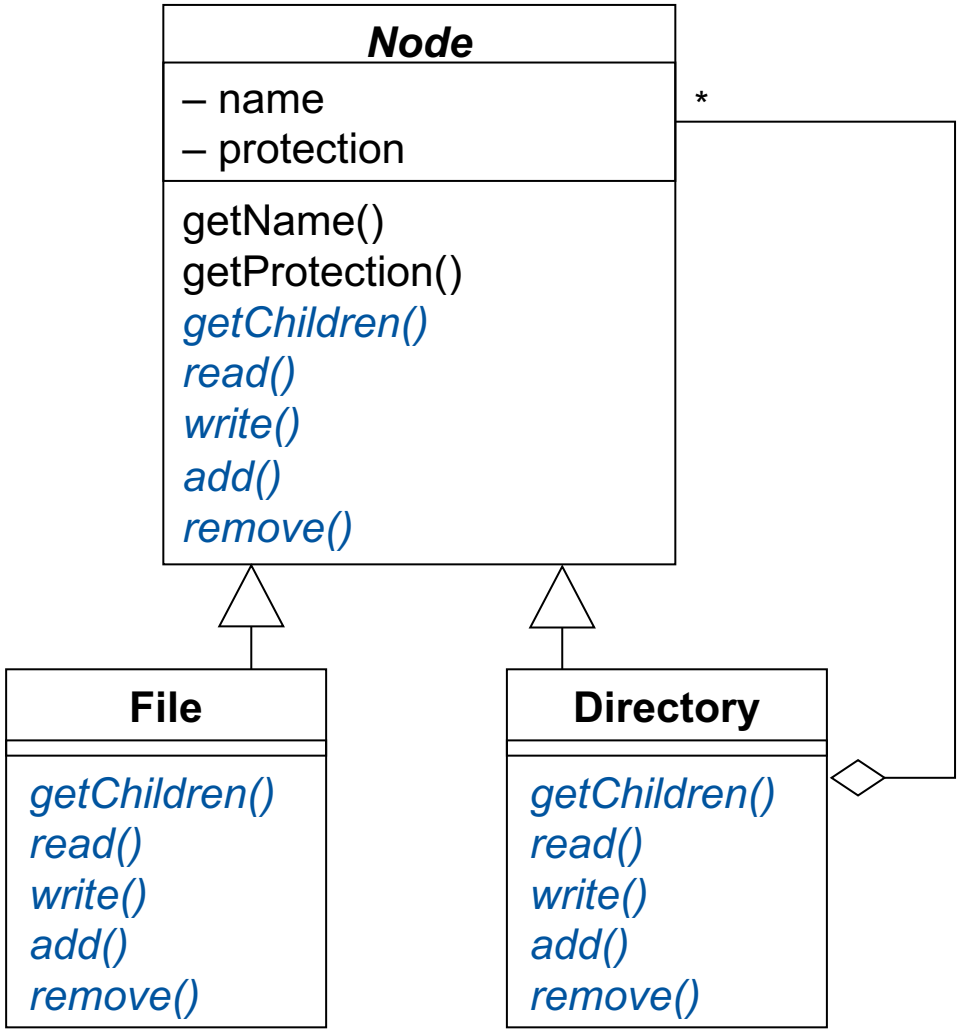
Aus der Analyse



Im Design



Anwendung des Composite-Musters

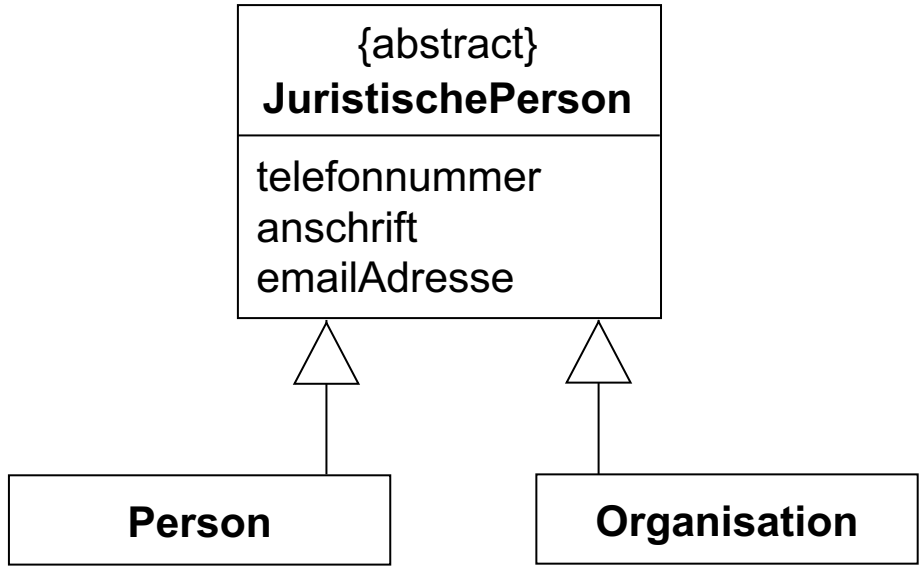


- flexible Zugriffsschicht auf Dateisysteme
  - Gemeinsame Operationen auf Dateien und Verzeichnissen:
  - Name, Größe, Zugriffsrechte, ...
- Teile-Strukturen für Geräte
- Ahnentafeln (Bäume...)

Juristische Person (Party)

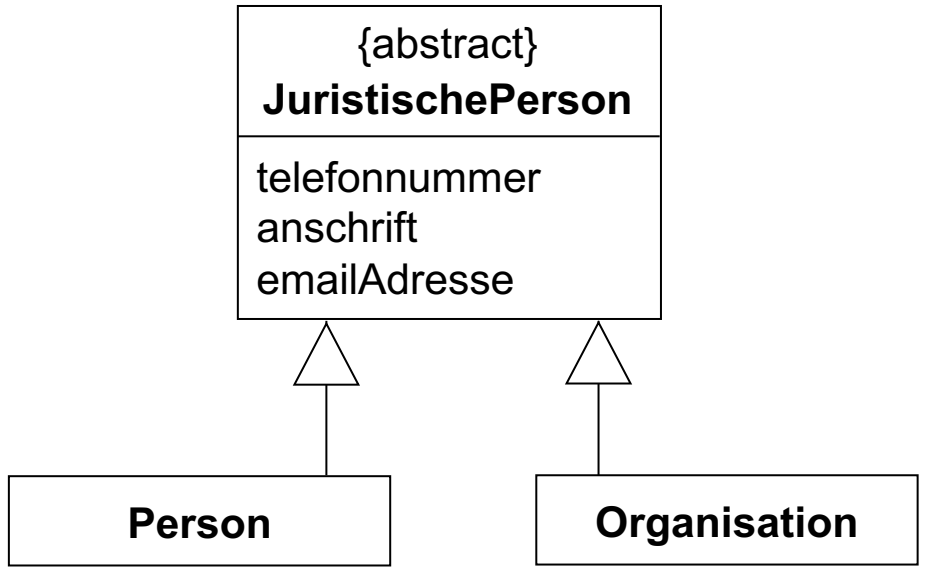
Spezifisches Analysemuster (nach Fowler 97)

- **Problem**: Organisationen und Personen spielen ähnliche Rollen gegenüber dem System
- **Lösung**: Abstrakte Oberklasse „JuristischePerson“ .

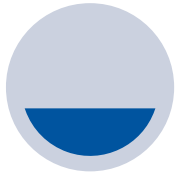


„JuristischePerson“ vs. „Abstrakte Oberklasse“

- Das Muster „JuristischePerson“ ist ein Spezialfall des Musters „Abstrakte Oberklasse“.
- Gründe für separate Darstellung:
  - Anpassung an ein Fachgebiet
  - Fachterminologie (!)
  - Geringerer Abstraktionsgrad: mehr Details aus dem Fachgebiet
- *Spezifische Analysemuster* sind meist Anwendungen und Zusammensetzungen universeller Muster
- Wesentlich ist auch die Vokabular-Bildung



Was haben wir gelernt?

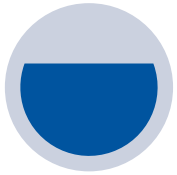


Muster …

… ist eine schematische Lösung für eine Klasse verwandter Probleme

…beschreibt Erfahrungswissen

… gibt es für unterschiedliche Bereiche: Analyse, Entwurf, Architektur, Test, …



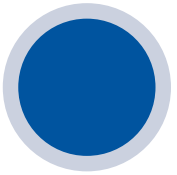
Muster in der OO Analyse

Universelle Muster sind ein „Leitfaden“ zur Modellierung und bieten Bausteine für andere Muster

Spezifische Muster sind fachgebietsübergreifend bzw. fachgebiets-spezifisch

Kataloge publiziert, z.B.:

- betriebswirtschaftliche Anwendungen (Fowler)
- Firmen- und projektspezifische Kataloge



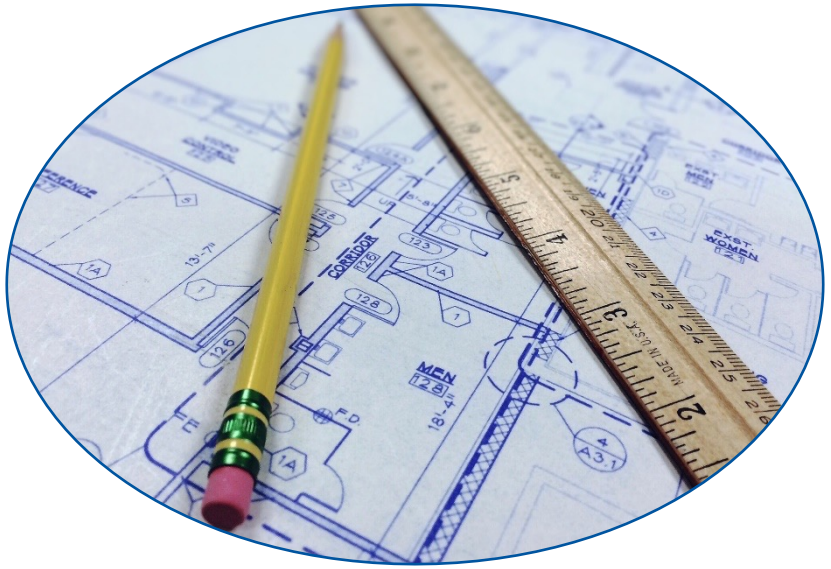
Einsatzzweck

Informationen aus der Analyse strukturieren und besser verwendbar machen

Wenn Ihre Erfahrungen mehr werden: Systematische Erkennung, Kommunikation und Anwendung von Mustern

Vorlesung Softwaretechnik

6. Software- und Systementwurf



Prof. Bernhard Rumpe
Software Engineering
RWTH Aachen

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

SE Software Engineering | RWTH AACHEN UNIVERSITY

Warum, was, wie und wozu? Entwurfsprinzipien

Warum?

- Hochgradig Praxisrelevant
- Von der Analyse zum Entwurf kommen
- Optimale Technologie gibt es nicht

Was?

- Grobentwurf: Architektur, Subsysteme und Schnittstellen
- Feinentwurf: Komponenten, Datenstrukturen, Algorithmen

Wie?

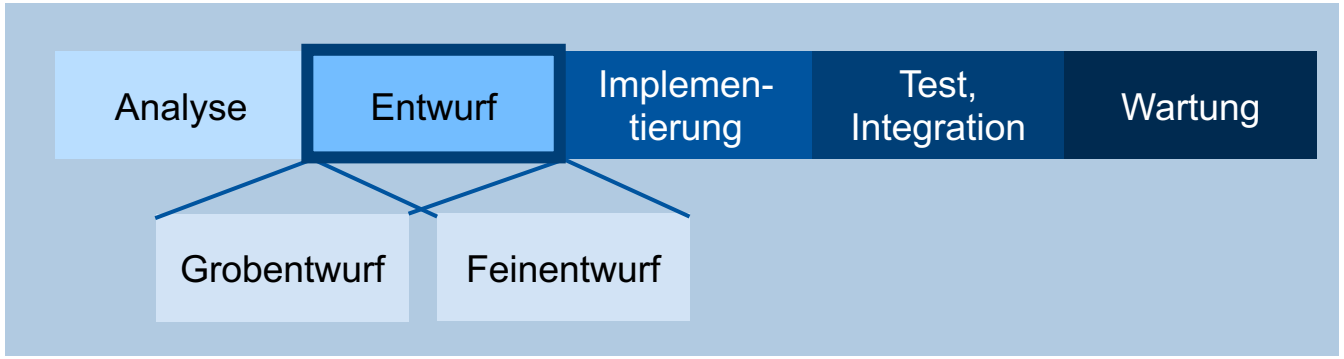
- Problem in kleine Einheiten zerlegen, Komponenten realisieren, zusammensetzen
- Softwarearchitekturen, Taktiken

Wozu?

- Vorbereitung für die Implementierung großer, komplexer Systeme, die in Teams entwickelt werden
- Kommunikation mit Stakeholdern

Softwaretechnik

6. Software- & Systementwurf
6.1. Entwurfsprinzipien



Prof. Bernhard Rumpe
Software Engineering
RWTH Aachen

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

Literatur:

- Sommerville 10
- Balzert Band I LE 23
- Balzert Band II LE 17

SE Software Engineering | RWTH AACHEN UNIVERSITY

Software-Entwurf

- Ausgangspunkt:
  - Anforderungsspezifikation (Pflichtenheft) und
  - Funktionale Spezifikation (Produktdefinition)
- Ziel:
  - Vom „WAS“ zum „WIE“: Vorgabe für Implementierung

Subsystem

- in sich geschlossen
- eigenständig funktionsfähig mit definierten Schnittstellen
- besteht aus Komponenten

Komponente

- Baustein für ein Softwaresystem (z.B. Modul, Klasse, Paket)
- benutzt andere Komponenten
- wird von anderen Komponenten benutzt
- kann auch aus Unterkomponenten bestehen

Gliederung des Entwurfsprozesses

- Architektur-Entwurf
- Subsystem- & Schnittstellen-Spezifikation

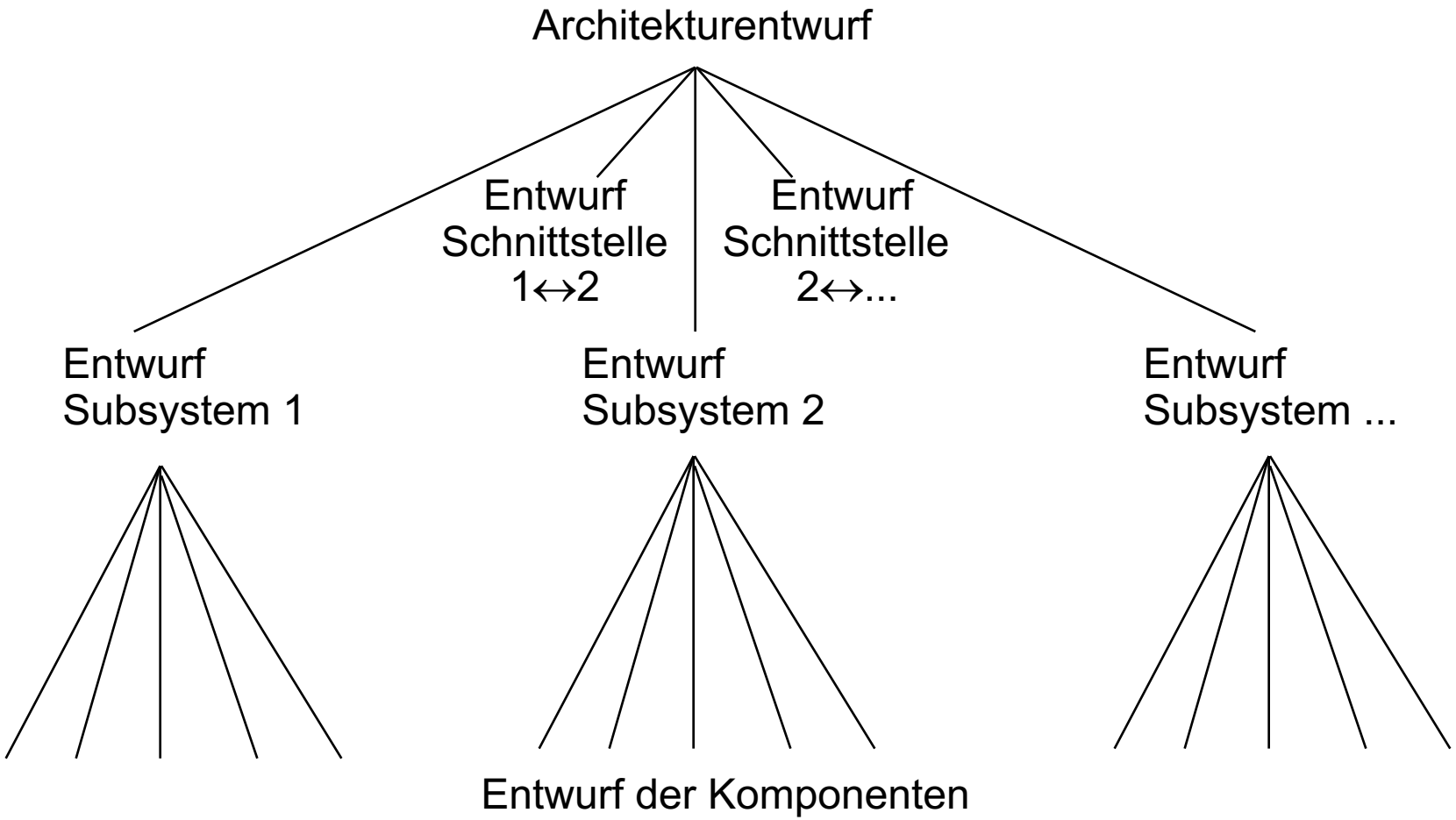
Gesamtstruktur des Systems (Grobentwurf)

- Komponenten-Entwurf
- Datenstruktur-Entwurf
- Algorithmen-Entwurf

Detailstruktur des Systems (Feinentwurf)

- Grobentwurf:
  - weitgehend unabhängig von Implementierungssprache
- Feinentwurf
  - angepasst an die Implementierungssprache und Plattform

Arbeitsteilung beim Entwurf



Kriterien für „guten“ Entwurf

- Korrektheit
  - Erfüllung der funktionalen Anforderungen
  - Sicherstellung der nichtfunktionalen Anforderungen
- Verständlichkeit & Präzision
  - Gute Dokumentation
- Anpassbarkeit
- Hohe Kohäsion innerhalb der Komponenten
- Schwache Kopplung zwischen den Komponenten
- Wiederverwendung

- Diese Kriterien gelten auf allen Ebenen des Entwurfs (Architektur, Subsysteme, Komponenten)

Kohäsion

- Kohäsion ist ein Maß für die *Zusammengehörigkeit der Bestandteile* einer Komponente.
- Hohe Kohäsion einer Komponente erleichtert Verständnis, Wartung und Anpassung.
- Kohäsion wird erreicht durch:
  - Prinzipien der Objektorientierung (Daten & Methoden-Kapselung)
  - Einhaltung von Regeln zur Paketbildung
  - Verwendung geeigneter Muster zu Kopplung und Entkopplung
- „Kohärente“ Klasse: Es gibt keine Partitionierung in unabhängige Untergruppen von zusammengehörigen Operationen und Attributen

Grade der Kohäsion (wenig bis hoch)

- Zufällige
  - Beliebige Bestandteile in einer Komponente zusammengefasst z.B. utility Klassen
- Logische
  - Zusammenfassung von Bestandteilen, die ähnliche Funktionen ausführen z.B. Eingabe, Fehlerbehandlung
- Zeit/Zeitliche
  - Die Bestandteile der Komponente werden zur gleichen Zeit aktiviert, Gruppierung nach der gemeinsamen Ausführungszeit z.B. bei Systemstart, in einem Konstruktor, Exception Handling nach dem Öffnen einer Datei
- Ablauf/Prozedurale
  - Die Bestandteile der Komponente werden als geschlossene Ablauffolge ausgeführt, d.h. Gruppierung nach der Ausführungsreihenfolge z.B. Rechteprüfung vor der Datenbankanfrage
- Aufruf/ Sequentielle
  - Ausgabe eines Bestandteils der Komponente = Eingabe eines anderen Bestandteils
- Daten/ Kommunikative/ Informationale
  - Bestandteile einer Komponente operieren auf gemeinsamen Daten (gleiche Eingabe- oder Ausgabedaten)
- Funktionale
  - Jeder Bestandteil der Komponente ist für die Ausführung der (einzigen) Gesamtfunktion der Komponente notwendig

Kopplung

- Kopplung ist ein Maß für die *Abhängigkeiten zwischen* Komponenten.
- Geringe Kopplung erleichtert die Wartbarkeit und macht Systeme stabiler.

- Arten der Kopplung:
  - Datenkopplung (gemeinsame Daten)
  - Schnittstellenkopplung (gegenseitiger Aufruf)
  - Strukturkopplung (gemeinsame Strukturelemente (z.B. Attribute))

- Reduktion der Kopplung:
  - Kopplung kann nie auf Null reduziert werden!
  - Schnittstellenkopplung ist akzeptabel, da Flexibilität in der OO gegeben
  - Datenkopplung soweit möglich vermeiden!
    - z.B. static oder public Attribute, Vererbung
  - Strukturkopplung vermeiden!
    - z.B. keine Vererbung über Paketgrenzen hinweg

- Entkopplungsbeispiel: get/set-Methoden statt Attribut-Zugriff

Wiederverwendung

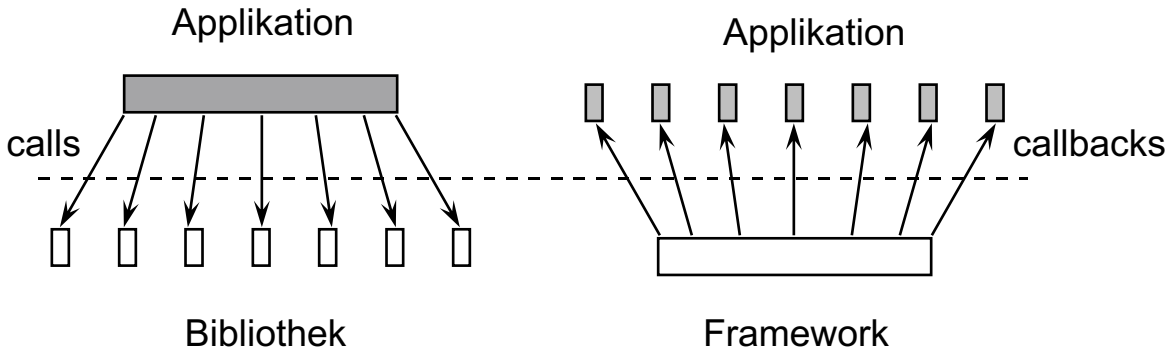
- Wiederverwendung (*reuse*) ist ein Maß für die Ausnutzung von Gemeinsamkeiten zwischen Komponenten
- Reduktion der Redundanz
- Erhöhung der Stabilität und Ergonomie

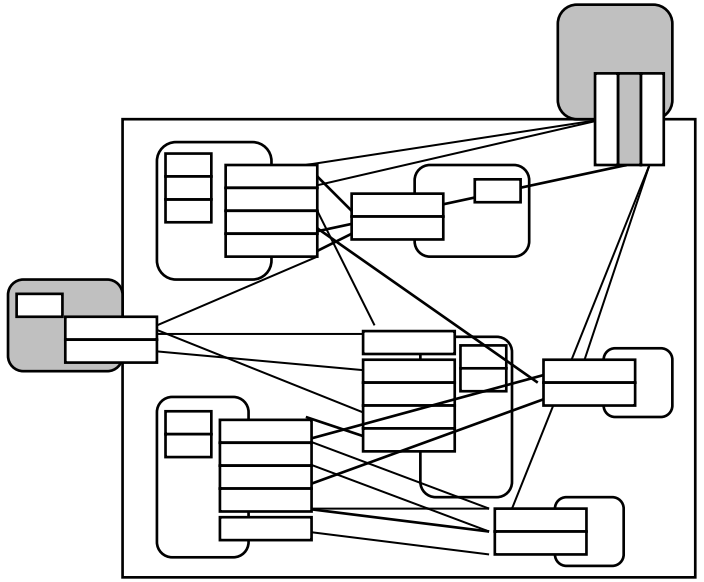
- Hilfsmittel für Wiederverwendung:
  - im objektorientierten Entwurf: Vererbung, Parametrisierung
  - im modularen und objektorientierten Entwurf:
    Module/Objekte mit allgemein nutzbaren Schnittstellen (Interfaces)

- Aber: Wiederverwendung kann die Kopplung erhöhen:
  - Schnittstellenkopplung und Strukturkopplung

Framework

- Framework
  - Eine Software, die durch Callback-Methoden erweiterbar ist
  - Dazu werden Subklassen gebildet und dem Framework gegeben
  - Aufrufe finden in entgegengesetzter Richtung statt:
    - das genutzte Framework ruft die eigentliche Applikation auf





Referenzmodelle und -architekturen

- Referenzmodell (für die Analyse)
  - Logische Aufteilung der Systeme einer Domäne in
    - Subsysteme
    - Verbindungen
    - Kommunikationskanäle zwischen diesen Subsystemen

- Referenzarchitektur (für die Architektur/Entwurfsphase)
  - Abbildung eines Referenzmodells auf Softwarekomponenten
  - Datenfluss, Kommunikation
  - Technische Aspekte werden hinzugefügt
  - Gruppierung der logischen Elemente auf Softwarekomponenten ist *-zu-*
    - Meistens realisieren mehrere Softwarekomponenten ein logisches Subsystem
    - Logische Elemente können mehrfach repliziert sein

Beispiel: Referenzmodell für Workflowmanagement Systeme

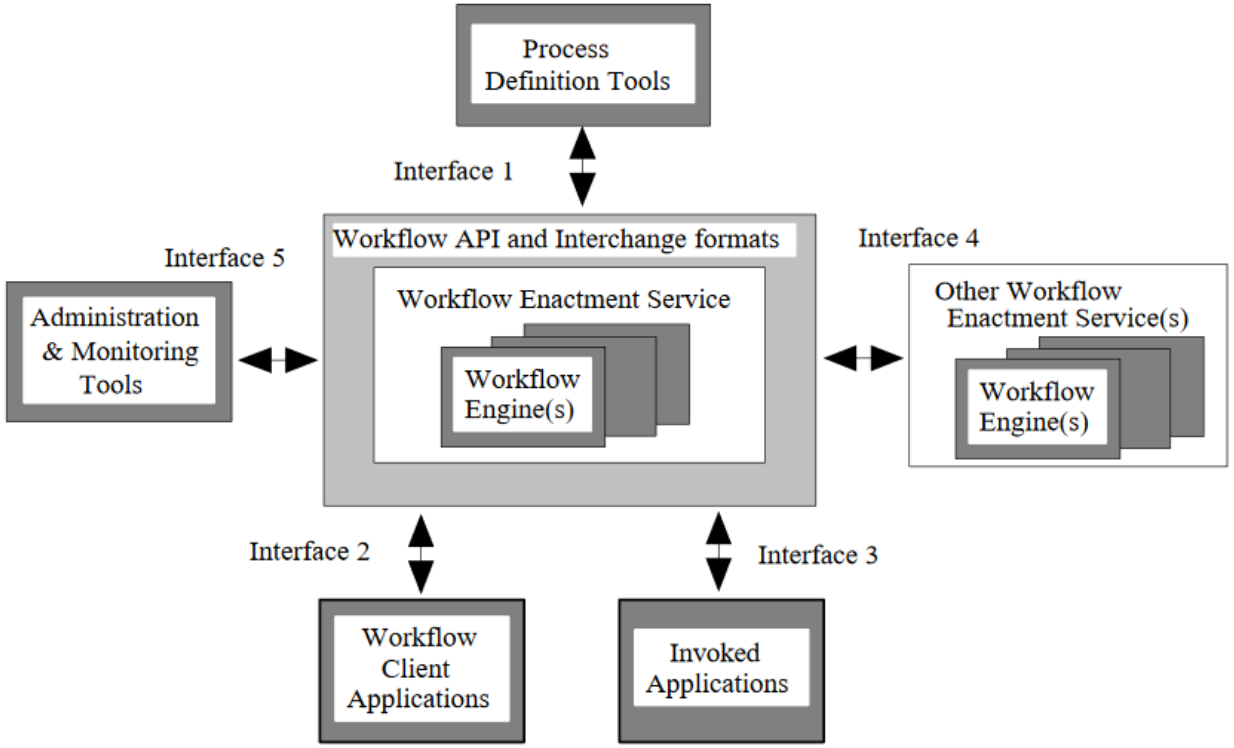


Fig 6 Workflow Reference Model - Components & Interfaces

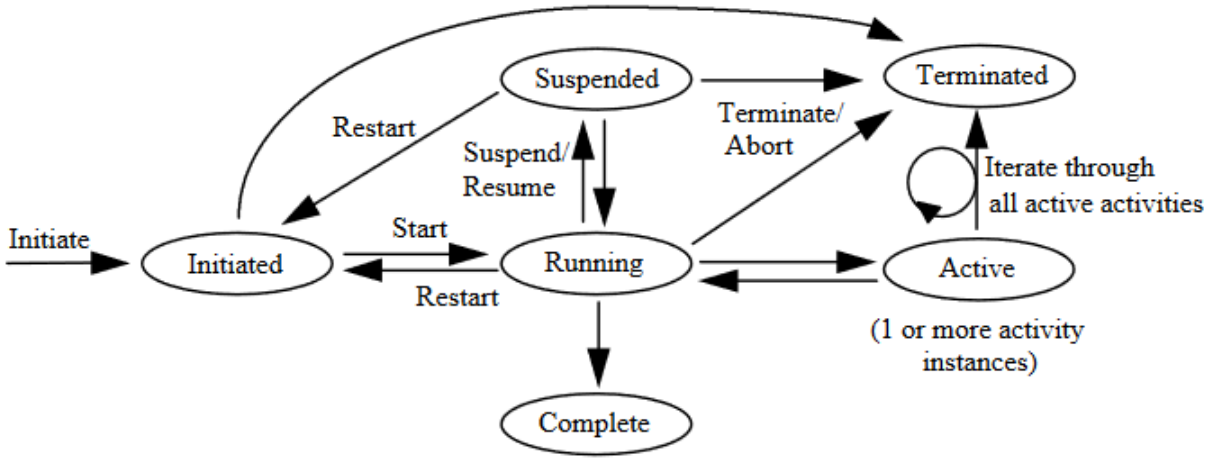
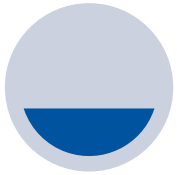


Figure 7 - Example state transitions for a process instances

Siehe: Workflow Management Coalition, http://www.wfmc.org/standards/docs/tc003v11.pdf

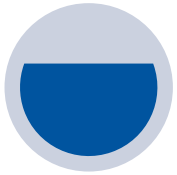
Was haben wir gelernt?



Entwurf

Aufteilen des Problems

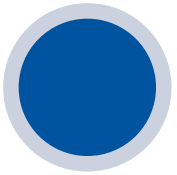
Subsysteme,
Schnittstellen,
Komponenten



Entwurfsprozess

Grobentwurf:
Architektur, Subsysteme und Schnittstellen

Feinentwurf:
Komponenten,
Datenstrukturen,
Algorithmen



Guter Entwurf

Hohe Kohäsion

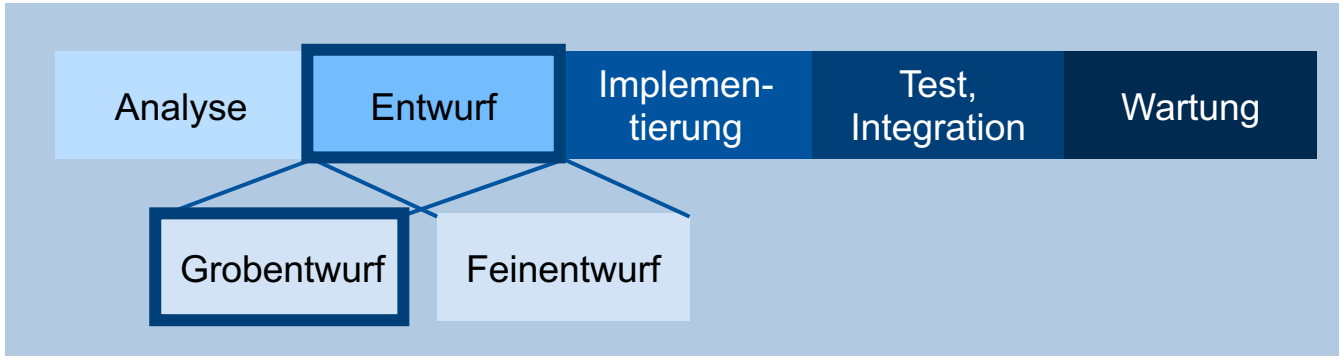
Wenig Kopplung

Wiederverwendung:

Frameworks,
Referenzmodelle,
Referenzarchitekturen

Softwaretechnik

6. Software- & Systementwurf
6.2. Softwarearchitektur



Prof. Bernhard Rumpe
Software Engineering
RWTH Aachen

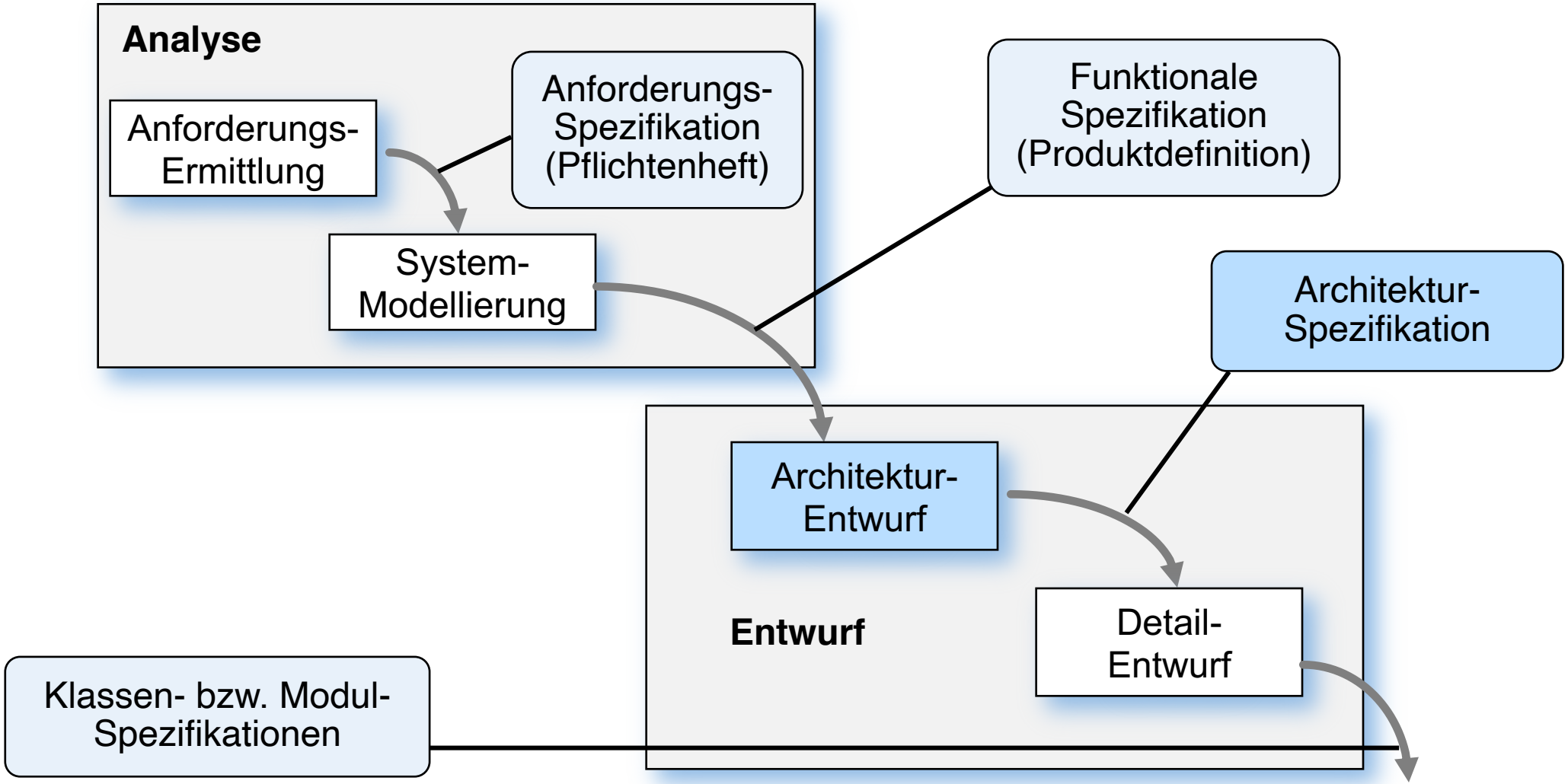
http://www.se-rwth.de/
@SE_RWTH

Literatur:

• Sommerville 10
• Balzert LE 23
• Shaw/Garlan: Software Architecture, 1996
• Bass/Clements/Kazman: Software Architecture in Practice, Addison-Wesley, 1998
• P. Kruchten, The 4+1 view model of architecture, IEEE Software, Nov. 1995, 12(6)

SE Software Engineering | RWTH AACHEN UNIVERSITY

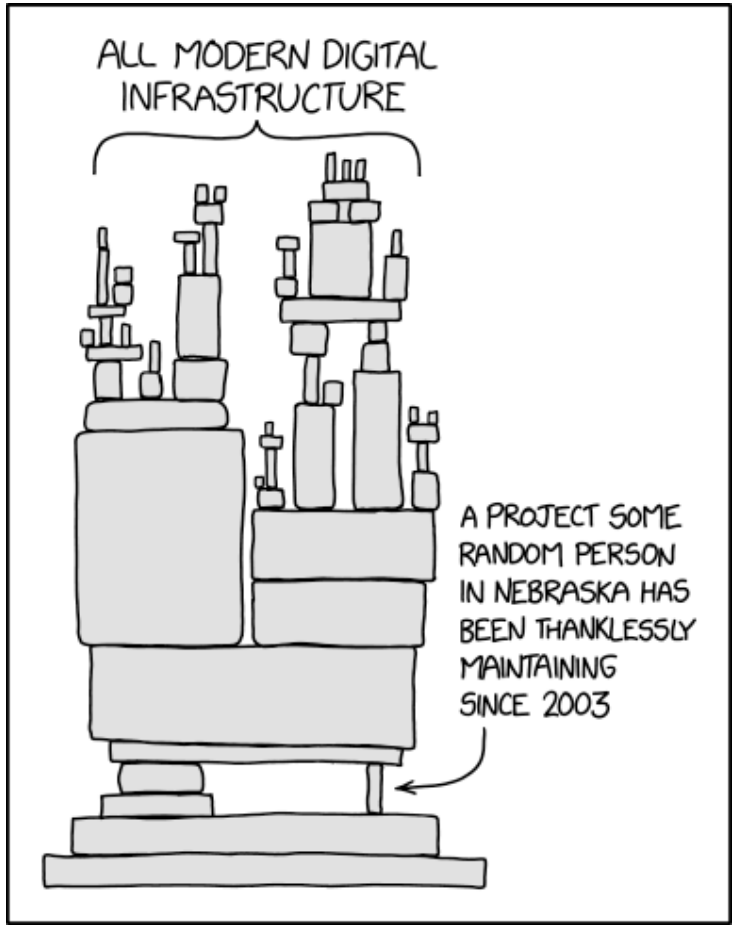
Von der Analyse zum Entwurf



Was ist Architektur?

- „[Architektur ist] Harmonie und Einklang aller Teile, die so erreicht wird, dass nichts weggenommen, zugefügt oder verändert werden könnte, ohne das Ganze zu zerstören.“

  [Leon Battista Alberti]



https://xkcd.com/

Softwarearchitektur: Definitionen aus der Literatur

- "The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them."

[BCK03]

- "Architecture is defined by the recommended practice as the fundamental organization of a system, embodied in its components, their relationships to each other and the environment, and the principles governing its design and evolution."

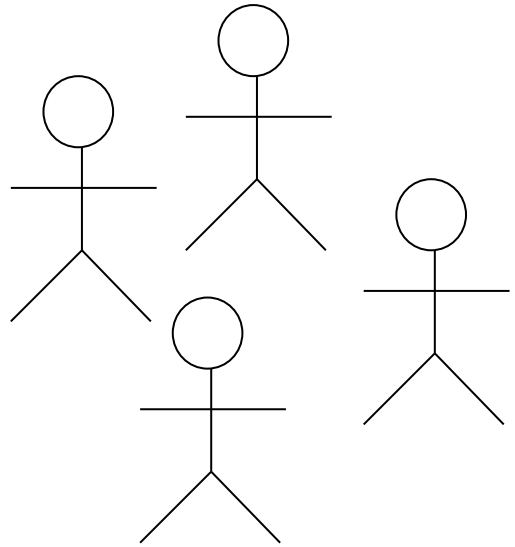
[IEEE Std. 1471-2000]

Softwarearchitektur: Unsere Sicht

- Softwarearchitektur beschreibt die Struktur eines Systems.
- Diese Beschreibung beinhaltet
  - die Komponenten,
  - deren Schnittstellen
  - und deren Beziehungen
- Sie beschreibt die wichtigsten strukturellen Eigenschaften eines Systems präzise, ist gleichzeitig aber kompakt.
- Sie beinhaltet die essentiellen Eigenschaften eines Systems, die sich auf Gesamtsystemsicht beschreiben und analysieren lassen

Stakeholder

- Ein System wird von vielen Faktoren beeinflusst
  - Kunden
  - Endnutzer
  - Entwickler
  - Projektmanager
  - Wartungspersonal (Konfiguration, Weiterentwicklung)
  - Vermarktung/Verkauf
  - …
- Diese werden als Stakeholder bezeichnet:
  Alle am Projekt direkt oder indirekt beteiligten Personengruppen



Stakeholder

Einfluss der Stakeholder auf die Softwarearchitektur

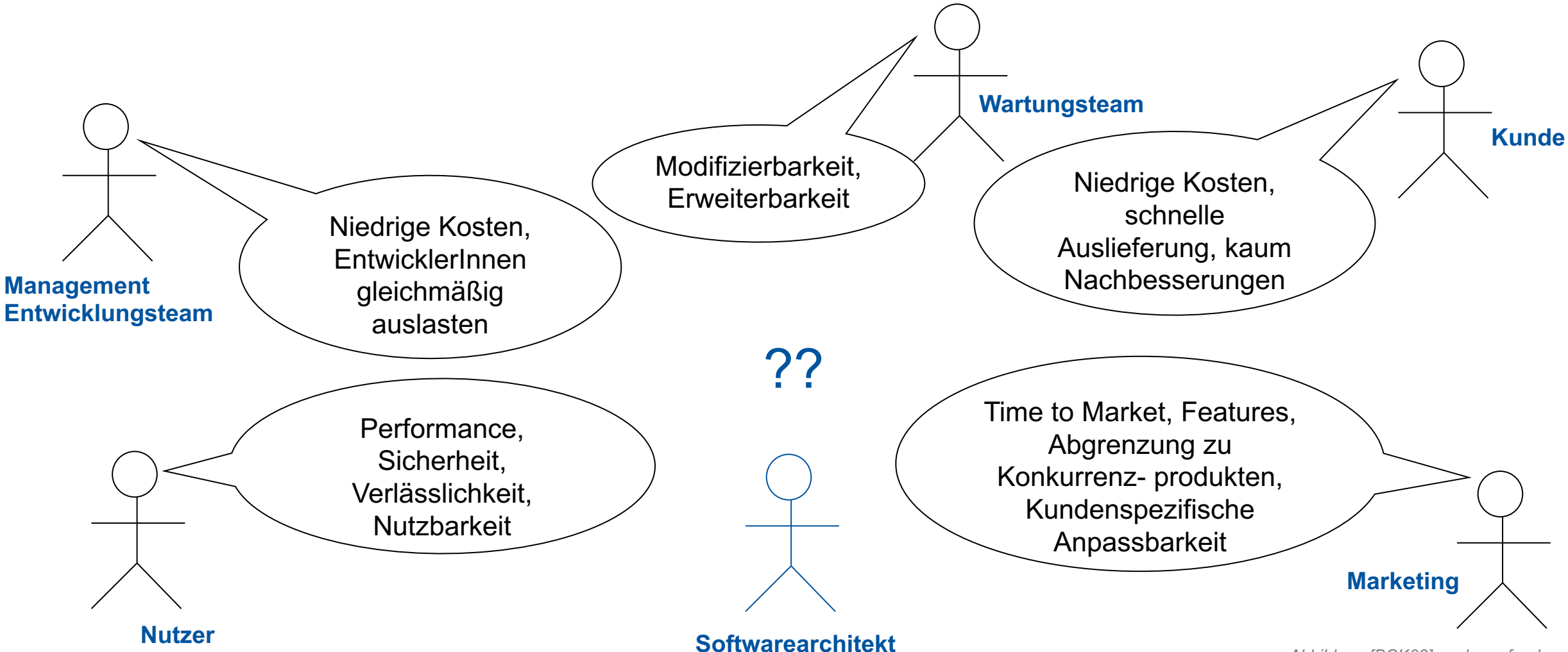


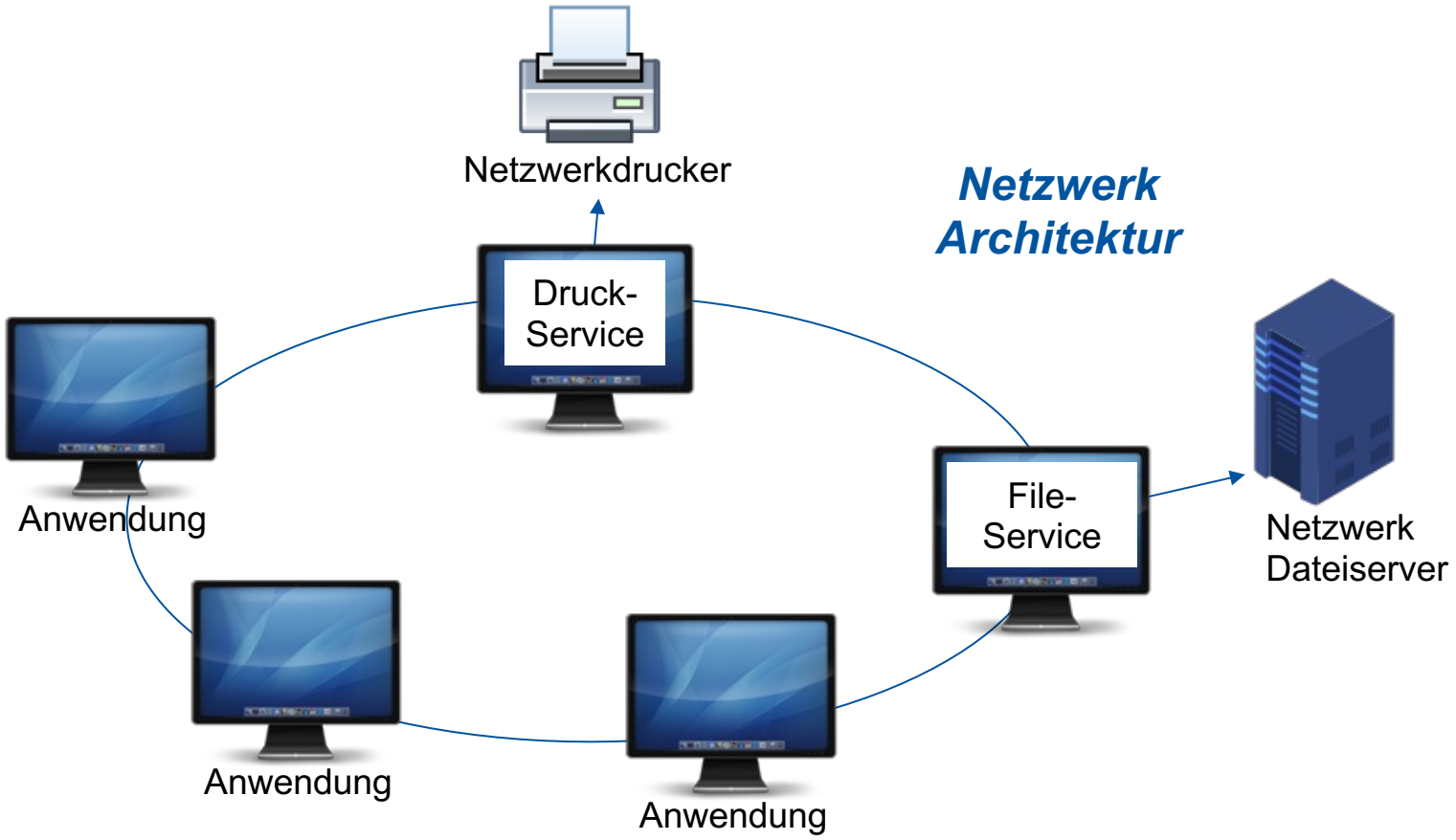
Abbildung [BCK03] nachempfunden

Nutzen einer Architekturbeschreibung

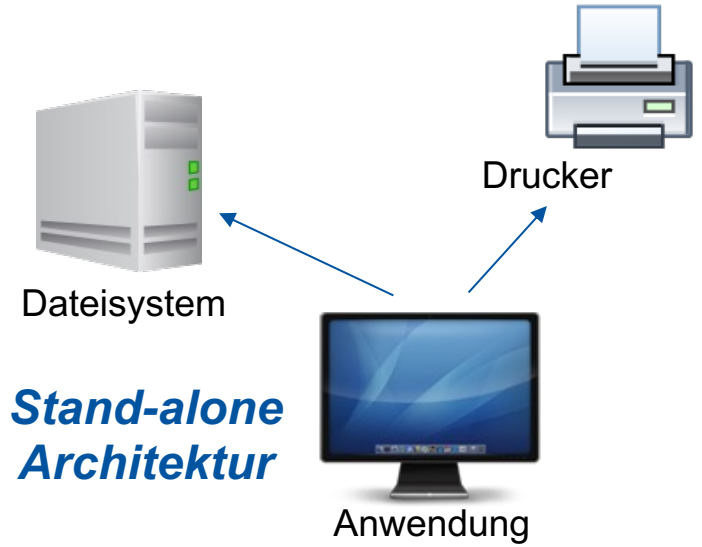
- Kommunikation der Stakeholder
  - gemeinsame Sprache
  - Abstraktion macht überhaupt Kommunikation möglich
  - Schulung der Entwickler
- Wesentliche Entwurfsentscheidungen
  - Beschränkung der Implementierungsmöglichkeiten
  - Legt Organisationsstruktur fest
  - Verhindert oder erlaubt bestimmte Stufen für Qualitätsattribute
  - Erlaubt das Urteilen über und Verwalten von Veränderungen
  - Zeit und Kostenschätzung
- Wiederverwendbare Abstraktion des Systems
  - Produktlinien haben gemeinsame Architektur
  - Wiederverwendung von Komponenten
  - Wiederverwendung von erprobten Designs
- Fokus auf spezifische Systemeigenschaften möglich
  - Getrennte Betrachtung der Aspekte
  - Trennung der Zuständigkeiten
  - Übersichtlichkeit
- Frühzeitige Analysemöglichkeiten
  - Prototypen
  - Risikoabschätzung
  - Zeit- und Kostenschätzung
  - Vorhersage von Eigenschaften des Systems

Architektur-Beispiel

- Physikalische Architekturen



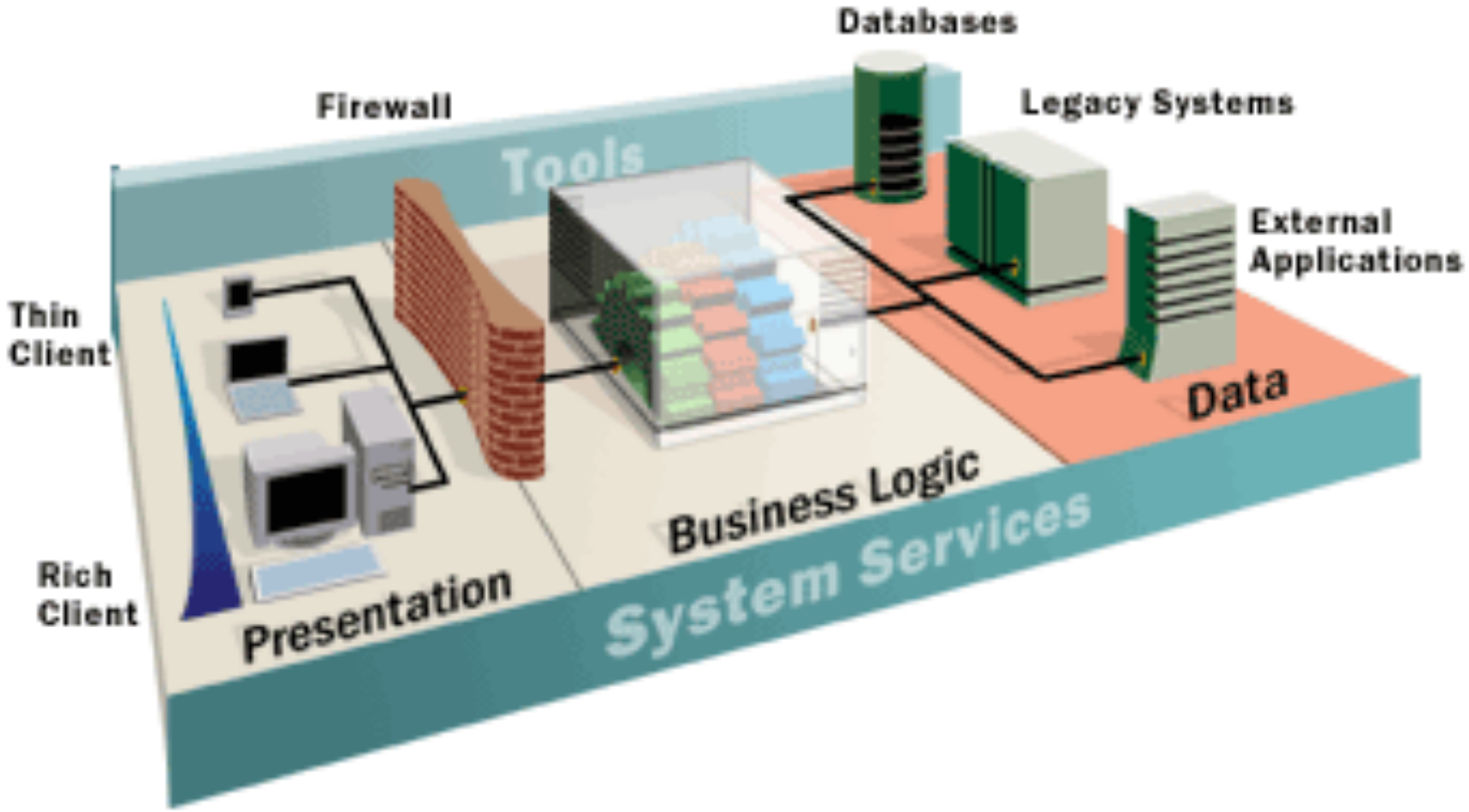
Netzwerk Architektur



Stand-alone Architektur

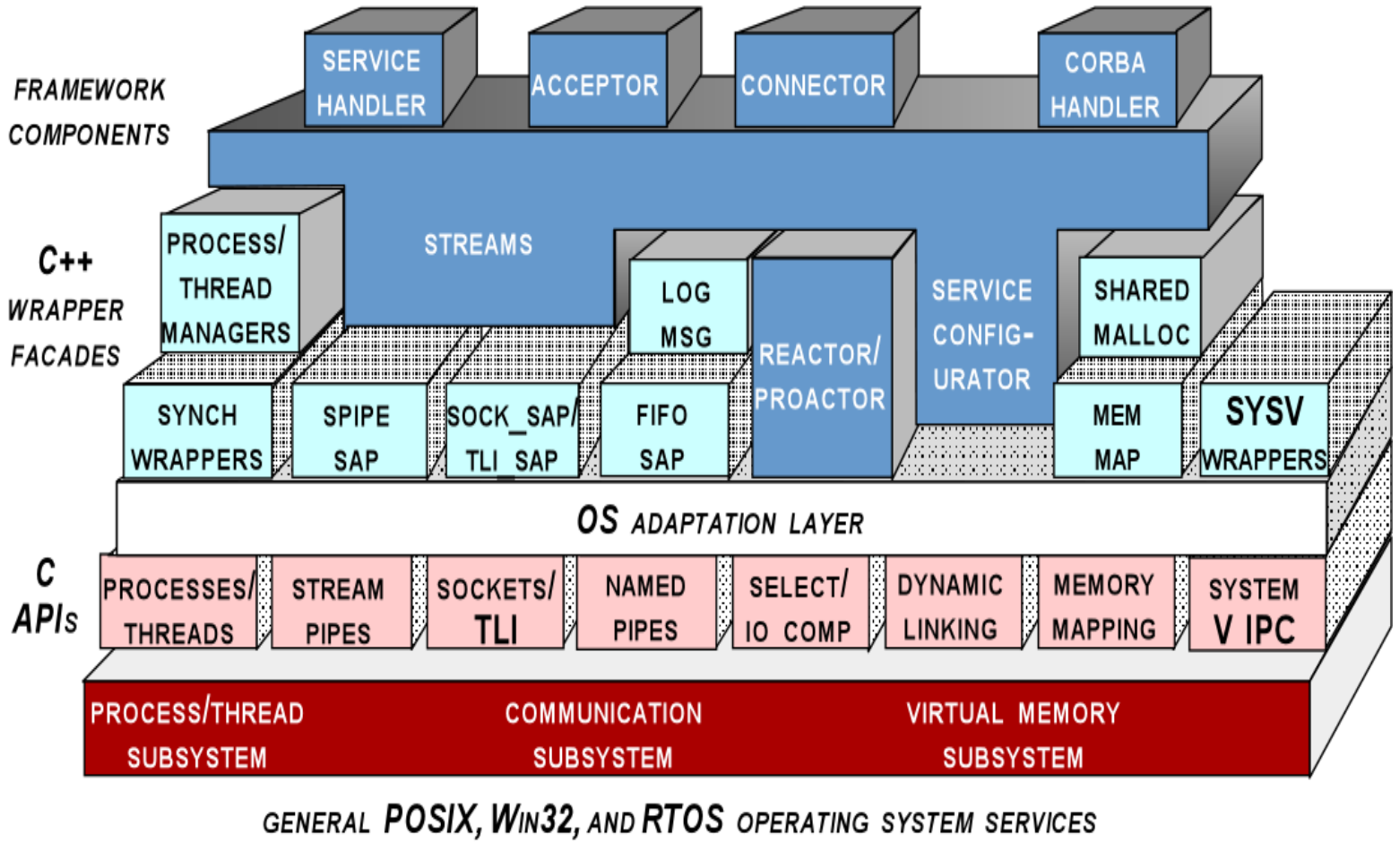
Architektur-Beispiel

- Physikalische Architekturen
  - Client
  - Firewall
  - Applikationsserver
  - Daten



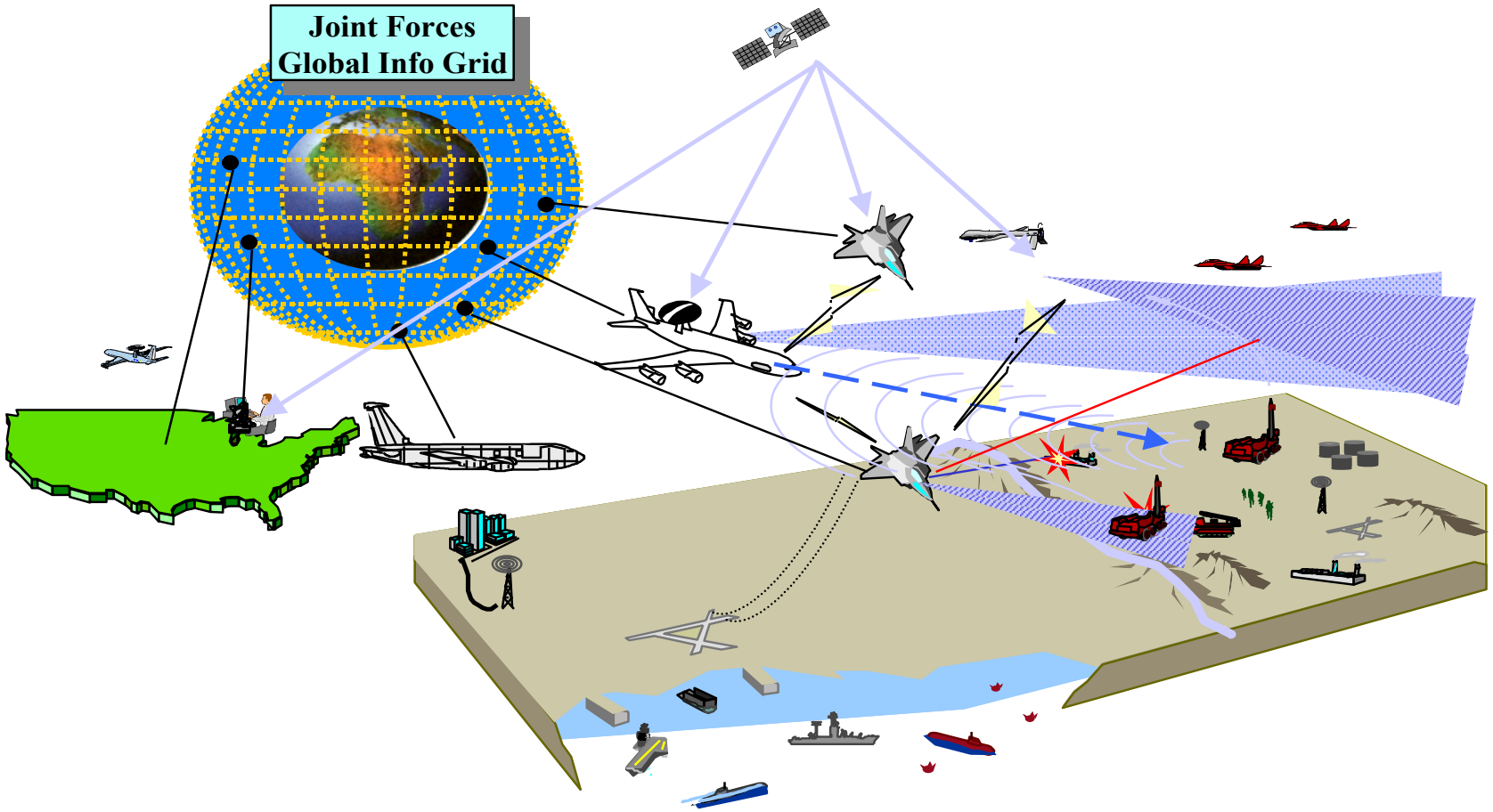
Architektur-Beispiel

- Schichten Architektur der Software (innerhalb eines Systems):

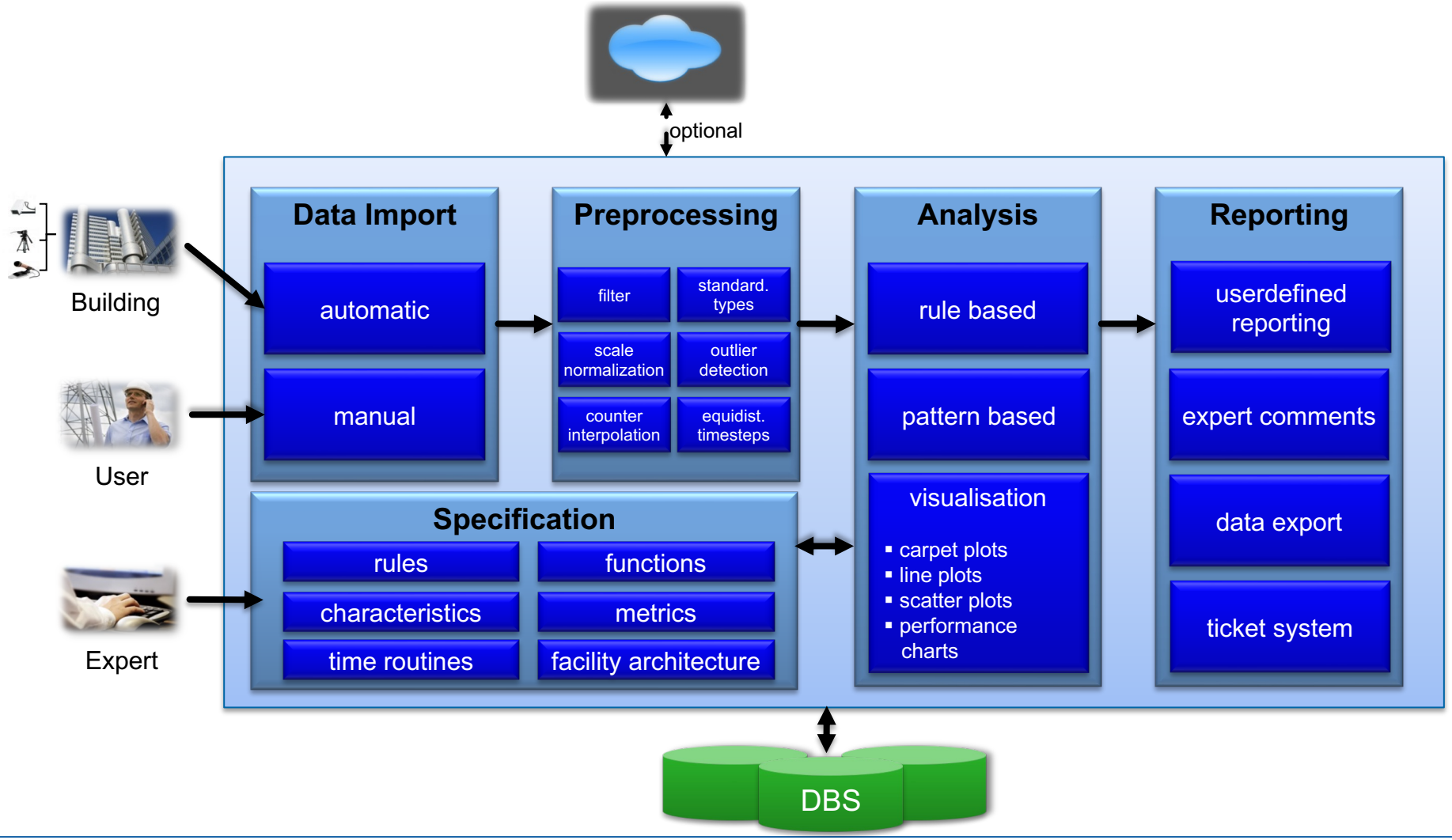


Architektur-Beispiel

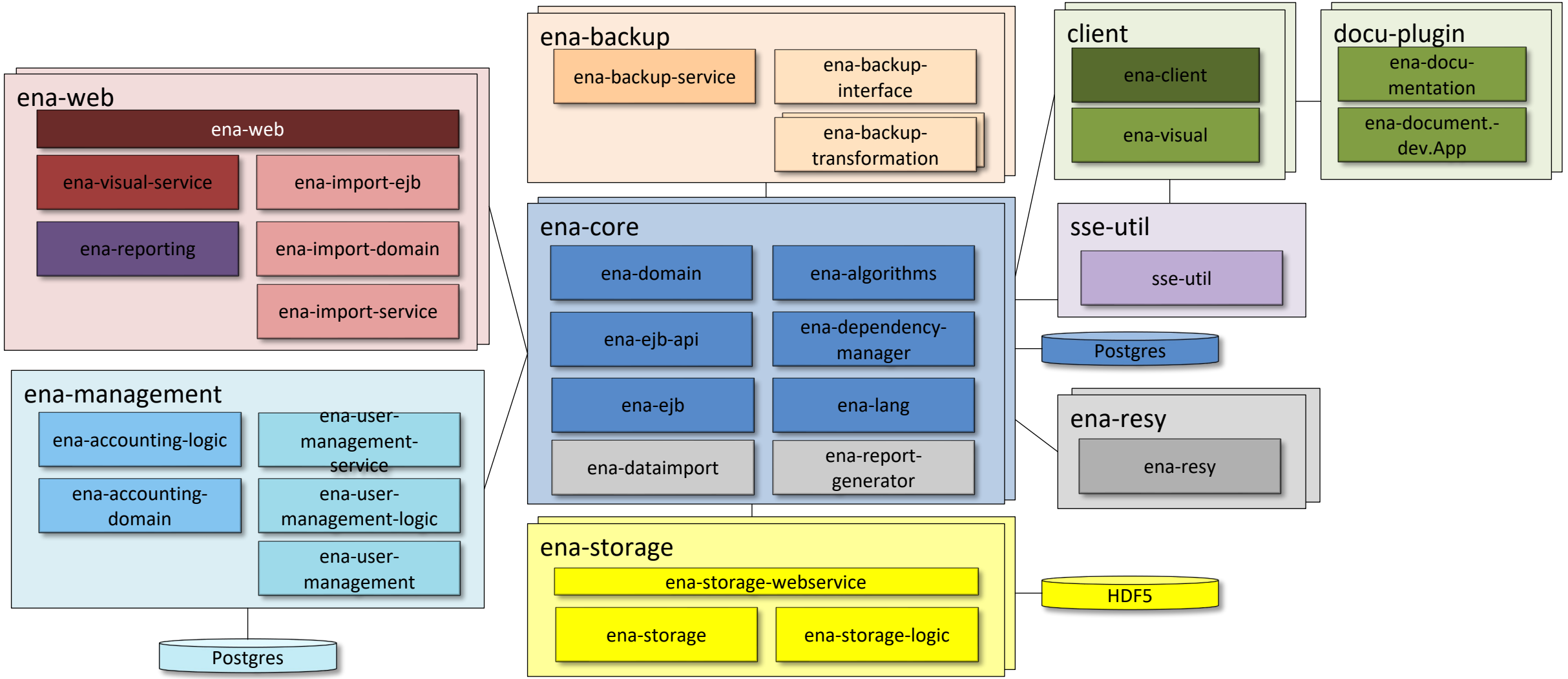
- Kommunikationsarchitektur eines Software-Intensiven Systems mit Echtzeitanforderungen:



Architektur der Energie Navigator Plattform (der Synavision GmbH)

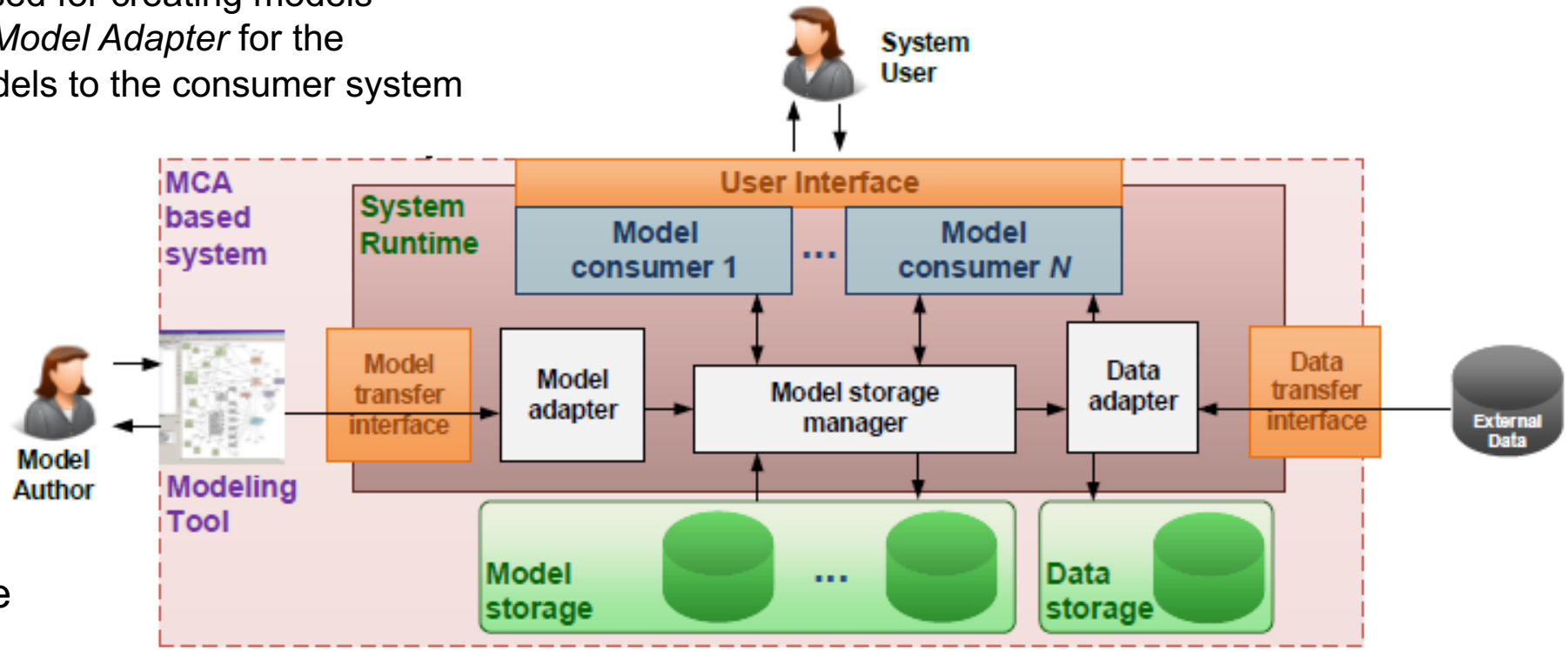


Architektur der Energie Navigator Plattform: Components



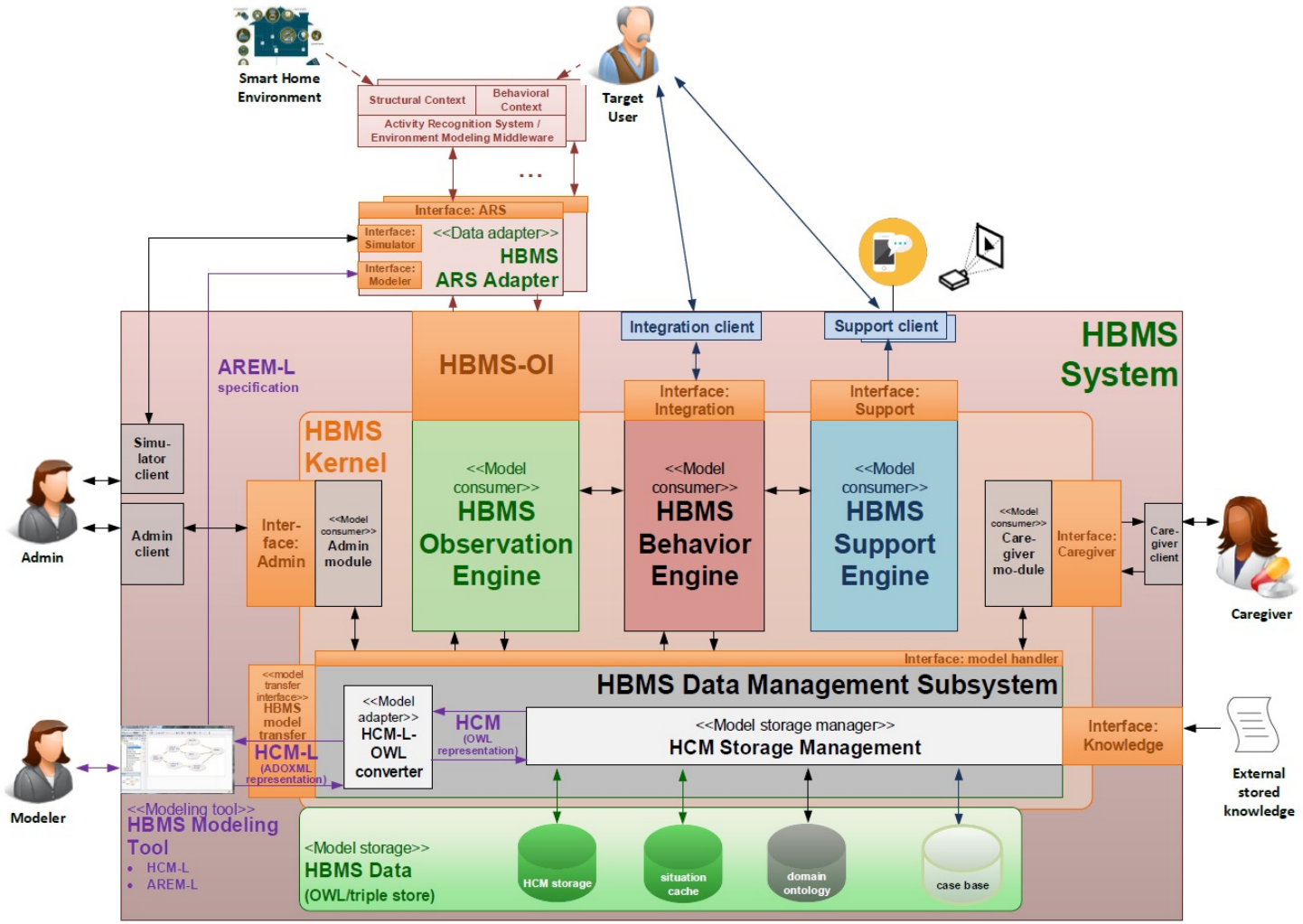
Architektur von Assistenzsystemen

- Model-Centered Architecture
  - *Modeling Tool*: components used for creating models
  - *Model Transfer Interface* and *Model Adapter* for the components providing the models to the consumer system
  - *Data Transfer* and *Data Adapter*: drive conversion of external data into internal representation; *Data Storage*
  - *Model Storage* and *Model Storage Manager* for model persistence
- *Model Consumer* describes the components which use the models to provide the functionality of the MCA-based solution)
  - *Device* and *User Interface*: model-based interfaces to the model consumers

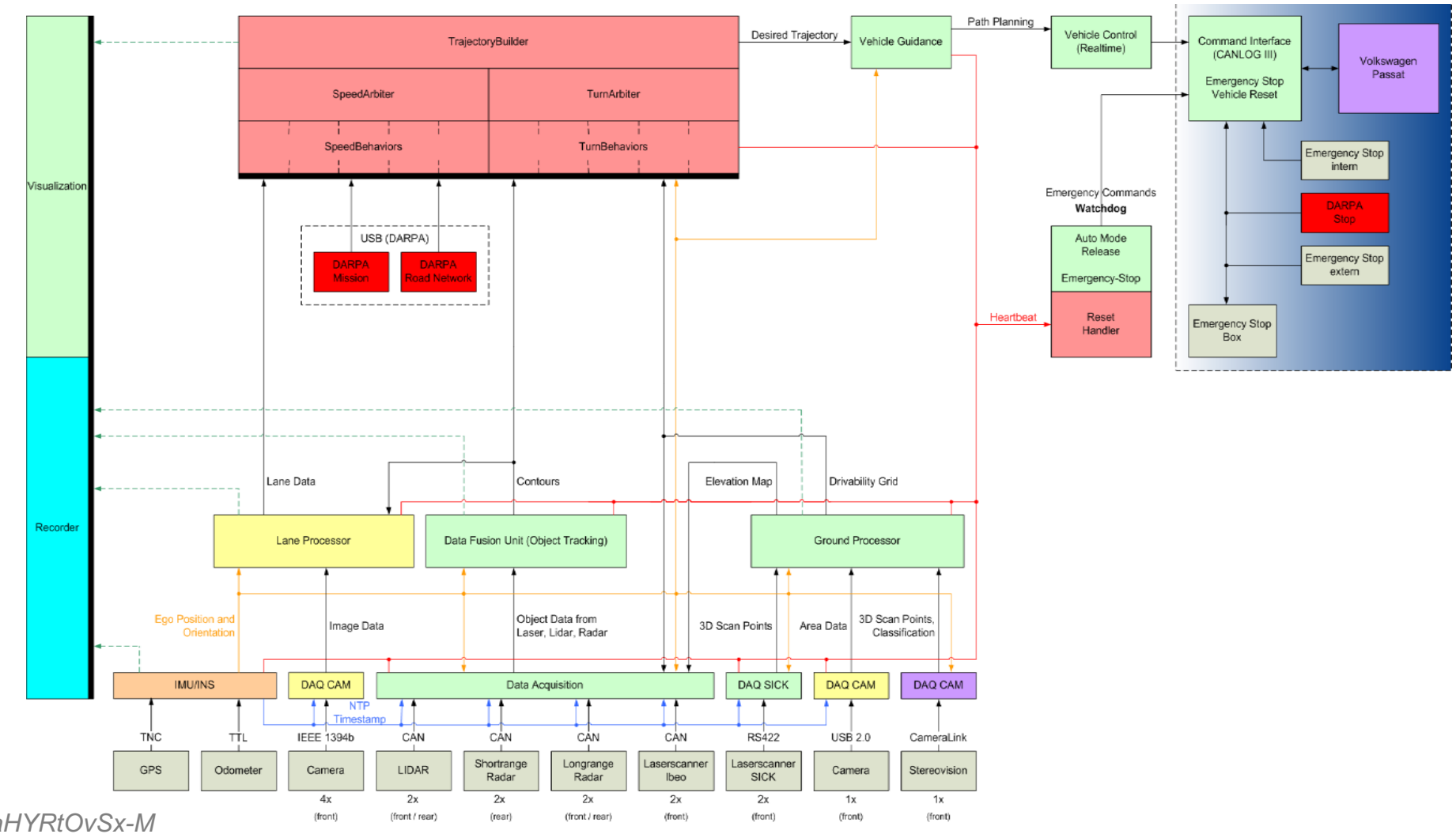


Architektur von Assistenzsystemen: Konkretes Beispiel

- Projekt: Human Behavior Monitoring and Support (HBMS)

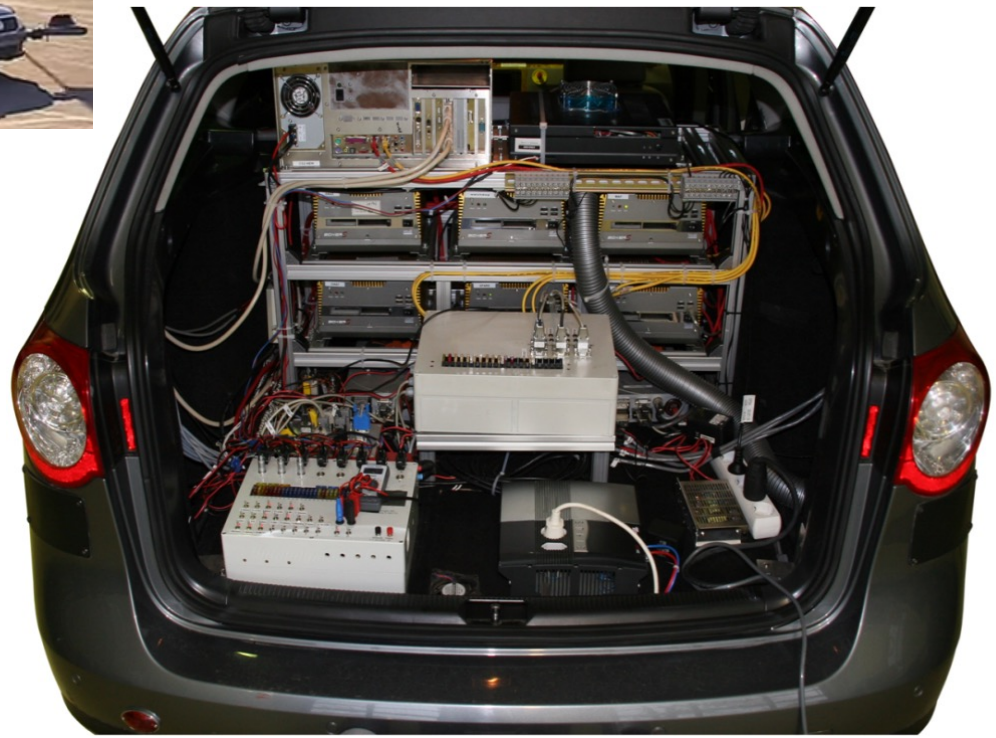


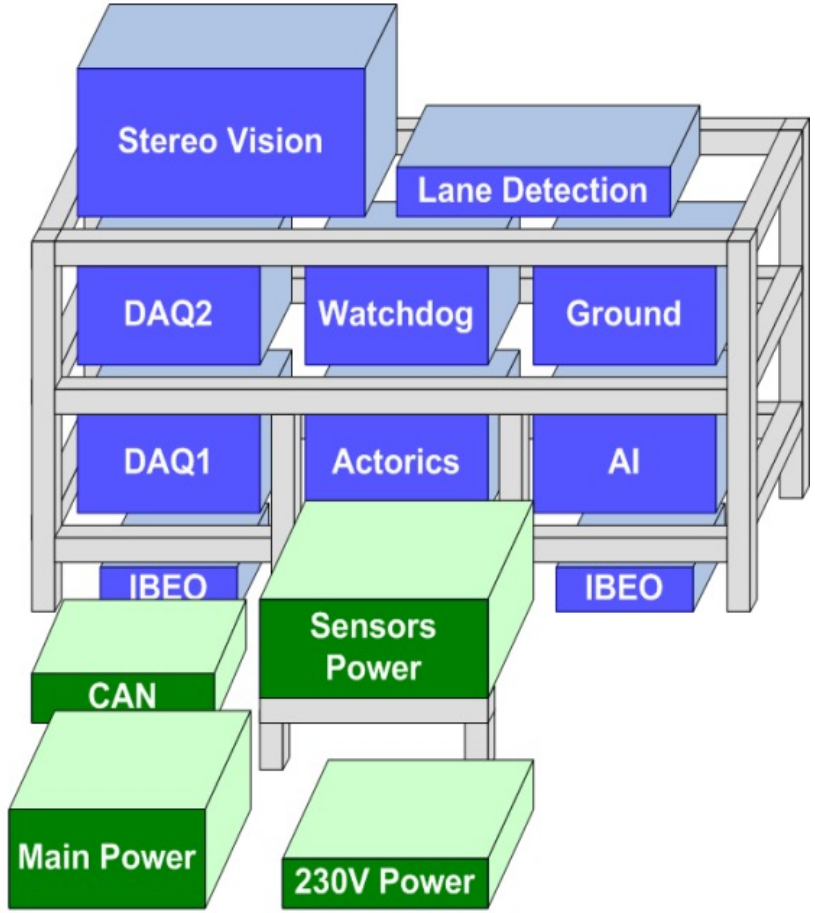
Caroline‘s Kommunikationsarchitektur



Video: https://www.youtube.com/watch?v=aHYRtOvSx-M

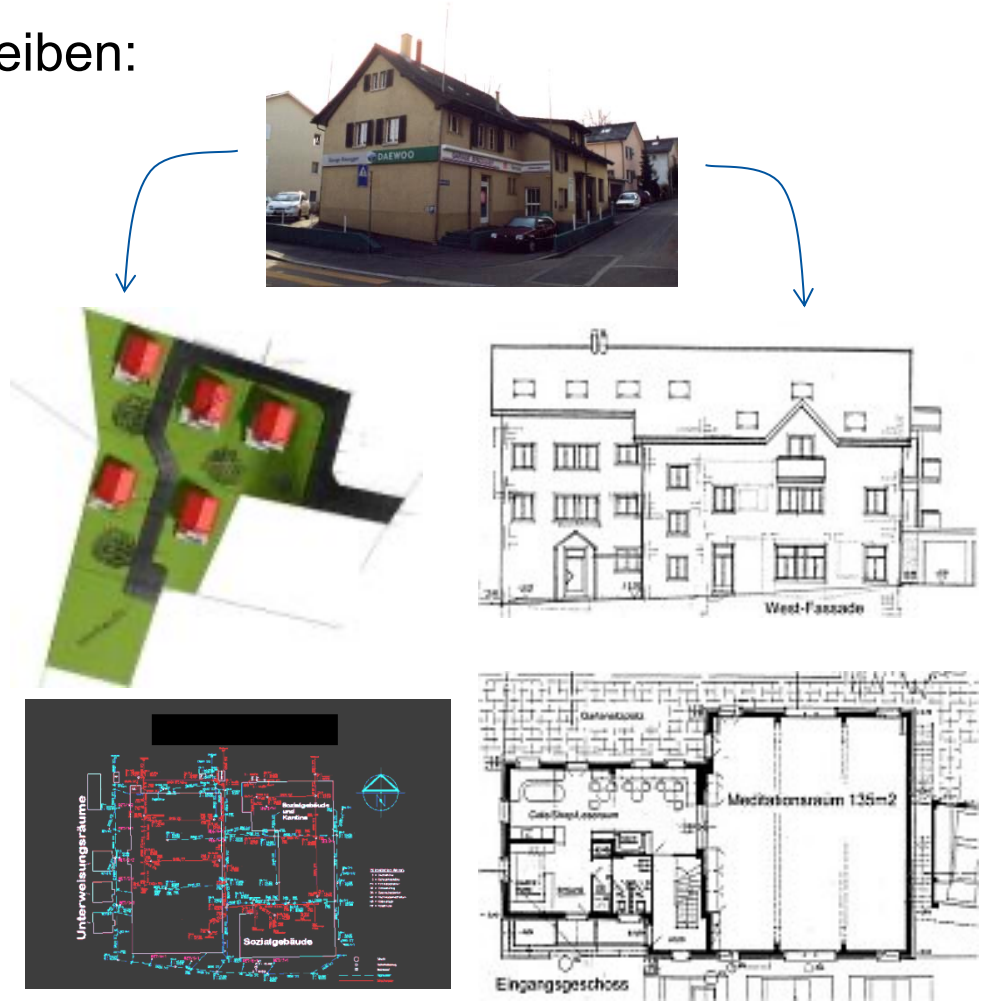
Caroline‘s Physische Komponenten-Architektur





Analogie: Hausbau

- Ein Haus lässt sich durch verschiedene Pläne beschreiben:
  - Grundriss
  - Aufriss
  - Lageplan
  - Elektrischer Anschlussplan
  - Kanalisation
  - …
- Jeder Plan
  - hat eine bestimmte Aufgabe
  - stellt einen Ausschnitt des Hauses dar
  - hat unterschiedliche Zielgruppen



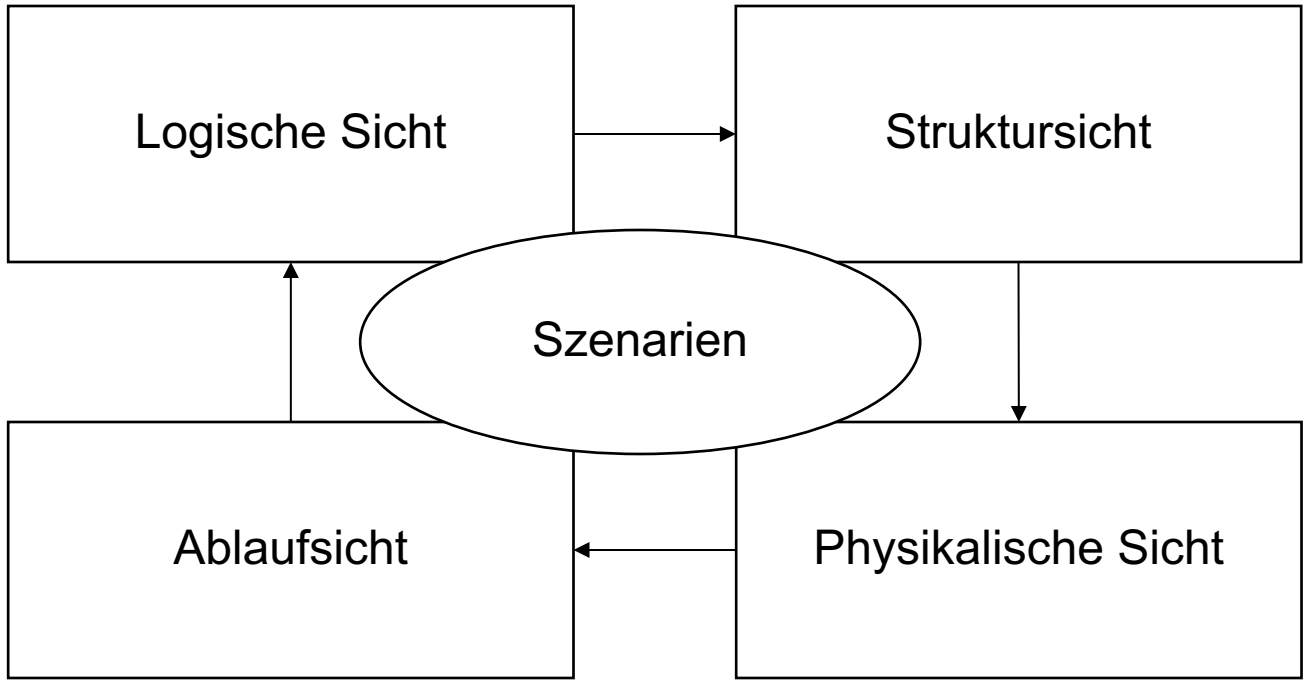
Sicht

- Eine Sicht ist eine Darstellung eines Systems, die nur die Elemente und Beziehungen enthält, die für eine bestimmte Perspektive relevant sind.

- Verschiedene Sichten bilden eine Gesamtspezifikation

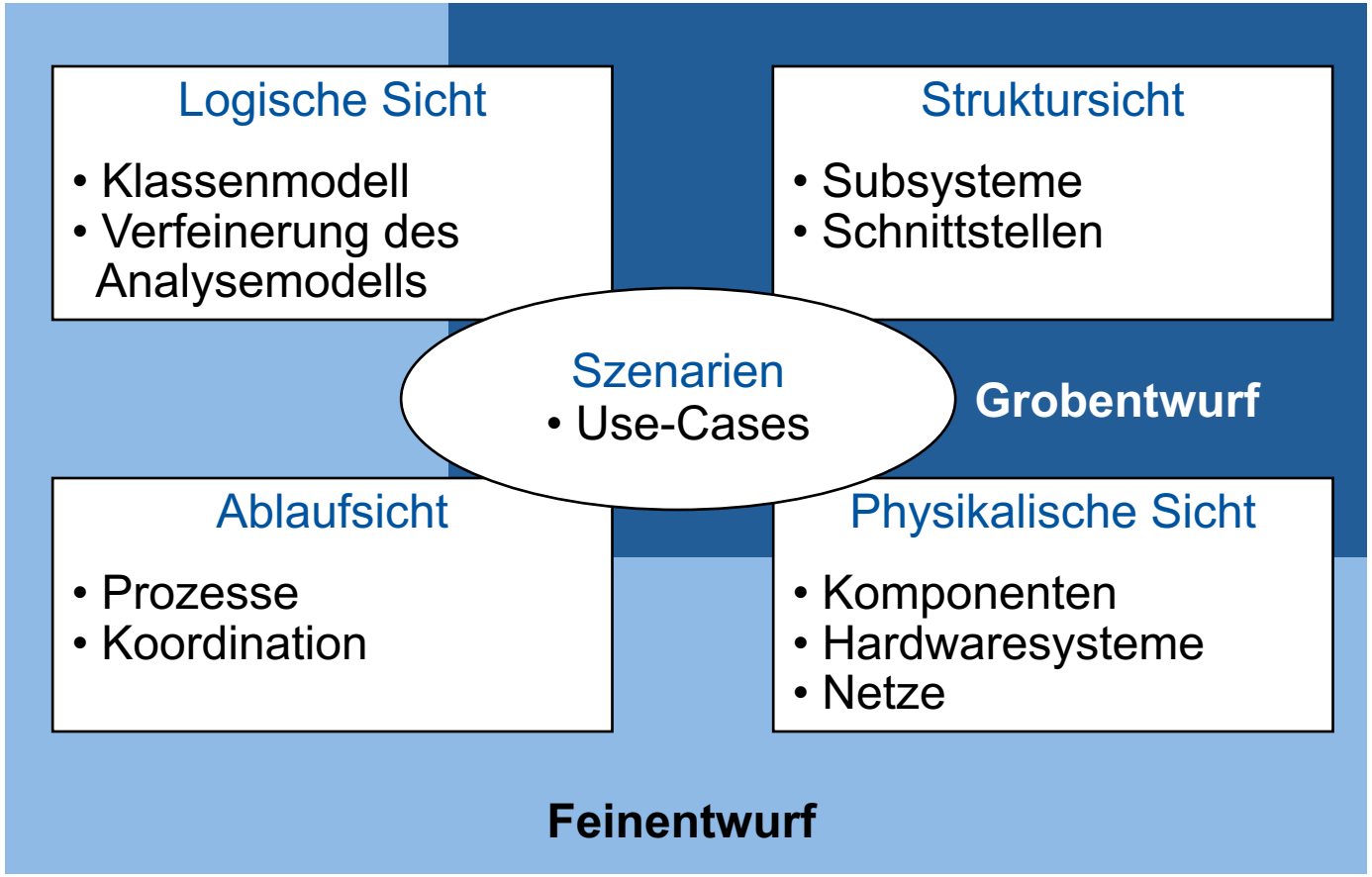
- Herausforderungen:
  - Konsistenz zwischen Sichten
  - Vollständigkeit
  - Möglichkeit zur Abstraktion
  - Übersichtlichkeit der Darstellung
  - Realitätstreue der Sicht
  - Beschreibungssprachen

„4+1 Sichten“-Modell der Softwarearchitektur (aus dem Rational Unified Process - RUP)

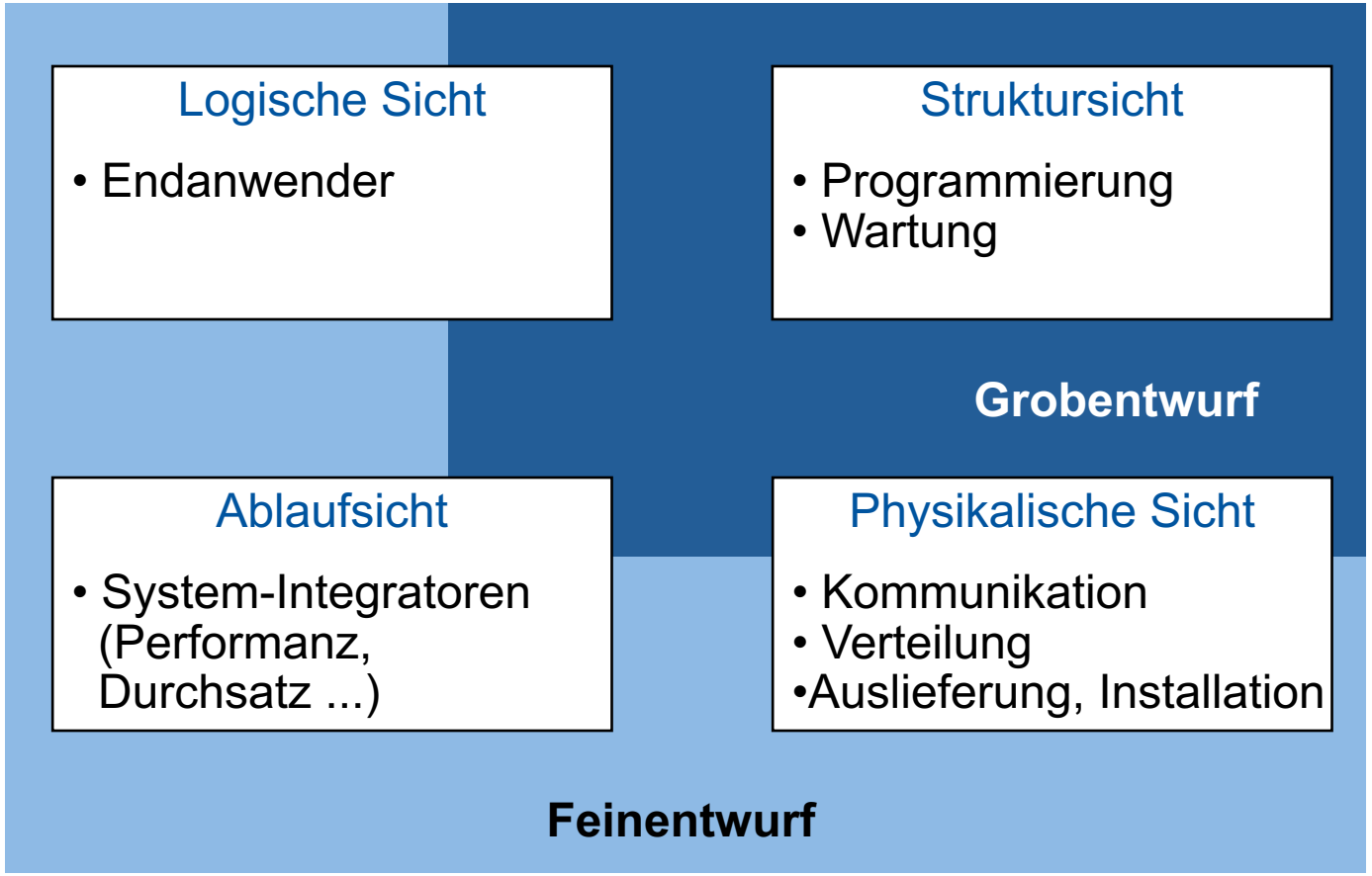


Philippe Kruchten, The 4+1 view model of architecture, IEEE Software, November 1995, 12(6), pp. 42-50

Modellierungstechniken und Kern-Elemente der 4+1 Sichten

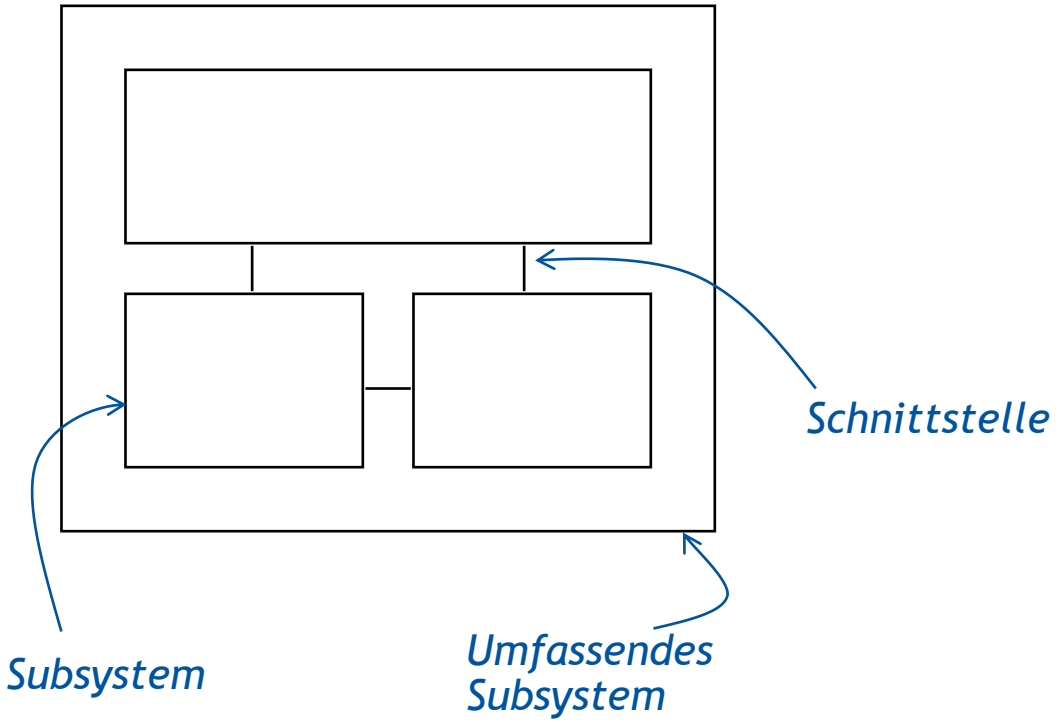


Primäre Zielgruppe/Aufgabe jeder der vier Sichten



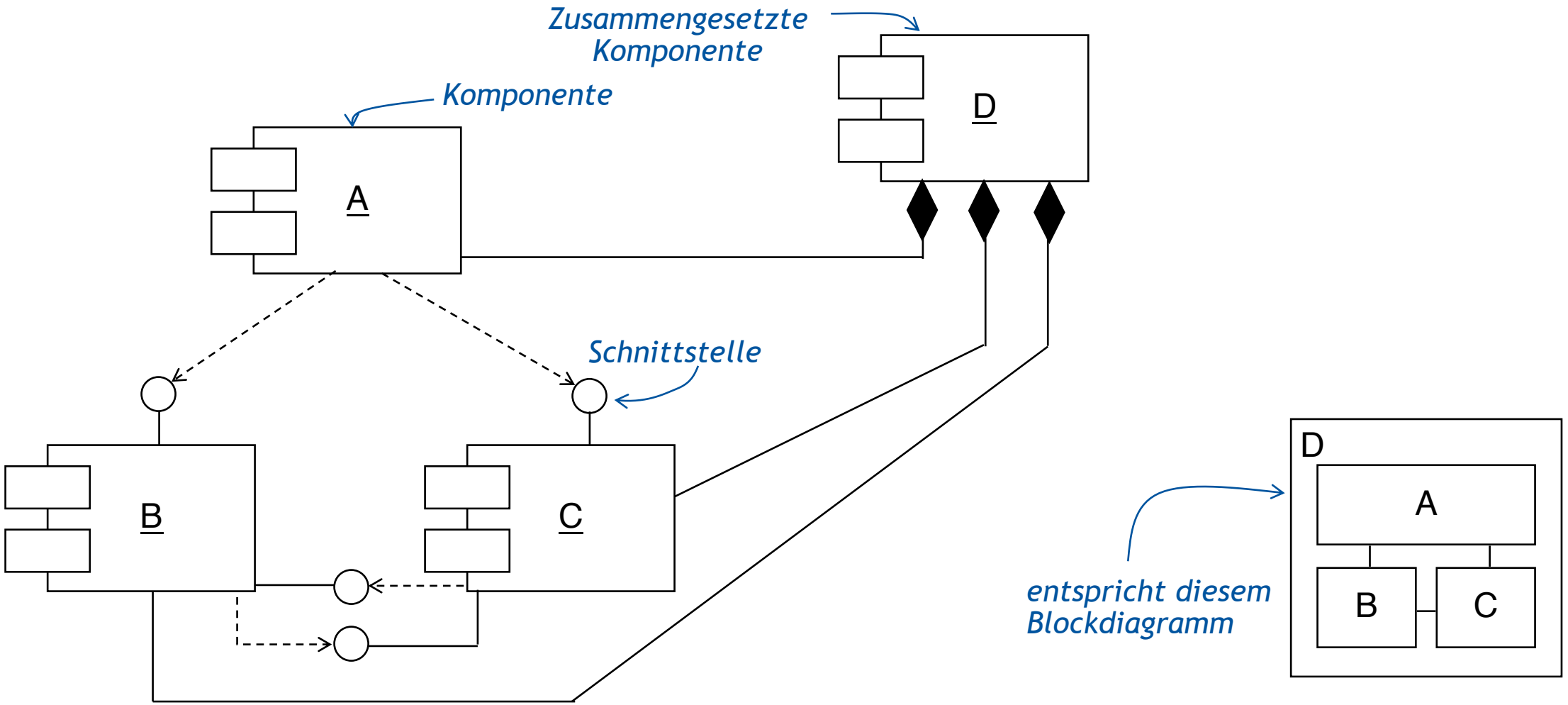
Blockdiagramme

- Blockdiagramme sind ein verbreitetes Hilfsmittel zur Skizzierung der logischen **Struktur** einer Systemarchitektur.



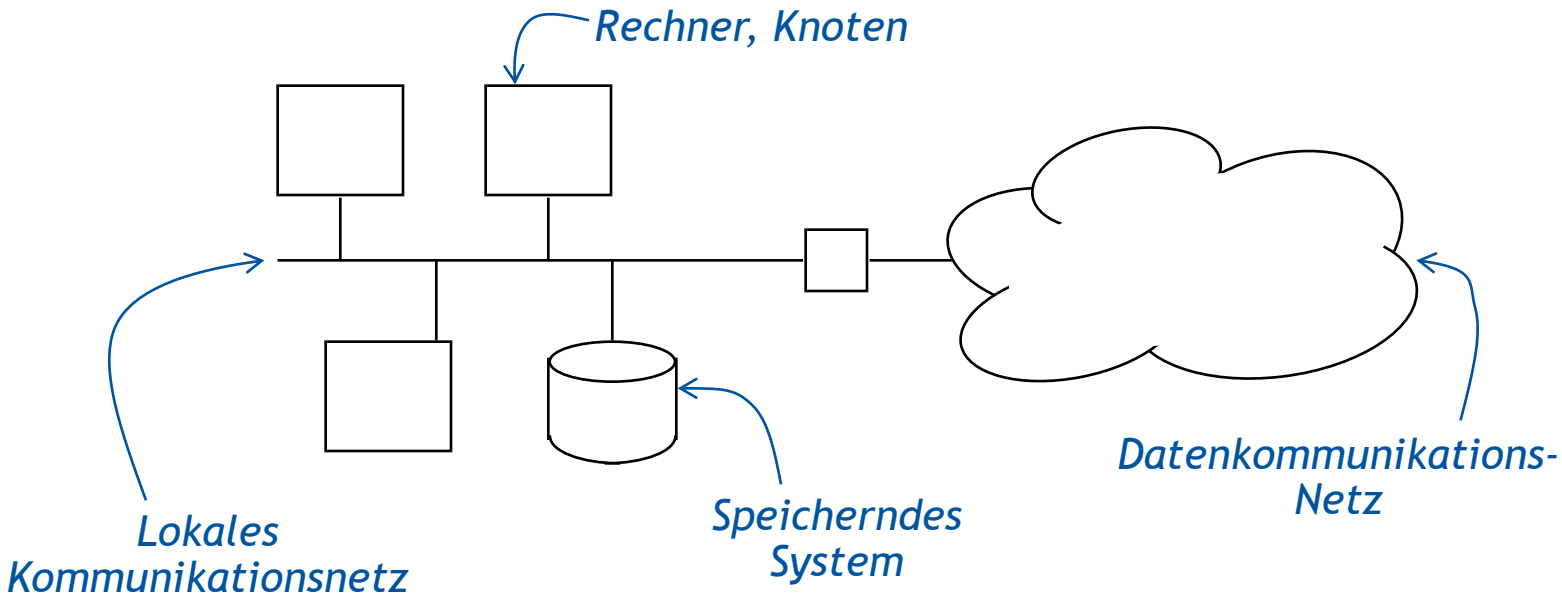
- Subsystem umfasst Objekte bestimmter Klassen
- Schnittstelle ist klar definiert (Aufrufschnittstelle, Kommunikationsprotokoll)

UML: Implementierungsdiagramm



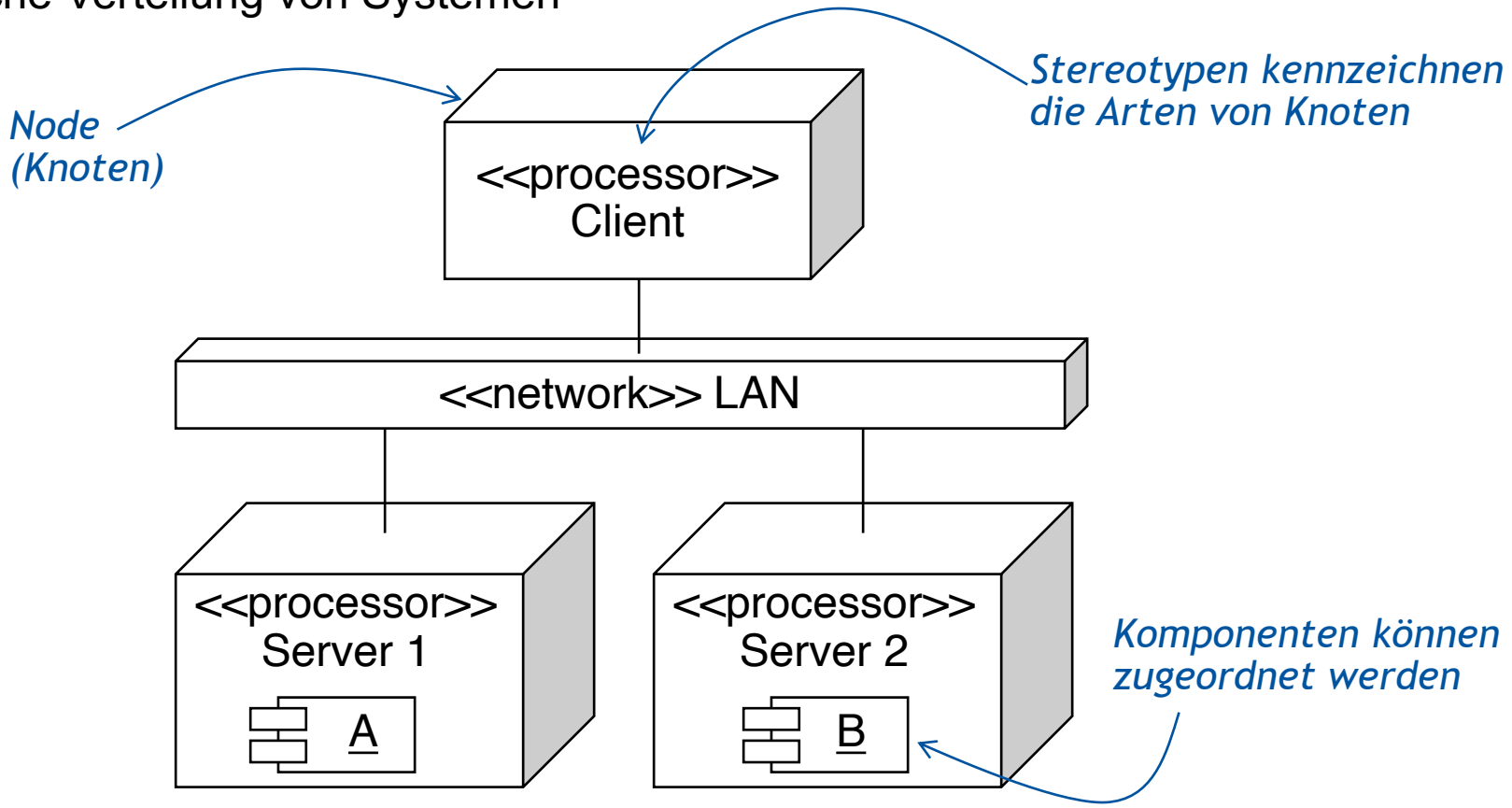
Konfigurationsdiagramme

- Konfigurationsdiagramme sind das meistverbreitete Hilfsmittel zur Beschreibung der physikalischen Verteilung von System-Komponenten.

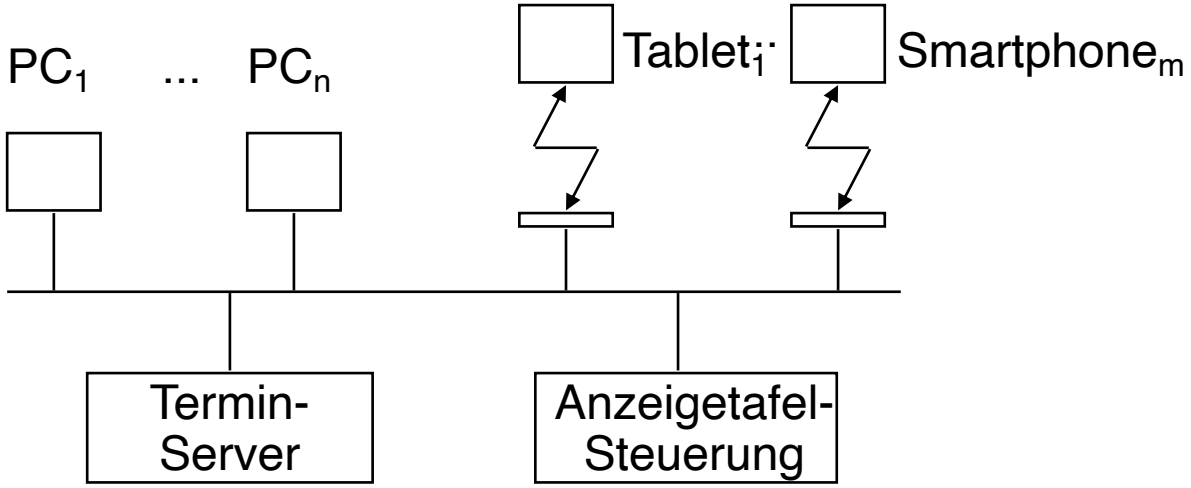


UML: Verteilungsdiagramm

- engl.: *deployment diagram*
- zeigt die physische Verteilung von Systemen

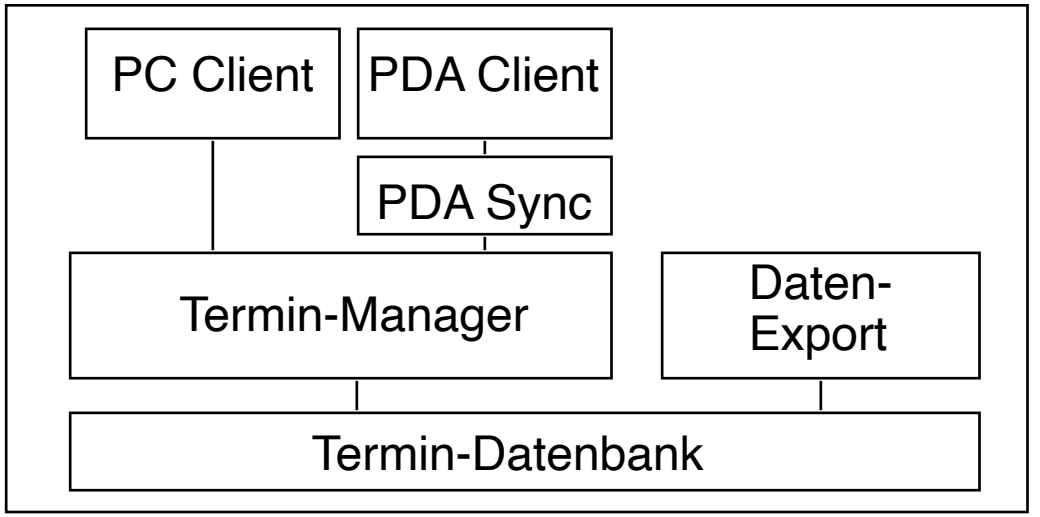


Beispiel Terminverwaltung

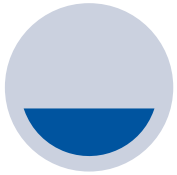


Physikalische Konfiguration

Blockdiagramm



Was haben wir gelernt?



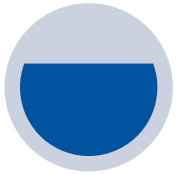
Software-architektur…

…beschreibt die Struktur eines Systems

… besteht aus Komponenten, Schnittstellen und Beziehungen

Essenzielle Eigenschaften; abstrahiert von Details

Unterschiedliche Sichten

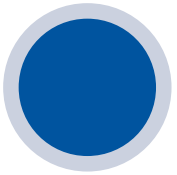


Software-architekt

…kann eine Person oder ein Team sein

… hat und braucht viel Erfahrung

…muss zwischen unterschiedliche Stakeholder-Interessen ausgleichen



Nutzen

Kommunikation mit Stakeholdern

Wesentliche Entwurfsentscheidungen

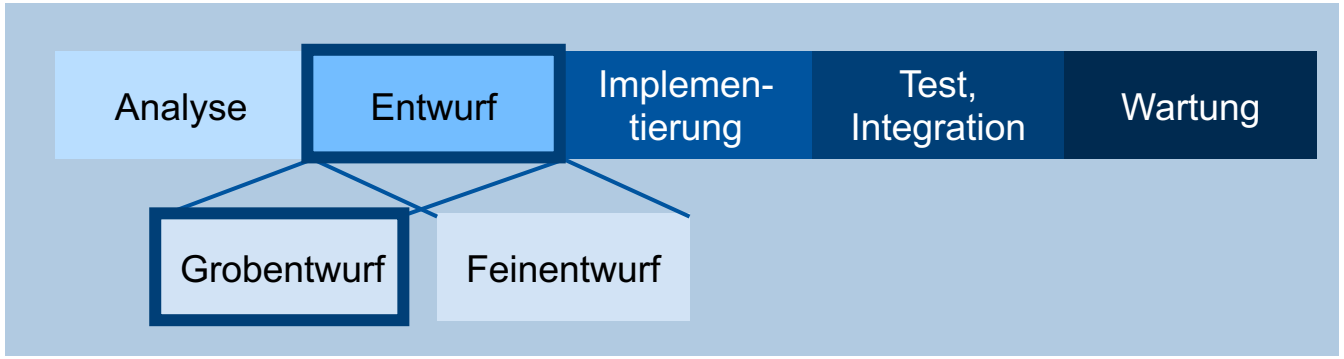
Wiederverwendbare Abstraktion

Fokus auf spezifische Systemeigenschaften

Frühzeitige Analyse möglich

Softwaretechnik

6. Software- & Systementwurf
6.3. Taktiken im Softwareentwurf



Prof. Bernhard Rumpe
Software Engineering
RWTH Aachen

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

Literatur:

- Sommerville 10
- Balzert Band I LE 23
- Balzert Band II LE 17
- Bass, Clements and Kazman (2003) Software Architecture in Practice

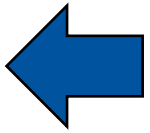
SE Software Engineering | RWTH AACHEN UNIVERSITY

Taktiken („Kleine Methodiken“)

- **Taktik**
  - Technik die Stufe eines Qualitätsattributs in einem System zu verändern
  - kann durch spezifischere Taktiken verfeinert werden
  - wird typischerweise durch Muster realisiert

- Sammlung von Taktiken für einzelne Qualitätsattribute möglich
- Bieten Vokabular für die Auswirkungen des Einsatzes von Mustern auf das Systemverhalten

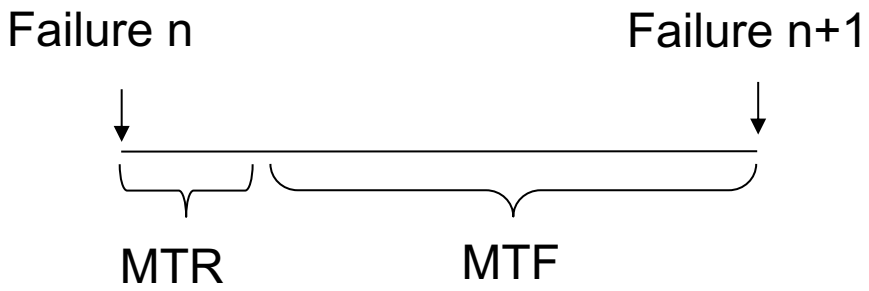
- [BCK03] enthält Sammlung von Taktiken für
  - Verfügbarkeit
  - Modifizierbarkeit
  - Performance
  - Security
  - Testbarkeit
  - Usability



Beispiele

Problem: Verfügbarkeit

- Begriffsbildung:
  - Failure
    - Von außen beobachtbarer Fehler eines Systems
    - ggf. mehrere Faults resultieren in einem Failure
  - Fault
    - Intern aufgetretener Fehler
    - Kann korrigiert werden oder wird zum Failure



- Mean Time to Repair (MTR) = Durchschnittliche Zeit zur Reparatur eines Failures
- Mean Time to Failure (MTF) = Durchschnittliche Zeit zwischen zwei Failures (Ohne Absichtliche Down-Zeiten)
- Verfügbarkeit = MTF/(MTF+MTR)
- Geplante Wartungsarbeiten werden nicht gerechnet.

Taktiken für Verfügbarkeit | 1

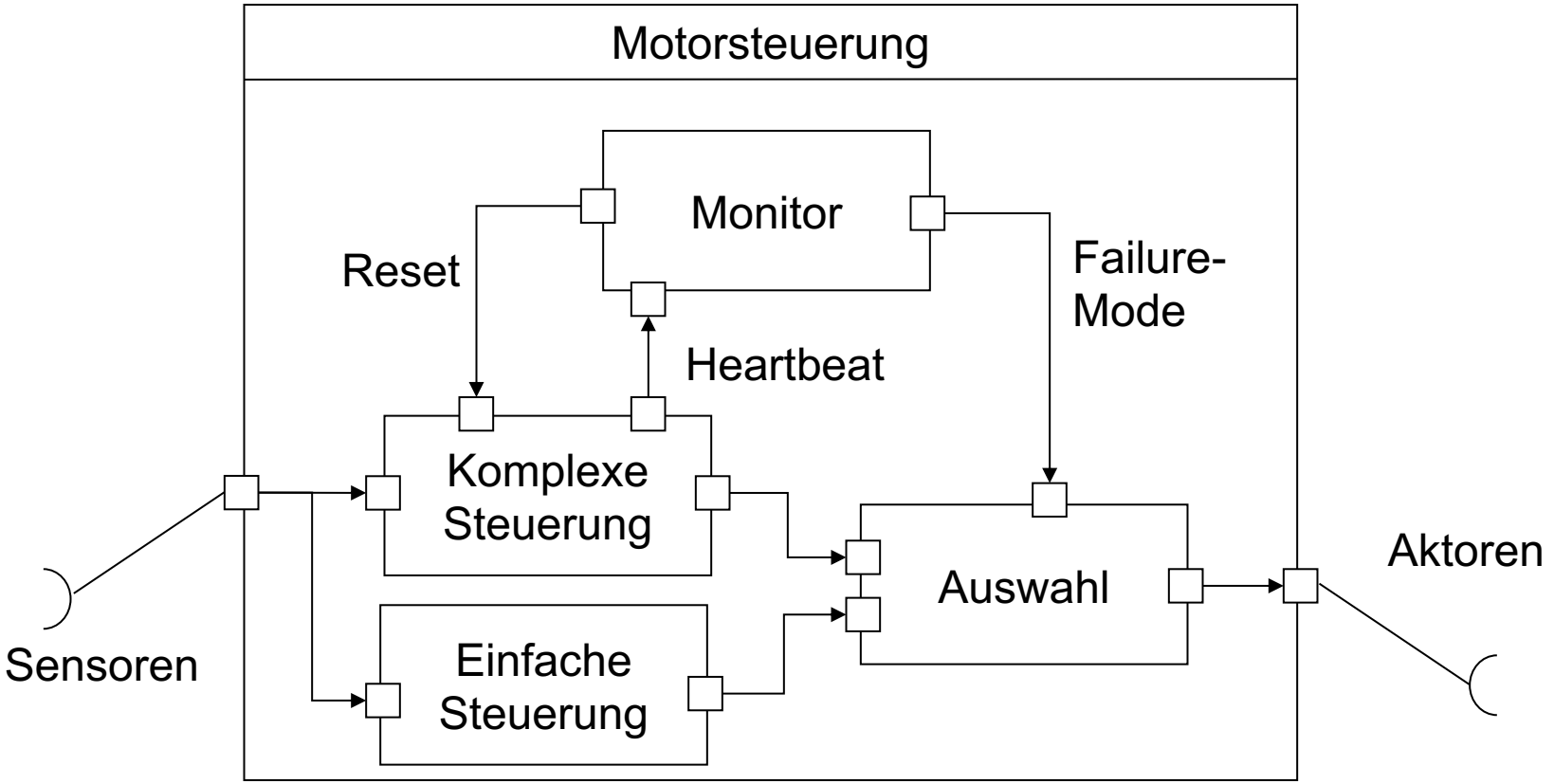
- Fehlererkennung (Fault)
  - Exception
    - Delegation der Fehlerbehandlung an eine Fehlerbehandlungskomponente
  - Ping/Echo
    - Eine Komponente pingt alle anderen in regelmäßigen Abständen an und kontrolliert die Antworten
  - Heartbeat
    - Komponenten müssen sich regelmäßig melden
    - Vorteilhaft falls bereits regelmäßige Kommunikation stattfindet

Taktiken für Verfügbarkeit | 2

- Fehlerbehandlung (Fault)
  - Abstimmen
    - Möglich bei redundanten Systemen
  - Aktive Redundanz
    - Redundante Komponenten laufen parallel und sind beide aktiv
    - Bei Ausfall einer Komponente bleibt das System lauffähig
  - Passive Redundanz
    - Nur eine Komponente interagiert mit der Umgebung
    - Die übrigen werden fortlaufend auf den neusten Stand gebracht
    - Bei Ausfall übernimmt eine passive Komponente die aktive Rolle
  - Spare
    - Redundantes System, das mehrere Komponenten übernimmt
    - Kann eine Rolle dynamisch übernehmen

Beispiel: Heartbeat + Redundanz

- Monitor und Auswahl haben zusätzlich eine einfache Steuerungsfunktion
- Beispiel: Motorsteuerung

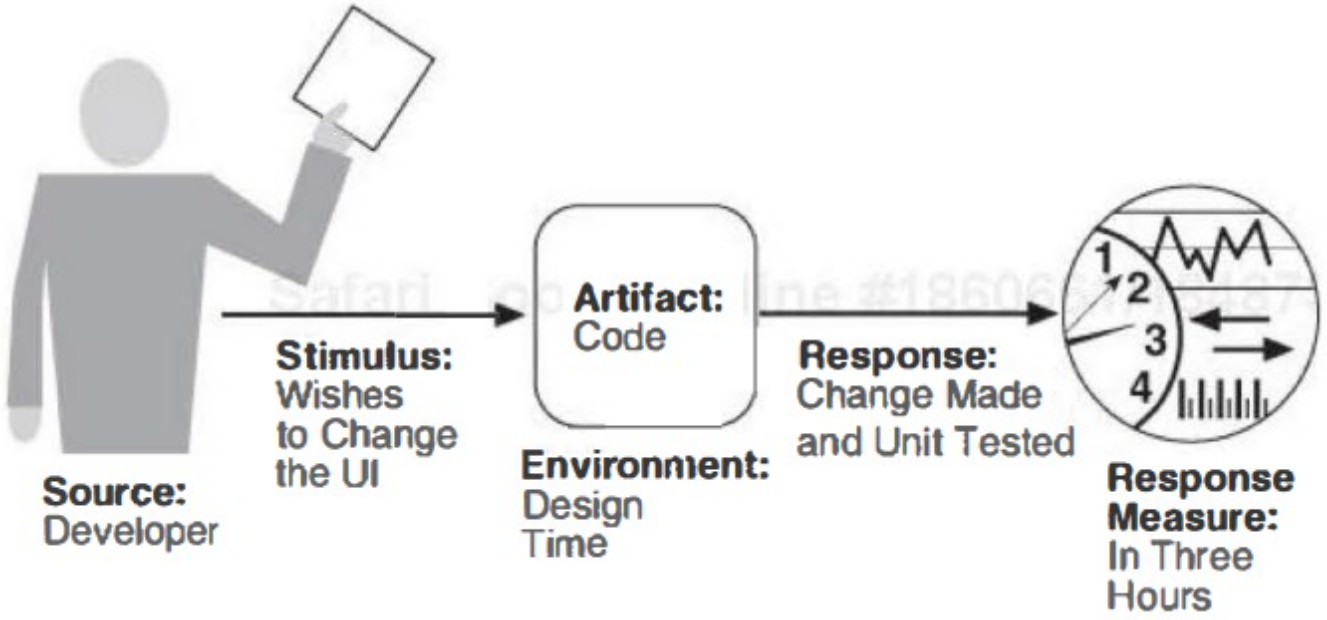


Taktiken für Verfügbarkeit | 3

- Redundante Komponente wiedereinführen (nach Abschaltung durch Failure)
  - Schattenoperation
    - Komponente mit Failure beobachtet zunächst das System
    - vergleicht Verhalten mit redundanten Komponenten
    - kehrt nach kurzer Zeit in den Betrieb zurück
  - Zustands-Resynchronisation der redundanten Komponenten
    - Komponenten zurücksetzen, upgrade auf den aktuellen Zustand
    - Kopie von redundanten Systemen
    - kann sehr aufwändig sein, wenn das Synchronisationsprotokoll komplex ist
  - Checkpoint/Rollback
    - Periodisch wird ein Checkpoint erstellt
    - In diesen Zustand kann das System zurückgesetzt werden (Rollback)

Problem: Modifizierbarkeit

- Modifizierbarkeit
  - Messbar durch Zeit bzw. Kosten für
    - die Implementierung
    - das Testen
    - Ausliefern von Änderungen
  - wird erschwert durch
    - die Abhängigkeit von Komponenten untereinander
    - fehlende Lokalisierung von Funktionalität
- „Ripple-Effekt“
  - Änderungen an anderen Komponenten, die indirekt notwendig werden, weil eine Komponente anzupassen ist.



Quelle: [BCK03]

Taktiken für Modifizierbarkeit | Reduktion der betroffenen Komponenten

- Kapselung (hohe Kohäsion)
  - Möglichst wenig öffentliche Schnittstellen
  - Private Methoden können leicht geändert werden (d.h. sind ohne Auswirkung auf andere Komponenten)
- Verbindungen reduzieren (geringe Kopplung)
  - Weniger Komponenten werden beeinflusst
- Vorhandene Schnittstellen erhalten
  - Neue Schnittstellen nur zusätzlich einführen
  - Gefahr:
    - Komplexer werdendes System
    - Spätere Änderungen aufwändiger (vgl. Agile Methoden und Simplicity-Prinzip, vgl. @deprecated)
- Vermittler (Broker)
  - Je nach Verbindungstyp, z.B.
    - Bus statt direkter Verbindung
    - Ort der Komponenten: Namensdienste
    - Entwurfsmuster Fassade, Adapter, Mediator

Taktiken für Modifizierbarkeit | Reduktion der notwendigen Änderungen

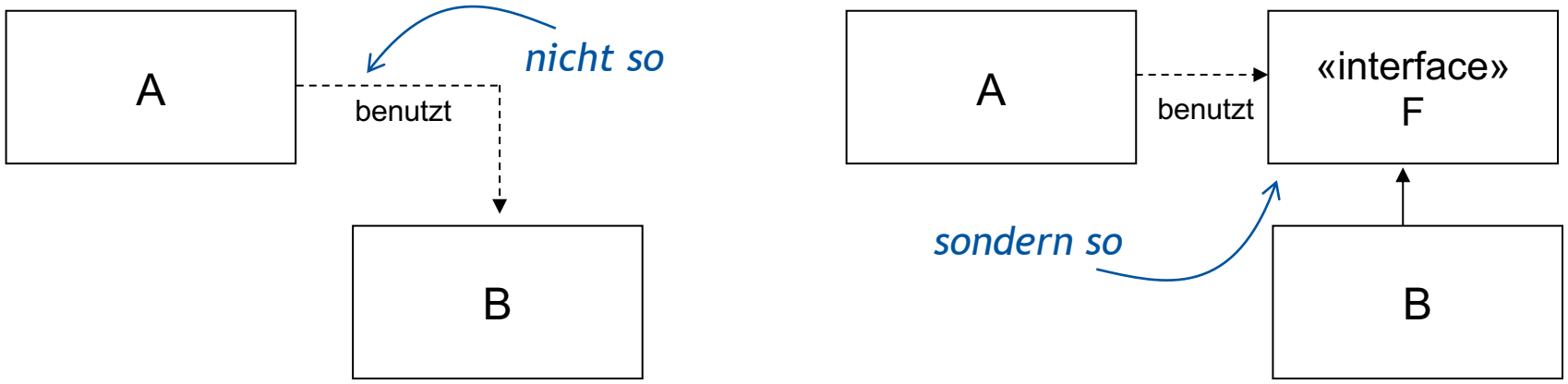
- Kohärenz erhalten
  gut sind:
  - Eine Aufgabe wird durch genau eine Komponente erfüllt
  - Die Realisierung eines Dienstes ist nicht über das System verteilt

- Änderungen antizipieren
  - Mögliche Änderungen gedanklich durchspielen
    (nicht implementieren! vgl. XP-Grundsätze)
  - Dabei Fragen beantworten:
    - Betreffen fundamental unterschiedliche Änderungen dasselbe System?
    - Betrifft eine Änderung mehrere Komponenten?
  - Falls ja, deutet das auf mangelnde semantische Kohärenz hin

Taktiken für Modifizierbarkeit | Verzögern des Bindungs-Zeitpunkts

- Registrierung zur Laufzeit
  - Plug-In-Fähigkeit von Komponenten
- Konfigurationsdateien
  - Konfiguration bei Auslieferung oder Rekonfiguration zur Laufzeit
- Polymorphie
  - Erlaubt das Ersetzen durch abgeleitete Klassen/Komponenten (zur Laufzeit)
- Komponentenersetzung
  - Dynamisches Zusammenstellen der Anwendung bei Auslieferung
- Standardisierte Protokolle
  - Erlauben die Kooperation unabhängiger Prozesse

1) High-Level-Komponenten sollen nicht von Low-level-Komponenten abhängen. Beide sollen nur von Abstraktionen (Interfaces) abhängig sein.
2) Abstraktionen sollen nicht von Details abhängen, sondern die Details von den Abstraktionen



- Vorteile:
  - Anpassungen unten beeinflussen obere Ebenen nicht!
  - leichtere Modifizierbarkeit und Testbarkeit

Robert C. Martin: The Dependency Inversion Principle. 1996

Problem: Software Security

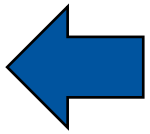
> “Software Security is the idea of engineering software so that it continues to function correctly under malicious attack.”
>
> *[Mc Graw]*

- Eine Schutzmaßnahme schützt vor einem oder mehreren Angriffen
  - es gibt nicht die „Silver Bullet“
  - Häufige Fehleinschätzung: „Wir haben eine Firewall also sind wir sicher.“
  - Firewall schützt auf Netzwerkebene nicht gegen Angriffe auf Applikationsebene wie SQL Injection
- Auswahl der richtigen Schutzmaßnahmen, die vor einer Bedrohung schützen
- Nur wenige Schutzmaßnahmen lassen sich kapseln z.B. Verschlüsselung

McGraw, Gary. Software security: building security in. Addison-Wesley Professional, 2006

Liste von Taktiken für Security

- Securing the Weakest Link
- Defense in Depth
- Failing Securely
- Least Privilege
- Separation of Privilege
- Economy of Mechanism
- Least Common Mechanism
- Reluctance to Trust
- Never Assuming that your Secrets are Safe
- Complete Mediation
- Psychological Acceptability
- Promoting Privacy



Beispiele

- Earn or give, but never assume, *trust.*
- Strictly *separate* data and control instructions, and never process control instructions received from *untrusted sources*
- Define an approach that ensures *all data* are explicitly *validated*
- Understand how *integrating external components* changes your attack surface
- ...

http://cybersecurity.ieee.org/center-for-secure-design/avoiding-the-top-10-security-flaws.html
https://buildsecurityin.us-cert.gov/articles/knowledge/principles/design-principles

Taktiken für Security | 1

Securing the Weakest Link

- Sicherheitsmaßnahmen ergeben eine Kette, die nur so stark ist wie ihr schwächstes Glied
- Angreifer greifen dort an, wo der Widerstand am geringsten ist

Taktik:

- Schwächste Stelle des Gesamtsystems identifizieren
- Dort Sicherheitsmaßnahmen einbauen

Failing Securely

- Standardmäßig Zugriff verweigern, Benutzern explizit Zugriff gewähren
- Im Fehlerfall keine (unsicheren) Aktionen zulassen

Taktiken für Security | 2

Economy of Mechanism

- Fehler in Sicherheitsmaßnahmen fallen im normalen Gebrauch und bei funktionalen Tests nicht auf.
- Daher notwendig: Aufwändiges Code Review, Sicherheitstests (Penetrationstests), u.U. Beweis von privacy Eigenschaften

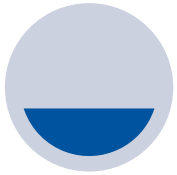
Taktik:

- Keep it simple (KISS)
- Sicherheitsmechanismen (Verschlüsselung, Authentisierung) wiederverwenden (nicht selbst implementieren)

Least Privilege

- Jeder Benutzer/Funktion bekommt *nur* die für ihre Arbeit notwendigen Rechte/Ressourcen
- *Ergebnis*: Übernimmt ein Angreifer die Kontrolle über die Funktion sind die Auswirkungen geringer

Was haben wir gelernt?

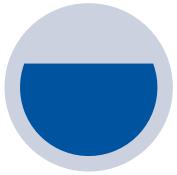


Taktiken…

…sind Techniken um die Stufe eines Qualitätsattributs in einem System zu verändern

…werden oft mit Mustern realisiert

Sammlungen bieten ein Vokabular für die Auswirkungen des Einsatzes von Mustern auf das Systemverhalten



Qualitätsattribute

Verfügbarkeit, z.B. Heartbeat und Redundanz

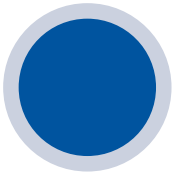
Modifizierbarkeit, z.B. Dependency Inversion

Performance

Security, z.B. securing the weakest link, KISS

Testbarkeit

Usability, etc.



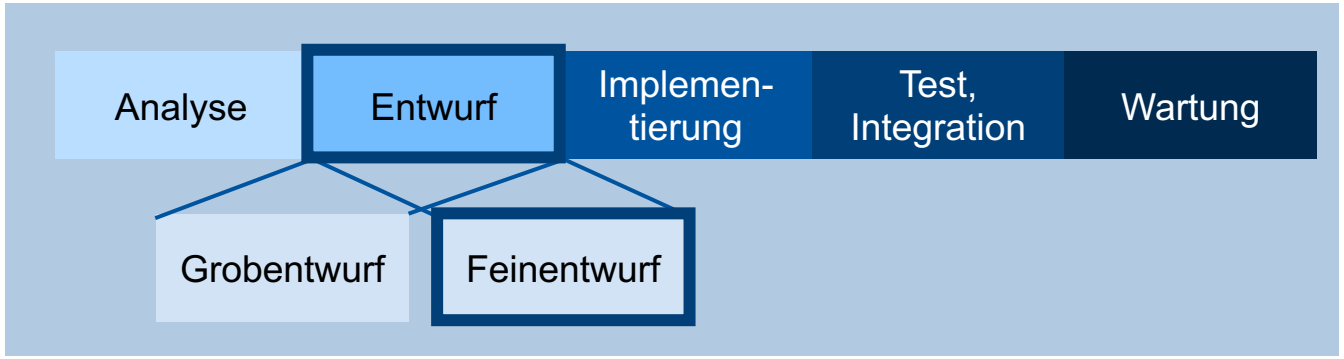
Einsatz

Betrachtung nicht-funktionaler Anforderung

Realisierung von Taktiken für die Verbesserung der Systemarchitektur

Softwaretechnik

6. Software- & Systementwurf
6.4. Objektorientierter Feinentwurf mit Klassendiagrammen



Prof. Bernhard Rumpe
Software Engineering
RWTH Aachen

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

SE Software Engineering | RWTH AACHEN UNIVERSITY

Warum, was, wie und wozu? Feinentwurf

Warum?

- Schritt zwischen Architektur und Implementierung
- Effizienz kommt von guter Planung

Was?

- Klassendiagramme
- Verfeinerung des Analysemodells

Wie?

- Mehr fachliche Details z.B. Vollständigkeit, Sichtbarkeiten und Datentypen, Navigationsrichtung,…
- Zusätzliche technische Klassen/Pakete z.B. Plattformabhängig

Wozu?

- Basis für die Entwicklung des Systems

Objektorientierter Feinentwurf

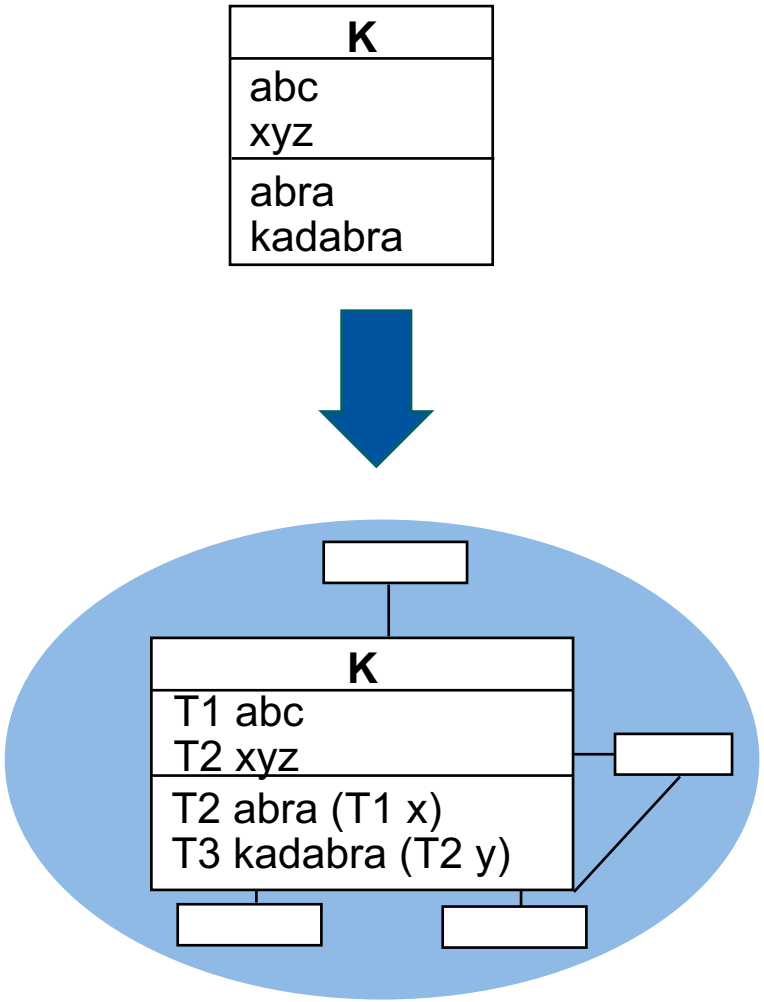
- Ausgangspunkt:
  - Grobdefinition der Architektur:
    - Zerlegung in Subsysteme (evtl. unter Verwendung von Standardarchitekturen)
    - Verteilungskonzept
    - Ablaufmodell

- Ergebnis des Feinentwurfs:
  - OO-Modell für jedes Subsystem der Architektur
  - OO-Modell für unterstützende Subsysteme
    - unter Berücksichtigung gewählter Technologien
  - Spezifikationen der Klassen
  - Spezifikationen von externen Schnittstellen

Verfeinerung und Umbau des Analysemodells

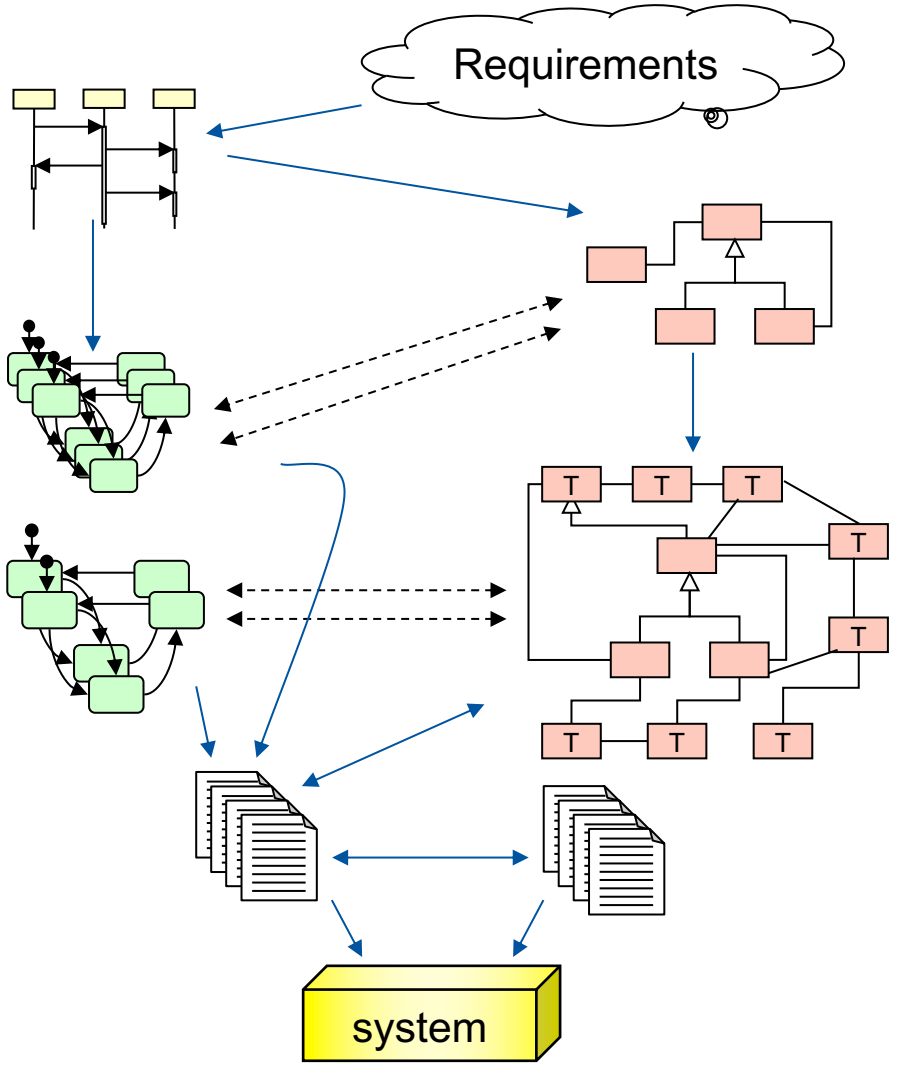
- Fachlicher Kern: Mehr Details als im Analysemodell
  - Listen der Attribute und Operationen: (einigermaßen) vollständig
  - Attribute und Operationen: Datentypen, Sichtbarkeit
  - Operationen: Spezifikation (z.B. Vor- und Nachbedingungen)
  - Assoziationen/Aggregationen: Navigationsrichtung, Ordnung, Qualifikation

- Zusätzliche "technische" Klassen/Pakete:
  - Einbindung in Infrastruktur, Frameworks, Altsysteme etc.
  - Anpassungs- und Entkopplungsschichten für gewählte Technologien
    (z.B. Datenzugriffsschicht, CORBA-Schnittstellen, XML-Anschluss...)



UML im Entwurf

- Generell: Analysemodelle werden im Entwurf umgebaut
- Insbesondere Klassendiagramme erhalten dazu eine neue Bedeutung:
  - In der Analyse repräsentieren Klassen Dinge der realen Welt
  - Im Entwurf stellen Klassen einen Teil des Softwaresystems dar
  - Es findet eine Detaillierung und Präzisierung statt
- Statecharts werden detailliert oder ein Einzelspezifikationen von Methoden zerlegt
- Andere UML-Diagramme werden im Feinentwurf vor allem als Vorlagen (Aktivitäts-, Sequenzdiagramme, Use Cases) oder zur Strukturierung im Grobentwurf (Komponentendiagramme) eingesetzt, selbst aber eher *nicht weiter detailliert*.

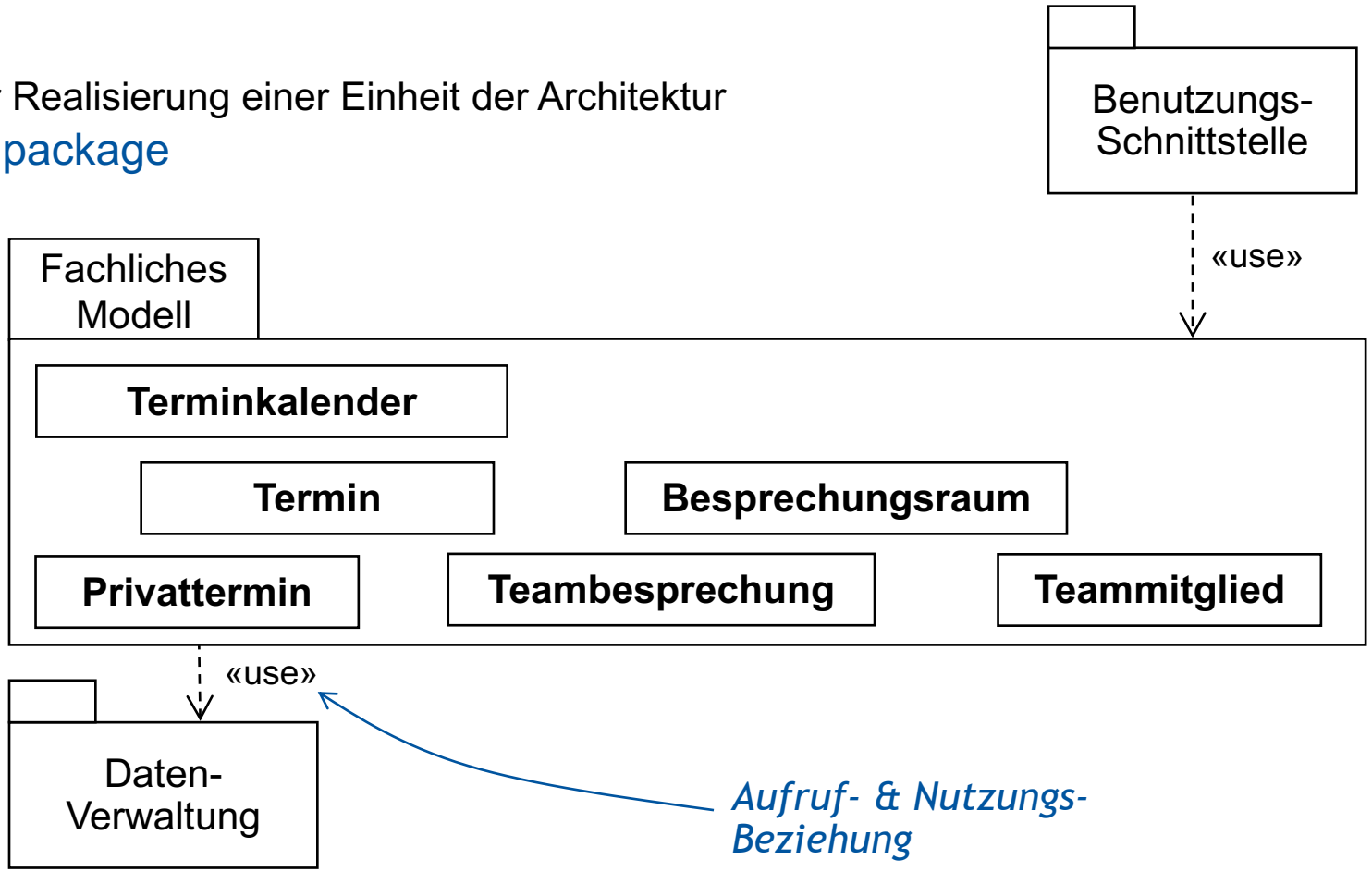


UML zum logischen Detailentwurf

Analyse-Modell	Entwurfs-Modell
Notation: UML	Notation: UML
Objekte: Fachgegenstände	Objekte: Softwareeinheiten
Klassen: Fachbegriffe	Klassen: Schemata
Vererbung: Begriffsstruktur	Vererbung: Programmableitung
Annahme perfekter Technologie	Erfüllung konkreter Rahmenbedingungen
Funktionale Essenz	Gesamtstruktur des Systems
Völlig projektspezifisch	Ähnlichkeiten zwischen verwandten Projekten
Grobe Strukturskizze	Genaue Strukturdefinition
	Mehr Struktur & mehr Details

Pakete und Subsysteme

- UML:
  - Pakete als „Ordner“
  - “Subsystem”: Paket zur Realisierung einer Einheit der Architektur
- Java-Sprachkonstrukt: package



Sichtbarkeit

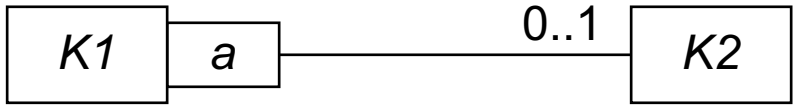
Benutzes Sichtbarkeits-Symbol …						
UML	+	#	–			
Java / C++	**public**	**protected**	**private**	*(default)*		
… führt zu Sichtbarkeit für:				Java	C++ class	UML, C++ struct
Gleiche Klasse	**ja**	**ja**	**ja**	**ja**	**ja**	**ja**
Andere Klasse, gleiches Paket	**ja**	**ja/nein**[1]	**nein**	**ja**	**nein**[2]	**ja**[2]
Andere Klasse, anderes Paket	**ja**	**nein**	**nein**	**nein**	**nein**[2]	**ja**[2]
Unterklasse, gleiches Paket	**ja**	**ja**	**nein**	**ja**	**nein**[2]	**ja**[2]
Unterklasse, anderes Paket	**ja**	**ja**	**nein**	**nein**	**nein**[2]	**ja**[2]

[1] In Java „ja“, in UML und C++ „nein“
[2] In C++ default bei Klassen: private, bei struct: public

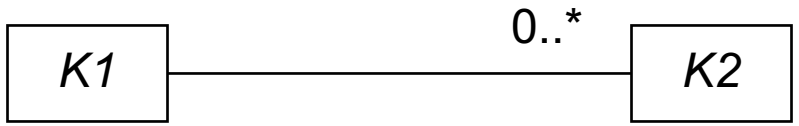
Qualifizierte Assoziation (Erinnerung)

- **Definition:** Eine Qualifikation (*Qualifier*) ist ein Attribut für eine Assoziation zwischen Klassen K1 und K2, durch das die Menge der zu einem K1-Objekt assoziierten K2-Objekte *partitioniert* wird.
  Zweck der Qualifikation ist direkter Zugriff unter Vermeidung von Suche.

- **Notation:**

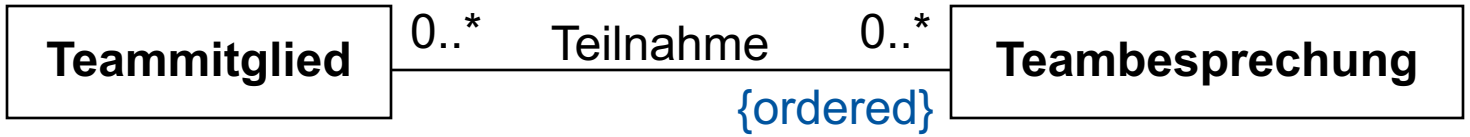


als Detaillierung von:



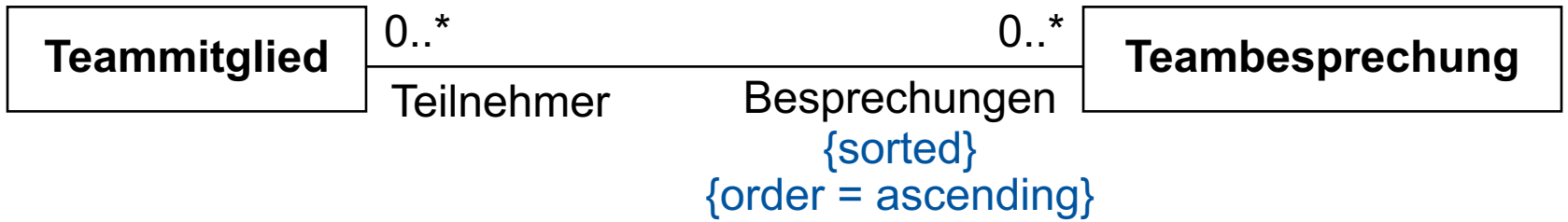
- Bedeutung vor allem im Zusammenhang mit *Datenbanken (Schlüssel, Indizes in Look-up Tabellen)*, aber auch mit geeigneten Datenstrukturen nach Java abbildbar.
  - Qualifizierung wird oft erst im Design eingeführt

Geordnete und sortierte Assoziation (Erinnerung)



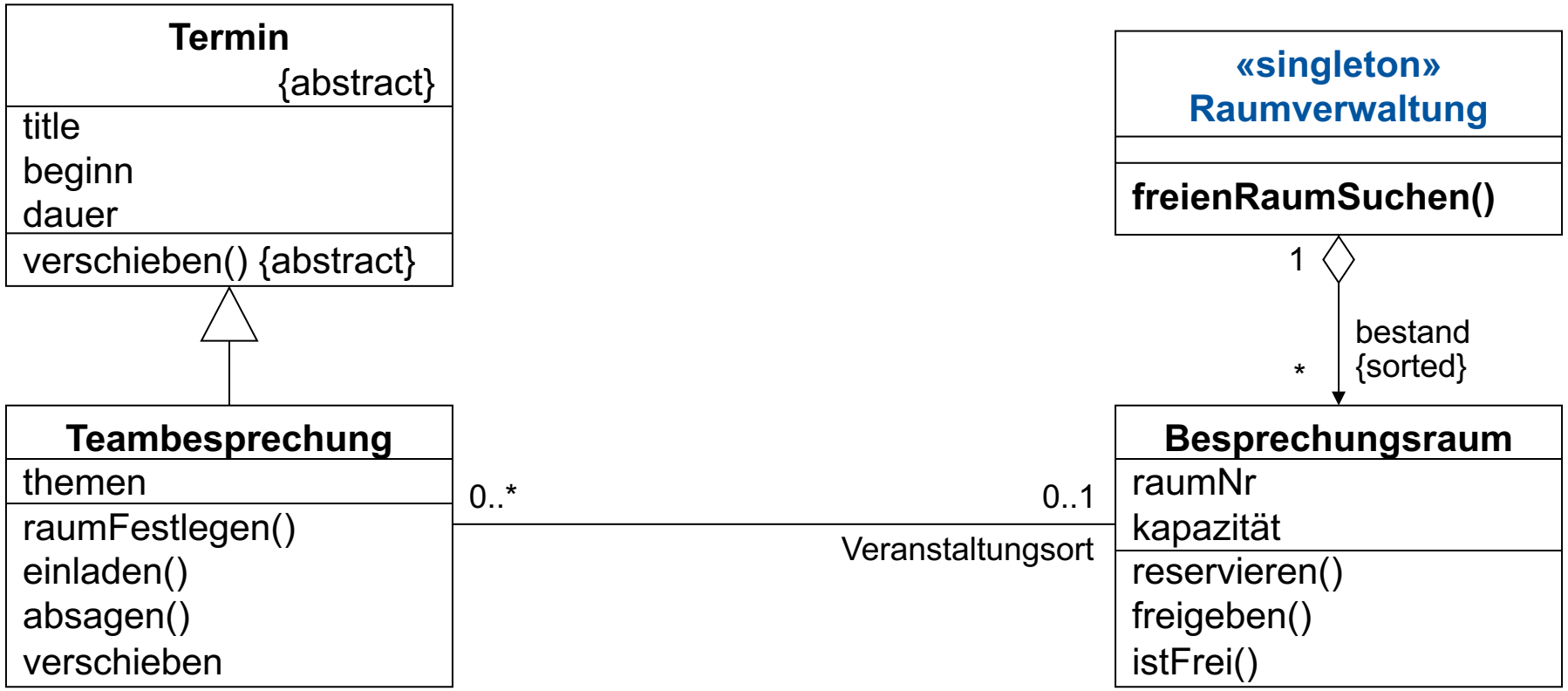
- {ordered} an einem Assoziationsende:
  - Es besteht eine feste Reihenfolge, in der die assoziierten Objekte durchlaufen werden können
  - Oft ist Zugriff über Listen, Iteratoren möglich

- Default: die assoziierten Objekte sind als *Menge* strukturiert.
- Weitere Einschränkungen möglich, z.B. die Forderung nach Sortierung gemäß bestimmter Attribute:



Verwaltungsklassen: Singletons

- Ein Singleton ist eine Klasse, die genau einmal instantiiert wird, zB Verwaltung kann damit gut modelliert werden

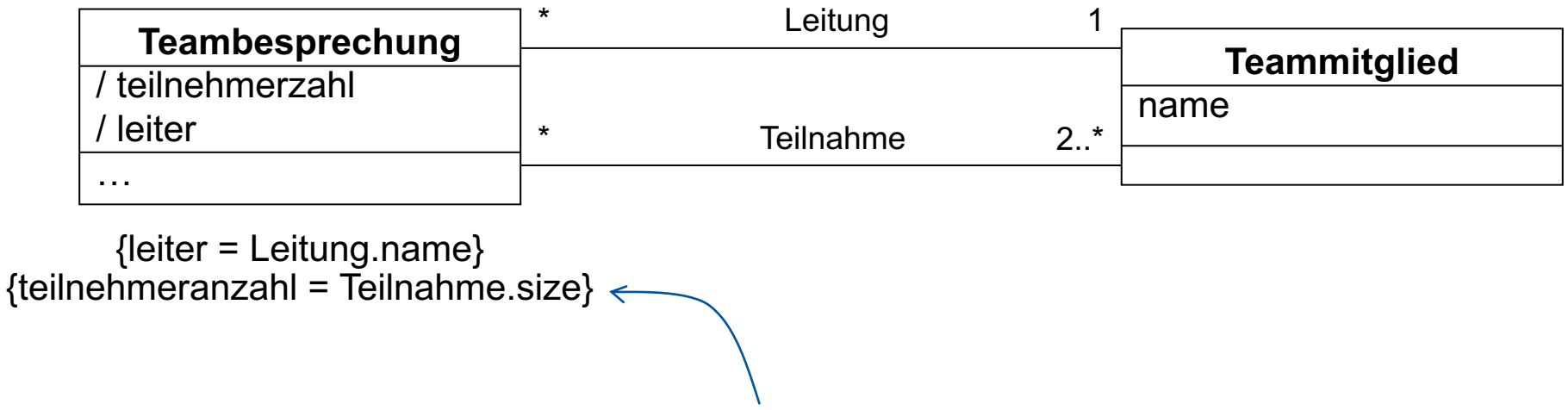


Abgeleitete (redundante) Elemente

- **Definition:** Ein *abgeleitetes* Modellelement (z.B. Attribut, Assoziation) ist ein Modell-Element, das aus anderen Elementen rekonstruiert werden kann.
- **Notation:**

  / Modellelement oder
  Modellelement {derived}

- **Beispiele:**



- Ableitung kann mit der Object Constraint Language OCL, ein weiterer Teil der UML, formuliert werden.

Parameter und Datentypen für Operationen

- Analysephase:
  - oft Operationsname ausreichend
  - ggf. Parameternamen ohne weitere Information

Besprechungsraum
raumNr kapazität
reservieren() freigeben() istFrei()

- Entwurfsphase:
  - Parameter und Datentypen der Operationen genau festlegen!
  - Sichtbarkeiten

Standard-Style

```
- istFrei(beginn: Date, dauer: int):boolean
+ reservieren (für: Termin):boolean
```

Java, C, C++-Style:

```
- boolean istFrei(Date beginn, int dauer)
+ boolean reservieren (Termin für)
```

Spezifikation von Operationen

- Definition:
  - Die Spezifikation einer Operation legt das Verhalten der Operation fest, ohne einen Algorithmus festzuschreiben.

- Grundprinzip:

> Es wird das „*Was*“ beschrieben und noch nicht das „*Wie*“.

- Häufigste Formen von Spezifikationen:
  - Text in natürlicher Sprache (oft mit speziellen Konventionen)
    - Oft in Programmcode eingebettet (Kommentare)
    - Werkzeugunterstützung zur Dokumentationsgenerierung, z.B. “Javadoc”
  - Vor- und Nachbedingungen
  - Tabellen, spezielle Notationen
  - „Pseudocode“ (Programmiersprachenartiger Text)
    - nur mit Vorsicht zu verwenden - oft zu viele Details festgelegt!

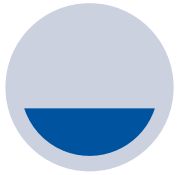
Schnittstellen („Interface”) und abstrakte Klassen

Abstrakte Klasse	Schnittstelle
Enthält Attribute und Operationen	Enthält nur Operationen (und ggf. Konstante)
Kann Default-Verhalten festlegen	Kann (seit Java 8) Default-Verhalten festlegen
Default-Verhalten kann in Unterklassen redefiniert werden	Unterklassen müssen Verhalten definieren
Java: Unterklasse erbt nur von einer Klasse; C++: von mehreren	Java, C++: Eine Klasse kann mehrere Schnittstellen implementieren
	Schnittstelle ist eine spezielle Sicht auf eine Klasse

Zusammenfassung: UML-Klassenmodelle in Analyse und Entwurf

Analyse-Modell	Entwurfs-Modell
Skizze: Teilweise unvollständig in Attributen und Operationen	Vollständige Angabe aller Attribute und Operationen
Datentypen und Parameter können noch fehlen	Vollständige Angabe von Datentypen und Parametern
Noch kaum Bezug zur Realisierungssprache	Auf Umsetzung in gewählter Programmiersprache bezogen
Keine Überlegungen zur effizienten Realisierung von Assoziationen	Navigationsangaben, Qualifikation, Ordnung, Verwaltungsklassen
	Entscheidung über Datenstrukturen
	Anbindung von Benutzungsoberfläche und Datenhaltung an fachlichen Kern

Was haben wir gelernt?

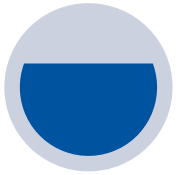


Feinentwurf

Mehr Details im fachlichen Kern

Zusätzliche Klassen und Pakete:

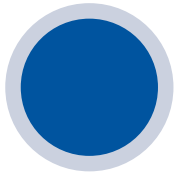
- Einbindung in Infrastruktur, Frameworks, Altsysteme etc.
- Anpassungs- und Entkopplungsschichten für gewählte Technologien



Entwurfsmodell

Klassen stellen einen Teil des Softwaresystems dar

Verwendung als Grundlage für die Implementierung



Details zu…

Paketen und Subsystemen

Sichtbarkeiten, Parameter und Datentypen

Abgeleitete Elemente

Spezifikation von Operationen

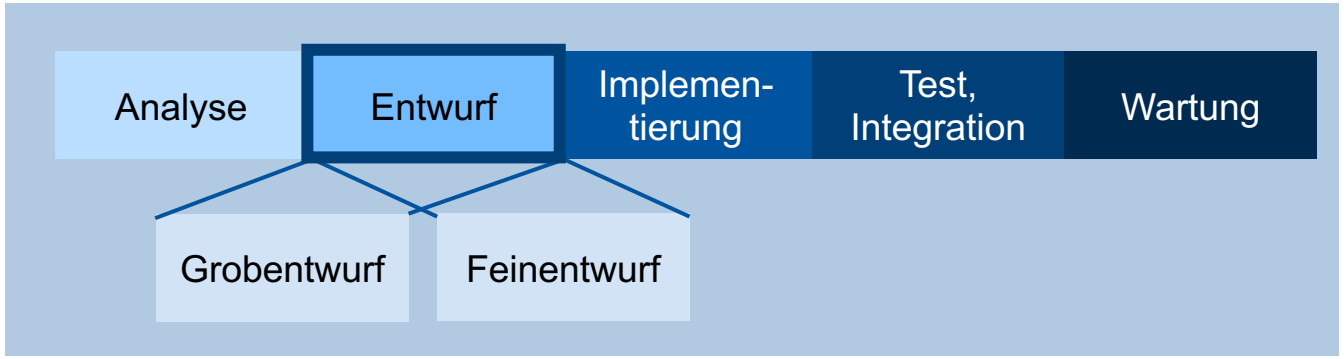
Assoziationen (qualifiziert, geordnet, sortiert)

Verwaltungsklassen (singletons)

abstrakte Klassen und Interfaces

Softwaretechnik

6. Software- & Systementwurf
6.5. Entwurf von Nutzeroberflächen



Prof. Bernhard Rumpe
Software Engineering
RWTH Aachen

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

Literatur:

- Sommerville, I.: Software Engineering, Kap.16
- Shneiderman, B.: Designing the User Interface: Strategies for Effective Human-Computer Interaction. 1998
- Balzert, LE 16 Software-Ergonomie

SE Software Engineering | RWTH AACHEN UNIVERSITY

Warum, was, wie und wozu? Entwurf von Nutzeroberflächen

Warum?

- In der Praxis werden Nutzeroberflächen immer wichtiger
- Überlegungen in der frühen Entwurfsphase

Was?

- Entwurf von Nutzeroberflächen

Wie?

- Entwurfsgrundsätze
- Interaktionsarten
- Darstellung von Information
- Aktivitäten des Entwurfs
- Benutzerfreundlichkeit

Wozu?

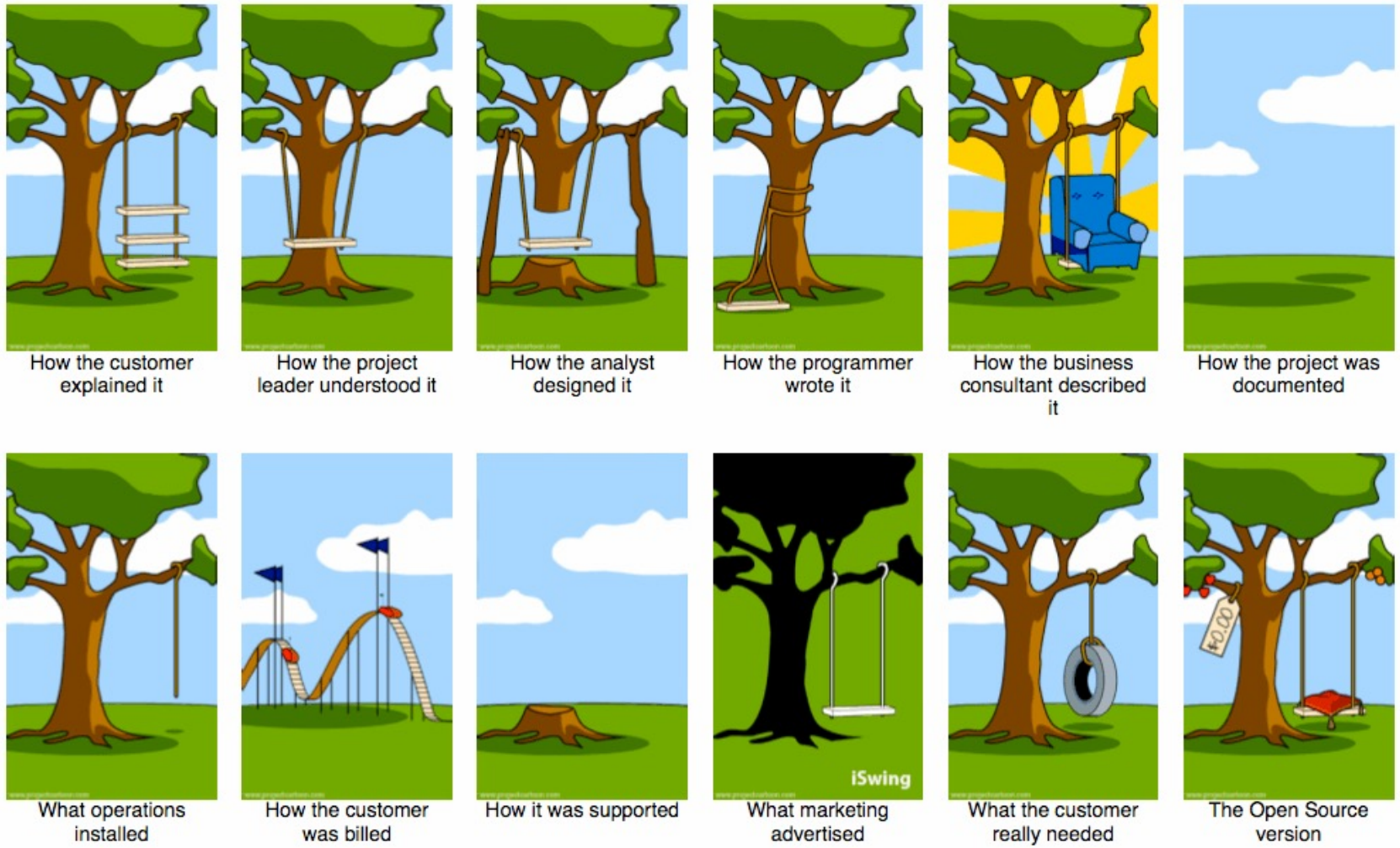
- Selbst konkret anwenden in eigenen Projekten
- Grundwissen über Entwurf von Oberflächen

Entwurf von Oberflächen

- Entwurf von Systemen
  - Architektur | Schnittstellen | Komponenten | Daten | … | Hardware
  - Entwurf der Oberflächen
- Graphische Oberflächen
  - Viele unterschiedliche Technologien und Frameworks
  - Unterschiedliche Hardware
- Faktor Mensch
  - Fähigkeiten, Erfahrungen und Erwartungen der Benutzer
- Spannungsfeld
  - Benutzerfehler vs. Fähigkeiten und Arbeitsumgebungen der Benutzer in der Entwicklung nicht bedacht
  - Unterstützung vs. Behinderung von Tätigkeiten

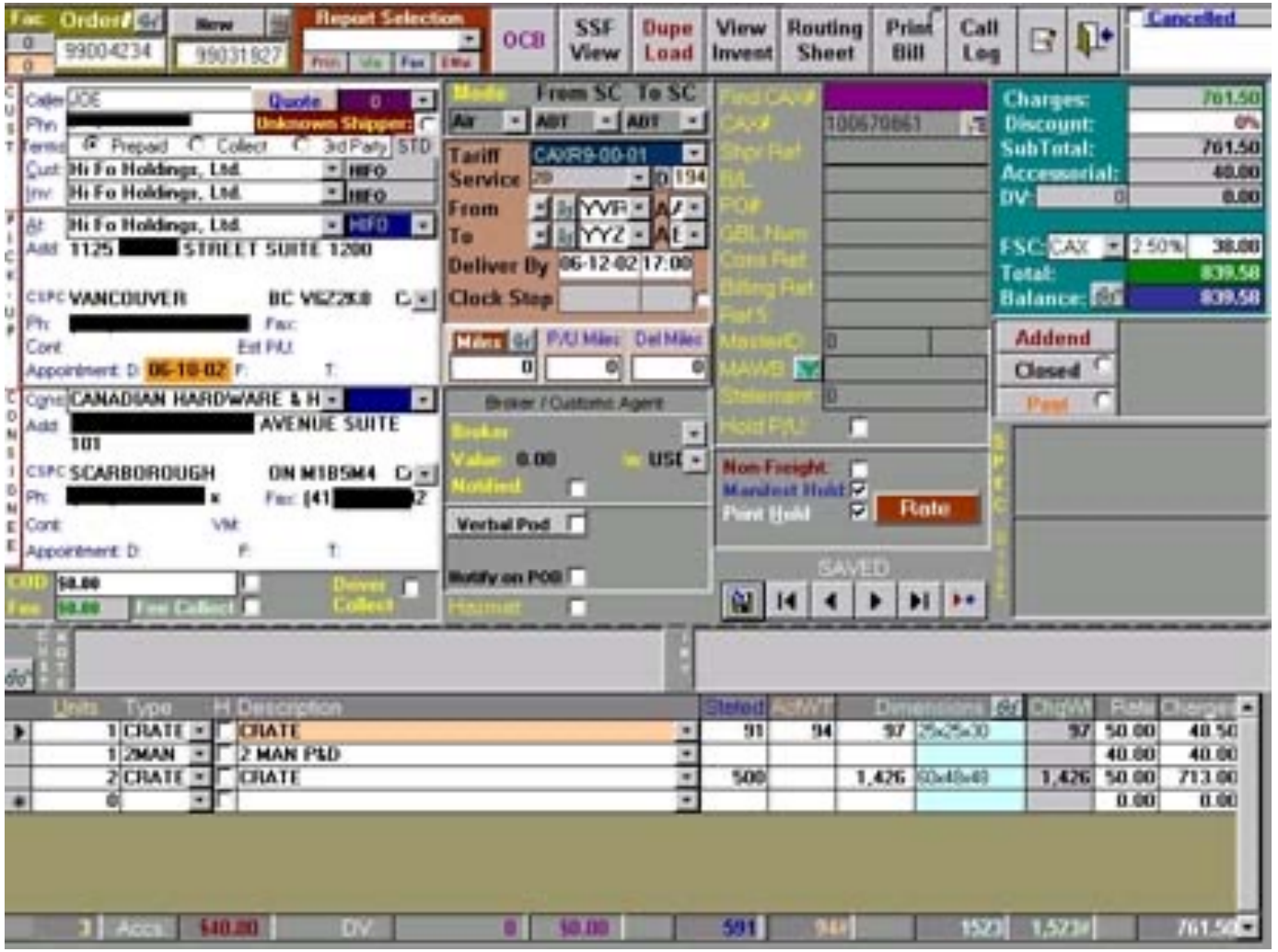
- Menschliche (Un-)Fähigkeiten bedenken
- Kurzzeitgedächtnis
  - ca. 7 einzelne Informationen zu gleich erinnerbar
  - → Nicht zu viel Information auf einmal anzeigen
- Alle machen Fehler
  - Gründe: zu viel Information zur gleichen Zeit, gesundheitliche Einschränkungen, Stress
  - Warnmeldungen bei Systemfehlern erhöhen den Stress
  - Wichtiges identifizieren
- Unterschiedliche körperliche Fähigkeiten
  - Sehkraft | Gehör | Erkennen von Farben | Motorische Fähigkeiten
  - Entwurf nicht an den eigenen Fähigkeiten ausrichten
- Unterschiedliche Vorlieben bei der Interaktion
  - Bildschirm | Text
  - Direkte Bearbeitung | Befehle geben

Herausforderungen



Quelle: https://www.conversationagent.com/2010/01/what-really-affects-behavior.html

Kognitiv gut erfassbar?



How the customer explained it

Entwurfsgrundsätze

- Benutzervertrautheit
  - Bezeichnungen und Begriffe aus der Erfahrungswelt der Nutzer, die am meisten vom System gebrauch machen
  - Domänenspezifische Sprachen helfen
- Konsistenz
  - Vergleichbare Operationen gleich veranlasst
- Minimale Überraschung
  - Keine Überraschung durchs Systemverhalten
- Wiederherstellbarkeit
  - Mechanismen zur Wiederherstellung bereitstellen
- Benutzerführung
  - Bei Fehlern aussagekräftige Rückmeldungen & kontextsensitive Hilfsmittel
- Benutzervielfalt
  - Geeignete Interaktionsmöglichkeiten für verschiedene Arten von Systembenutzern

What the customer really needed

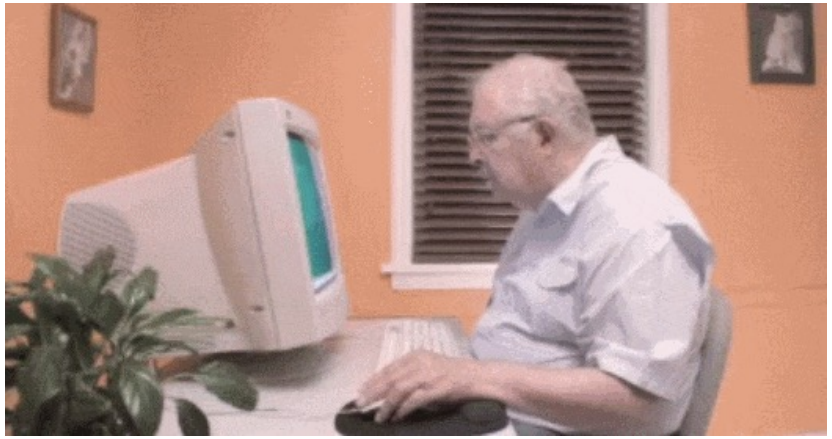
Entwurfsfragen

Wie sollen Benutzer mit dem System interagieren?

Wie sollen Informationen des Systems den Benutzern dargestellt werden?

Interaktionsarten

- Direkte Manipulation
  - Direkte Interaktion mit Objekten auf dem Bildschirm
  - Beteiligte: Zeigegerät (z.B. Maus, Finger bei Touchscreens), zu bearbeitendes Objekt, Aktion die erfolgen soll
- Menüauswahl
  - Befehl aus einer Liste aller Möglichkeiten auswählen
  - Kombination mit direkter Manipulation: Objekt auswählen, auf die sich der Befehl dann auswirkt
- Ausfüllen einer Eingabemaske
  - Felder ausfüllen
  - Kombination mit Menüs, Buttons
- Befehlssprache
  - Spezieller Befehlt mit Parametern
- Natürliche Sprache
  - Meist Frontend für eine Befehlssprache
  - Analyse der natürlichen Sprache und Übersetzung in einen Systembefehl



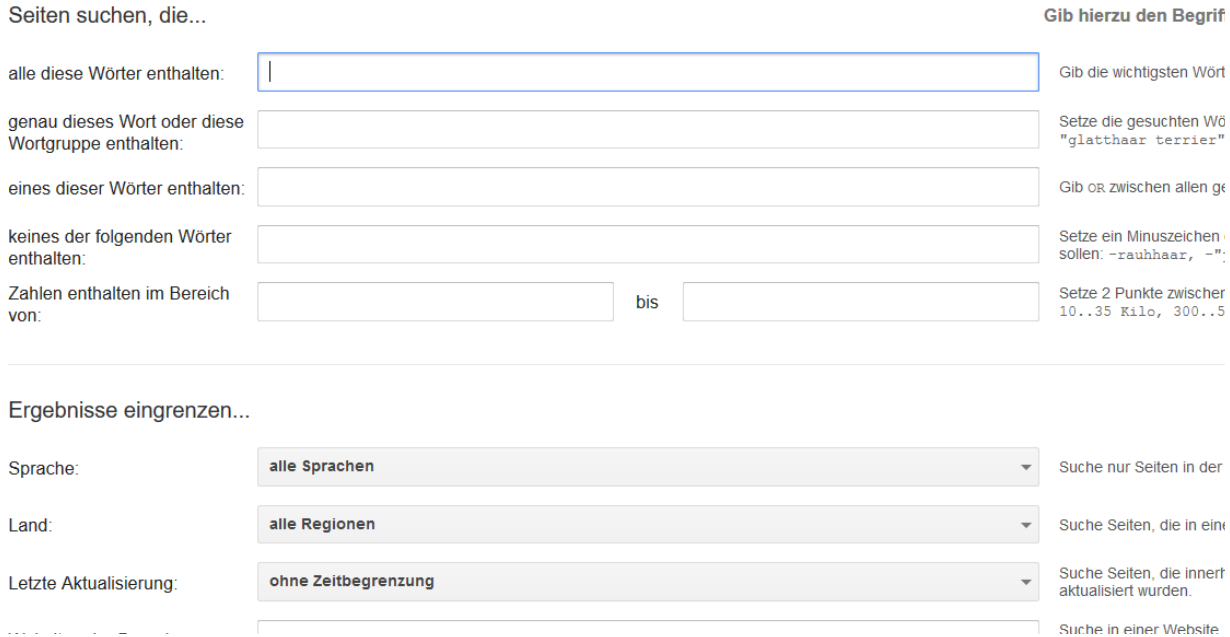
Vor- und Nachteile der Interaktionsarten

Art der Interaktion	Wichtigste Vorteile	Wichtigste Nachteile	Anwendungsbeispiele
Direkte Manipulation	Schnelle und intuitive Interaktion Leicht zu erlernen	Möglicherweise schwierig zu implementieren Nur angemessen, wenn es optische Metaphern, Aufgaben und Objekte gibt	Videospiele CAD-Systeme
Menüauswahl	Verhindert Benutzerfehler Wenig Tippen erforderlich	Für erfahrene Benutzer zu langsam Kann komplex werden, wenn viele Menüoptionen vorhanden sind	Die meisten allgemein eingesetzten Systeme
Ausfüllen einer Eingabemaske	Einfache Dateneingabe Leicht zu erlernen Überprüfbar	Benötigt viel Platz auf dem Bildschirm Führt zu Problemen, wenn die Benutzeroptionen nicht mit den Formularfeldern übereinstimmen	Die meisten allgemein eingesetzten Systeme
Befehlssprache	Leistungsfähig und flexibel	Schwer zu erlernen Schwaches Fehlermanagement	Betriebssysteme, Shells, Steuerungssysteme
Natürliche Sprache	Zugänglich für Gelegenheitsnutzer Leicht erweiterbar	Erfordert mehr Tippen / Spracherkennung Systeme, die natürliche Sprache verstehen, sind unzuverlässig	Systeme zum Abrufen von Informationen, SmartHome

Interaktion in der Praxis

- Gemischte Interaktionsarten
  - Direkter Zugriff & Menübasiert
  - Befehlssprache mit speziellen Eingabemasken

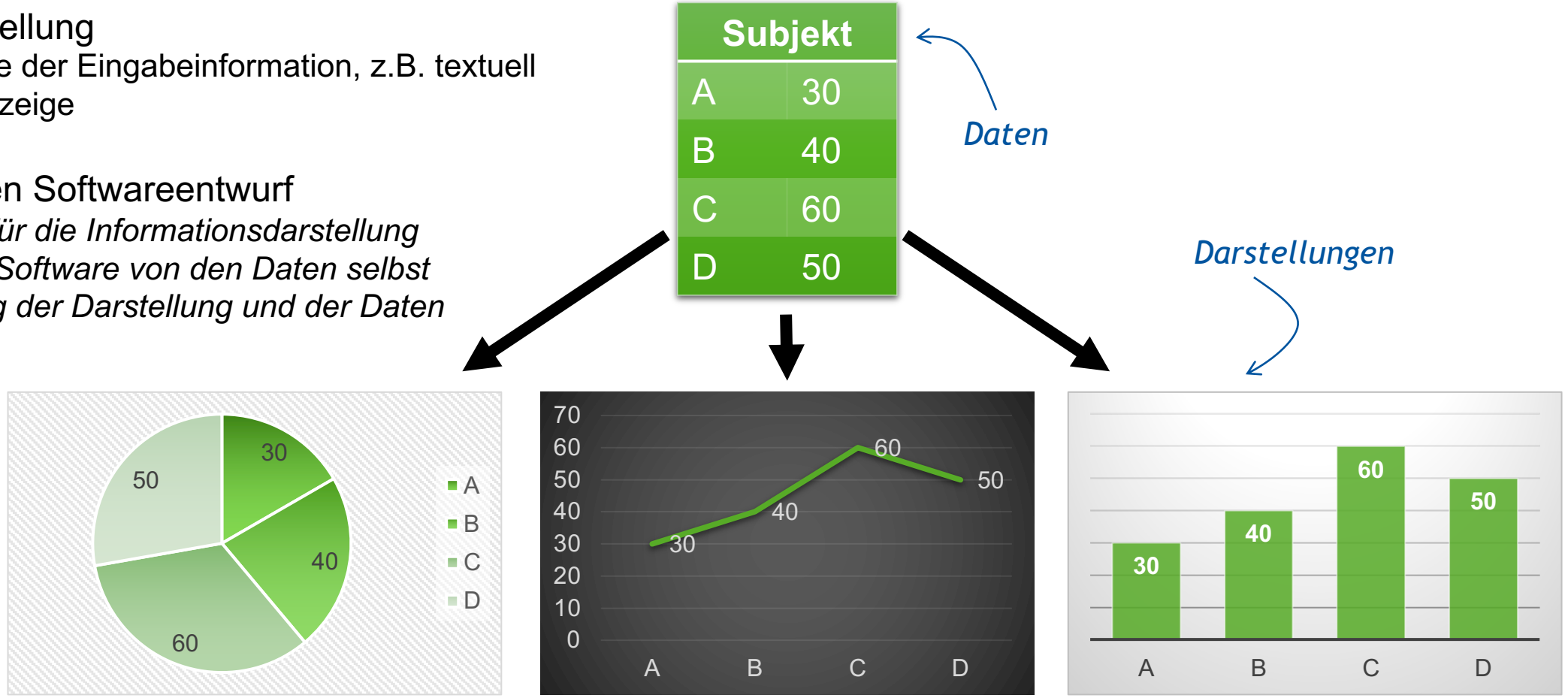
- Individualisierte Oberflächen für unterschiedliche Benutzerklassen
  - Gelegenheits- und erfahrene Benutzer, z.B.
    - Betriebssysteme
      (Befehlssprache vs. graphische Oberflächen)
    - Suche im Web
      (Formular inkl. komplexer Befehlssprache)



Darstellung von Information

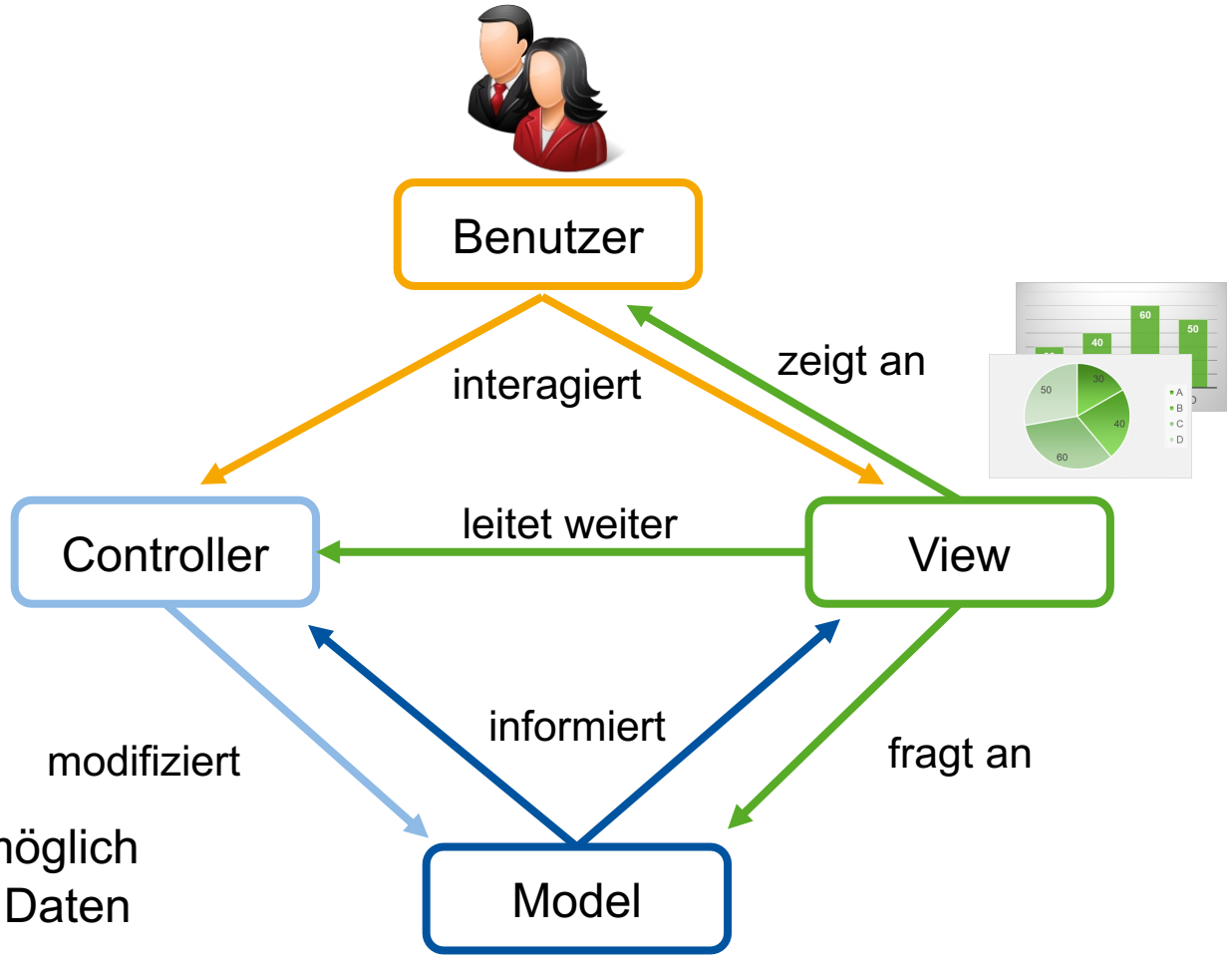
- Arten der Darstellung
  - Direkte Anzeige der Eingabeinformation, z.B. textuell
  - Graphische Anzeige

- Richtlinie für den Softwareentwurf
  - *Trennung der für die Informationsdarstellung erforderlichen Software von den Daten selbst*
  - *→ Entkopplung der Darstellung und der Daten*



MVC-Modell der Benutzerinteraktion

- Architekturmuster: Model-View-Controller (MVC)
  - Model
    - Datenhaltung des Systems
  - View
    - Sichten auf die Daten (mehrere je Modell)
  - Controller
    - Benutzerschnittstelle und Modifikation der Daten (je View ein Controller zugeordnet)

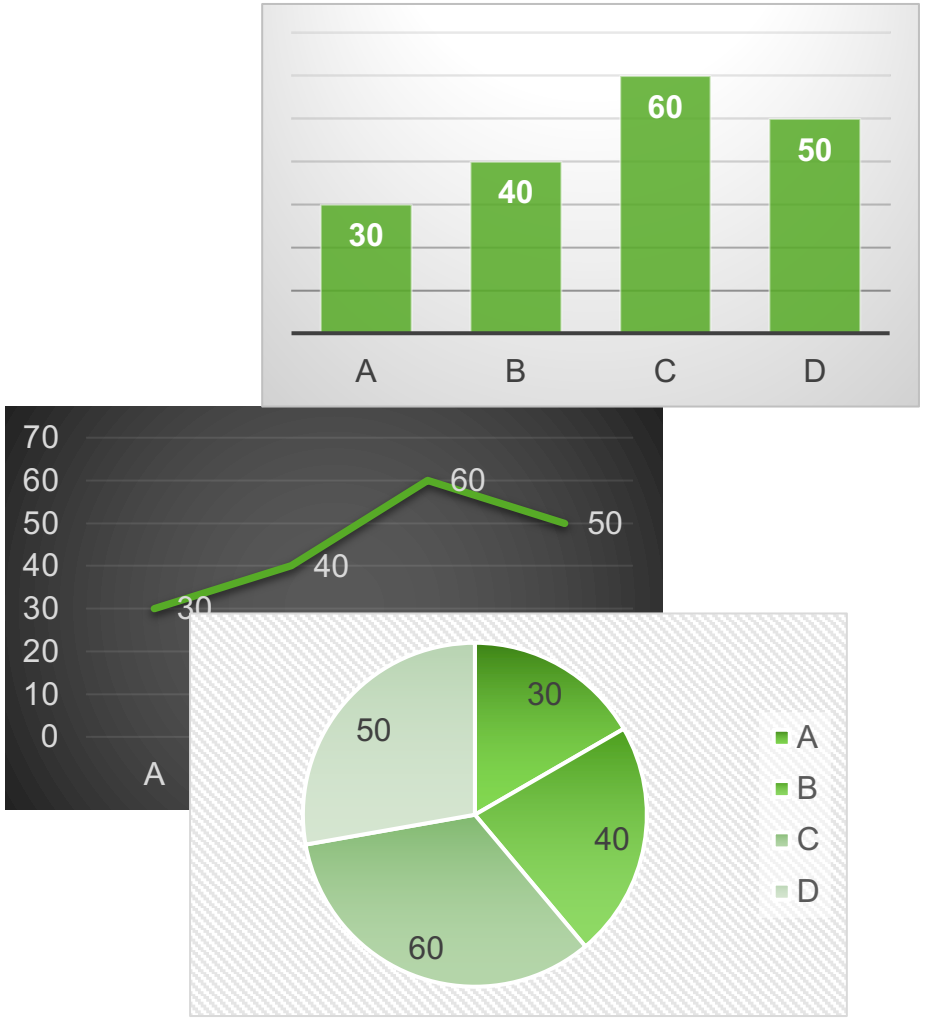


- Getrennte Interaktion mit jeder der Darstellungsarten möglich
- Aktualisierung aller Darstellungen bei Änderungen der Daten

Darstellung von Information

Fragestellungen, z.B.

- Genaue Information oder Beziehung zwischen Datenwerten?
- Häufigkeit der Veränderung von Informationswerten
  - Unmittelbare Anzeige dieser Veränderungen?
- Maßnahmen durch den Benutzer notwendig bei Informationsänderungen?
- Direkte Interaktionsmöglichkeit in der Oberfläche oder reine Anzeige?
- Art der Information?
  - Text
  - Numerische Daten
  - Visuelle Graphik
  - Diagramm (Graph)



Informationen, die sich nicht verändern

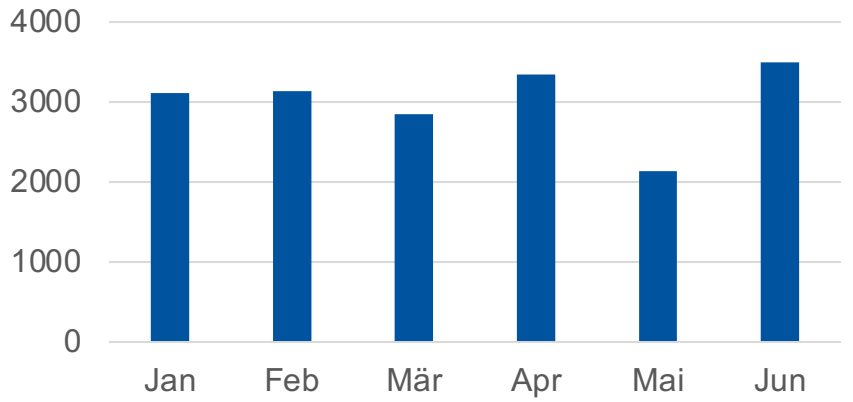
- Text
  - Genaue Angaben erforderlich
  - Relativ langsame Änderung von Werten

  - Pro: geringere Bildschirmfläche
  - Kontra: nicht auf einen Blick erfassbar

- Graphische Darstellung
  - Schnelle Änderungen
  - Nicht genaue Werte sondern Beziehungen zwischen Daten und Tendenzen im Vordergrund

  - Pro: schnellere Erfassung
  - Kontra: größere Bildschirmfläche, Download/Rendering dauert länger

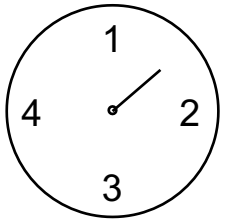
Jan	Feb	Mär	Apr	Mai	Jun
3111	3133	2848	3345	2133	3493



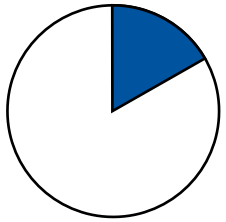
- Unterscheidung von dynamischen Daten durch andere Darstellungsart

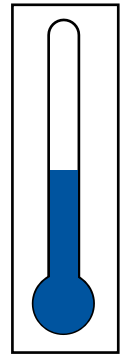
Dynamisch ändernde Informationen

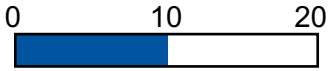
- Numerische Informationen
  - Analoge Darstellungen
  - Bei Bedarf: Ergänzung um eine Digitalanzeige



Skala mit Zeiger

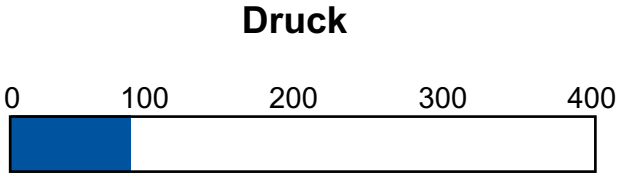

Tortendiagramm

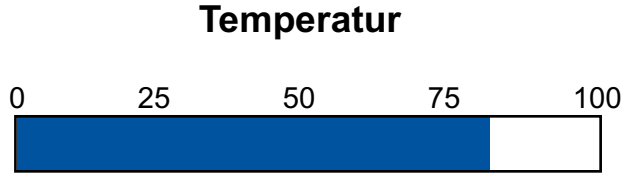

Thermometer



Horizontaler Balken

- Kontinuierliche analoge Anzeigen
  - Relativer Bezug der Werte z.B. zu Maximalwerten



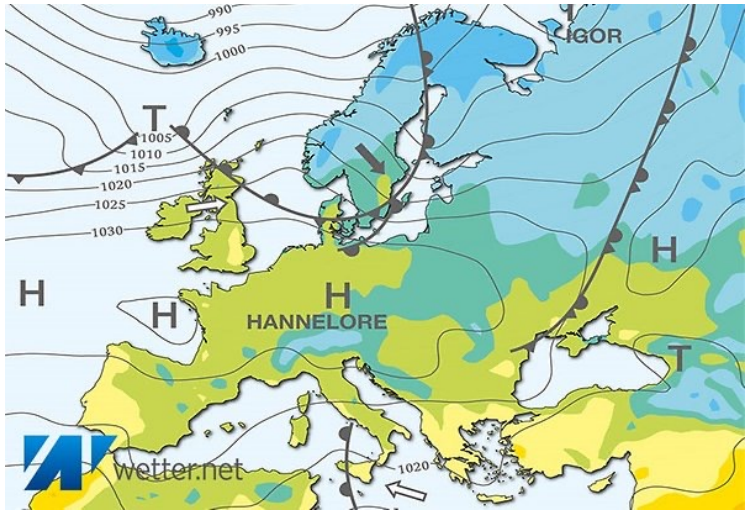


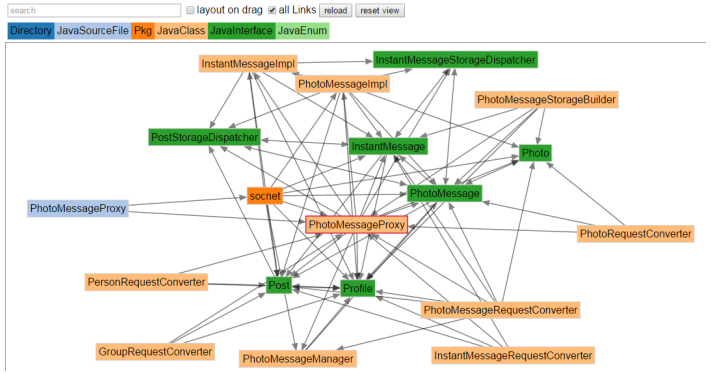
Große Informationsmengen

- Abstrakte Anzeigeformen
- Verknüpfung von in Beziehung stehenden Daten

- Zwei- und Dreidimensionale Darstellung
- Bäume
- Netze

- Karten als Grundlage





Bilder: https://www.berliner-zeitung.de/berlin/wechselhafter-fruehlingsbeginn-auf-hoch-hannelore-folgt-tief-jerry-32250088; https://de.wikipedia.org/wiki/Soziale_Netzwerkanalyse#/media/Datei:Social_Network_Analysis_Visualization.png sowie SH: Artefaktmodell

Farben

- Aufwertung von Oberflächen durch Farbe: Komplexität handhaben (ohne zu übertreiben!!!)

Richtlinien für wirkungsvolle Verwendung von Farben (Auszug)

- Anzahl der Farben beschränken & zurückhaltend verwenden
  - Fenster: 4-5 Farben; Oberfläche: max. 7 Farben
- Farbänderungen zur Anzeige von Änderungen des Systemzustands
  - Wichtige Ereignisse markieren | wichtig für Komplexe Anzeigen mit vielen Elementen
- Farbcodes verwenden um Aufgaben der Benutzer zu unterstützen
  - Außergewöhnliche Objekte markieren
- Farbcodes bedacht und konsistent verwenden
  - z.B. Rot an einer Stelle für Fehlermeldungen verwendet
- Sorgsam mit Farbpaaren umgehen
  - Lesbarkeit, Kontrast, Bedeutung unterschiedlich je Kontext

[Shneiderman 1998]

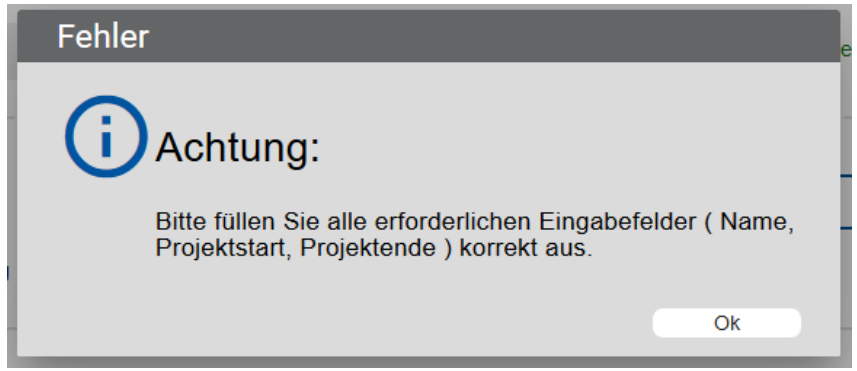
Weiß

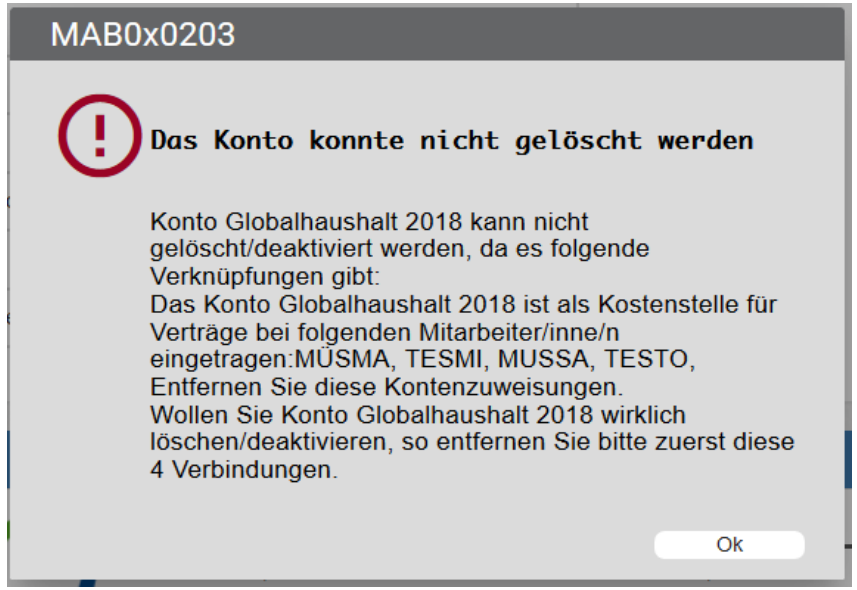
Rot

Schwarz

Entwurf von Systemmeldungen

- Kontext
  - Widerspiegelung des aktuellen Benutzerkontexts
  - Was hat der Benutzer gerade getan, wie hängt die Fehlermeldung damit zusammen
- Erfahrung
  - Experten: Irritiert durch lange „bedeutungsvolle“ Meldungen
  - Anfänger: Problemverständnis schwierig wenn zu kurz und knapp & Technikdetails (z.B. Stack Traces helfen nicht)
- Fähigkeiten
  - Zugeschnitten auf Fähigkeiten und Erfahrungen der Benutzer (Terminologie)
- Stil
  - Positiv statt negativ | aktiv statt passiv | Nie: verletzend, komisch, sarkastisch

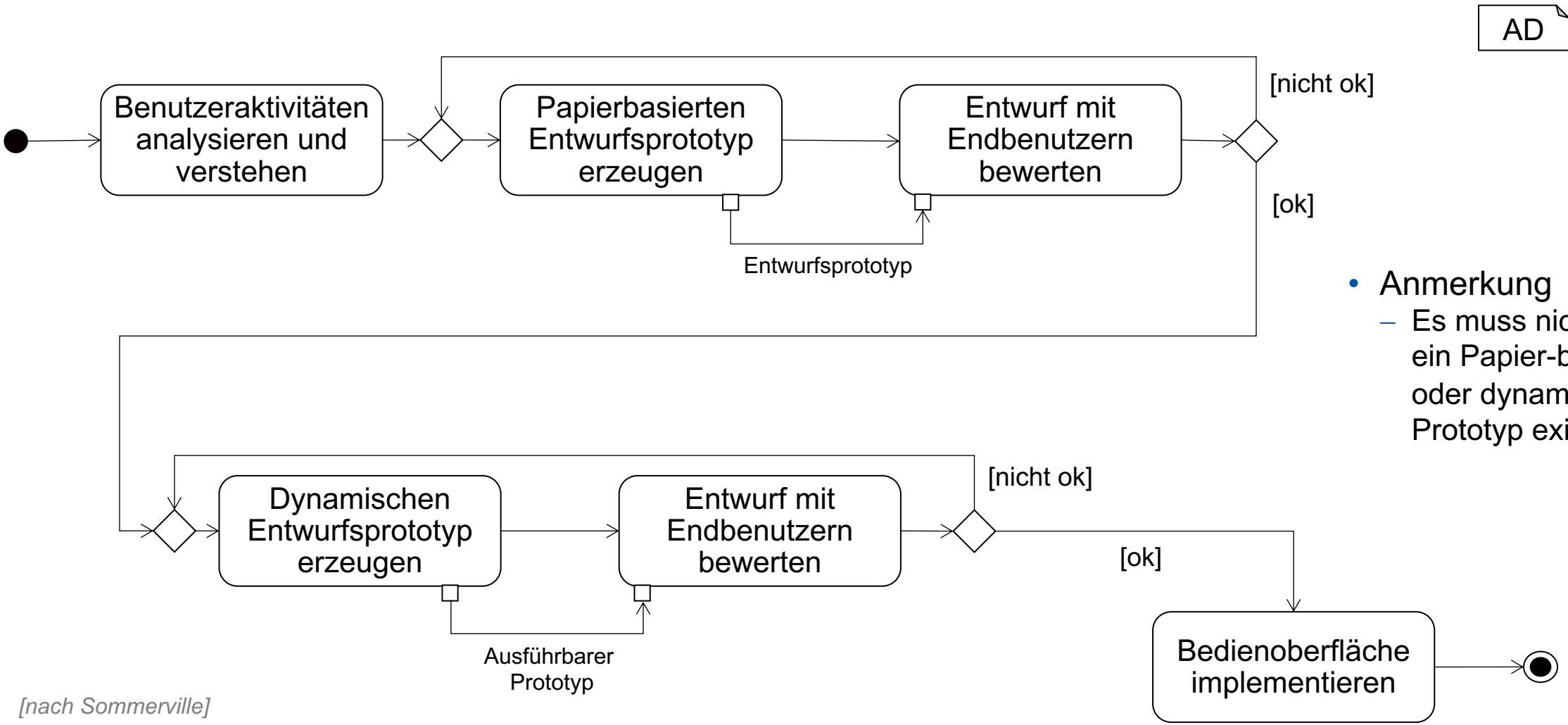




Klassifikation von Prototyping

	Weiterverwendung des Prototyps	Phase im Phasenmodell (vorwiegend)	Zielgruppe
explorativ	wegwerfen	Analyse	Anwender, Systemanalytiker
experimentell	wegwerfen	(Analyse,) Entwurf, Implementierung	Entwickler
evolutionär	ausbauen (inkrementelle Entwicklung)	(sinnvoll nur bei evolutionärer Entwicklung)	Entwickler, Anwender

Mögliche Vorgehensweise zum Entwurf von Oberflächen



- Anmerkung
  - Es muss nicht immer ein Papier-basierter oder dynamischer Prototyp existieren

[nach Sommerville]

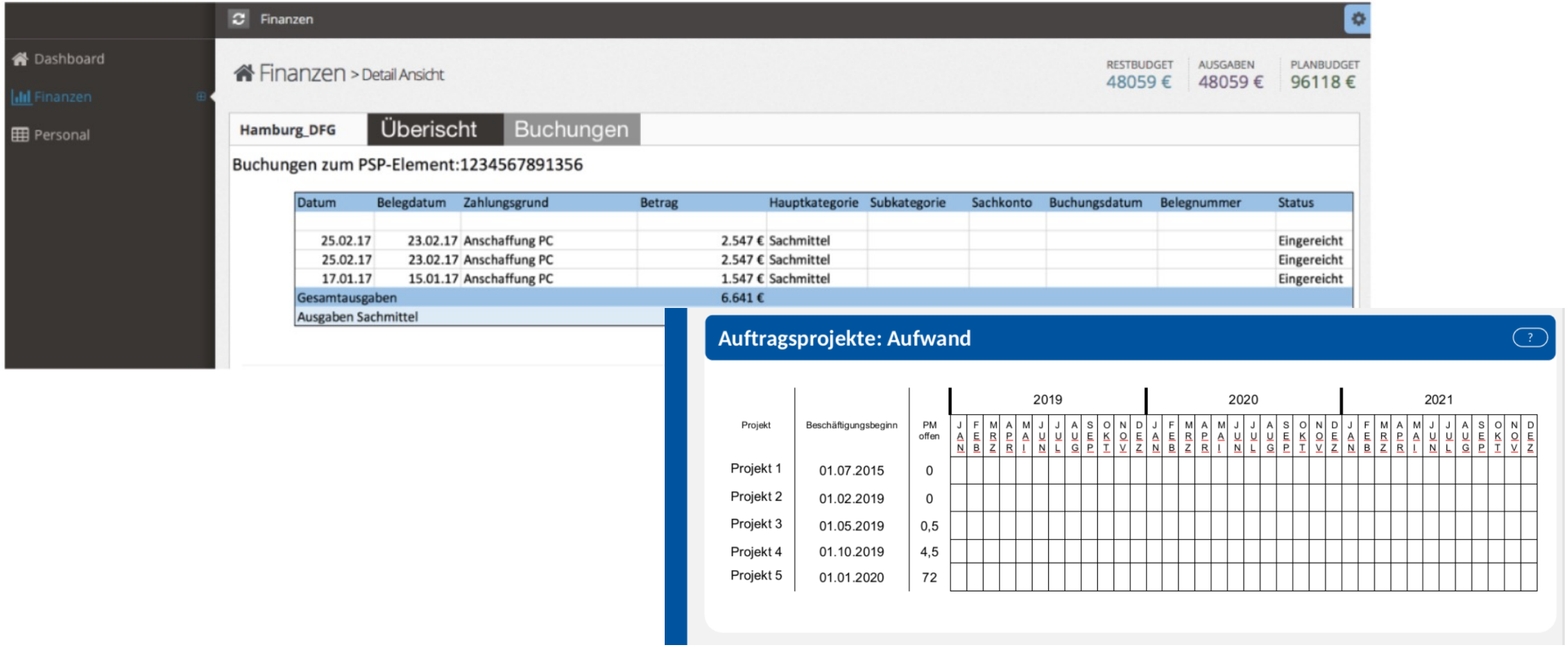
Benutzeranalyse

- Analyse der Benutzeraktivitäten
  - Siehe UML-Modelle zur Anforderungsermittlung
    - Use Case
    - Aktivitätsdiagramme
- Techniken
  - Aufgabenanalyse: einzelne Personen
  - Interviews und Fragebögen: einzelne Personen bzw. Gruppen von Personen
  - Ethnographische Studien: Interaktion zwischen unterschiedlichen Personen
- Gesamtbild was Benutzer tun
  - Durch keine der Methoden erreichbar
- Ergänzende Ansätze
  - Rückschlüsse auf Oberfläche
  - Feedback von Benutzern unbedingt notwendig

Erstellen des Systemprototyps

- Evolutionäre oder explorative Prototyperstellung
  - Einbeziehung der Endbenutzer: Benutzerzentriertes Design
    - Entwurfsphilosophie für interaktive Systeme
- Beispieloberfläche
  - Leichter Eigenschaften zu nennen, die uns gefallen oder nicht
- Einfache Prototypen
  - Papierprotoyp: billig und effektiv | Bildschirme des Systems ODER
  - Storyboard: Folge von Skizzen (=Abfolge von Interaktionen)
- Systemprototypen
  - Throw-Away Protoyp: einmalige Verwendung (z.B. GUI-Mockup, Klickdummy, Prototyping Tools)
    - Wizard-of-Oz: Person im Hintergrund simuliert Antworten des Systems
  - Evolutionärer Prototyp: Weiterentwicklung im Projekt (evolutionäre Entwicklung)

Beispiel: fachlicher Prototyp (Information, aber optimierbare Darstellung & Auswahl)



Oberflächenbewertung 1/2

Selbststudium!

- Beurteilung der Benutzerfreundlichkeit & Prüfung der Benutzeranforderungen
  - Erlernbarkeit
    - Wie lange dauert es, bis ein neuer Benutzer produktiv mit dem System arbeiten kann?
  - Antwortgeschwindigkeit
    - Wie gut passt die Antwortzeit des Systems zur Arbeitsweise des Benutzers?
  - Stabilität
    - Wie tolerant ist das System gegenüber fehlerhaften Eingaben?
  - Wiederherstellbarkeit
    - Wie gut kann sich das System von Eingabefehlern erholen?
  - Anpassungsfähigkeit
    - Wie eng ist das System auf ein einzelnes Arbeitsmodell zugeschnitten?
- Metriken?
  - z.B. nach 4h Einschulung können 80% der Systemfunktionalitäten angewandt werden
    - schwierig zu beurteilen & messen

Oberflächenbewertung 2/2

Selbststudium!

- Systematische Bewertung
  - Labore mit Überwachungseinrichtungen
  - viele Experten (Kognitionswissenschaftler, Grafiker,…)
  - Statistisch signifikante Anzahl an Experimenten (große Anzahl an typischen Benutzern)

  - Langwierig & teuer
    - wirtschaftlich unrealistisch

- Punktuelle Bewertung
  - Fragebögen
  - Beobachtungen
  - Videoaufnahmen beim typischen Systemgebrauch
  - Logs & deren Auswertung (Gebrauchsstatistik)
    - welche Fehler auftreten, welche Hilfsmittel verwendet werden
  - Feedbackmöglichkeit für Benutzer

Gedanken zur Überprüfung der Benutzerfreundlichkeit 1/2

Selbststudium!

- Sichtbarkeit des Systemstatus
  - Genügend Feedback über laufende Funktionen oder den Systemstatus ist stets gegeben
- Verknüpfung zwischen dem System und der realen Welt
  - Es ist hilfreich, entsprechende Konzeptmodelle zu erarbeiten, die auf einer Analogie aus der realen Welt basieren. Dadurch ist die Funktionsweise für den Benutzer viel schneller nachvollziehbar oder sogar vorhersehbar.
- Kontrolle und Freiheit des Benutzers
  - Der Benutzer sollte immer das Gefühl haben, die Kontrolle auszuüben. Dennoch sollte man dem Benutzer nicht grenzenlose Freiheit einräumen, da er ansonsten überfordert sein und Fehler machen könnte.
- Konsistenz und Standards
  - Man sollte immer auf entsprechende interne und externe Konsistenz achten. Darüber hinaus sind, außer in speziellen Ausnahmefällen, immer bestehende Standards zu berücksichtigen.
- Fehler-Vorbeugung
  - Potentielle Fehlerquellen sollten frühzeitig eliminiert werden. Der Benutzer muss zudem ausreichend Anleitung erhalten, um keine Fehler zu verursachen.

[Nielsen, 1994]: https://www.nngroup.com/articles/ten-usability-heuristics/

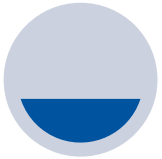
Gedanken zur Überprüfung der Benutzerfreundlichkeit 2/2

Selbststudium!

- Wiedererkennen vor Überlegen
  - Bevor der Benutzer nachdenken muss, wie eine Funktion zu bewerkstelligen oder wo sie zu finden ist, sollte er sie direkt anhand des Interfaces wiedererkennen können.
- Flexibilität und Effizienz der Benutzung
  - Dies bezieht sich auf die nötige Balance zwischen Abkürzungen (shortcuts) für Experten und ausführlicher Hilfestellung für Anfänger.
- Ästhetisches und minimalistisches Design
  - Ein Design sollte stets ästhetisch ansprechend, aber dennoch minimalistisch sein, um unnötige Verwirrung und Übersichtsverlust zu vermeiden.
- Benutzerhilfe und Support bei Fehlern
  - Fehlermeldungen müssen klar und verständlich sein. Sie müssen das Problem und eine mögliche Lösung aufzeigen.
- Hilfe und Dokumentation
  - Auch wenn es besser ist, ein System ohne Dokumentation betreiben zu können, so ist es manchmal doch nötig, eine Hilfe und eine Dokumentation anzubieten. Diese sollten einfach zu durchsuchen sein und sollten Schritte hin zur Lösung eines Problems aufzeigen.

[Nielsen, 1994]: https://www.nngroup.com/articles/ten-usability-heuristics/

Was haben wir gelernt?

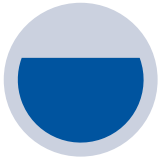


Entwurfs-grundsätze

Passende Darstellungsart für Daten & Interaktion auswählen

Farben: sparsam & einheitlich

Systemmeldungen: Kontextinformation & Rücksicht auf Erfahrungen der Benutzergruppen

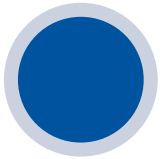


Aktivitäten

Analyse der Szenarien (Analysephase)

Papier-basierter und/oder dynamischer Prototyp

Bewertung mit Nutzer:innen & wenn notwendig Iteration



Unterstützung für NutzerInnen

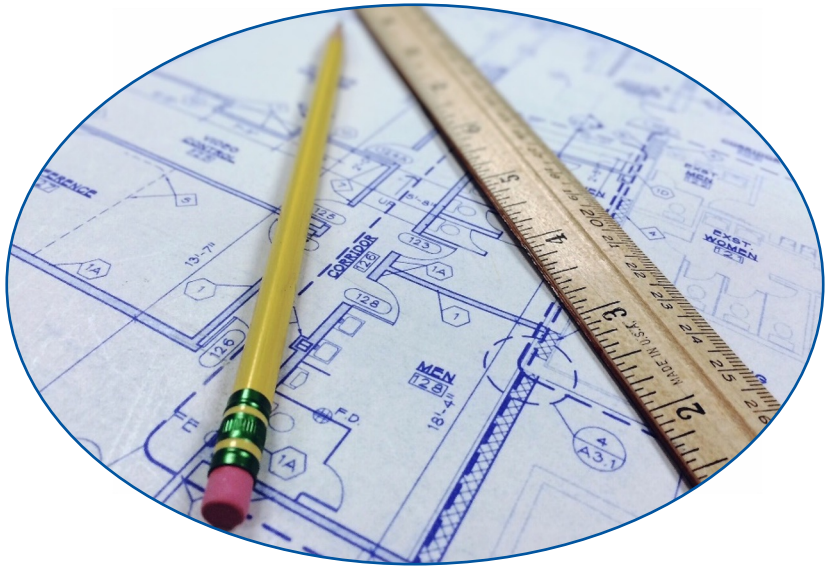
Feedbackmöglichkeit

Handbücher, Tutorials, Erklärvideos

Interaktive Unterstützungs-systeme: Lernen aus Benutzerverhalten

Vorlesung Softwaretechnik

6a. Software Security



Dr. Lars Hermerschmidt
REWE digital
Product Owner Security Engineering

https://www.rewe-digital.com/
@bob5ec

SE Software Engineering | RWTH AACHEN UNIVERSITY

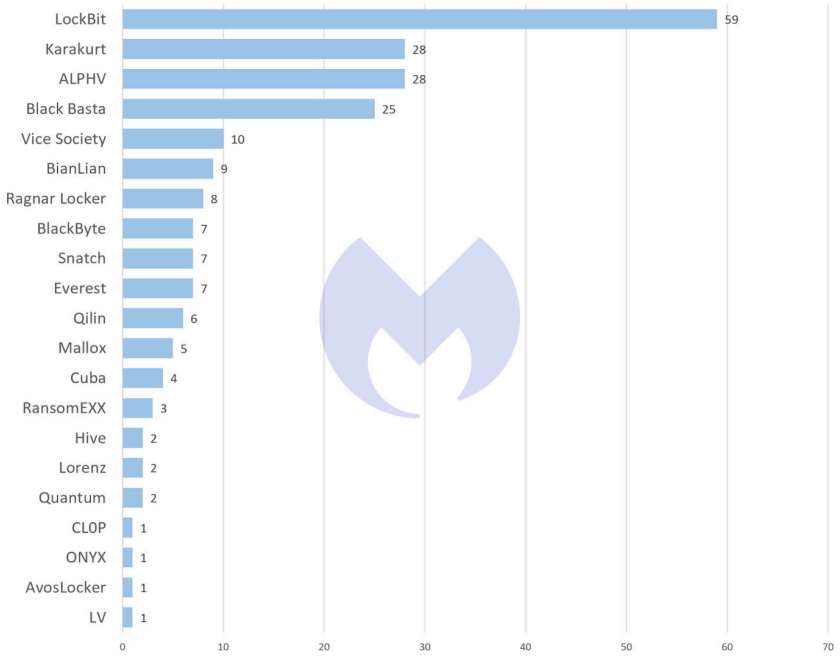
Security is an Important Quality

- We rely on software every few minutes
- Software influences people: Twitter Chief Security Officer (Mudge) states the security controls are inadequate
  - Now Elon Musk screwed up Twitter verified accounts
- Software influences the real world
  - Empty supermarket shelves
  - broken nuclear program in Iran (Stuxnet)

- Security is an abstract value like environmental protection (tragedy of the commons dilemma applies)
  - Everyone wants it
  - Nobody feels personal consequences and personal responsibility
  - Nobody acts

Threat Actors and Risk

Compromised Enterprises by Hacker Group in 2022



https://www.malwarebytes.c
intelligence/2022/11/ransom

https://www.weforum.org/reports/global-risks-report-2022/

- Market size in 2020: $350 million (ransom paid)
- 7 German enterprises got compromised in October 2022
- Recent victims in Germany: Sixt, Dpa, Metro, …

Agenda

- Definitions
- Security Bugs in Code
- Language-theoretic Security (LangSec)
- Security Flaws in Design
- Security Processes

What is Security?

Assume you work at REWE digital. Which topics would you ask me?

Definitions

> “Software Security is the idea of engineering software so that it continues to function correctly under malicious attack.”
>
> *[Mc Graw]*

Information Security Protection goals:

- Confidentiality
- Integrity
- Availability

Security Bugs

Vulnerability

- Common Vulnerability & Exposures (CVE)
  - Example: Log4Shell https://nvd.nist.gov/vuln/detail/CVE-2021-44228
- Common Vulnerability Scoring System (CVSS)
  - CVSS Calculator https://nvd.nist.gov/vuln-metrics/cvss/v3-calculator
- Keep up to date: 55 new CVEs per day
  - OWASP Dependency Check Maven Plugin https://jeremylong.github.io/DependencyCheck/dependency-check-maven/
  - Dependency Track, Software Bill of Material (SBOM)

Weakness

- Common Weakness Enumeration (CWE)
  - Example: Improper Input Validation https://cwe.mitre.org/data/definitions/20.html

Open Web Application Security Project (OWASP)

- OWASP Top 10
  - A01:2021-Broken Access Control
  - A02:2021-Cryptographic Failures
  - A03:2021-Injection ← CWE-20
  - A04:2021-Insecure Design
  - A05:2021-Security Misconfiguration
  - A06:2021-Vulnerable and Outdated Components ← OWASP Dependency Check
  - A07:2021-Identification and Authentication Failures
  - A08:2021-Software and Data Integrity Failures
  - A09:2021-Security Logging and Monitoring Failures*
  - A10:2021-Server-Side Request Forgery (SSRF)*
- (Mobile) Application Security Verification Standard (ASVS / MASVS)
  - Long list of weaknesses and how to assess them
- OWASP Web Security Testing Guide

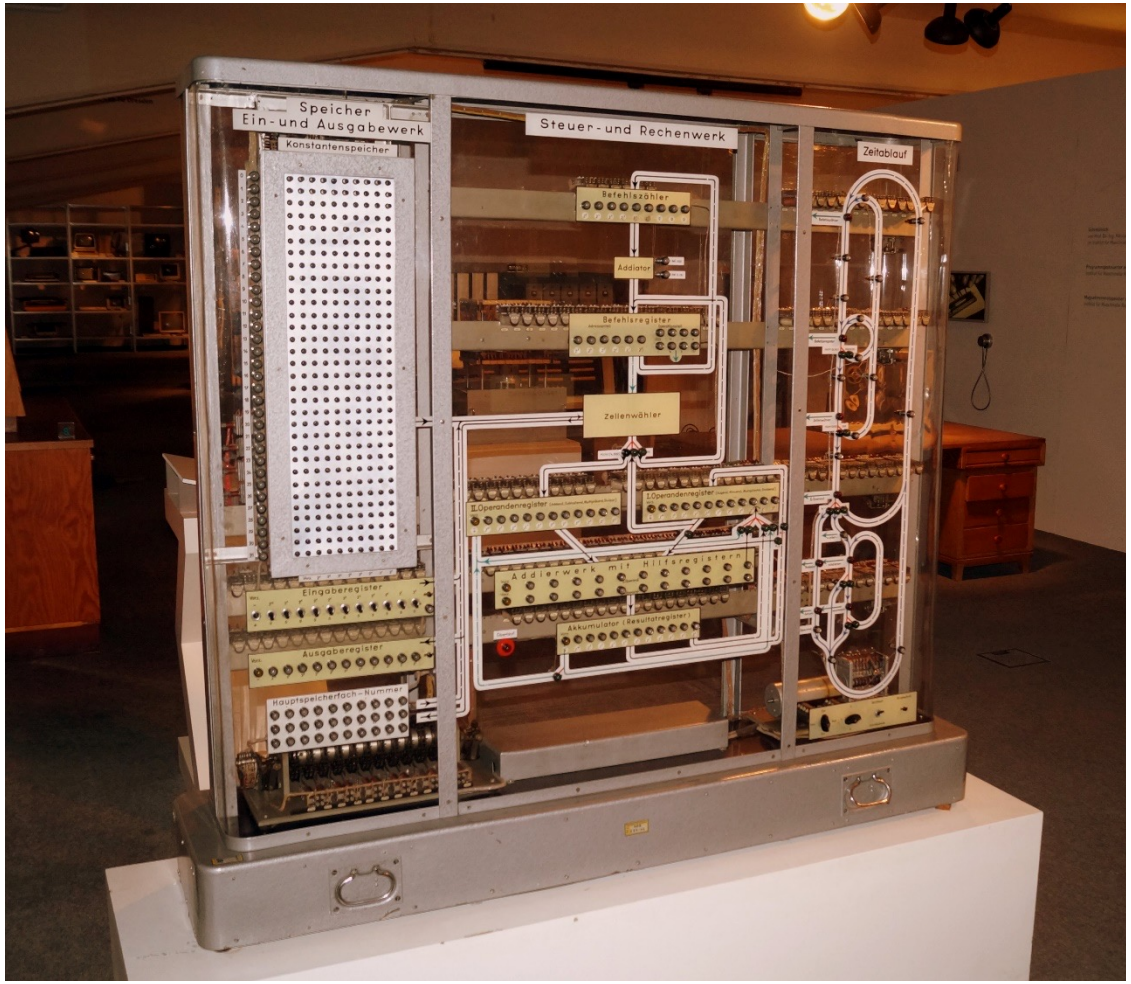
No more FUD (fear, uncertainty, doubt)

LangSec

Software is Input

Von Neumann architecture:
- General purpose machine
- Software is loaded and executed by the machine
- Software is stored like data

Input drives Software
- „Datenverarbeitung“ is an illusion
- Software reacts on input
- Attackers can use all offered processing capabilities



Protection Against Input

Common Bad Advices
- “Don’t trust input data”

Input Validation
- Validate all inputs against a specification
- Accept only those input fulfilling the specification
- Reject all others, NEVER try to guess/heal bad input

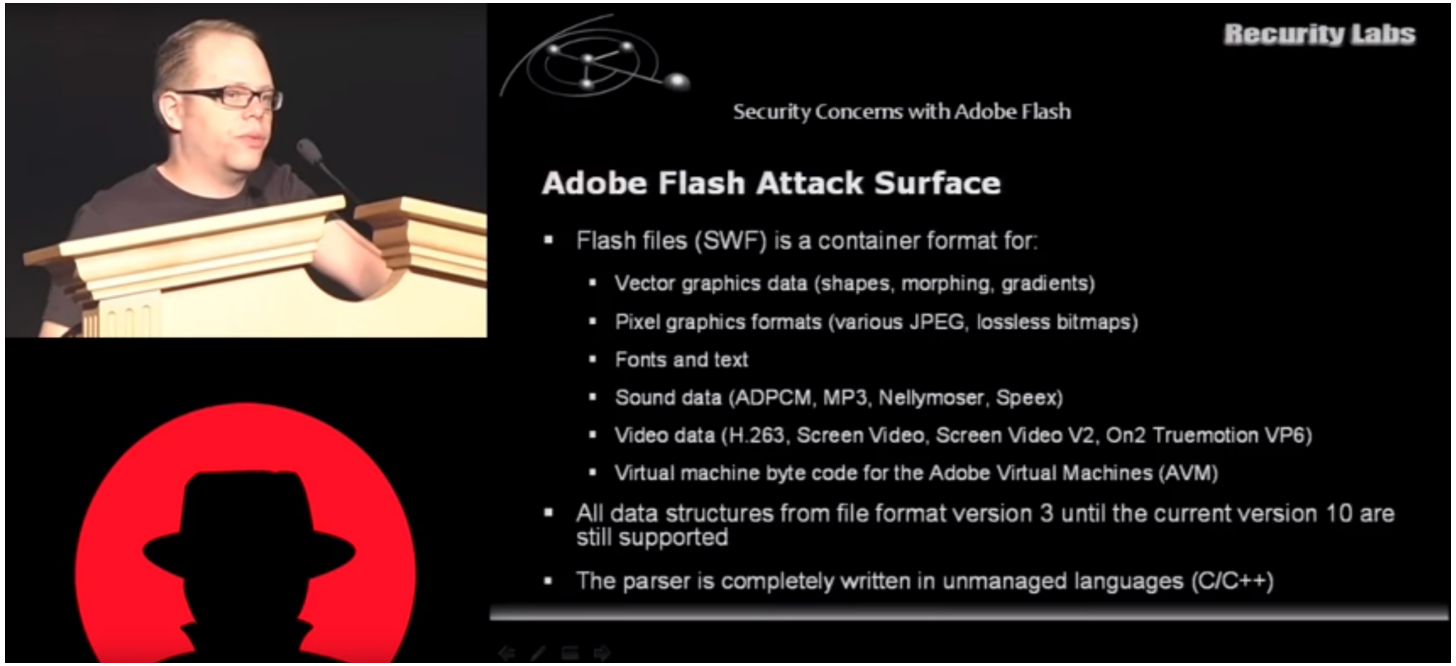
Input specification
- Blacklist: ^[^a-z]+$
  - Reject Known bad
- Whitelist: ^[0-9]+$
  - Only accept known inputs
- Beware: Regular Expressions can only recognize regular languages

FX: Blitzableiter (2010)

- FX exploited countless vulnerabilities in Flash
- Adobe did not manage to patch them

- FX invented Blitzableiter a Filter used in front of the vulnerable Adobe Flash Plugin
- It is a rigor parser for Flash files that
  1. Recognize only valid Flash
  2. Create a new Flash file from the accepted contend
  3. Passes the new Flash file to the Flash Plugin
- The technique is now called Content Disarm and Reconstruction

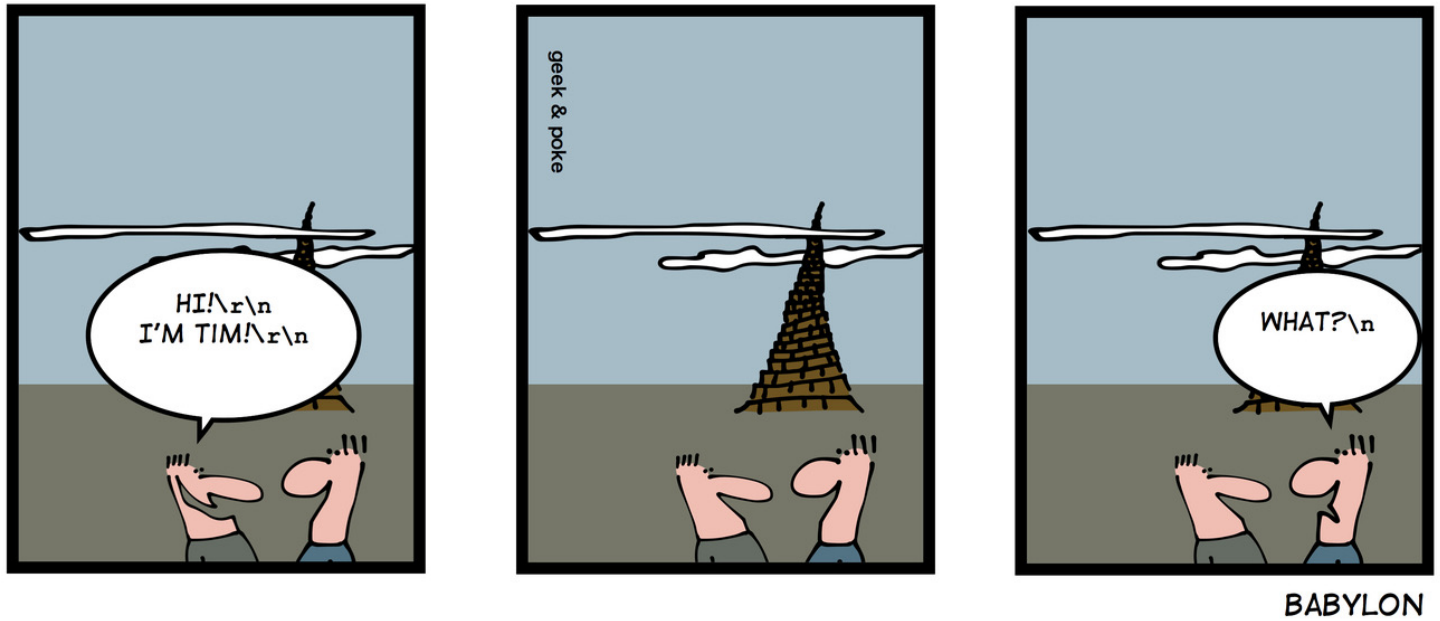
- Blitzableiter defended all exploitation attemts
- FX found LangSec



https://www.youtube.com/watch?v=o34s_J_wylQ

LangSec: Language-theoretic Security
"The View from the Tower of Babel"

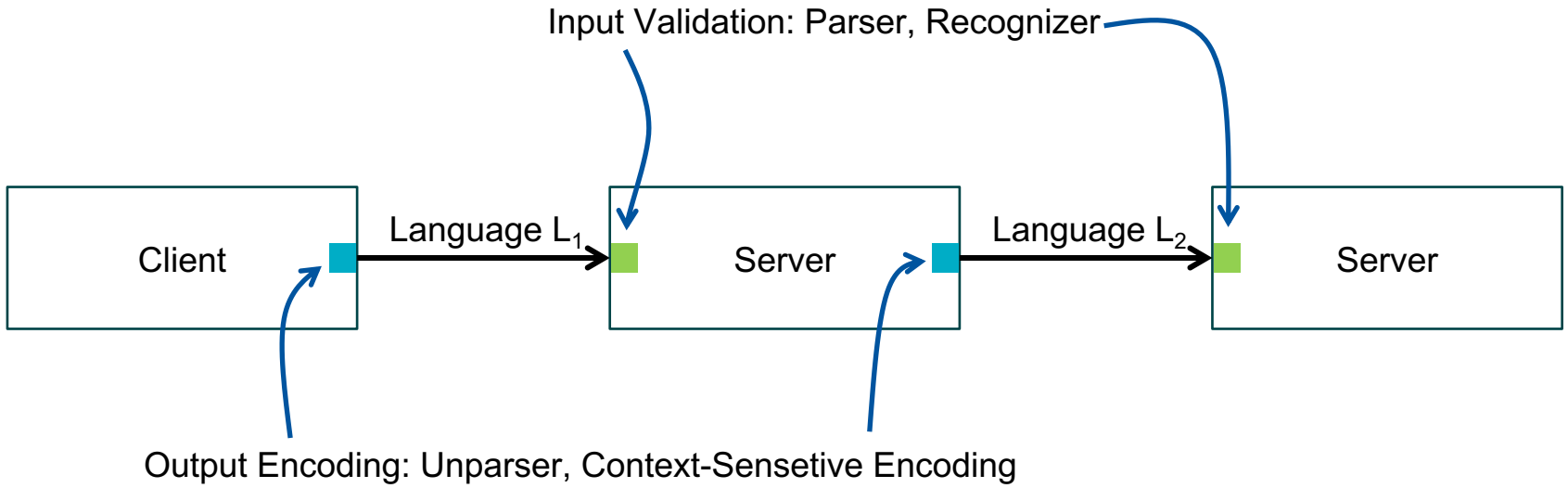
LangSec attributes the insecurity of software to ad hoc programming of input handling



- Treat all input as formal language (remember FOSAP)
- Input handling code is a parser for that language
- That parser implementation must match the grammar used to define the language

- Constructing secure parsers is as complex as secure cryptography, so reuse known good libraries

Definitions



Antipattern Shotgun Parser

Shotgun Parser
- Hand-written parser
- The recognition logic is spread all over the application
- State is allocated before the input is fully recognized

- Apache Tika uses Apache PDFBox Parser
- Hand-written parser
- Does it comply to some PDF grammar?

```
186     /**
187      * This will parse the stream and populate the PDDocument object. This will close the keystore stream when it is
188      * done parsing.
189      *
190      * @param lenient activate leniency if set to true
191      * @throws InvalidPasswordException If the password is incorrect.
192      * @throws IOException If there is an error reading from the stream or corrupt data is found.
193      */
194     public PDDocument parse(boolean lenient) throws IOException
195     {
196         setLenient(lenient);
197         // set to false if all is processed
198         boolean exceptionOccurred = true;
199         try
200         {
201             // PDFBOX-1922 read the version header and rewind
202             if (!parsePDFHeader() && !parseFDFHeader())
203             {
204                 throw new IOException( "Error: Header doesn't contain versioninfo" );
205             }
206
207             if (!initialParseDone)
208             {
209                 initialParse();
210             }
211             exceptionOccurred = false;
212             PDDocument pdDocument = createDocument();
213             pdDocument.setEncryptionDictionary(getEncryption());
214             return pdDocument;
215         }
216         finally
217         {
218             if (exceptionOccurred && document != null)
219             {
220                 IOUtils.closeQuietly(document);
221                 document = null;
222             }
223         }
224     }
```

https://svn.apache.org/viewvc/pdfbox/trunk/pdfbox/src/main/java/org/apache/pdfbox/pdfparser/PDFParser.java?revision=1878544&view=co

Don’t implement parsers in $YourFavoredProgrammingLanguage!

Writing Secure Parsers is hard

- Regular Expression Catastrophic Backtracking

```
^([a-zA-Z0-9])(([\-.]|[_]+)?([a-zA-Z0-9]+))*(@){1}[a-z0-
9]+[.]{1}(([a-z]{2,3})|([a-z]{2,3}[.]{1}[a-z]{2,3}))$
```

- Catastrophic Backtracking leads to Denial of Service

Preventing Shotgun Parsers

Standard language

1. Find existing good (Un)Parser
2. Review Parser (with LangSec knowledge)
3. Work on Parsetree to access the data

Your own language

1. Define language through grammar
2. Generate (Un)Parser or use (Un)Parser Combinator
3. Work on Parsetree to access the data

CSV Example Grammar

```
29  grammar CSV;
30
31  csvFile: hdr row+ ;
32  hdr : row ;
33
34  row : field (',' field)* '\r'? '\n' ;
35
36  field
37      : TEXT
38      | STRING
39      |
40      ;
41
42  TEXT   : ~[,\n\r"]+ ;
43  STRING : '"' ('""'|~'"')* '"' ; // quote-quote is an escaped quote
```

S

P

Keywords (part of T)

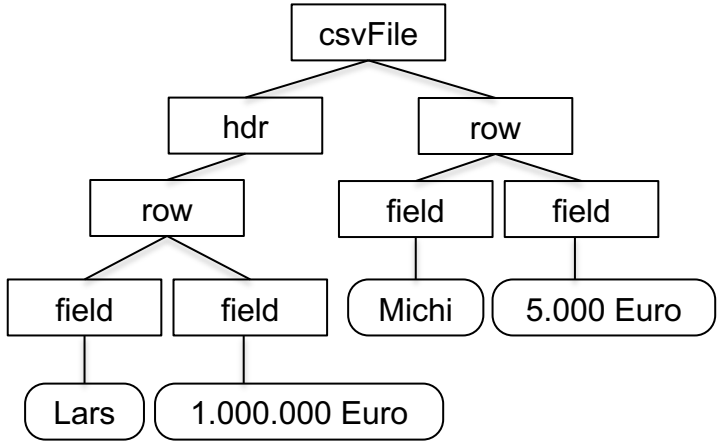
https://github.com/antlr/grammars-v4/blob/master/csv/CSV.g4

Grammar G = (N, T, P, S)
- **N**onterminal Symbols
- **T**erminal Symbols
- **P**roductions
- **S**tart Symbol

CSV Example

Lars,1.000.000 Euro
Michi,5.000 Euro

Parsing

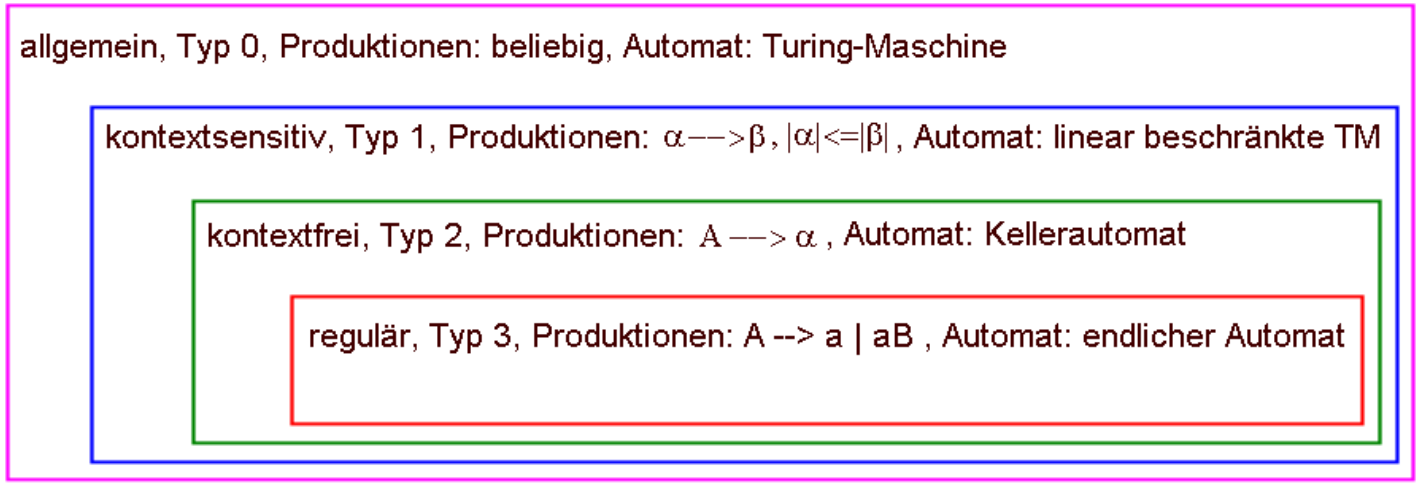


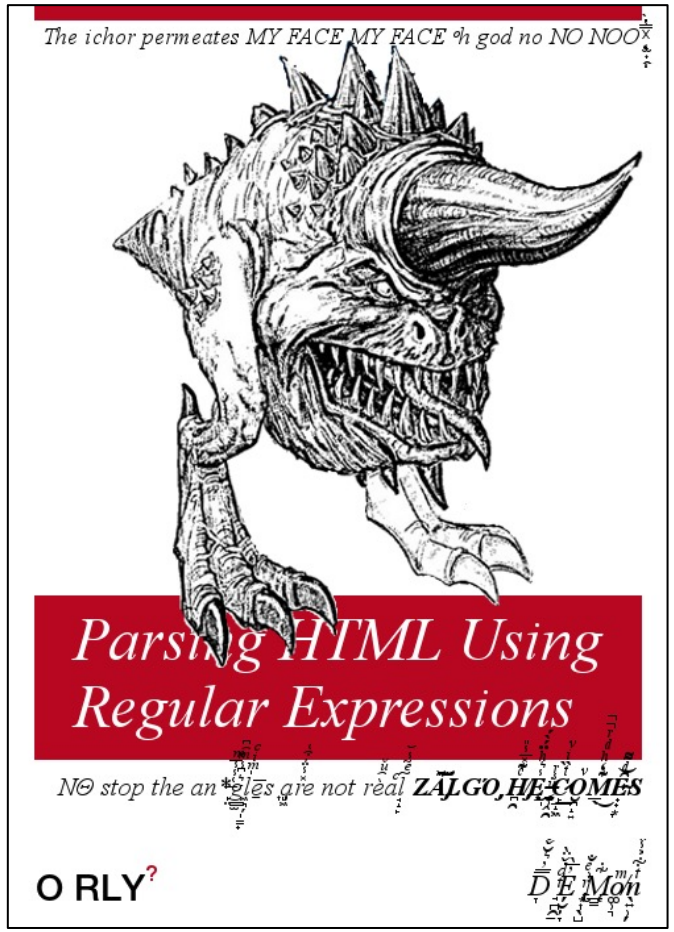
Do not use string splitting and concatenation in applications

Use (Un)Parsers and interact with Parstree

Computational Cliff

Chomsky Hierarchy for Transport Languages





- Context free Grammar: Halting problem decidable and computable
  - „One can construct a parser and proof it‘s correctness.“
  - Proof is challenging (ASN.1 compiler has been verified using Coq)
- Above Context free Grammar = Remote Code Execution
  - Parser has to be a Turing Machine
- Never use regular Expressions to parse more complex languages:

```
<html>.*?<head>.*?<title>.*?</title>.*?</head>.*?<body[^>]*>.*?</body>.*?</html>
```

Do not specify formats that are more complex than context free

Syntax and Semantics

Example: XML External Entity Injection (XXE)
- XML parsers load external entities from filesystem or network (XML language designers thought that would be a good idea)
- A Webservice processing this XML would return the /etc/passwd file to the caller

```
1  <?xml version="1.0" encoding="ISO-8859-1"?>
2  <!DOCTYPE foo [
3    <!ELEMENT foo ANY >
4    <!ENTITY xxe SYSTEM "file:///etc/passwd" >]>
5  <foo>&xxe;</foo>
```

Syntax
- Parsing enforces syntactical correctness on input
- Programs can decide which part of that input to process
  - Filtering of instructions is possible
- Fix XXE Injection by configuring XML parsers to not load XXE

Semantics
- Processing Input results in semantic action
  - Execute parsed instructions on Turing Machine
- (unintentionally) exposing semantic actions may lead to vulnerabilities, e.g., Log4Shell

Rice's theorem, Code Scanners, and Antivirus

Programming Languages are **awesome**

- Turing complete
- Open to solve all problems

Programming Languages are **scary**

- Turing complete
- Unintended functionality possible

Rice's theorem: “non-trivial, semantic properties of programs are undecidable”

It is impossible to predict what code or scripts are doing before executing them.

Implications for file formats
- Antivirus does not work.
- Executing in a sandbox and observing is no better.
- Solution: Remove semantic actions, i.e., scripts from file formats.

Implications for programming languages
- All static code analysis searching for bugs have to solve an undecidable problem and will fail.
- Dynamic analysis, i.e., security scanners is no better.

Solving LangSec in Programming Languages

Remember: Interact with a language, not with bytes or strings

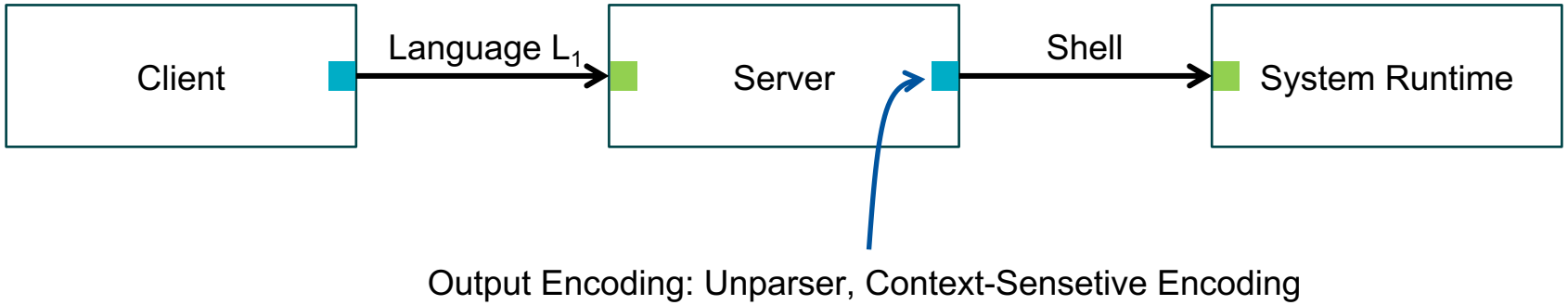
- Make using good (un)parsers for existing languages easy
- Make constructing (un)parsers easy
- Build in (un)parser construction capabilities into programming languages core

- Remove simple Input Output functions like byte wise read and println()
- **Remove string concatenation**

Protection Against Input during Output Creation
Command Injection

```
1   String btype = request.getParameter("backuptype");
2   String cmd = new String("cmd.exe /K \"c:\\util\\rmanDB.bat "
3   +btype+
4   "&&c:\\utl\\cleanup.bat\"")
5
6   System.Runtime.getRuntime().exec(cmd);
```

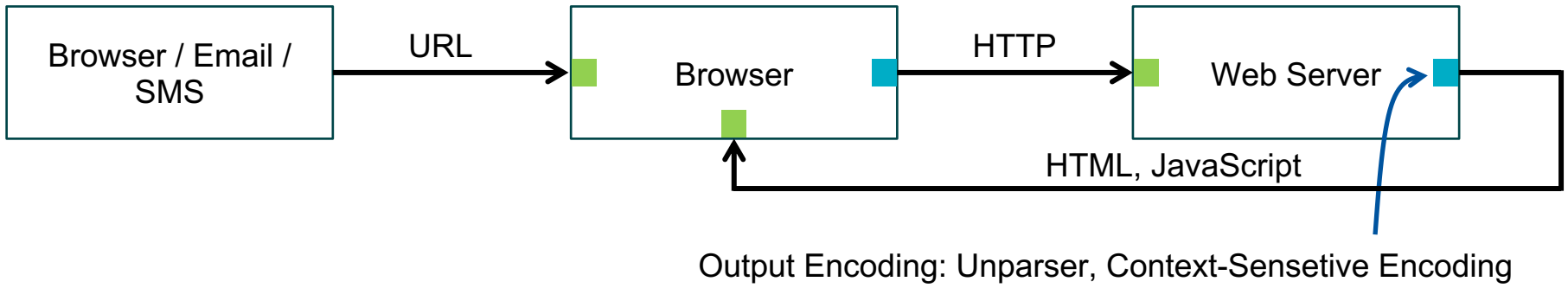
https://cwe.mitre.org/data/definitions/77.html



Cross Site Scription (XSS)

```
1   $username = $_GET['username'];
2   echo '<div class="header"> Welcome, ' . $username . '</div>';
```

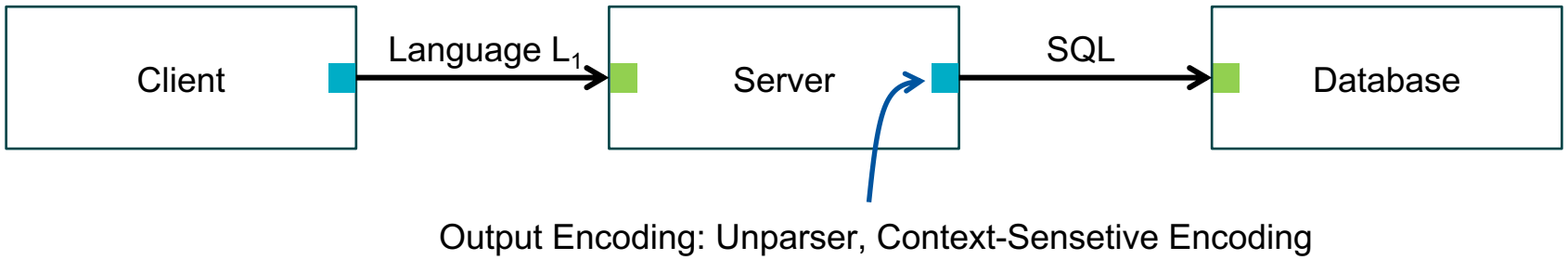
https://cwe.mitre.org/data/definitions/79.html



SQL Injection

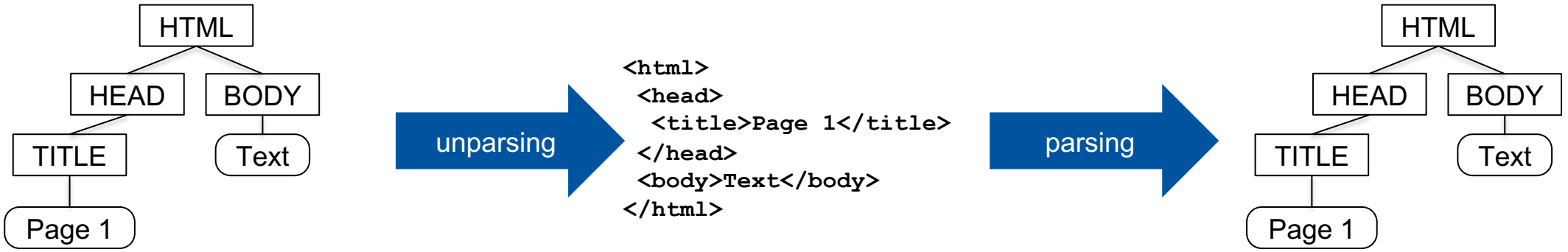
```
1   String firstname = req.getParameter("firstname");
2   String lastname = req.getParameter("lastname");
3
4   String query = "SELECT id, firstname, lastname FROM authors WHERE "
5       + "firstname = "+firstname+" and lastname = "+lastname;
6   PreparedStatement pstmt = connection.prepareStatement( query );
7   try
8   {
9       ResultSet results = pstmt.execute( );
10  }
```

https://owasp.org/www-community/attacks/SQL_Injection




String concatenation leads to
Injections

(Un)Parser Roundtrip



(Un)Parser Roundtrip

CSV Injection

Enter your Name:

```
Lars
```

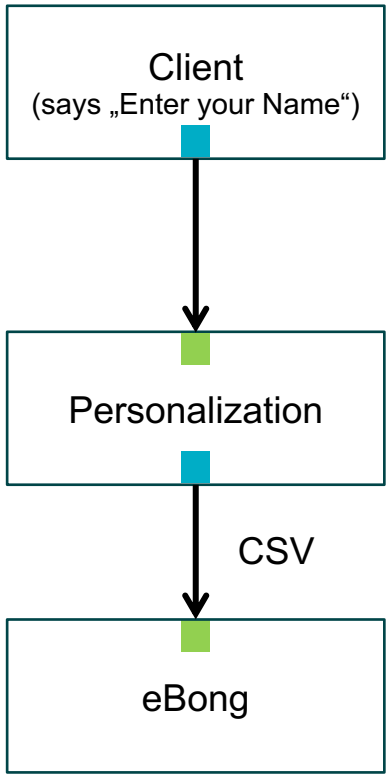
Call service

Personalization:

```
Lars,10.000 Euro
```

eBong:

```
---------------------- eBon ----------------------
Name: Lars
Sum: 10.000 Euro
```



CSV Injection Demo

Demo

CSV Injection Demo

Enter your Name:

```
Lars
```

Call service

Personalization:

```
Lars,10.000 Euro
```

eBong:

```
---------- eBon ----------
Name: Lars
Sum: 10.000 Euro
```

Enter your Name:

```
Lars,42
```

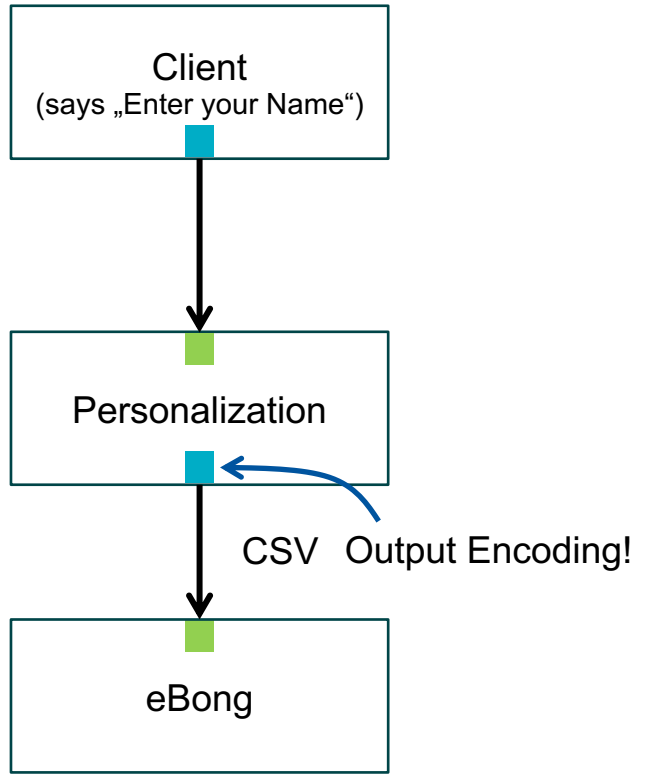
Call service

Personalization:

```
Lars,42,10.000 Euro
```

eBong:

```
---------- eBon ----------
Name: Lars
Sum: 42
```



Bad Implementation

```java
@RestController
public class BadController {

  @GetMapping("/bad/contributions/{name}")
  public String getContribution(@PathVariable("name") String name) {
    return name + ',' + "10.000 Euro";
  }

  @GetMapping("/bad/bill")
  public String printBill(@RequestParam("contribution") String contribution) {
    String[] fields = contribution.split(regex: ",");

    StringBuilder out = new StringBuilder();
    out.append(str: "--------------------- eBon ---------------------\n");
    out.append("Name: " + fields[0] + "\n");
    out.append("Sum: " + fields[1] + "\n");
    return out.toString();
  }

}
```

Bad unparser: String concatenation

Bad parser: Regular Expression for a context free language

Good Implementation

```java
@RestController
public class GoodController {

  @GetMapping("/good/contributions/{name}")
  public String getContribution(@PathVariable("name") String name) throws IOException {
    StringBuilder out = new StringBuilder();
    try (CSVPrinter printer = new CSVPrinter(out, CSVFormat.DEFAULT)) {
      printer.printRecord(name, "10.000 Euro");
    }
    return out.toString();
  }

  @GetMapping("/good/bill")
  public String printBill(@RequestParam("contribution") String contribution) throws IOException {
    CSVParser reader = CSVFormat.DEFAULT.parse(new StringReader(contribution));

    StringBuilder out = new StringBuilder();
    reader.forEach(record -> {
      out.append(str: "---------------------- eBon ----------------------\n");
      out.append("Name: " + record.get(0) + "\n");
      out.append("Sum: " + record.get(1) + "\n");
    });
    return out.toString();
  }
}
```

Use an existing unparser
Work with the Unparser API

Use an existing parser
Get data from the parser API

Injections from a Language Perspective

- “malicious input” is made of keywords from the grammar
- Keywords separate data from code

- Injection vulnerabilities are rooted in language design
- All non-trivial languages with keywords are injectable

- Languages using length field seem to be secure
  - “Total Length” in IPv4 Header
  - IEEE 802.3 MAC

```
29  grammar CSV;
30
31  csvFile: hdr row+ ;
32  hdr : row ;
33
34  row : field (',' field)* '\r'? '\n' ;
35
36  field
37      : TEXT
38      | STRING
39      |
40      ;
41
42  TEXT   : ~[,\n\r"]+ ;
43  STRING : '"' ('""'|~'"')* '"' ; // quote-quote is an escaped quote
```

Keywords (part of T)

https://github.com/antlr/grammars-v4/blob/master/csv/CSV.g4

Recipe to Prevent Injections

- A Correct Unparser prevents Injections
  - Don’t write encoding code yourself
  - Context-sensitive Encoding, e.g. in HTML, is too complex to get it right

- When using an existing language (HTML, SQL, CSV, …): Use Open Source (Un)Parsers library
  1. Find existing good (Un)Parser.
     - Apply Fuzzing, i.e. handling of malformed input
     - Test encoding to make sure the (Un)Parser handles it
     - Review (Un)Parser with LangSec knowledge
  2. Work on Parsetree to access the data

- When creating your own communication language:
  1. Ask yourself: Why is XML, JSON, YAML, or Cap’n Proto not working for your case?
  2. Define language through grammar
  3. Generate (Un)Parser
  4. Work on Parsetree to access the data

Summary

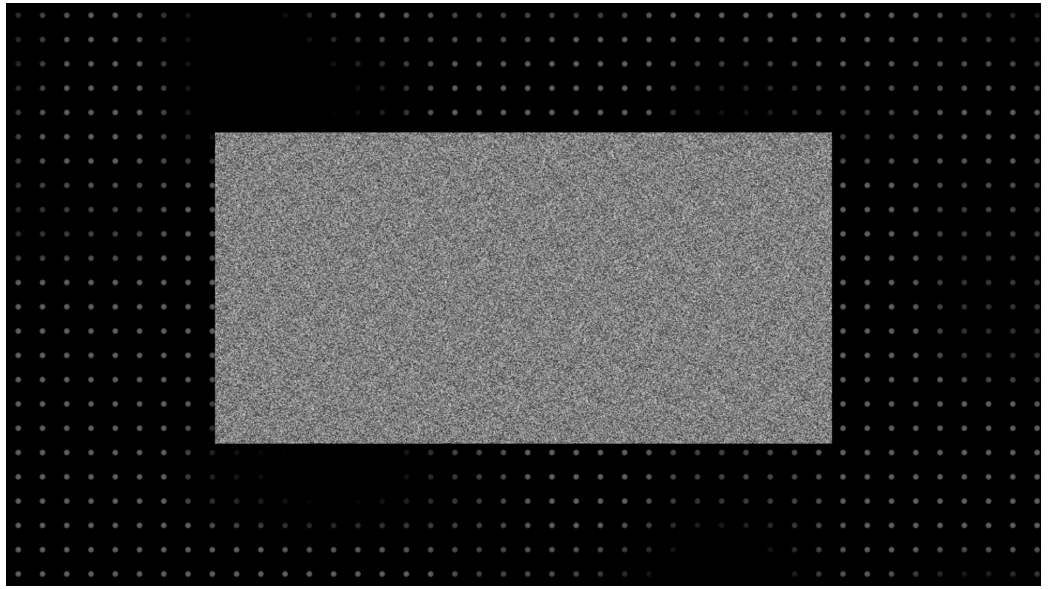
- Computers (including AI) can not determine what programs do without executing them
- Communication language above deterministic context free introduce Remote Code Execution by design

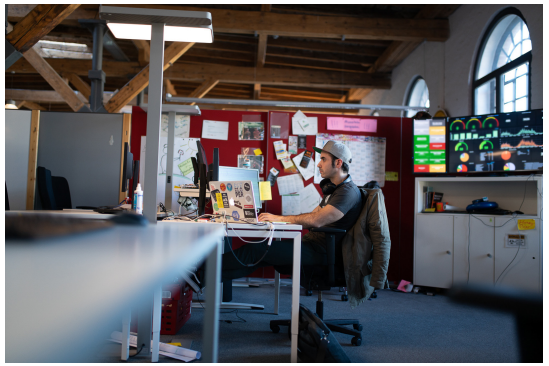
- “Rolling your own (un)parser is like rolling your own crypto algo”
  - Always work on (un)parse trees, not on strings
  - Use existing good (un)parsers
    or
  - Define your own languages with a grammar and generate them

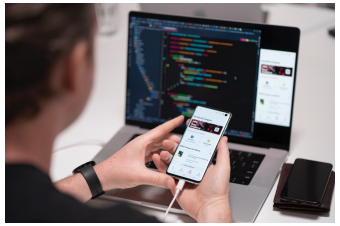
- Correct Unparser prevents Injections

REWE DIGITAL

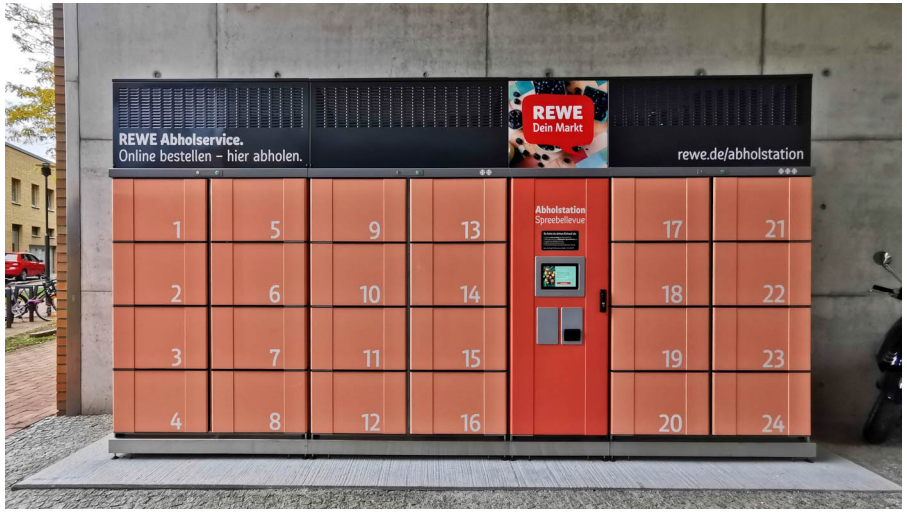
REWE digital ist das Home of IT eines der größten Handels- und Touristikkonzerne Europas







Wir entwickeln die digitale Zukunft des Handels – und probieren uns immer gerne aus …



REWE digital gibt es seit 2014 – und die *neue* REWE digital seit Oktober 2022

- Gegründet: 2014 in Köln
- Zusammenschluss mit REWE Systems: Oktober 2022

- Standorte:
  - Deutschland: Köln, Kiel, Frankfurt, Berlin, Ilmenau und viele weitere kleinere Regionalstandorte
  - Bulgarien: Sofia und Varna
  - Spanien: Màlaga
  - Österreich: Graz

- Verantwortlich für:
  - Onlineshop rewe.de
  - REWE Apps für Android und iOS
  - Teilautomatisierte Food Fulfillment Center in ganz Deutschland
  - Technologische Marktinfrastruktur der REWE Märkte, digitale Innovationen wie Pick&Go, cloudnative Lösungen und richtig, richtig viele weitere Dinge (und Millionen Zeilen Code)

- Und sonst so? Wir haben auch noch einige spannende Tochterfirmen und Start-Ups …

REWE digital ist ein Tech-Unternehmen – und Innovationshub, Inkubator oder Investor

Wir lieben agile Softwareentwicklung und Technologien aller Art. Daher können wir gut mit Coding, Lebensmitteln oder UX. Aber wir …

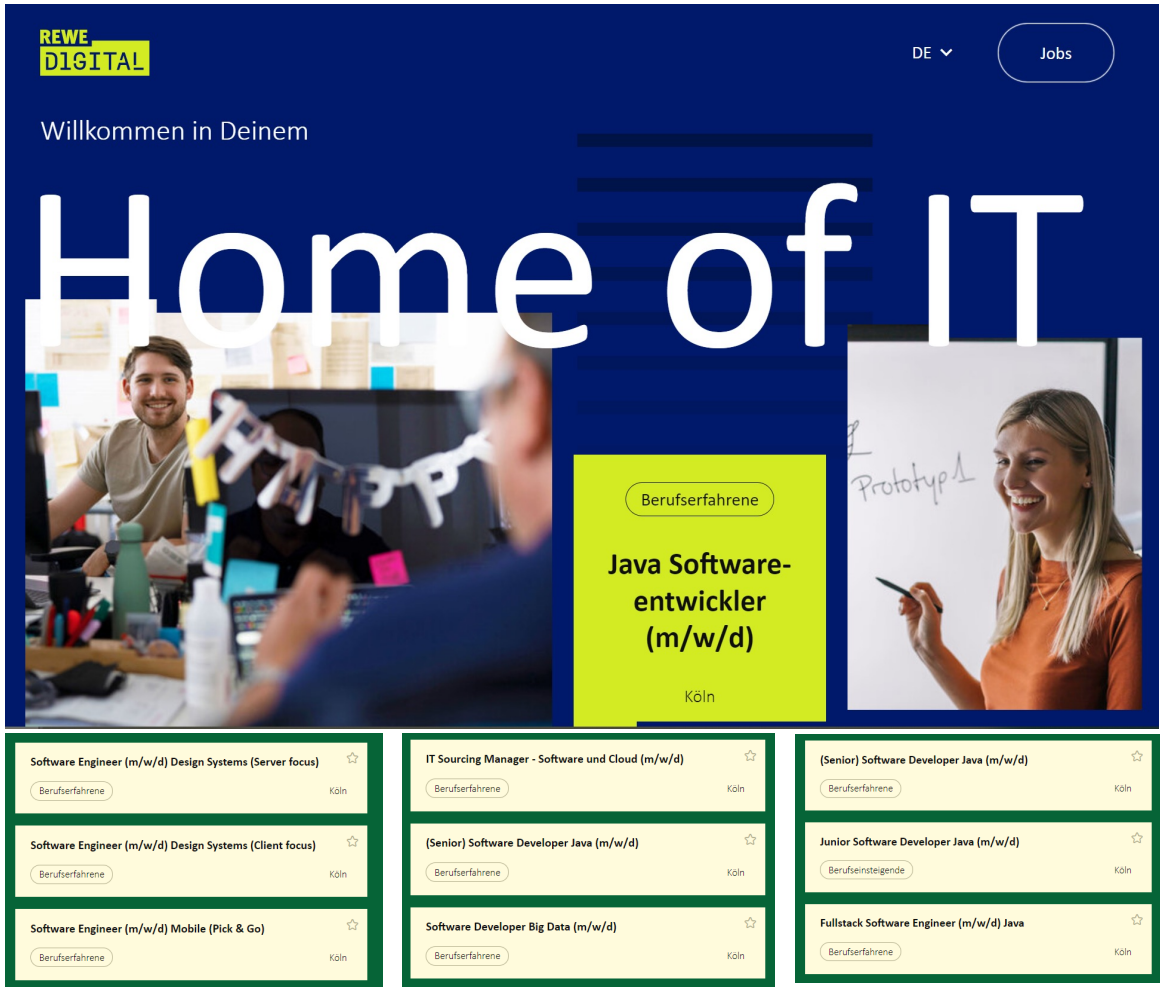
- … können auch Fulfillment für andere mit fulfillmenttools by REWE digital
- … entwickeln Payment für das Universum mit paymenttools
- … lieben Code und Tiere mit ZooRoyal
- … machen next-gen Commerce für alle commercetools Next generation commerce
- … oder haben ein lachendes Auge und einen Wein. WEINFREUNDE

Neugierig geworden? Dann bleibt mit uns in Kontakt!

Alle offene **Jobs** findest Du unter **rewe-digital.com**.
Wir suchen z.B. Werkstudent:innen für DevOps, ERP Systeme, Software Engineers, IT Business Analyst, Operations, Office IT und viele weitere.

Du willst nichts mehr verpassen? Dann folge uns auf **LinkedIn**, **Twitter** oder **Instagram**.

Schau auch auf **GitHub** oder **YouTube** vorbei und höre in unseren **Tech-Podcast** „Codes und schmerzlos“ rein.
Bis bald!



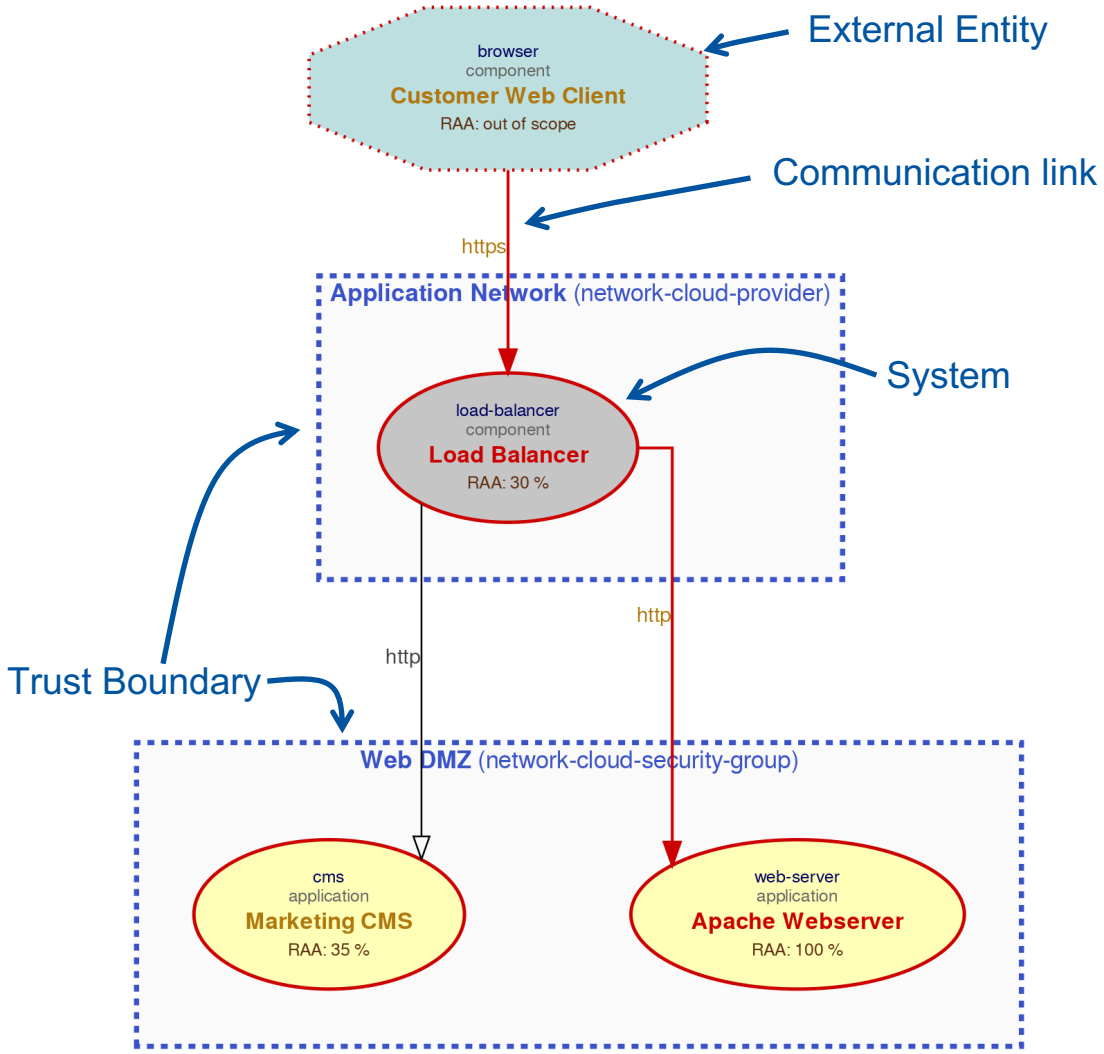
REWE DIGITAL

Security Flaws in Design

- Security Flaws exist in the architecture
- You will not find them with code scanners

Detection Technique: Threat Modeling

- Write down the architecture as Data Flow Diagram (DFD)
- Apply STRIDE Attacks to communication links:
  - Spoofing
  - Tampering
  - Repudiation
  - Information disclosure
  - Denial of service
  - Elevation of privileges



Apply STRIDE in Practice

- Run a card game when a component is changed
  - To score with a card you must explain where the attack at hand is applicable
  - Disagreement in how the system behaves reveals spots where flaws are likely

- Threagile
  - Write DFD in YAML
  - Commit to Git
  - Run automated analysis (based on STRIDE) in CI
  - Review identified flaws (risks.xlsx)
  - Use Risk Tracking in YAML file to document treatment of identified architectural flaws

ISO 27001 Information Security Management System (ISMS)

- ISO 27001 standard for building and improving an Information Security Management System (ISMS)
- ISO 27002 describes 114 controls in 18 anexes
- Policy Documents define how to implement security regarding different topics
- Policies are published such that every employee must follow them
- External Auditor asserts every two years if the policies are followed

Technical

- A.9: Access control
- A.10: Cryptography
- A.12: Operations security
- A.13: Communications security
- A.14: System acquisition, **development** and maintenance
- A.16: Information security incident management

Organizational

- A.5: Information security policies
- A.6: Organization of information security
- A.7: Human resource security
- A.8: Asset management
- A.11: Physical and environmental security
- A.15: Supplier relationships
- A.17: Information security aspects of business continuity management
- A.18: Compliance

Taylorism

Scientific Management
- Separate complex planning from easy execution work
- Lower costs by outsoucing execution work
- Idea: Create the perfect (project) plan upfront

Convay's Law
Any organization that designs a system will produce a design whose structure is a copy of the organization's communication structure.

In tayloristic Organizations Management decides upon the System Architecture

In tayloristic Organizations Expert Teams are organized by…
- Layer: Frontend, Database, Network
- Function: Business Process Step 1, …, n, Architecture, Test, Security

Frederick Taylor

Taylorism is dead since the 1970's

Jez Humble
@jezhumble

- Software Development is a creative process
- Software cannot be perfectly planed
- Customer feedback is required per Feature
- Teams by layer or function cannot serve a feature; They cannot learn

Agile

- "Customers don't know what they want. They only know what they don't want when you built it for them." – Jez Humble
- Requirements are hypotheses
- Feedback results in learning

Requires at least daily releases to production to learn every day.

Frederick Taylor 1856 - 1915

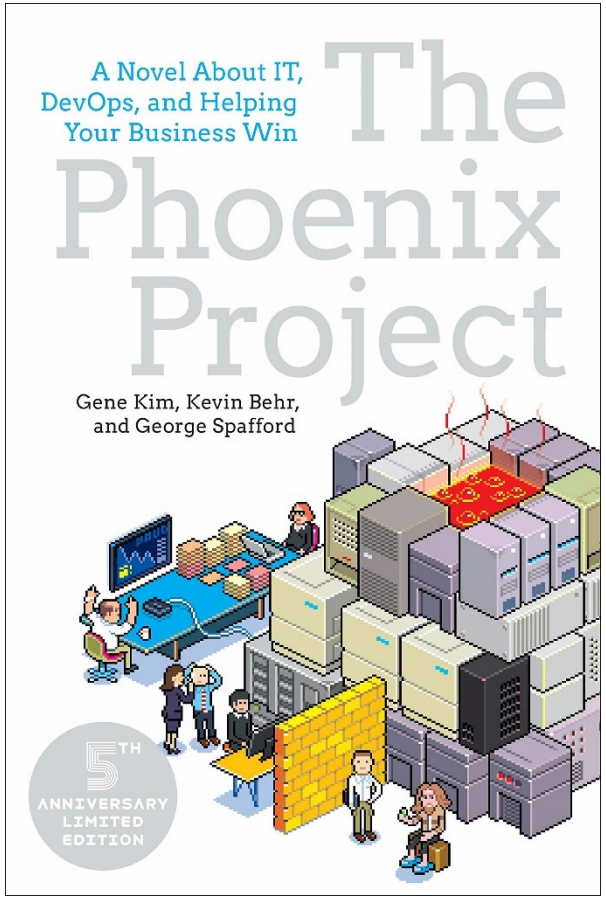
Continuous Delivery

- Automate build, test, and deploy into pipelines so that you can deploy to production at any time with no effort.
- “There is no trade-off of quality vs. speed in software. … The only way to go fast is to go well.” – Uncle Bob

- Shift Left for non-functional requirements (security, performance, …)
- Integrate them as tests into pipelines
- Manual security reviews must be automated and replaced.
- This is totally different to the standard ISO27001 implementation but can produce even better evidences for auditors to assert that controls are executed as planed.

DevOps Princibles & Ideals

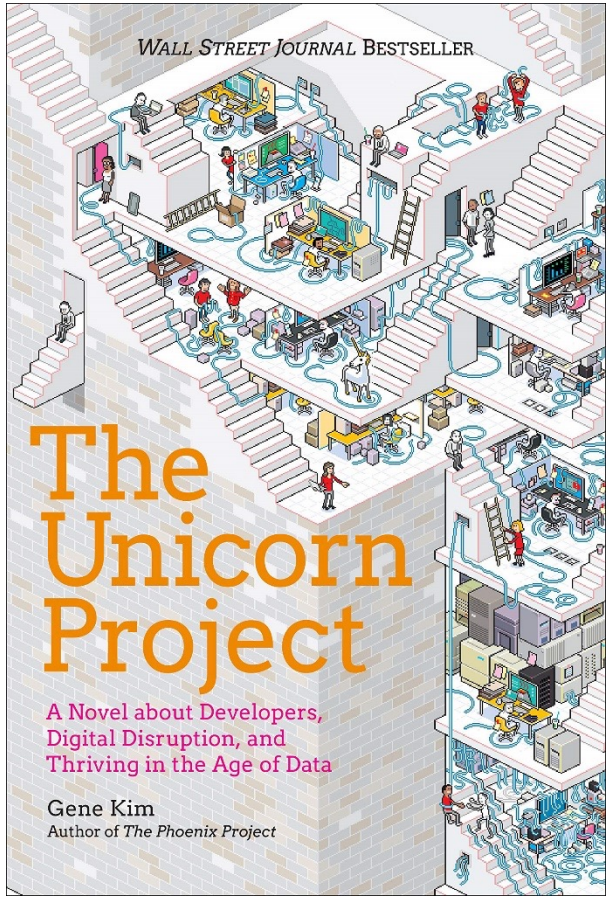
- The first Way: Flow Thinking
  - Structure Systems by flow of change, not by manager
- The second Way: Amplify Feedback Loops
  - Bring feedback from production to development teams
  - Let Dev-Teams operate their software
- The third Way: Culture of Continual Experimentation and Learning
  - Never let you boss decide (since (s)he is more far away from the problem), never fear failure
  - A failed experiment reveals more knowledge to the observer then one that goes as planed.
  - You shall only conduct single variable experiments



https://itrevolution.com/articles/the-three-ways-principles-underpinning-devops/

The Five Ideals of DevOps

- Locality and Simplicity
  - A development team can make local code changes in a single location without impacting various teams.
  - Decoupled Architecture is essential.
- Focus, Flow, and Joy
  - Focused work on value that is delivered to production makes developers happy.
  - Flow is interrupted by meetings, asking other teams, waiting for slow computers.
- Improvement of Daily Work
  - Improvement of daily work is more important than daily work itself.
  - Improved working procedures result in more, better work getting done.
- Psychological Safety
  - The organization is improved by feedback.
  - Leaders must support and reward it, so that everyone feels safe in giving feedback.
- Customer Focus
  - Only products that customers use can be developed hypothesis driven with customer feedback
  - Stop custom software development, where you have no customer feedback.



https://itrevolution.com/articles/five-ideals-of-devops/

Security in DevOps

- Security is everyone’s job.
  - Every Stream-Aligned Team must have enough security competency for daily work
- Security is a platform team
  - Common services are provided as tool
    - Monitor known vulnerabilities in artefacts
    - Static analysis tooling
    - …
  - It is ok for stream-aligned teams to be dependent on tools
  - It is not ok to be dependent on people from a platform team
- Security is an enabling team
  - Security team performs liaison with other teams to educate teams in security tools and practices

Learning more

Web Security
- https://portswigger.net/web-security/all-labs
- https://www.hacksplaining.com/
- https://wehackpurple.com/

General Offensive Security Skills
- https://www.hackthebox.com/
- https://tryhackme.com/

OWASP Stammtisch Köln https://owasp.org/www-chapter-germany/stammtische/koeln/

LangSec http://langsec.org
- Curing the Vulnerable Parser: Design Patterns for Secure Input Handling https://www.usenix.org/publications/login/spring2017/bratus

Devops: All Books from IT Revolution Press

Vorlesung Softwaretechnik

7. Muster in der Softwareentwicklung (Teil 2)

Prof. Bernhard Rumpe
Software Engineering
RWTH Aachen

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

SE Software Engineering | RWTH AACHEN UNIVERSITY

Warum, was, wie und wozu? Muster in der Softwareentwicklung

Warum?

- In Praxis weit verbreitet
- Hilfreich um Probleme zu lösen

Was?

- Entwurfsmuster: Struktur, Erzeugungen, Verhalten
- Strukturturmuster "im Kleinen"

Wie?

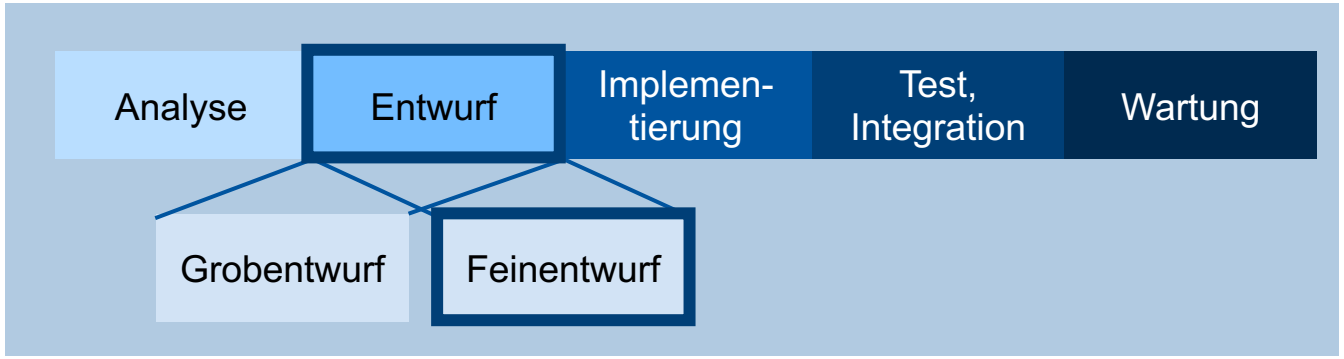
- Identifikation eines Problems
- Suche nach passendem Muster
- Anwendung auf konkretes Problem

Wozu?

- Strukturbildung für Daten- und technische Klassen
- Wiederverwendung von Design-Erfahrung

Softwaretechnik

7. Muster
7.1. Entwurfsmuster



Prof. Bernhard Rumpe
Software Engineering
RWTH Aachen

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

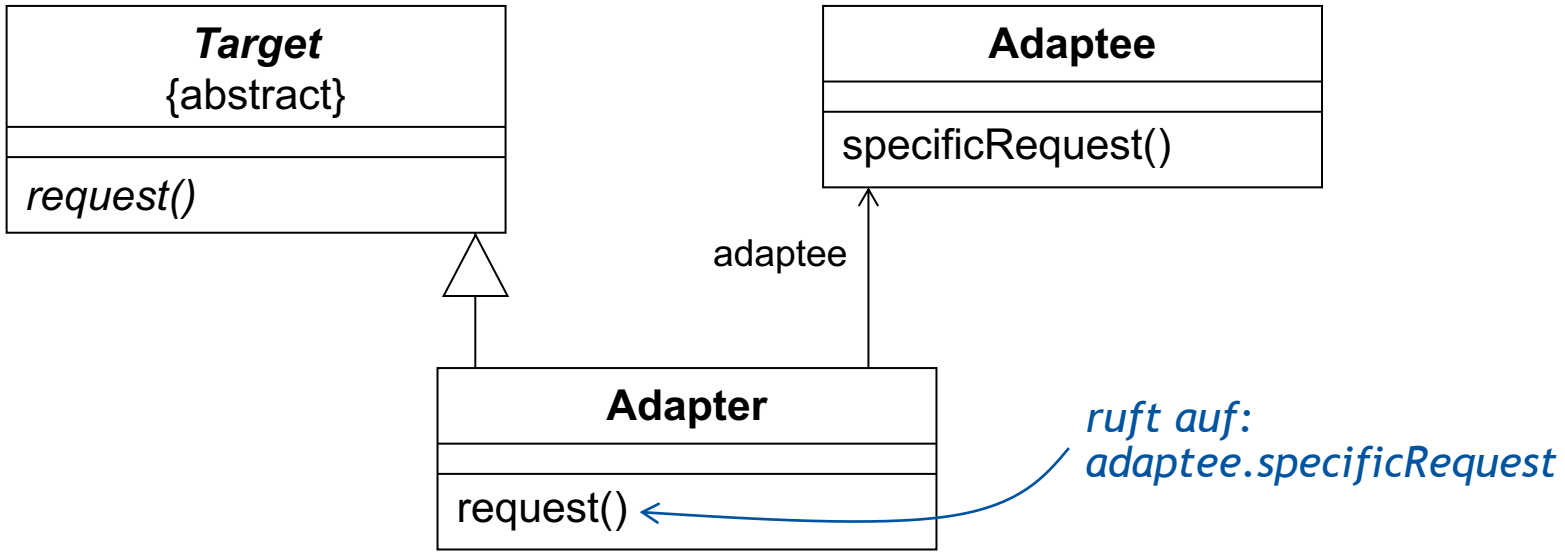
Literatur:

- Gamma/Helm/Johnson/Vlissides: Design Patterns, Addison-Wesley 1994 (= „Gang of Four“, „GoF“)
- Buschmann/Meunier/Rohnert/Sommerlad/Stal: A System of Patterns, Wiley 1996

Strukturmuster Adapter | Variante 1: Objektadapter

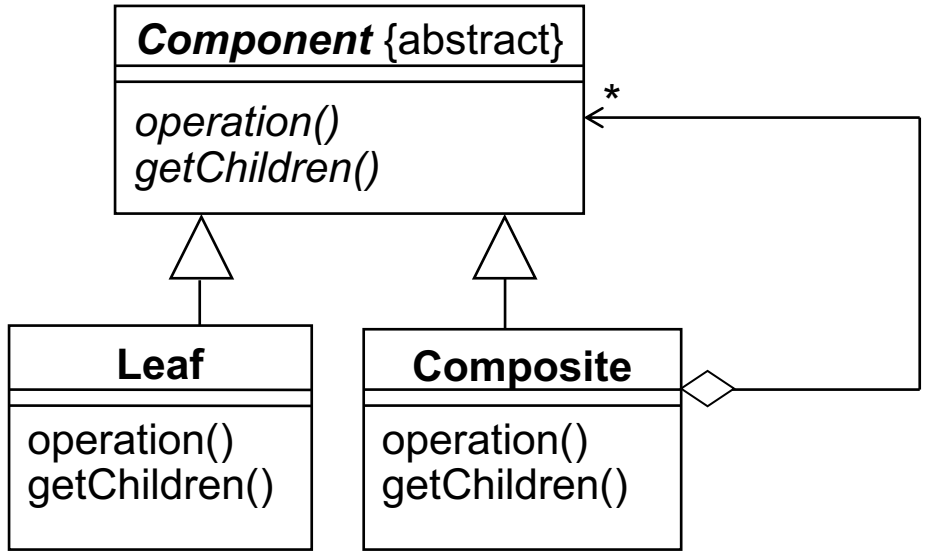
Wiederholung

- Name: **Adapter** (auch: Wrapper)
- *Problem*:
  - Anpassung der Schnittstelle eines vorgegebenen Objekts (*adaptee*) auf eine gewünschte Schnittstelle (*target*)
- *Lösung*:



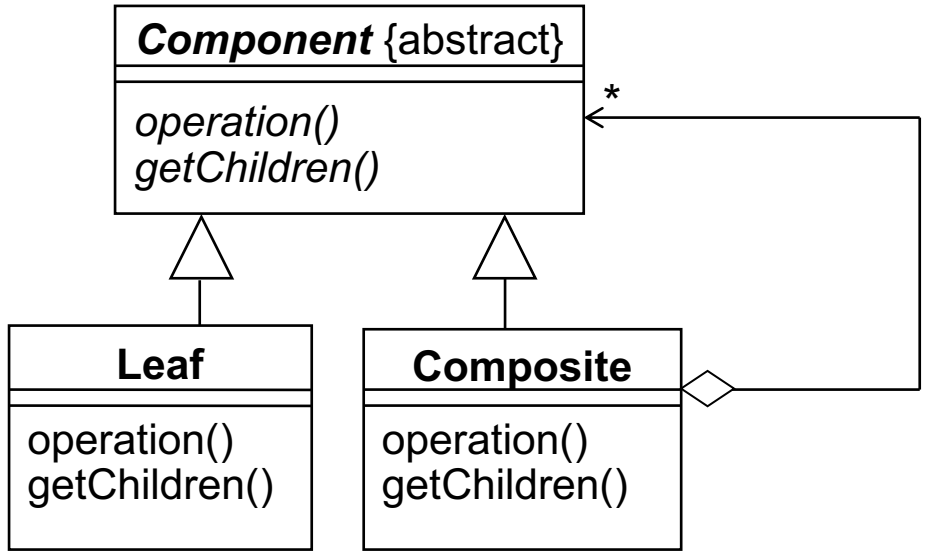
Entwurfsmuster Composite

- … ist ein Strukturmuster
- *Problem*: Hierarchische Struktur von Objekten
- *Lösung*: Einheitliche abstrakte Schnittstelle für „Blätter“ und Verzweigungsknoten eines Baumes
- … und wurde auch als Analysemuster schon identifiziert und diskutiert



Grundidee: Verantwortlichkeiten trennen

- “Separation of concerns”:
  - Jede Einheit soll einen Aufgabenkomplex gut lösen.
  - Der Aufgabenkomplex einer Einheit soll in sich geschlossen sein (hohe *Kohäsion*).
  - Die Einheit soll so wenig wie möglich von anderen Einheiten abhängen (niedrige *Kopplung*).
- Praktische Anwendung im Composite-Beispiel:
  - Die Mechanismen zur Realisierung der Komposition in einer Klasse zusammengefasst
  - Einige Operationen sind für (fast) alle Dokument-Elemente einheitlich verwendbar: diese werden in einer Klasse zusammengefasst.

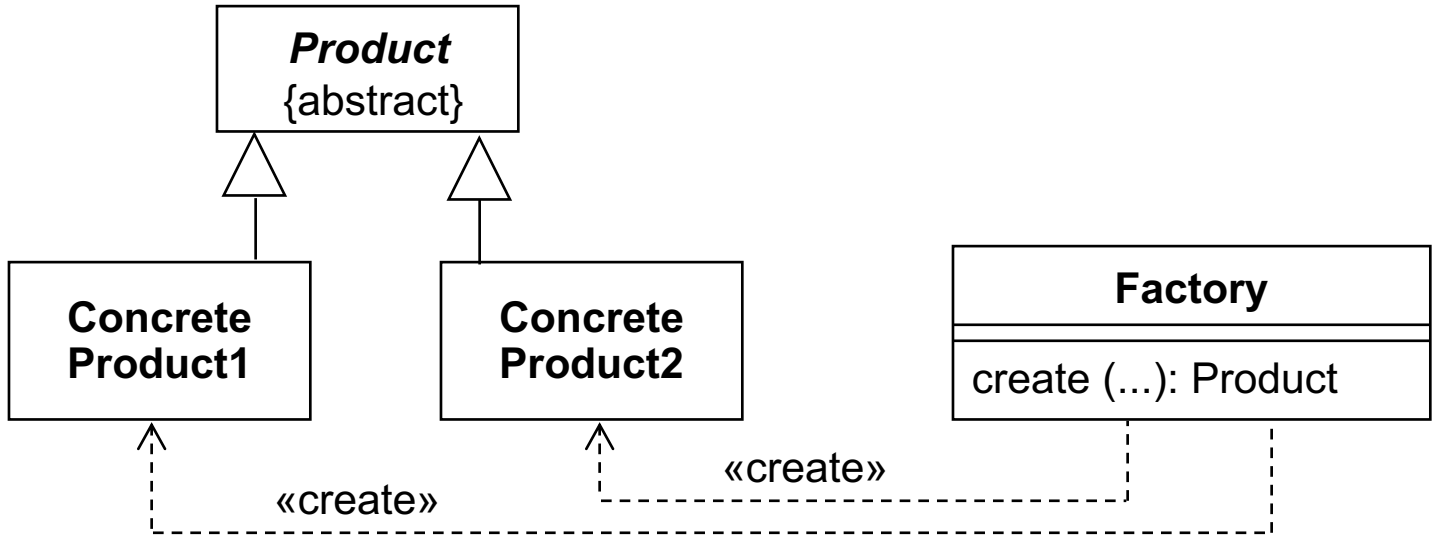


Erzeugungsmuster Factory Method

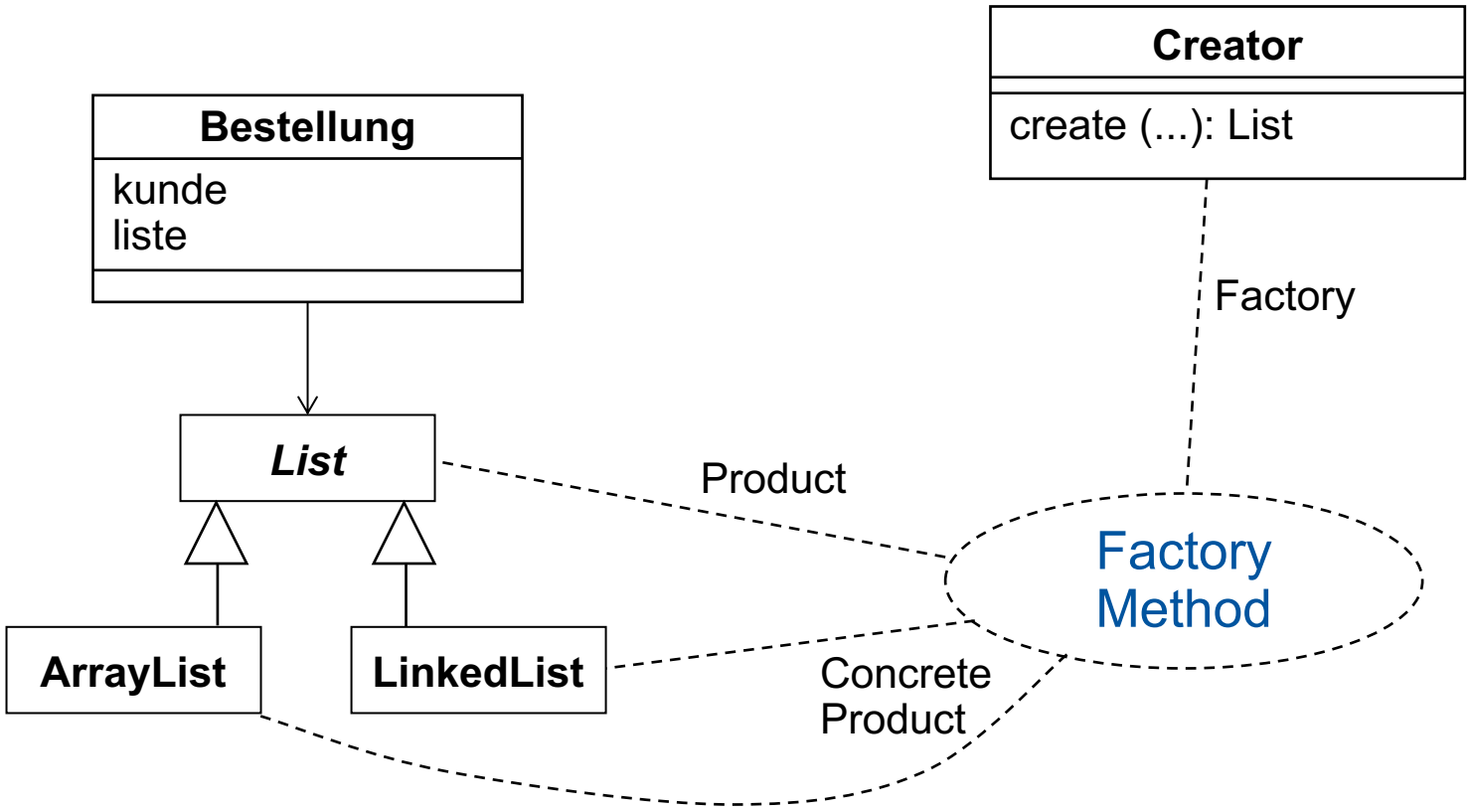
- Name: **Factory Method** (dt.: Fabrikmethode)

- *Problem*:
  - Bei der Erzeugung von Objekten soll zwischen Varianten gewählt werden; dies soll aber zum Zeitpunkt der Erzeugung geschehen, ohne dass der Auftraggeber der Erzeugung damit beschäftigt ist.
  - Besonders geeignet auch fürs Testen (um Dummies ins System zu injizieren)

- *Lösung*:

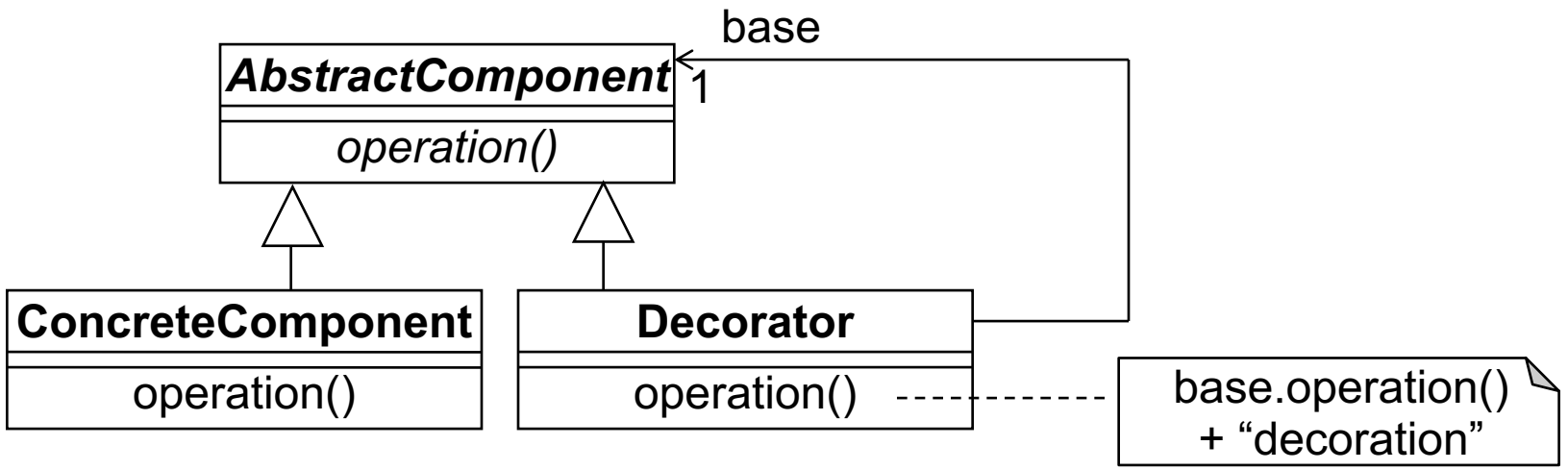


Factory Method - Beispiel



Entwurfsmuster Decorator

- *Problem*: Zu einer Klasse, die eine abstrakte Schnittstelle implementiert, sollen flexibel weitere Eigenschaften hinzugefügt werden, so dass eine neue Klasse entsteht, die die gleiche Schnittstelle implementiert.
- *Lösung*: Definition einer Klasse für Zwischenobjekte, die die Operationen an die Originalklasse delegieren, nachdem sie ggf. zusätzliche Funktionalität erbracht haben.



Achtung: Dies ist etwas vereinfachte Version (ohne abstrakte Decorator-Oberklasse)

Entwurfsmuster Singleton

- … ist ein Erzeugungsmuster

- *Problem*: Manche Klassen sind nur sinnvoll, wenn sichergestellt ist, dass immer höchstens eine Instanz der Klasse besteht (und diese bei Bedarf erzeugt wird).

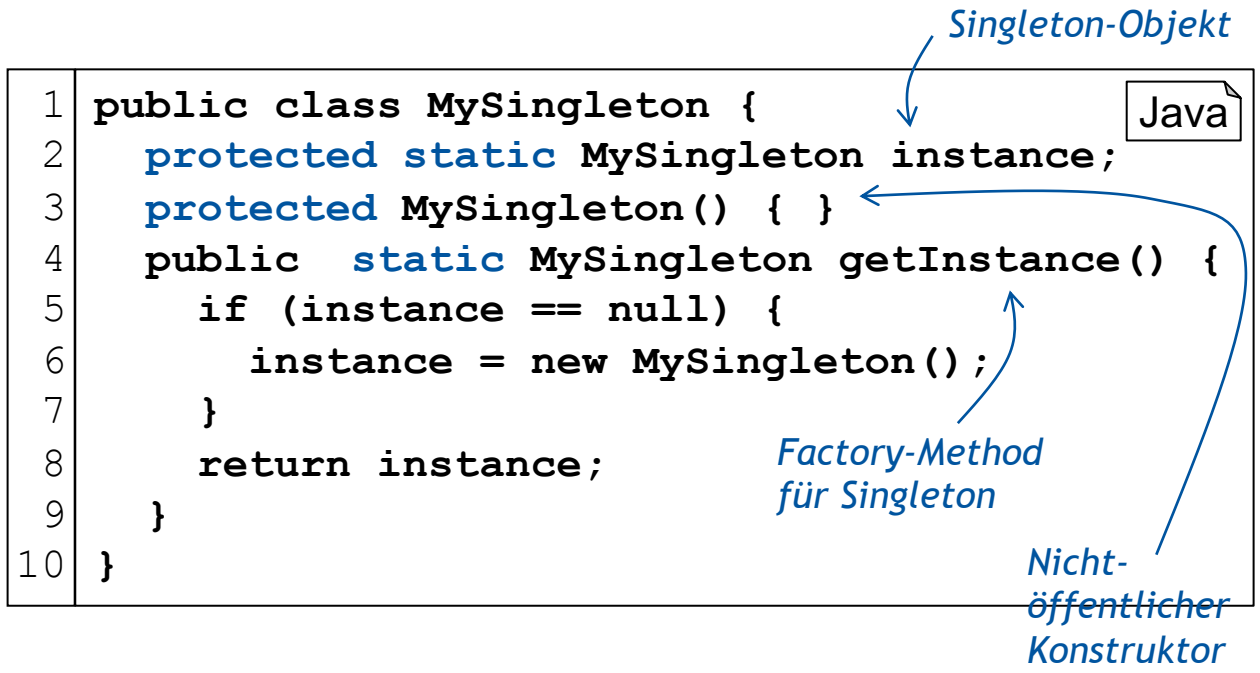
- *Lösung*:
  - Modellebene: Klasse als Singleton auszeichnen
  - Programmebene: Sprachabhängig

Beispiele:
- Protokoll-, DB- Schnittstellen
- „Manager“-Klassen sind meist Singletons.

- Lösungen sind technisch verschieden je nach Programmiersprache, umfassen aber immer die selben wesentlichen Elemente (eine globale (statische) Variable theInstance, Modifikationen beim Konstruktor).

Singleton

- Singleton Strukturteil (Z. 1-10):
  - Management des einen Singleton-Objekts
  - Automatische Instanziierung, wenn noch keine Instanz vorhanden



Singleton – mit „Override Static“ Erweiterung

- Singleton Strukturteil (Z. 1-10):
  - Management des einen Singleton-Objekts
  - Automatische Instanziierung, wenn noch keine Instanz vorhanden
- Funktionsteil:
- Erlaubt Nutzung der statischen Methode (von überall im Code):

```
MySingleton.someMethod()
```

- aber auch das dadurch aufgerufene Verhalten anzupassen in Subklassen, weil an das Singleton Objekt delegiert wird

Java

```
1  public class MySingleton {
2    protected static MySingleton instance;
3    protected MySingleton() { }
4    private static MySingleton getInstance() {
5      if (instance == null) {
6        instance = new MySingleton();
7      }
8      return instance;
9    }
10
11   public static void someMethod() {
12     getInstance()._someMethod();
13   }
14
15   protected void _someMethod() {
16     /* Do something */
17   }
18 }
```

Factory-Method jetzt private

Statische Methoden delegieren an Singleton-Objekt

Override Static: Verhalten verändern

- Notwendig: `MySubSingle.init()` einmal zur Initialisierung aufrufen
- *Vorteile* des Singleton mit Override für Methoden:
  - Global bekannter Zugriff (wie bei static üblich)
  - Überschreibbare Funktion (Adaption) bei den Aufrufen statischer Funktionen
    - `MySingleton.someMethod()`
  - Objekt wird nicht nach außen gegeben: kompakt (Hiding) und stateless (Neuinitialisierung jederzeit möglich)
  - Nutzende Klassen kennen Subklasse(n) nicht → Konfigurierbarkeit

```java
public class MySubSingle extends MySingleton {

  protected MySubSingle() {};

  public static void init() {
    instance = new MySubSingle();
  }

  @override
  protected void _someMethod() {
    /* do something else */
  }
}
```

Instanziierung des Singleton durch Subklasse

Überschreiben der Funktionalität

```java
public class MySingleton {
  protected static MySingleton instance;
  protected MySingleton() { }

  protected void _someMethod() {
    /* Do something */
  }
}
```

Singleton – mit „Override Static“ Erweiterung – C++ Variante

- Ähnlich zu Java ist, aber:
  - Nötige Initialisierung des Singleton-Objekts (Z. 18)
  - Muss genau ein mal gemacht werden
  - Kann nicht wiederholt werden
- Memory-Leak, Speicher des Objekts wird nicht freigegeben!
  - Nur begrenzt schlimm, da ja Singleton, aber
  - Lösung zB. durch sogenannte „Unique Pointers“ möglich, wenn das Singleton mit der Zeit oft ausgetauscht werden soll

C++

```
1  class MySingleton {
2    protected:
3      static MySingleton *instance;
4      MySingleton() {}
5      virtual void _someMethod() {/*...*/}
6    private:
7      static MySingleton* getInstance() {
8        if (instance == nullptr) {
9          instance = new MySingleton();
10       }
11       return instance;
12     }
13   public:
14     static void someMethod() {
15       getInstance()->_someMethod();
16     }
17 };
18 MySingleton* MySingleton::instance
19     = nullptr;
```

Singleton-Objekt

Nicht-öffentlicher Konstruktor

Factory-Method für Singleton

Statische Methoden delegieren an Singleton-Objekt

Einmalige Initialisierung

Singleton – mit „Override Static“ Erweiterung – Speicher-behandelnde C++ Variante

- „Unique Pointers“ managen die Speicherfreigabe

C++

```
1  #include <memory>
2  class MySingleton {
3    protected:
4      static std::unique_ptr<MySingleton> instance;
5      MySingleton() {}
6      virtual void _someMethod() {/*...*/}
7    private:
8      static MySingleton& getInstance() {
9        if (instance.get()==0) {
10         instance.reset(new MySingleton());
11       }
12       return *instance;
13     }
14   public:
15     static void someMethod() {
16       getInstance()->_someMethod();
17     }
18 };
19 std::unique_ptr<MySingleton>
20   MySingleton::instance(nullptr);
```

Import

Singleton-Objekt

Unique Pointer

Nicht-öffentlicher Konstruktor

Factory-Method für Singleton

Statische Methoden delegieren an Singleton-Objekt

Einmalige Initialisierung

Override Static: Verhalten verändern – C++ Variante

- Ähnlich zu Java

```cpp
class MySubSingle : public MySingleton {

  protected:
    MySubSingle() {}
    void _someMethod() {
      /* do something else*/
    }

  public:
    static void init() override {
      instance.reset(new MySubSingle());
    }
};
```

C++

Überschreiben der Funktionalität

Instanziierung des Singleton durch Subklasse

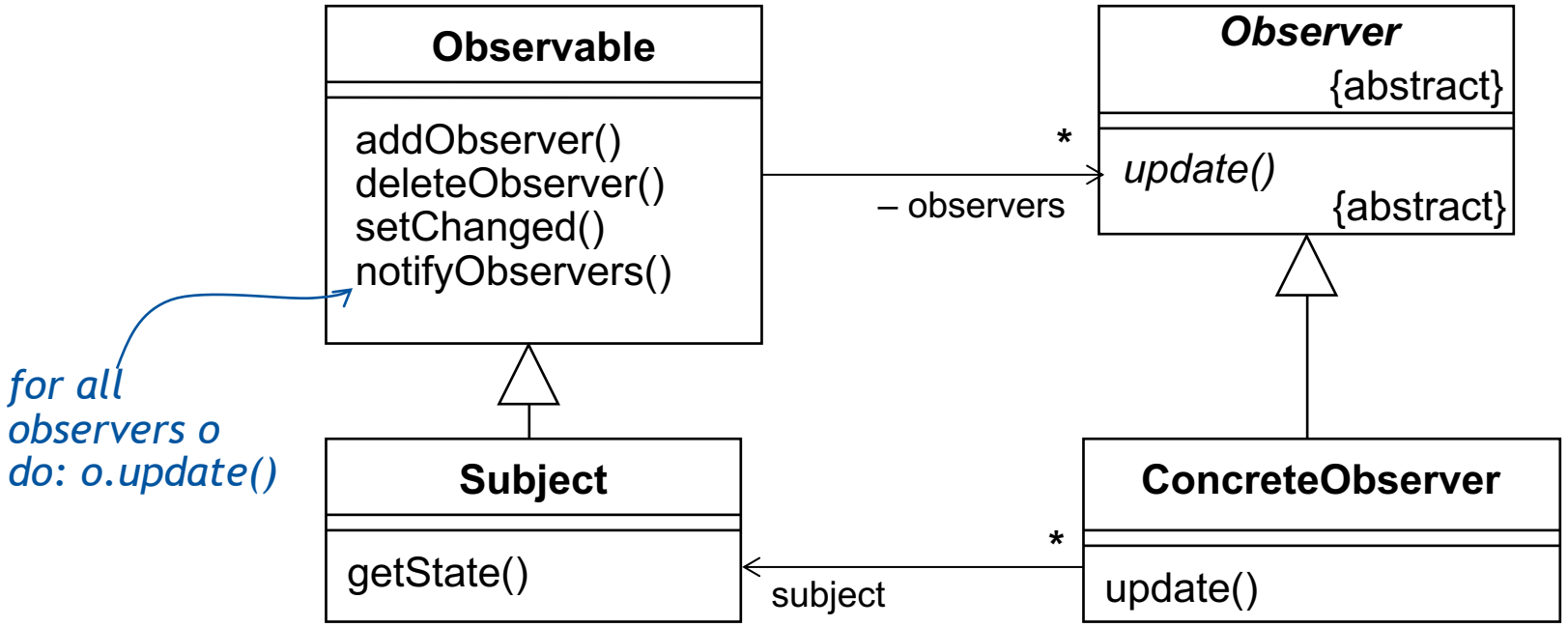
```cpp
#include <memory>
class MySingleton {
  protected:
    static std::unique_ptr<MySingleton> instance;
    MySingleton() {}
    virtual void _someMethod() {/*...*/}
```

C++

Verhaltensmuster Observer

- Name: **Observer** (dt.: Beobachter)
- *Problem*:
  - Mehrere Objekte sind interessiert an bestimmten Zustandsänderungen eines Objektes

- *Lösung*:



Konkrete Realisierungen weichen meist in Details ab (z.B. Interface Observer) !

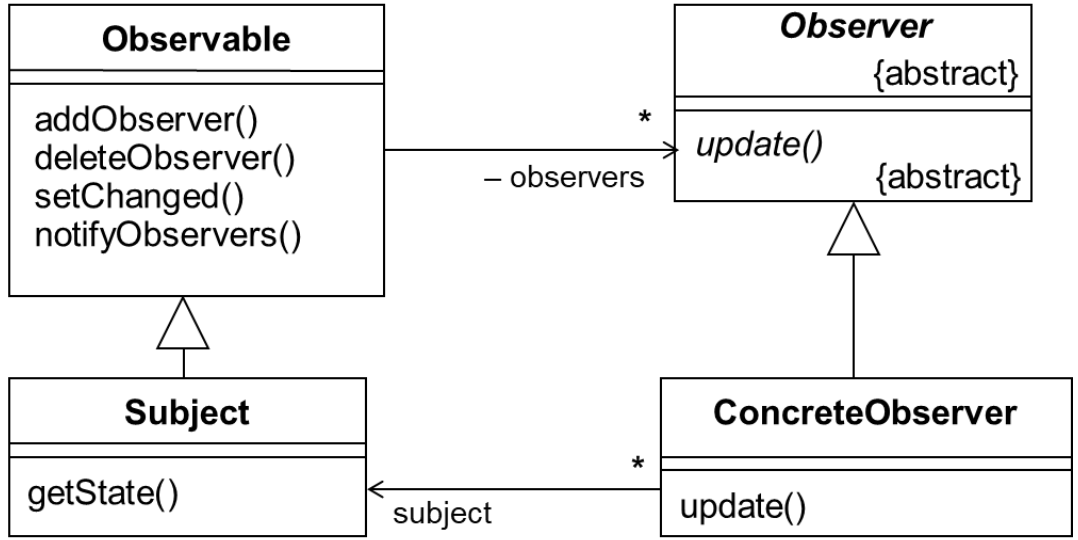
java.util.Observable, java.util.Observer

```java
public class Observable {
  public void addObserver(Observer o);
  public void deleteObserver(Observer o);

  protected void setChanged();
  public void notifyObservers();
  public void notifyObservers(Object arg);
}
```

Java

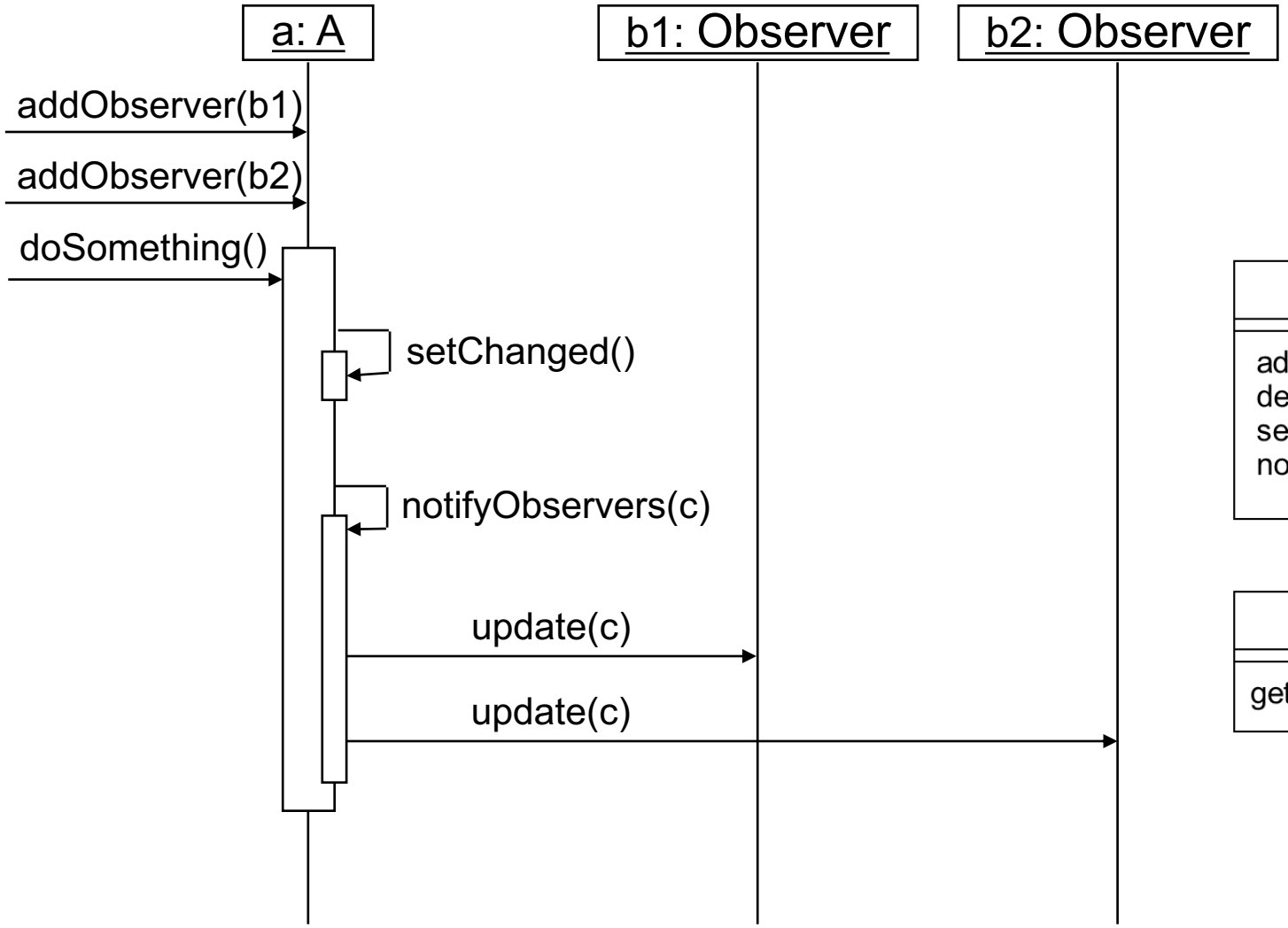
```java
public interface Observer {
  public void update(Observable o, Object arg);
}
```

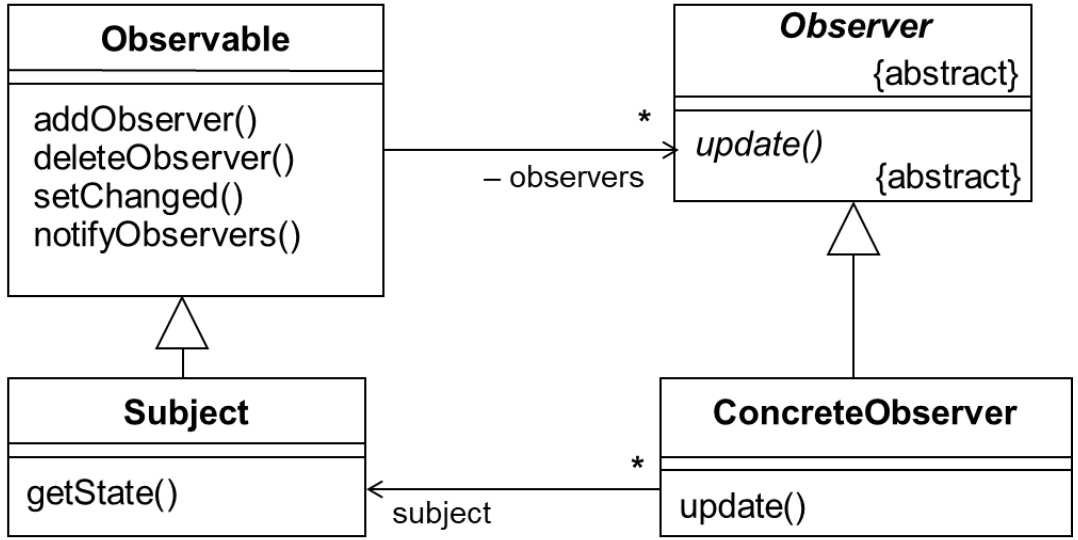
Java



- Parameter von `notifyObservers()`:
  - meist nur Art der Änderung, nicht gesamte Zustandsinformation
  - Beobachter können normale Methodenaufrufe nutzen, um sich näher zu informieren.
- C++, other languages: Design the classes yourself

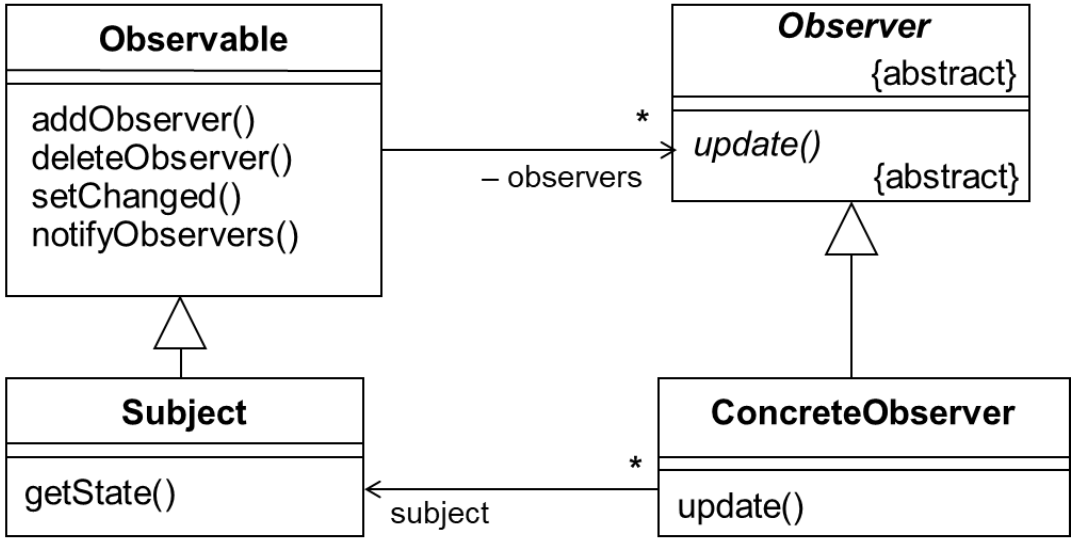
Beispielablauf beim Observer



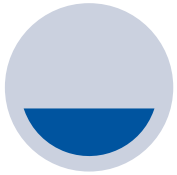


Vorteile des Observer-Musters

- Jede Klasse des Modells definiert lokal,
  - welche Veränderungen beobachtbar sind (d.h. ***aktiv*** der Umwelt mitgeteilt werden) und
  - wann die Mitteilung erfolgen soll.
- Eine beobachtete Klasse
  - hat keine im Code verankerte Kenntnis und über ihre Beobachter;
  - muss nicht geändert werden, wenn sich Beobachter verändern;
  - muss nicht geändert werden, wenn neue Beobachter dazukommen oder Beobachter wegfallen.
- Methodik:
  - Hinweise auf wichtige Zustandsveränderungen gibt das UML-Zustandsdiagramm.



Was haben wir gelernt?

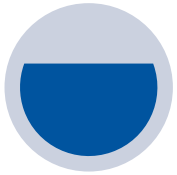


Kategorien und Arten

es gibt schon in dem GoF-Buch 23 Entwurfsmuster in drei Kategorien:

- Struktur-,
- Erzeugungs- und
- Verhaltensmuster

Viele weitere Entwurfsmuster für teilweise spezifische Problemstellungen



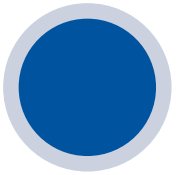
Muster sind hilfreich zur…

…Verbesserung der Struktur des Codes

Kommunikation über Entwurfs-entscheidungen

Einsatz:

- den Anforderungen angepasst
- diesen als *Schablonen* für den Entwurf



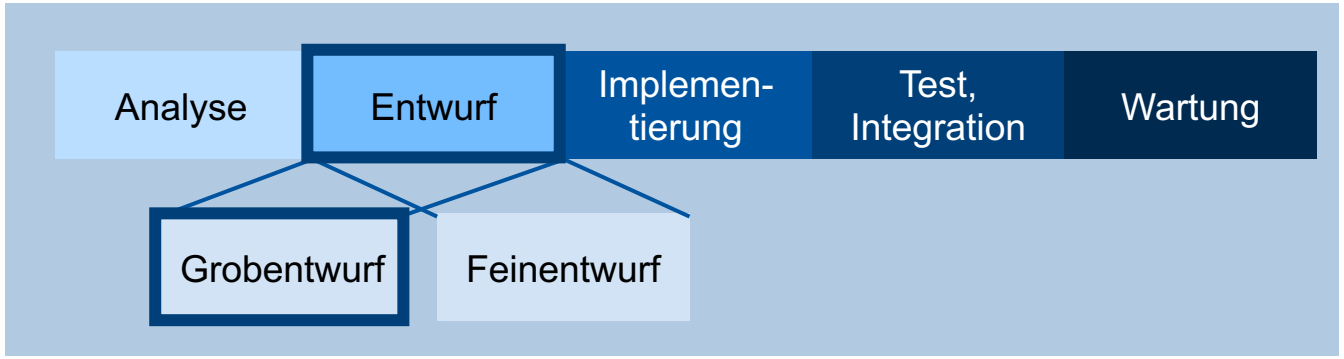
Diese Vorlesung

… behandelt einige der wesentlichen Muster

Für Einsatz in der Praxis: Weiteren Ausbau der Kenntnisse & ggf. Recherche nach passenden Mustern, wenn konkreter Bedarf besteht

Softwaretechnik

7. Muster
7.2. Architekturmuster für die Struktur



Prof. Bernhard Rumpe
Software Engineering
RWTH Aachen

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

Literatur:

- Gamma/Helm/Johnson/Vlissides: Design Patterns, Addison-Wesley 1994 (= „Gang of Four“, „GoF“)
- Buschmann/Meunier/Rohnert/Sommerlad/Stal: A System of Patterns, Wiley 1996

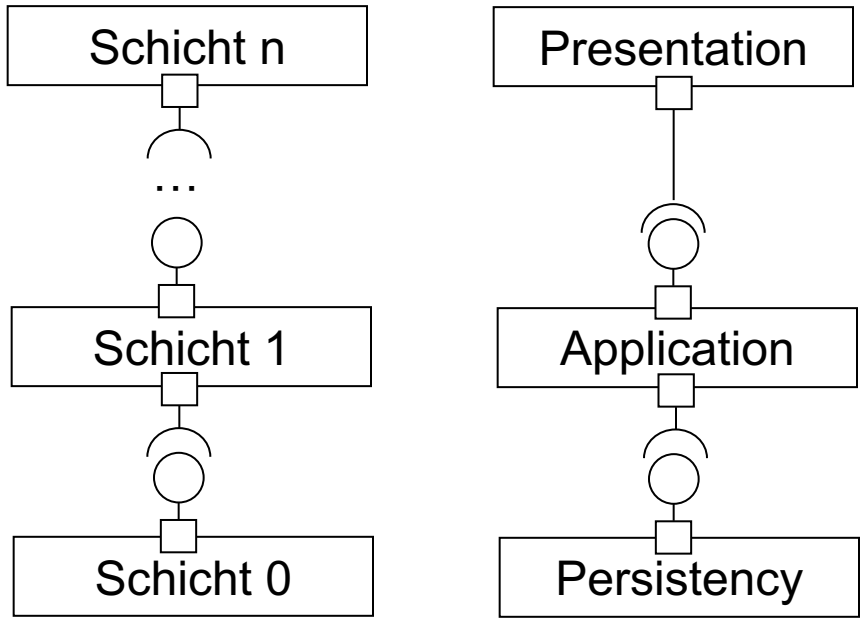
Architekturmuster: Schichtenbildung (Layers)

- Das Layer-Architekturmuster hilft bei der Strukturierung von Anwendungen.
- Indikatoren:
  - Unteraufgaben haben einen eigenen Abstraktionslevel
  - Abstraktere Aufgaben hängen jeweils von konkreten ab
  - Teile des Systems sollen austauschbar sein, wobei die Schnittstellen erhalten bleiben
- Beispiel: ISO/OSI-Modell

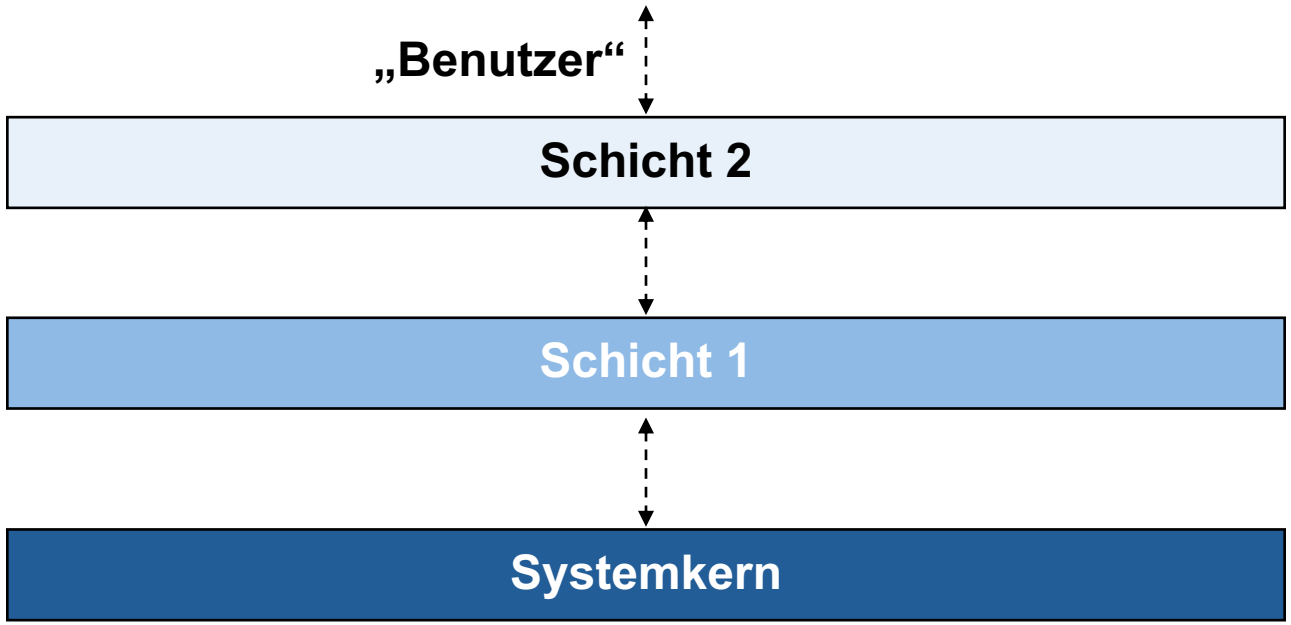
Architekturmuster: Schichtenbildung (Layers)

- Das Layer-Architekturmuster hilft bei der Strukturierung von Anwendungen.
- Indikatoren:
  - Unteraufgaben haben einen eigenen Abstraktionslevel
  - Abstraktere Aufgaben hängen jeweils von konkreten ab (und nutzen diese)
  - Teile des Systems sollen austauschbar sein, wobei die Schnittstellen erhalten bleiben
- Beispiel: ISO/OSI-Modell
- Einführung verschiedener Schichten
  - Dienste werden nach oben angeboten
  - Unteraufgaben werden nach unten delegiert
- Schichten können übersprungen werden (Relaxed Layered System)

CSD

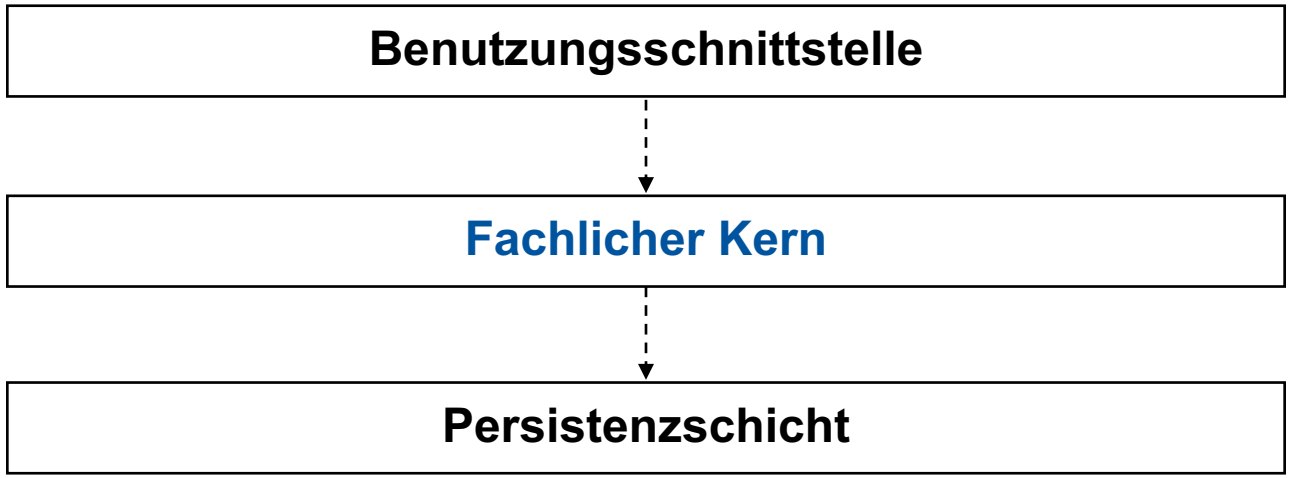


Architekturmuster „Schichten“



- Jede Schicht bietet Dienste (nach oben) und nutzt Dienste (von unten)
- Beispiele:
  - Kommunikationsprotokolle
  - Datenbanksysteme, Betriebssysteme

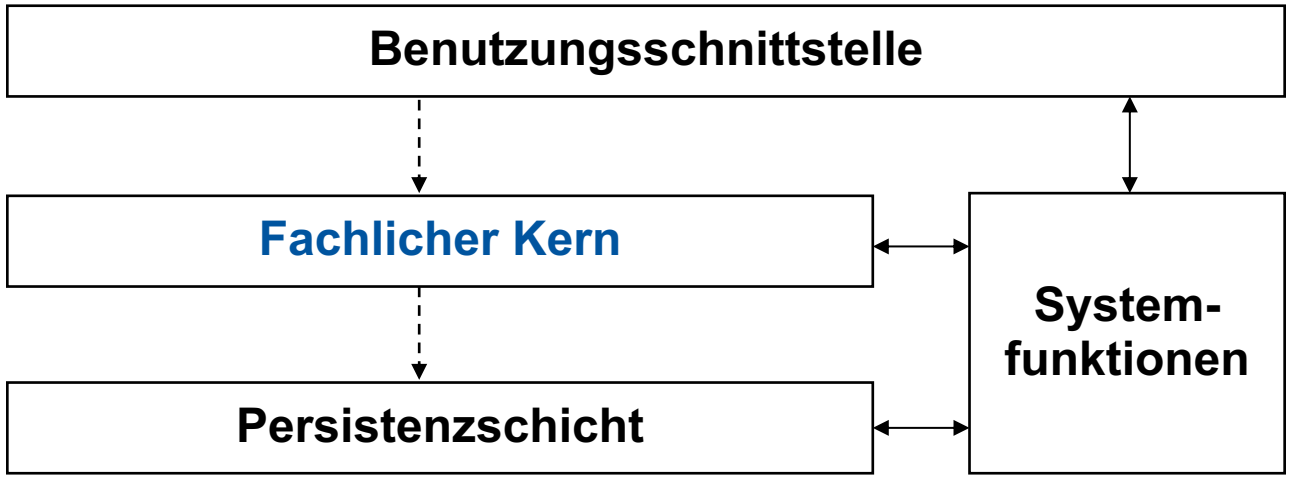
Beispiel: 3-Schichten-Referenzarchitektur



- *Entwurfsregeln*:
  - Benutzungsschnittstelle greift ***nie*** direkt auf Datenhaltung zu.
  - Persistenzschicht verkapselt Zugriff auf Datenhaltung, ist aber nicht identisch mit dem Mechanismus der Datenhaltung (z.B. Datenbank).
  - Fachlicher Kern basiert auf dem Analyse-Modell

- Erlaubt das Aufsetzen von interaktiven, batch, etc. Benutzerschnittstellen und den Austausch von Datenbanken

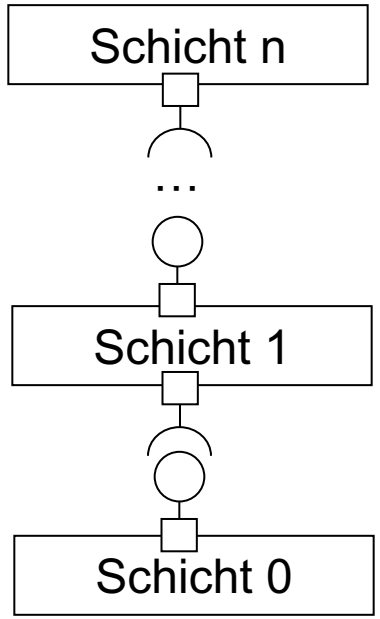
Variante: 3-Schichten-Referenzarchitektur



- Beispiele für Systemfunktionen:
  - Verkapselung von plattformspezifischen Funktionen
  - Schnittstellen zu Fremdsystemen
  - Fehlerbehandlung

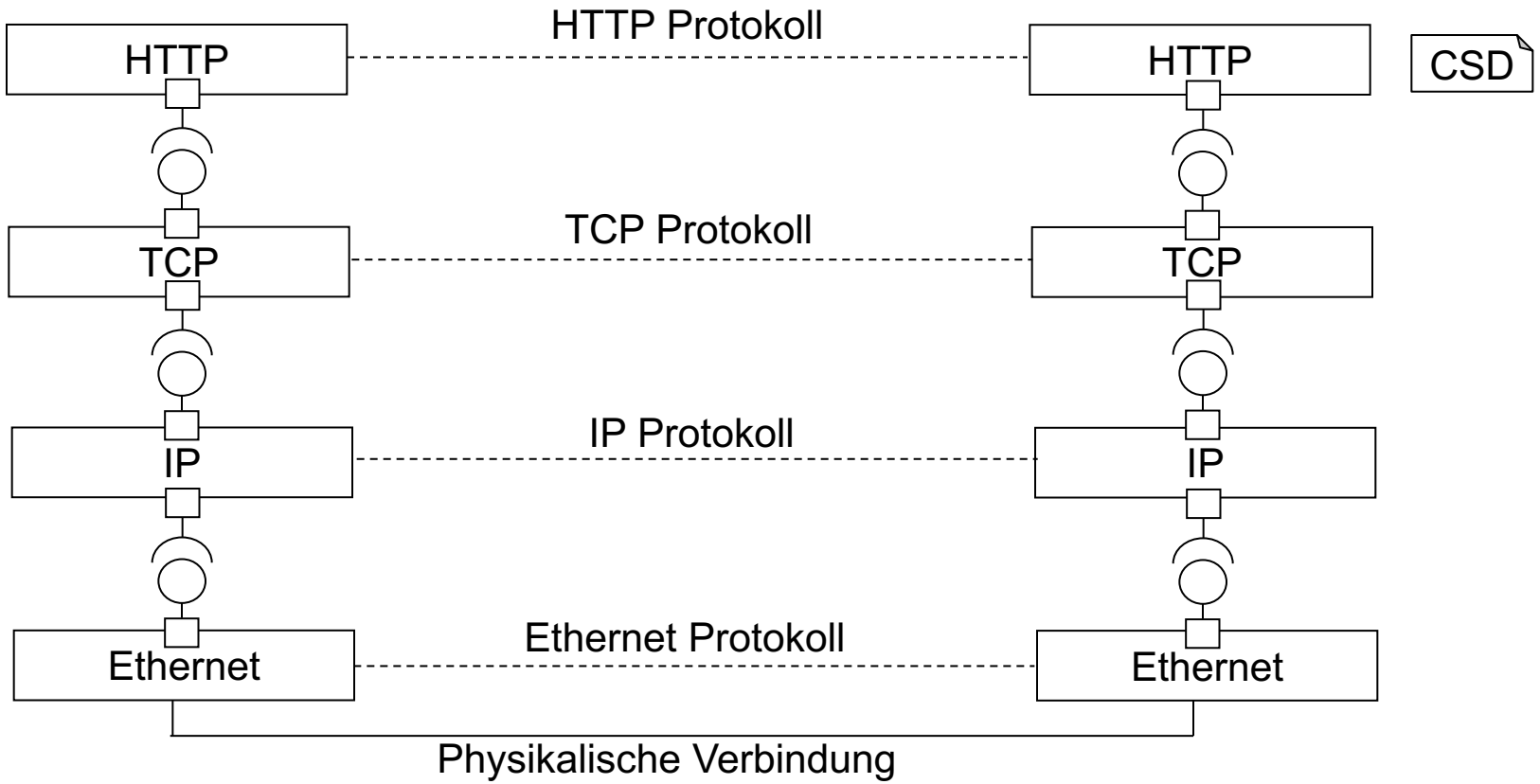
Schichten: Vor- und Nachteile

- Vorteile:
  - Wiederverwendbarkeit einzelner Schichten
  - Standardisierung einzelner Schichten möglich
  - Geringe Abhängigkeit zwischen den Komponenten: Gute Modifizierbarkeit
  - Austauschbarkeit einzelner Schichten



- Probleme:
  - Geringe Effizienz bei vielen Schichten
    - Daten durchlaufen viele Schichten und werden nur an wenigen Stellen bearbeitet
    - Jede Schicht erhält alle Daten, benötigt aber eventuell nur wenige davon
  - Anzahl der Schichten
    - Zu wenig: Muster entfaltet nicht das volle Potential
    - Zu viele: Muster führt zu unnötigem Overhead.

Beispiel: Schichtenbildung TCP/IP

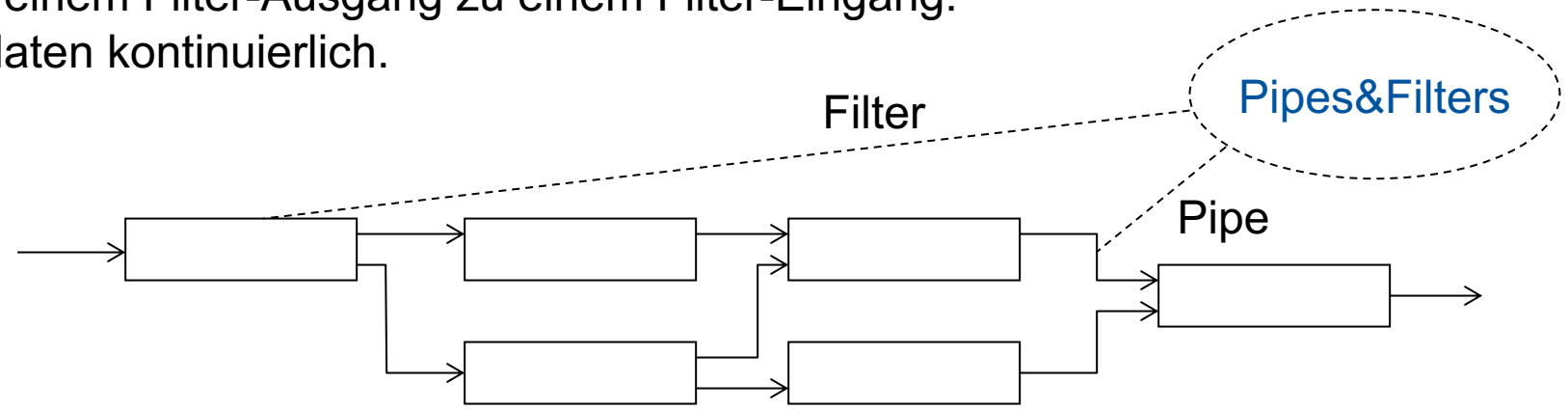


- Logische Kommunikation zwischen den Diensten
- Tatsächliche Kommunikation nur über die physikalische Verbindung

Architekturmuster: Pipes and Filters

- *Problem*:
  In einem System durchlaufen Datenströme verschiedene Bearbeitungsstufen.
  - Der Datenfluss soll leicht änderbar sein.
  - Bearbeitungsstufen sollen austauschbar, erweiterbar und wiederverwendbar sein.

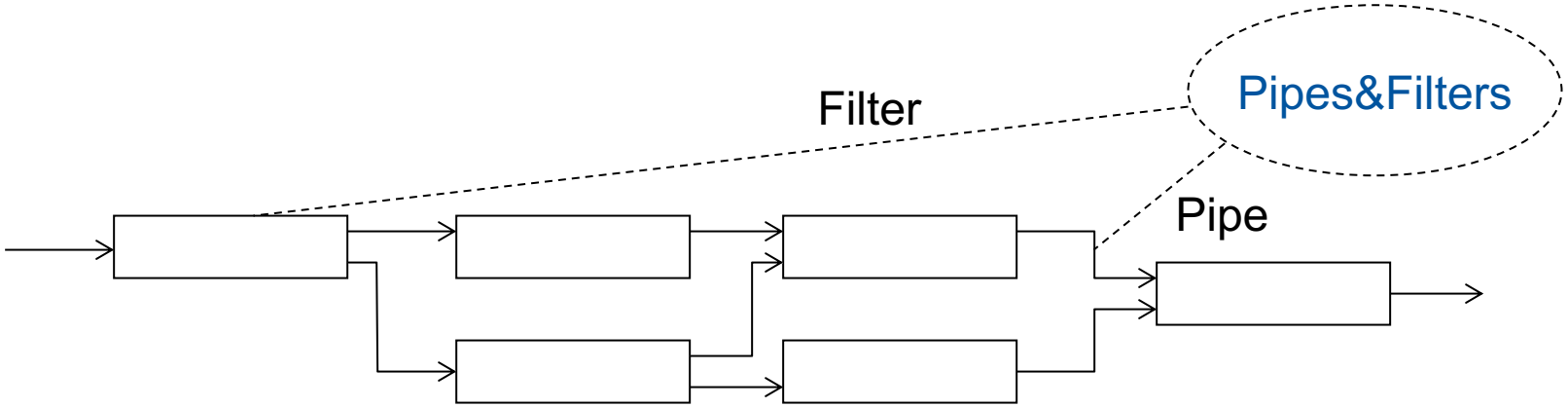
- *Lösung*:
  Beschreibung von Bearbeitungsstufen durch Filters und Datenströme durch Pipes.
  - Filter-Komponenten haben Ein/Ausgänge und bearbeiten einen Datenstrom.
  - Pipes sind ein Datenstrom von einem Filter-Ausgang zu einem Filter-Eingang.
  - Filter bearbeiten die Eingangsdaten kontinuierlich.



Pipes and Filters

- *Vorteile*:
  - Keine temporären Dateien erforderlich
  - Flexibilität durch Austausch und Rekombination von Filtern
  - Rapid-Prototyping

- *Nachteile*:
  - Keine globalen Daten vorhanden
  - Jeder Filter muss Daten eventuell erneut parsen
  - Fehlerbehandlung schwierig, da verteilt

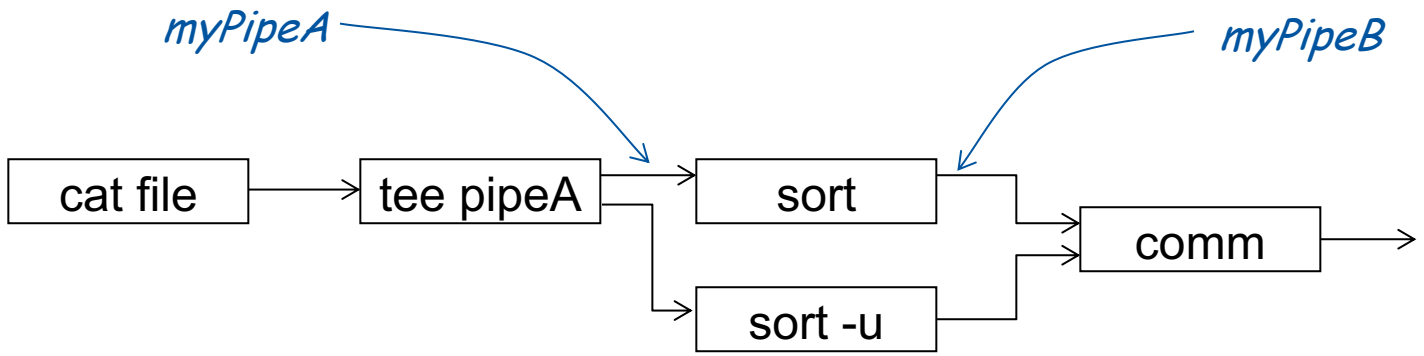


Beispiel: UNIX Pipes und Filters

- Ausgabe der doppelten Zeilen einer Datei in sortierter Reihenfolge
- Folgendes shell programm macht was?

shell

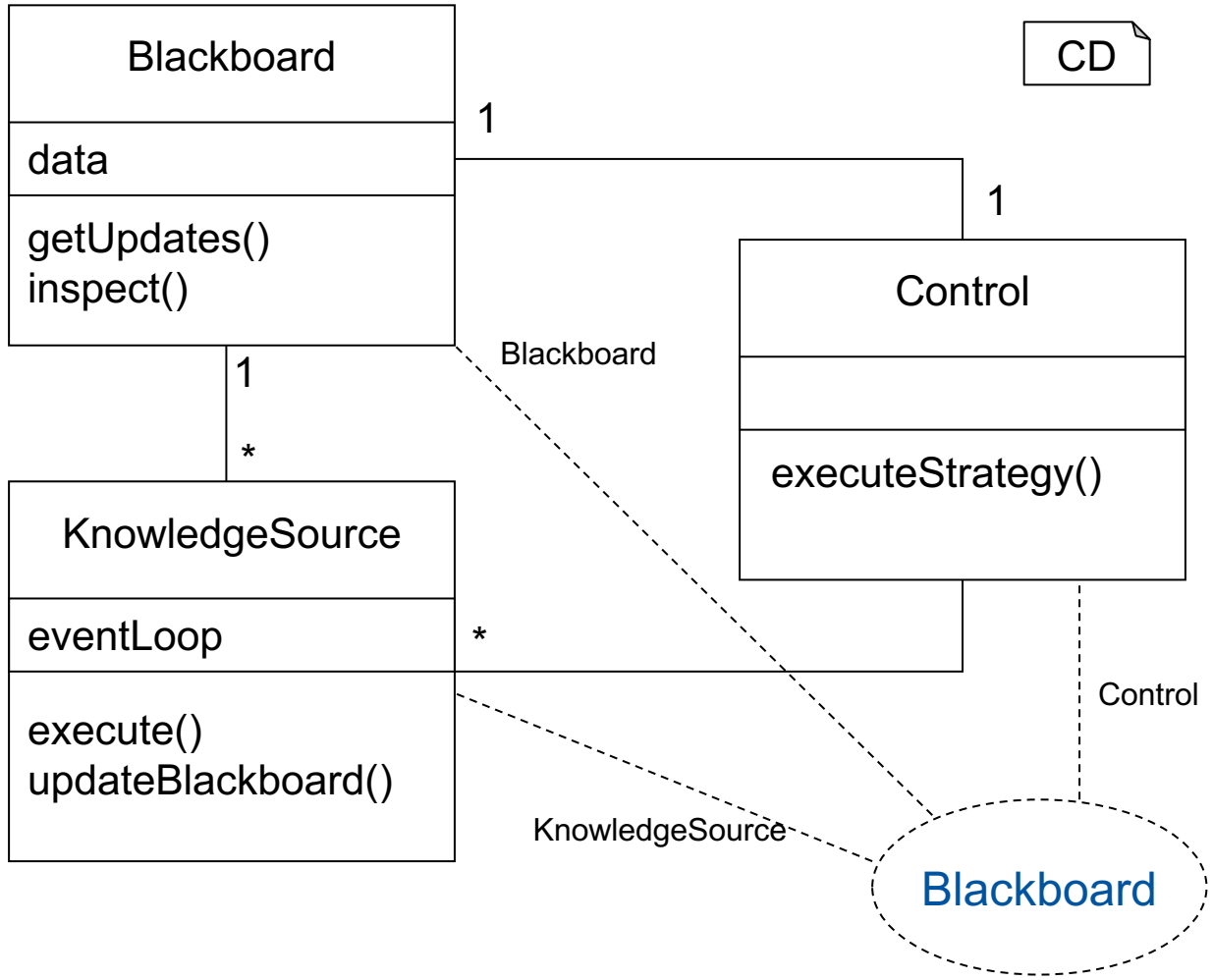
```
1 mknod myPipeA p
2 mknod myPipeB p
3 sort myPipeA > myPipeB &
4 cat file  |  tee myPipeA  |  sort -u  |  comm -13 -myPipeB
```



Architekturmuster: Blackboard

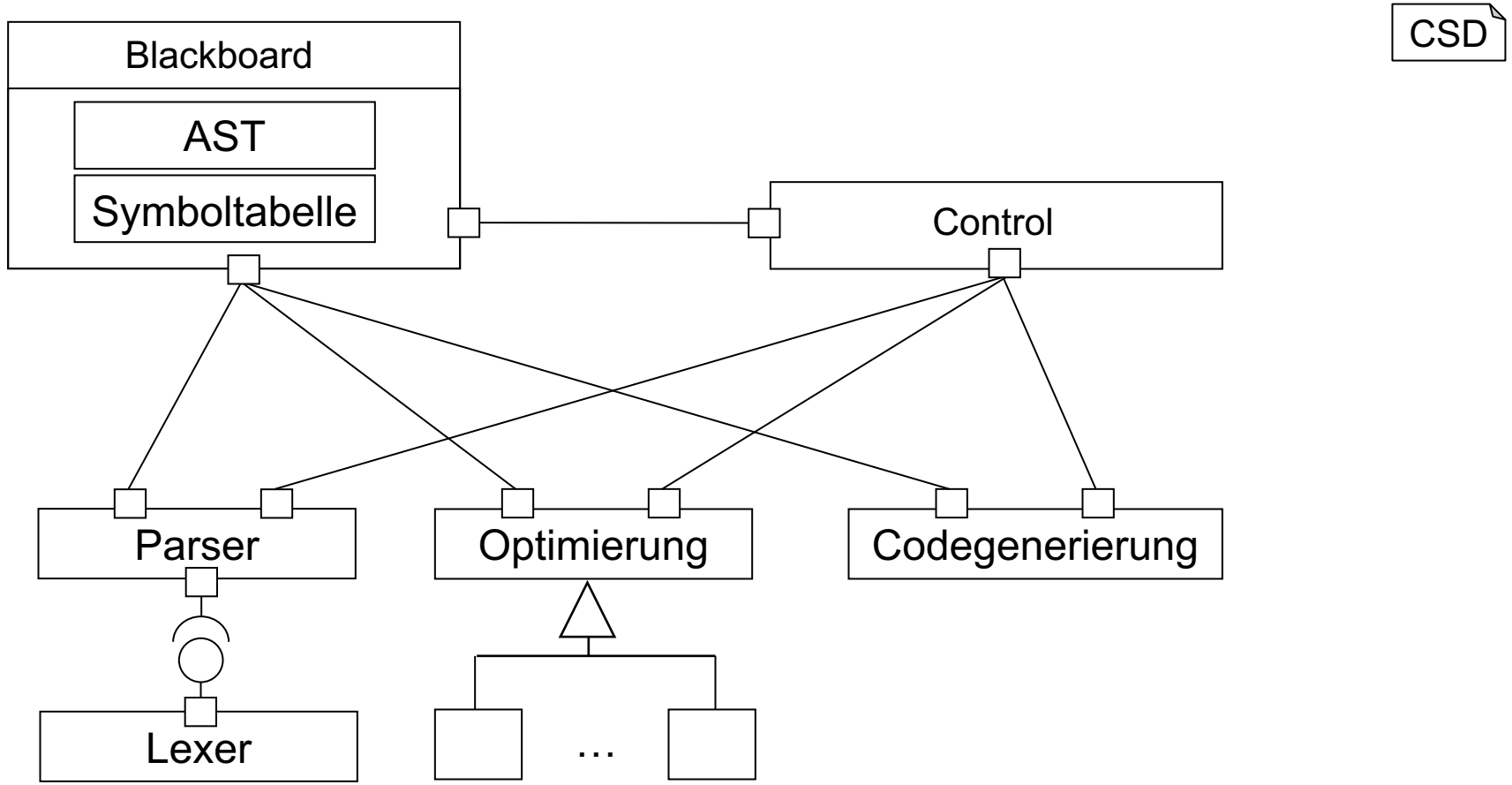
- Verschiedene unabhängige Subsysteme nutzen und bilden gemeinsame Daten
- Problem:
  - Gesucht eine flexible Architektur für experimentelle Algorithmen
  - Zusammenspiel der Komponenten durch Lösungsstrategie beschreibbar
- Lösung:
  - Sammlung der Daten im Blackboard
  - Beschreibung der Lösungsstrategie in der Komponente Control
  - Verwendung voneinander unabhängiger KnowledgeSources
- Schwierigkeiten
  - Zusammenspiel der KnowledgeSources schlecht testbar
  - Nebenläufigkeit der KnowledgeSources

- Muster auch bekannt als *Repository*



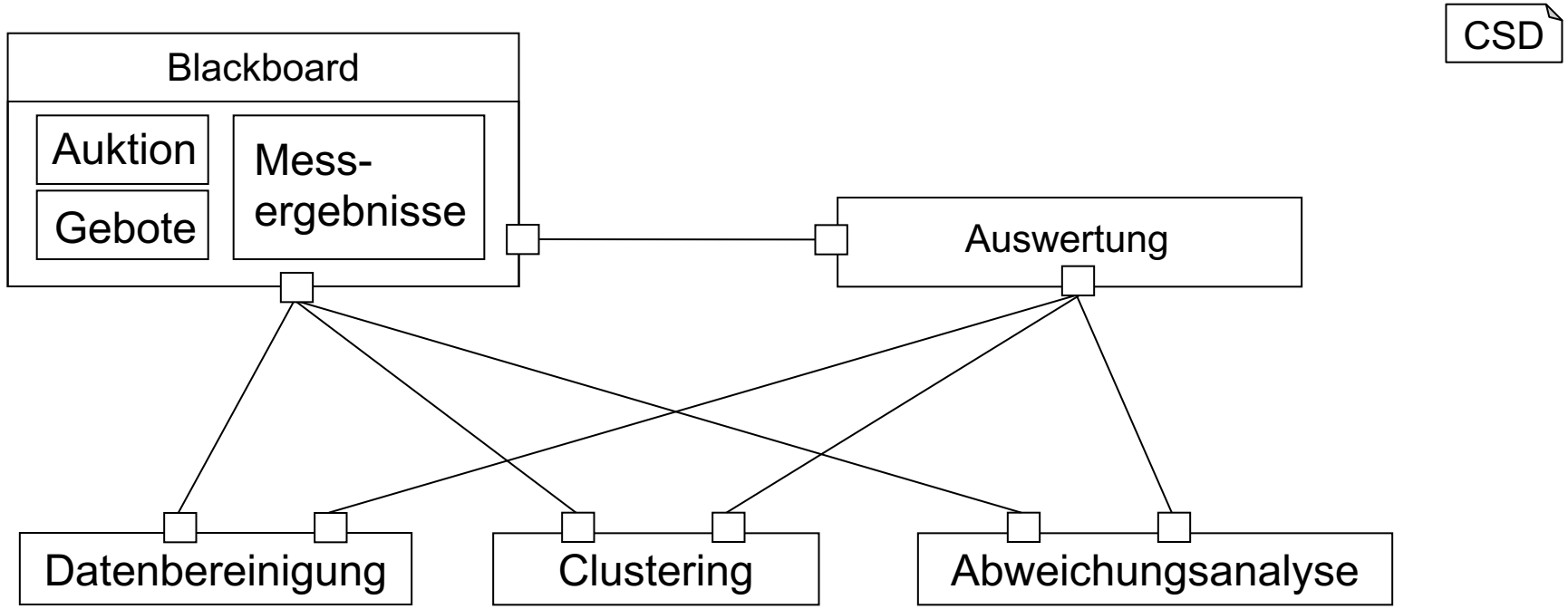
Beispiel: Blackboard im Compiler

- Blackboard-Architektur mit relativ klarer Kontrollstrategie: Compiler



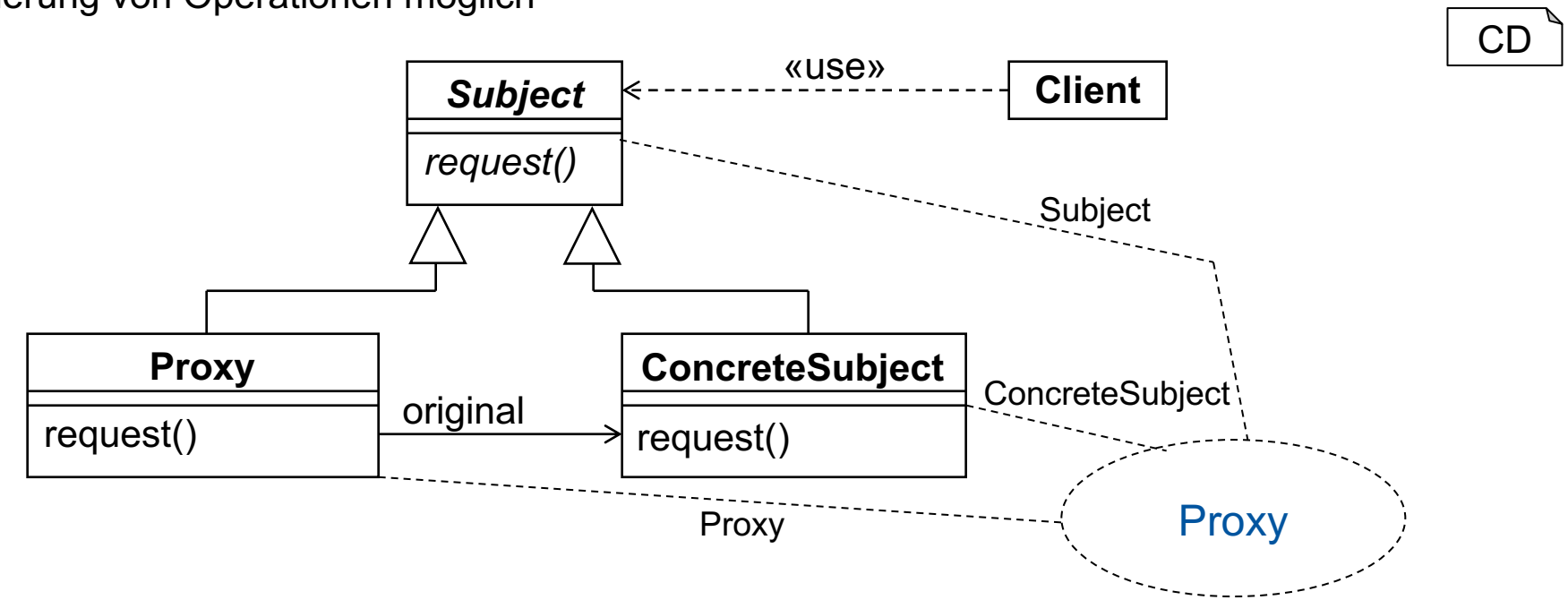
Beispiel: Blackboard im Auktionssystem

- Einsatz von Data-Mining-Techniken zur Detektion von Absprachen zwischen Bietern
- Die Auswertung startet die Analysen
- Die Auswertung greift auf die Ergebnisse verschiedener Algorithmen zurück

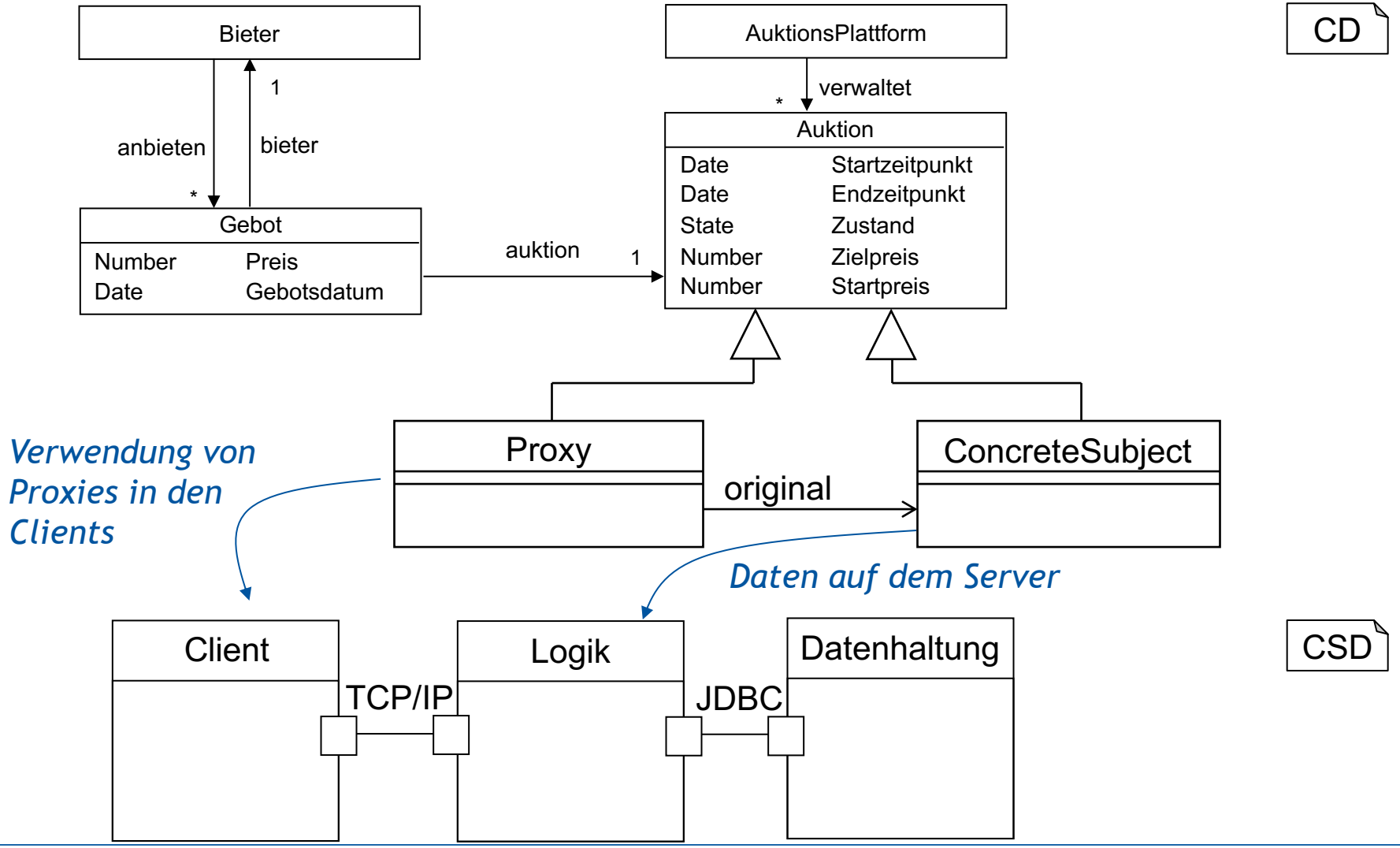


Architektur- (und Entwurfs-)muster Proxy

- *Problem*: Der direkte Zugriff auf ein Objekt ist problematisch.
  (z.B. großer Aufwand, Sicherheitsprobleme, physische Entfernung)
- *Lösung*: Stellvertreter-Objekt
  - Alias erlaubt alle Operationen, die auf Originalen möglich sind
  - Weiterleitung von Operationen an das Original
  - Spezielle Realisierung von Operationen möglich



Beispiel: Auktionsplattform



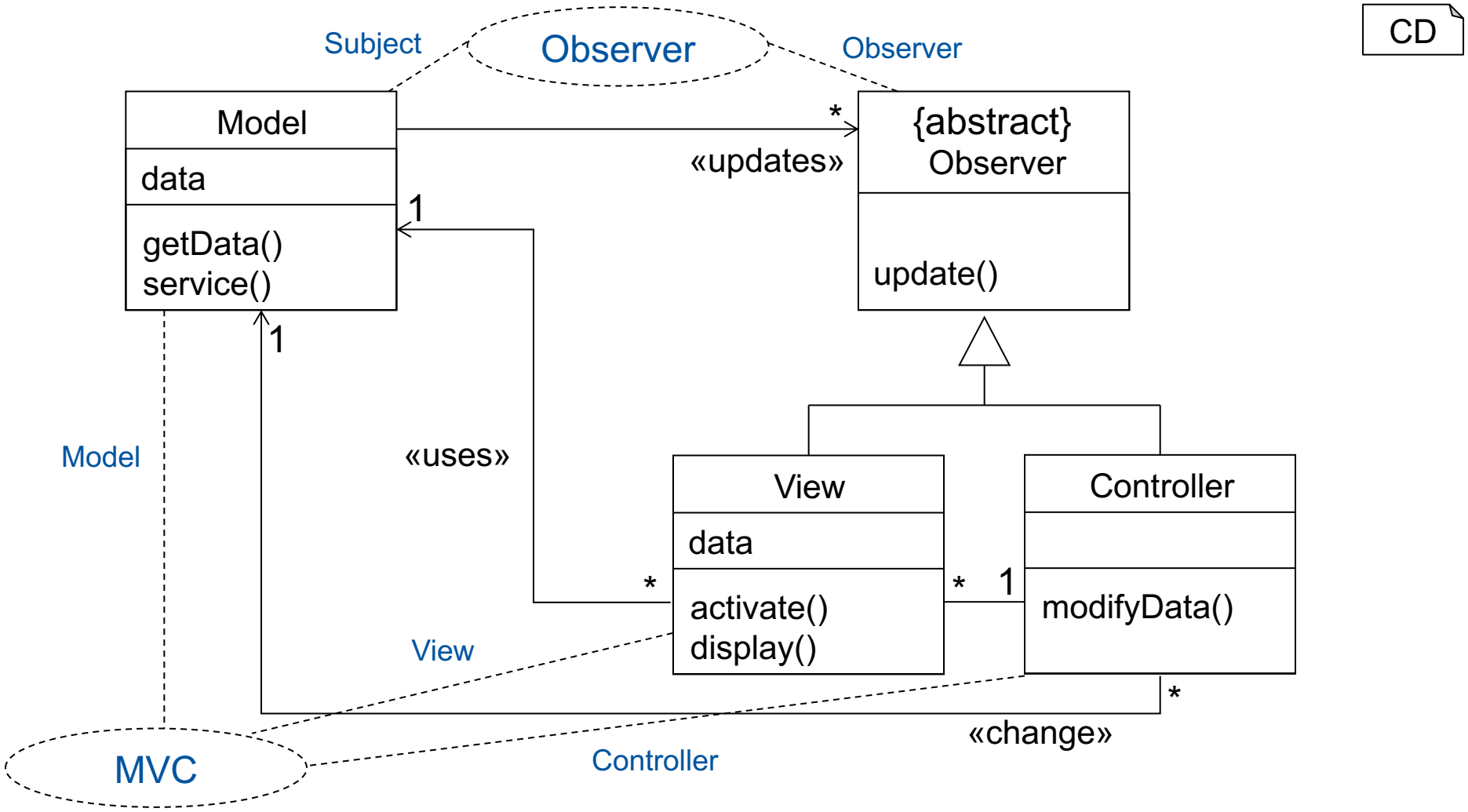
Architekturmuster: Model-View-Controller

- Einsatzbedingungen:
  - Eine Anwendung enthält mehrere Sichten auf dieselben Daten
  - Der Benutzer kann die Daten in den Sichten verändern
  - Die Sichten müssen stets aktuell gehalten werden und Änderungen der Daten dargestellt werden

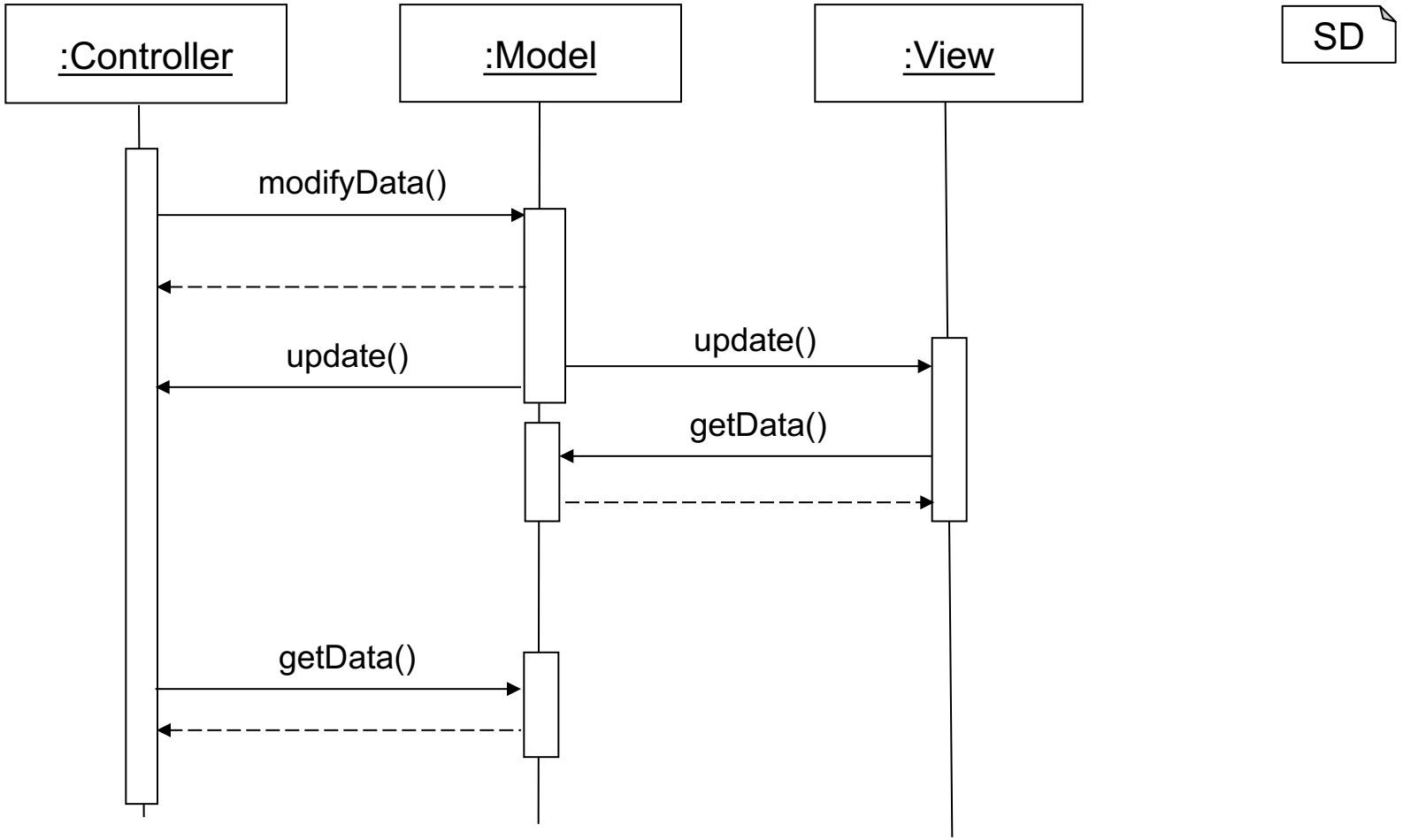
- *Lösung*:
  - Aufteilung des Systems in drei Komponenten
    - Model
      - Datenhaltung des Systems
    - View
      - Sichten auf die Daten
    - Controller
      - Benutzerschnittstelle und Modifikation der Daten

  - oft: verschränkter Einsatz des Observer-Musters zur Benachrichtigung

Model-View-Controller: Klassendiagramm (Variante nutzt Observer)



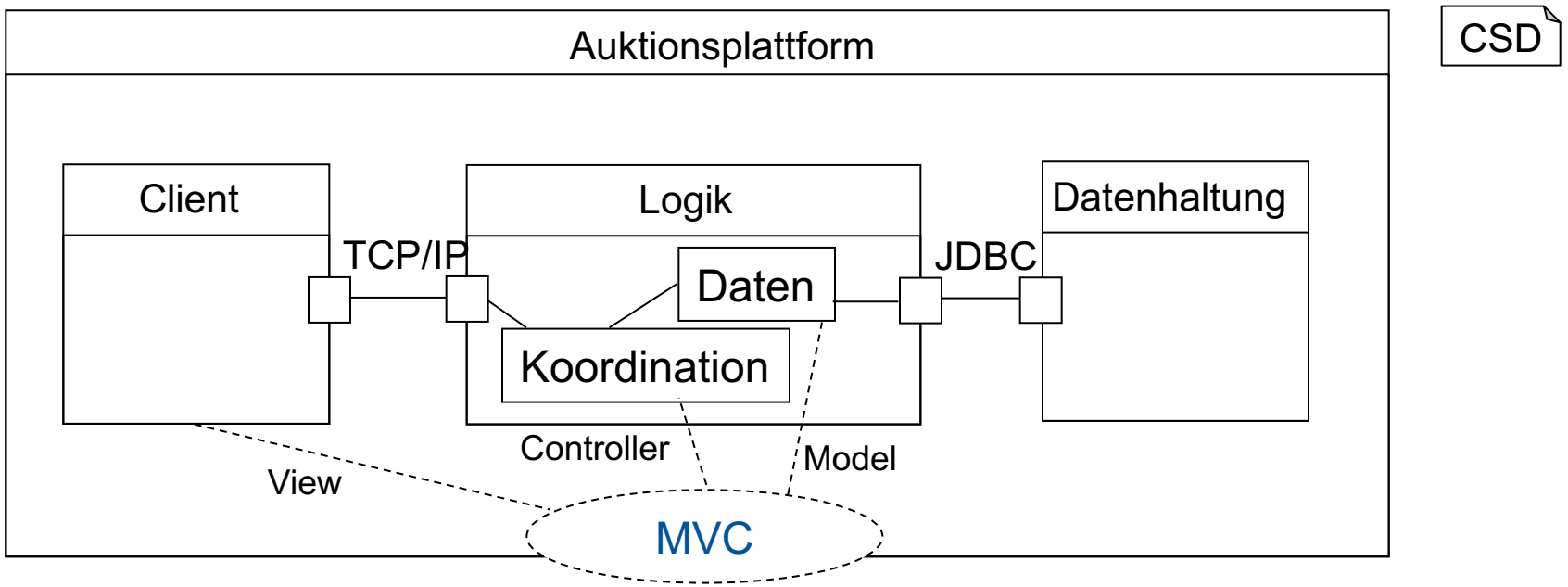
Model-View-Controller: Ablauf eines Datenupdates (Observer-Variante)



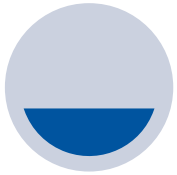
Beispiel: Auktionsplattform

- Anbieter
  - sehen alle Gebote & können eigene Gebote abgeben

- Koordination ist nötig, damit alle Anbieter die aktuellen Gebote sehen → Model-View-Controller-Muster



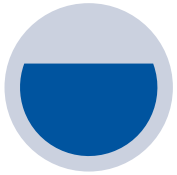
Was haben wir gelernt?



Architekturmuster

… helfen bei der Strukturierung von Anwendungen

… wirken eher theoretisch und trocken – solange man sie noch nicht selbst angewandt hat – sind aber hoch praxisrelevant



Behandelte Muster

Schichten (Layers)

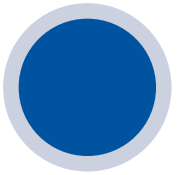
Pipes and Filters

Blackboard

Proxy

Model-View-Controller (mit Observer)

… es gibt noch viele weitere



Praxis

Entwickler kommunizieren in Mustern

Muster werden in der Praxis häufig angewandt & sind hilfreich

Vorlesung Softwaretechnik

8. Implementierung

Prof. Bernhard Rumpe
Software Engineering
RWTH Aachen

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

SE Software Engineering | RWTH AACHEN UNIVERSITY

Warum, was, wie und wozu betrachten wir die Implementierung

Warum?

- Zur Implementierung gehört mehr als nur Code zu schreiben
- Praxisrelevant

Was?

- Ergebnisse der Implementierung
- Sprachparadigmen
- Guter Programmierstil
- Datenstrukturen

Wie?

- Unterschiedliche Paradigmen kennenlernen
- Tipps für guten Code-Style
- Ausnutzung vorhandener Datenstrukturen

Wozu?

- Einschätzen was man für welches Anwendungsgebiet oder welche Aufgabe verwenden kann

Definition: Implementierung

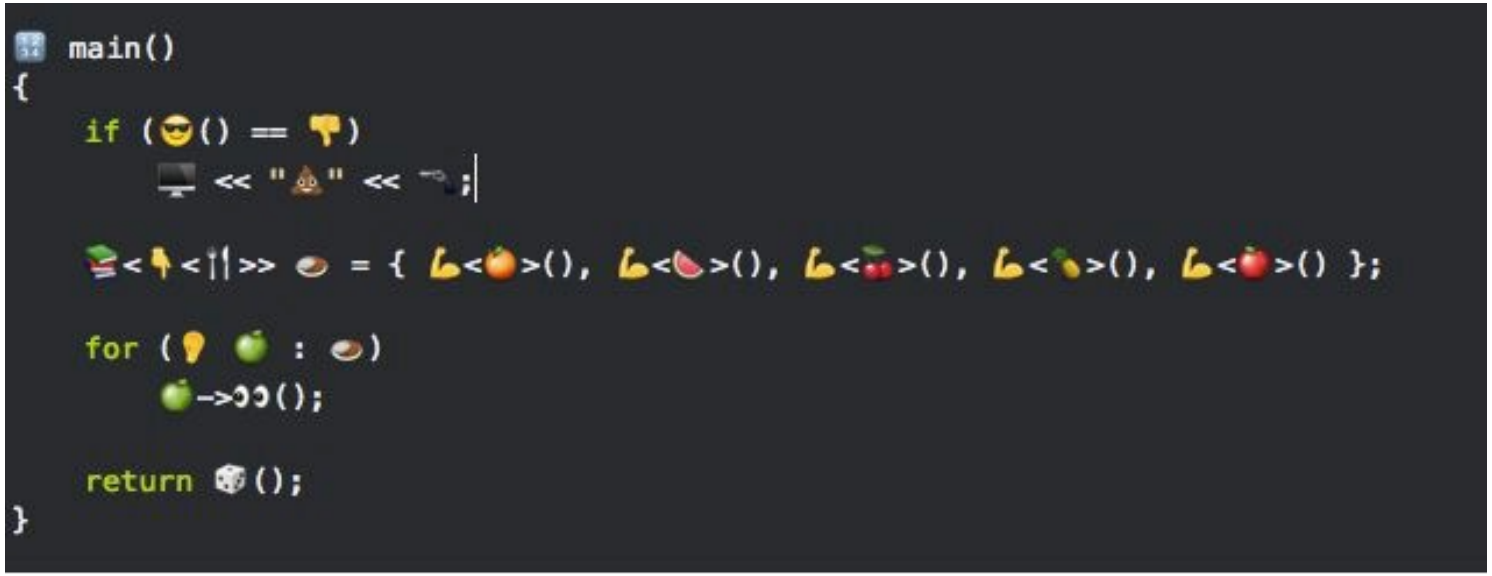
Implementierung ist die Menge aller Programmier-Aktivitäten.

- Die Implementierung geht von einer *System-Architektur* und detaillierten *Spezifikation der Funktionalität* aus.
- Ihre Ergebnisse sind:
  - Quellprogramm einschließlich Kommentierung
  - Lauffähiges System
  - Testplanung und Testdokumentation
  - Beschreibung des Kompiliervorganges (gradle, make, notfalls maven)
- Die Implementierung ist stark mit den Test-Aktivitäten verzahnt, um die Qualität des Systems sicher zu stellen.
- Varianten der „Implementierung“:
  - Legacy-System ist zu *erweitern* oder *anzupassen*
  - Agile Development: direkt von den Requirements zum Code+intensiven Tests;
    auf *Design und Detailspezifikation* wird dabei u.U. *verzichtet*

Einige Zahlen zum Thema Implementierung

- 1 Zeile Code (Lines of Code, LOC)
  nach COCOMO (Constructive Cost Mode, Kosten- bzw. Aufwandsschätzung)
  bei mittlerer Komplexität ca. 16 $ teuer
- 1 Personen Jahr = ca. 3300 LOC
  - (ca. 15 LOC pro Arbeitstag)
- Firefox hat 7,6 Mio Lines of Code,
  - 128 Mio $ Gesamtkosten
  - Davon 280.000 Zeilen alleine das Build Script
- Unser Energie Navigator (synavision.de) hat 650.000 LOC
  - Wert: ca. 10 Mio $
- Unser MaCoCo hat (Stand 2019)
  - 41.000 + 38.000 + 4.000 = 83.000 LOC handgeschriebenen Code, hwc (Java, TS, HTML)
  - 189.000 + 79.000 + 13.000 = 281.000 LOC generiert
  - Wert: ca. 1,3 Mio $ für hwc, gesamt 5,8 Mio $?

Coole Sprache ey?



C++

```
namespace 🔵 = std;
using 🔢 = int;
using 💀 = void;
using 🕐 = time_t;
using 👌 = bool;
#define 👂 auto
#define 👀 enum
#define 👎 false
#define 👍 true
#define 🍎 "evil"
#define 💪 🔵::make_shared
#define 🍸 virtual
#define 🖥️ 🔵::cout
#define 🦾 🔵::endl
template<class 🔮>
using 📚 = 🔵::vector<🔮>;
template<class 🔮>
using 👇 = 🔵::shared_ptr<🔮>;

👀 🐒 { 🙈, 🙉, 🙊, 🐵 };
🔢 🎲() { return 🔵::rand(); }
👌 😎() { return 👎; }

struct 🍴 { 🍸 💀 👀() = 0; };
struct 🍎 : 🍴 { 🍸 💀 👀() { 🖥️ << "🍎" << 🦾; }; };
struct 🍉 : 🍴 { 🍸 💀 👀() { 🖥️ << "🍉" << 🦾; }; };
struct 🍒 : 🍴 { 🍸 💀 👀() { 🖥️ << "🍉" << 🦾; }; };
struct 🍓 : 🍴 { 🍸 💀 👀() { 🖥️ << "🍓" << 🦾; }; };
struct 🍐 : 🍴 { 🍸 💀 👀() { 🖥️ << "🍐" << 🦾; }; };
struct 🍎 : 🍴 { 🍸 💀 👀() { 🖥️ << "🍎" << 🦾; }; };

🔢 main()
{
    if (😎() == 👎)
        🖥️ << "💩" << 🦾;

    📚<👇<🍴>> 🍽️ = { 💪<🍎>(), 💪<🍉>(), 💪<🍒>(), 💪<🍐>(), 💪<🍎>() };

    for (👂 🍏 : 🍽️)
        🍏->👀();

    return 🎲();
}
```

Softwaretechnik

8. Implementierung
8.1. Auswahl der Implementierungssprache

Prof. Bernhard Rumpe
Software Engineering
RWTH Aachen

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

SE Software Engineering | RWTH AACHEN UNIVERSITY

Objektorientierte Programmierung

- Beispiel: Java (C++, Oberon, Modula-3, Smalltalk, C#, ...)
- Die Merkmale der OO:
  - Objekte (Klassen) zur Kapslung von Daten und Funktionen
  - Objekte dynamisch erzeugen: Objektidentität
  - Vererbung (in ihren verschiedenen Ausprägungen)
- OO Sprachen subsumieren prozedurale Programmierung
- Es erfordert aber eine OO-spezifische Vorgehensweise, um Vorteile der OO zu nutzen:
  - Vererbung zur Anpassung und zur Verbesserung der Wiederverwendung
  - „Gutes Design“, um Wartbarkeit und Erweiterbarkeit zu stützen
  - Codingstandards für Lesbarkeit

- Fazit:
  - OO ist heute das Mittel der Wahl für große Projekte, effizienter geht es aber öfter auch mit Spezialsprachen

Prozedurale Programmiersprachen

- Beispiele: Modula-2, (Pascal, C, Fortran)
- Ideen zum Modul-Konzept teilweise vorhanden
- Komfortable Definition von Datenstrukturen im Speicher
- Seiteneffekte
- Trennung von Datenstruktur und Funktionen entspricht „mathematischer Tradition“, aber macht Wartbarkeit schwieriger

- Fazit:
  - Für kleinere und mittlere Systeme geeignet
  - Prozedurale Programmierung ist in der objektorientierten Programmierung subsumiert und wird deshalb kaum mehr in Reinform verwendet

  - Leider: eingebettete Systeme immer noch in C
  - Wissenschaftliches Rechnen: immer noch Fortran

Funktionale Programmiersprachen

- Beispiele: Gofer, Haskell, ML
- Seiteneffektfrei und dadurch leicht verstehbar
- Oft sehr mächtiges Typsystem
- Patternmatching auf Argumenten

- Kompakte Formulierung
- Fazit:
  - Effektive Programmierung aber langsamere Ausführungszeiten.
  - Geeignet für schnelle Entwicklung,
  - skaliert nicht für große Programmsysteme
  - Schwierigkeiten mit interaktiven Systemen (z.B. GUI, Persistenz)

- Effiziente Definition von Datenstrukturen und Funktionen

```
01 data Tree = Leaf(Int) | Node(Tree,Int,Tree)
```

- Funktionen höherer Ordnung (Funktionen über Funktionen)

```
11 twice f x = f(f(x))
```

Flavor of Haskell

Beispiel in Java:

- Berechnung durch Variablen-Zuweisung

Java

```
01 total = 0;
02 for (i = 1; i ≤ 10; ++i)
03    total = total+i;
```

Beispiel in Haskell:

- Berechnung durch Funktionsaufrufe:

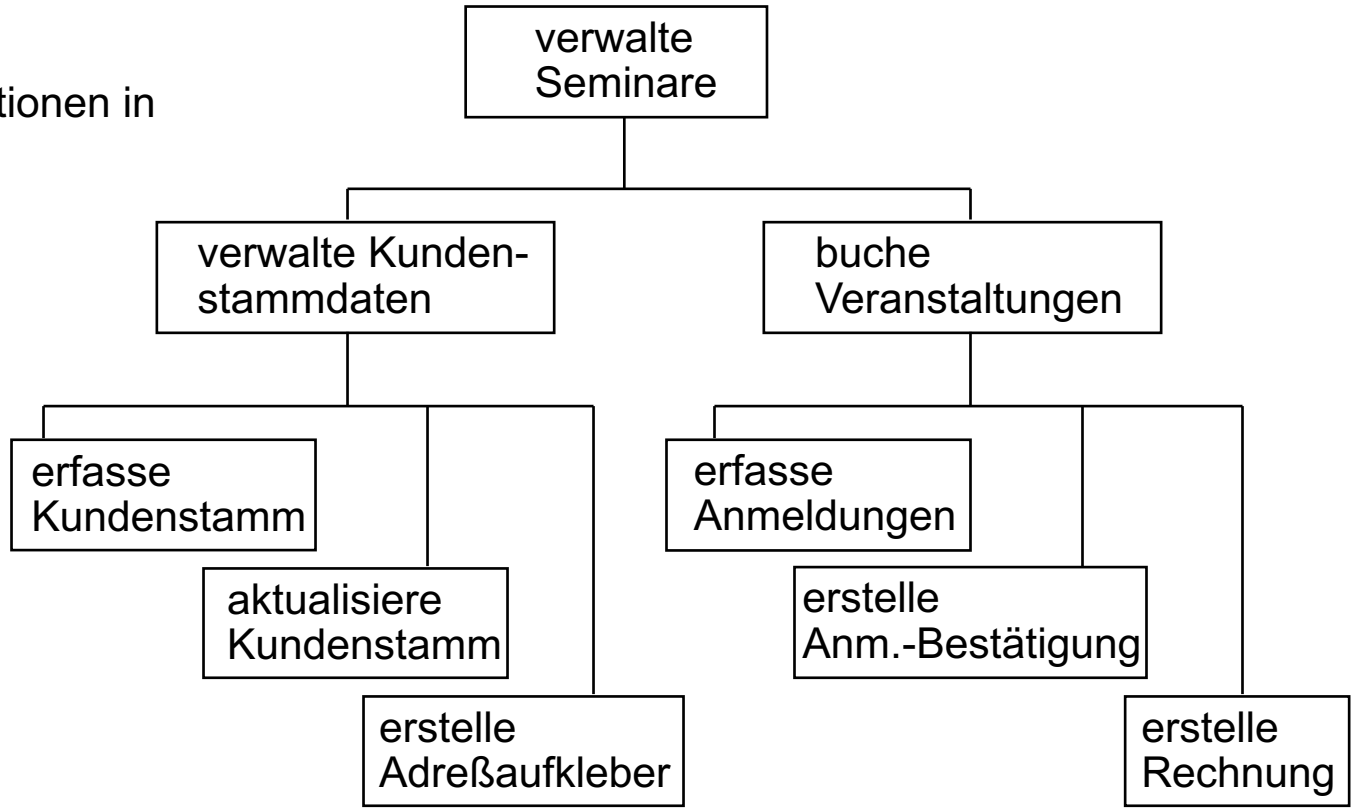
Haskell

```
11 foldl' (+) 0 [1..10]
```

Paradigma der Funktionsmodellierung

- Funktionsmodellierung
  - betrachtet ein System als Funktion
  - beschreibt die hierarchische Zerlegung von Funktionen in Teilfunktionen

- Beispiel: Funktionsbaum



Funktionale Programmierung (FP)

- Funktionale Programmierung mit Haskell hat interessante Charakteristika:
- Haskell ist funktional / deskriptiv:
  - Keine Seiteneffekte
  - Keine „Zustände“
  - Nur Werte und Funktionen
- Verzögerte Auswertung (nur in manchen Sprachen, z.B. Haskell)
  - Funktionsaufruf wird erst ausgewertet, wenn notwendig
- Mächtiges Typsystem
  - Polymorph: einmal definiert, auf allen Typen verfügbar
- Automatische Speicherverwaltung

Funktionen Höherer Ordnung: Lambdas

- Haskell erlaubt Funktionen Höherer Ordnung:
- Funktionen sind selbst Werte und können als Argumente eingesetzt werden,
- Beispiele für (anonyme) Funktionen:
  - `(+)` ist die Addition (als Wert)
  - `\x y -> x+y` addiert eine 1
  - `\f y -> 2*f(y)+5` Argument ist selbst Funktion f
- Anwendung die Summe von 0 bis 10:

Haskell

```
01 foldl' (+) 0 [1..10]
```

- Java bietet nun auch Lambdas:
-
- Beispiele für (anonyme) Funktionen:
  - `Integer::sum` ist die Addition (als Wert)
  - `(x y)-> x+y` addiert eine 1

Java

```
11 List<Integer> x = Arrays.asList(new
12         Integer[]{1,2,3,4,5,6,7,8,9,10});
13
14 x.stream().reduce(0, Integer::sum)
```

Lambda-Ausdrücke in Java

- Nimmt Parameter und berechnet einen Rückgabewert
  - Aggregierende Berechnungen (z.B. Summe)
  - Mappings (überall +1)
  - Prädikate, Filter (boolsche Rückgabe)
- Wie Namenlose Methoden
  - an Ort und Stelle implementiert
  - limitierte Funktionalität, direkte Rückgabe
  - keine Variablen, Zuweisungen
- Oft auf Collections (siehe später) angewandt
  - **`forEach`**, **`map`**, **`removeIf`**, etc.
  - besonders stark mit **`Streams`**
- See also: java.util.function

Java

```
01 List<Integer> x = Arrays.asList(
02                   new Integer[]{1,2,3});
03
04
05 // prints: 1 2 3
06 x.forEach(n -> System.out.println(n));
07
08 // value: false
09 bool t = x.anyMatch(n -> n > 4)
10
11 // ergibt r1=r2=6
12 int r1 = x.stream().reduce(0, (a,b)->a+b);
13 int r2 = x.stream().reduce(0, Integer::sum);
14
15 // Interface mit eigener Methode
16 interface Add { int run(int a, int b); }
17 Add myFun = (a, b) -> a + b;
18 r = l.stream().reduce(0, myFun::run);
```

Skriptsprachen

- Beispiele: Python, Ruby, Perl, PHP, JavaScript, Unix Shell, …

- Interpretiert statt kompiliert
- Meist dynamische Typisierung ohne Deklarationszwang von Variablen

- Auf schnelle Entwicklung mit geringem Aufwand und wenig Restriktionen ausgelegt

- Oft auf eine Domäne spezialisiert, z.B. Web oder Textverarbeitung

- Einsetzbar z.B. für Automatisierung in der Entwicklung, Konfiguration, Deployment

Python

```
01 # calculating sqrt
02 a = float(input("value a: "))
03 x = float(input("start x1: "))
04 for i in range(1,6):
05   x = 0.5*(x+a/x)
06   print ('   ',i,'    ',x)
```

```
11 mixedList = ["dog", 1, 3.5];
12
13 for i in range(10):
14     print(i)
15
16 def foo(value)
17   return int(value[2] * 14)
```

Spezialsprachen

- Logikprogramme: Prolog
  - Logische Aussagen in Hornklauselform als Programm
- Visuelle Programmierung
  - a) Komposition des Programms aus Bausteinen
  - b) Modellierung z.B. mit ausführbaren Statecharts
- Parallele Programmiersprachen
  - für massiv verteilte Systeme
- Maschinennahe
  - Assembler
- Weitere Spezialsprachen: HTML, JSP, XML, SQL, ...

Prolog

Clauses

```
01 likes(mary,food).
02 likes(mary,wine).
03 likes(john,wine).
04 likes(john,mary).
```

Questions

```
11 | ?- likes(mary,food).
12 yes.
13 | ?- likes(john,wine).
14 yes.
15 | ?- likes(john,food).
16 no.
```

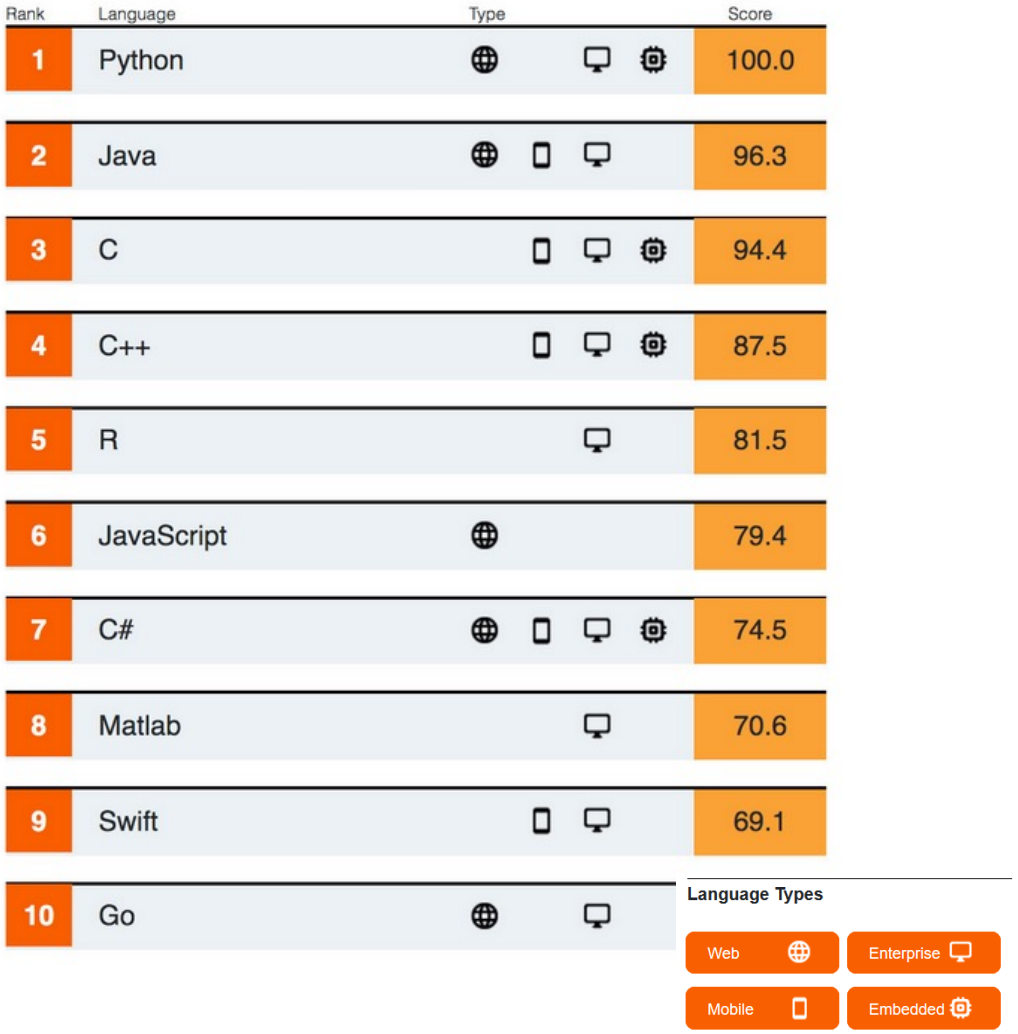
Kategorisierung für Programmiersprachen

- Programmierparadigmen
  - Programmentwicklung konzentriert sich auf die Beschreibung von Algorithmen, Objekten, funktionale Zusammenhänge oder Logik des Problems
  - OO | deskriptive (funktionale, logische) | imperative/algorithmische (maschinennah, höhere/prozedurale)
- Anwendungsgebieten
  - z.B. Naturwissenschaftlich, kommerziell, KI, Systemsoftware (Compiler, Betriebssysteme), Echtzeit, Datenbanken
  - Aber: Viele Sprachen sind universell anwendbar
- Historie
  - Welche Sprachen haben sich aus welchen entwickelt
  - z.B. Algol -> … -> Pascal -> Ada -> Ada95; C++ und Modula -> Java
- Programmiersprachengenerationen
  - 1. (Maschinensprachen), 2. (Assemblersprachen), 3. (höher algorithmisch und OO), 4. (visuell, Tabellenkalkulation, Datenbanksprachen), (5. deskriptive Sprachen?/KI?)
- Verbreitungsgrad und Nutzungsgrad
  - Messung?

Verbreitungsgrad von Programmiersprachen

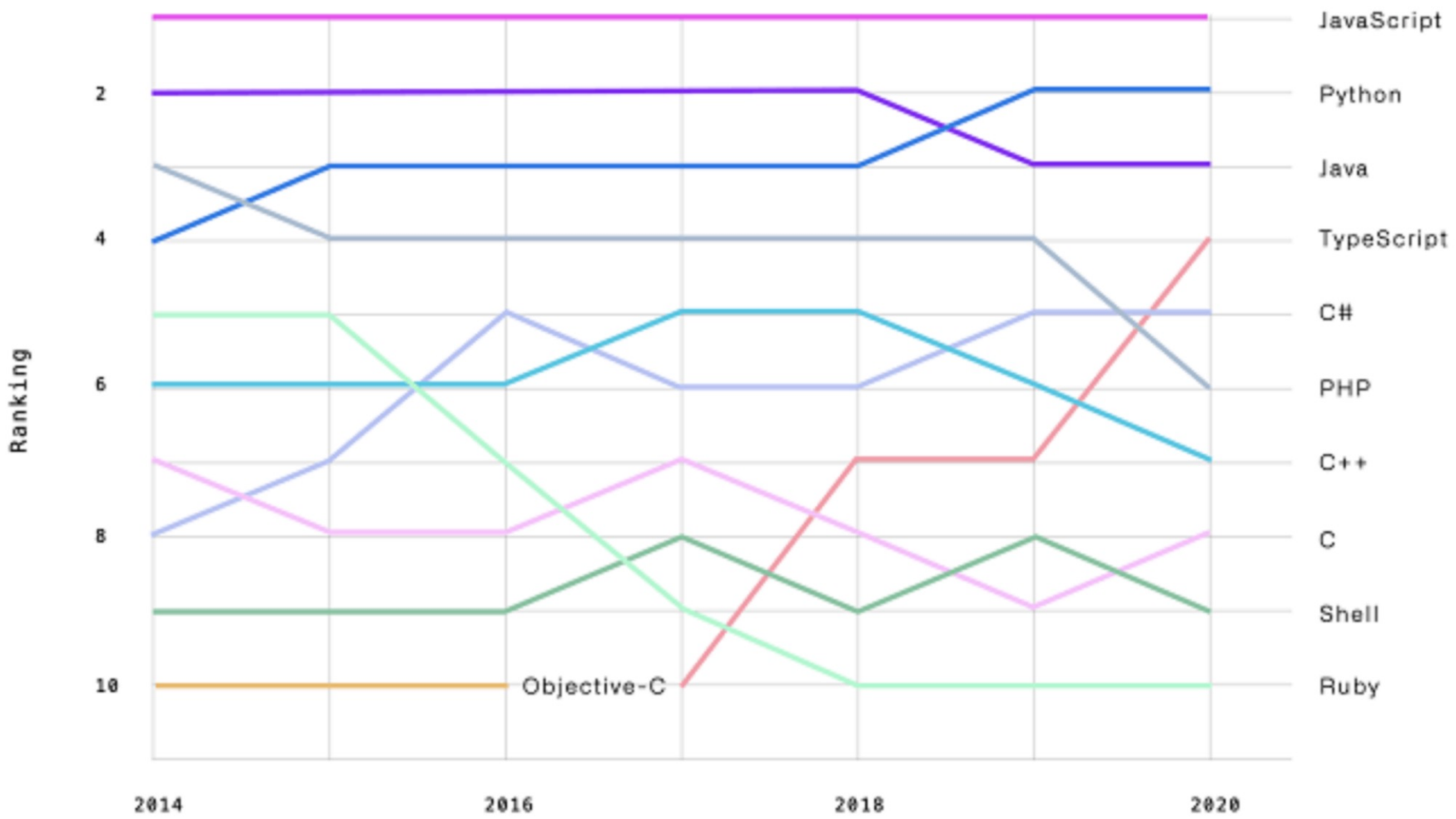
- Welche ist die am stärksten verbreitete Programmiersprache?
  - Antwort: It depends…
  - Anwendungsgebiet, Projektgröße, Unternehmen, …

- Achtung:
  - Aussagekraft solcher Statistiken immer hinterfragen
    - Welche Daten werden für das Ranking verwendet?
      - Hier: Google Suche/Trends, IEEE Publikationen, IEEE Job portal, gitHub (open source), StackOverflow, Twitter
    - Welchen Kategorien werden Sprachen zugeordnet?
    - Welche Größe haben die dahinterliegenden Projekte/ LOC?
    - Wie werden Bibliotheken mit einbezogen?
    - Welche „Sprachen“ werden aufgenommen (z.B. Arduino)?



https://spectrum.ieee.org/computing/software/the-top-programming-languages-2019

Verbreitungsgrad von Programmiersprachen: GitHub-Ranking-2020



Fragen für Ihre Zukunft

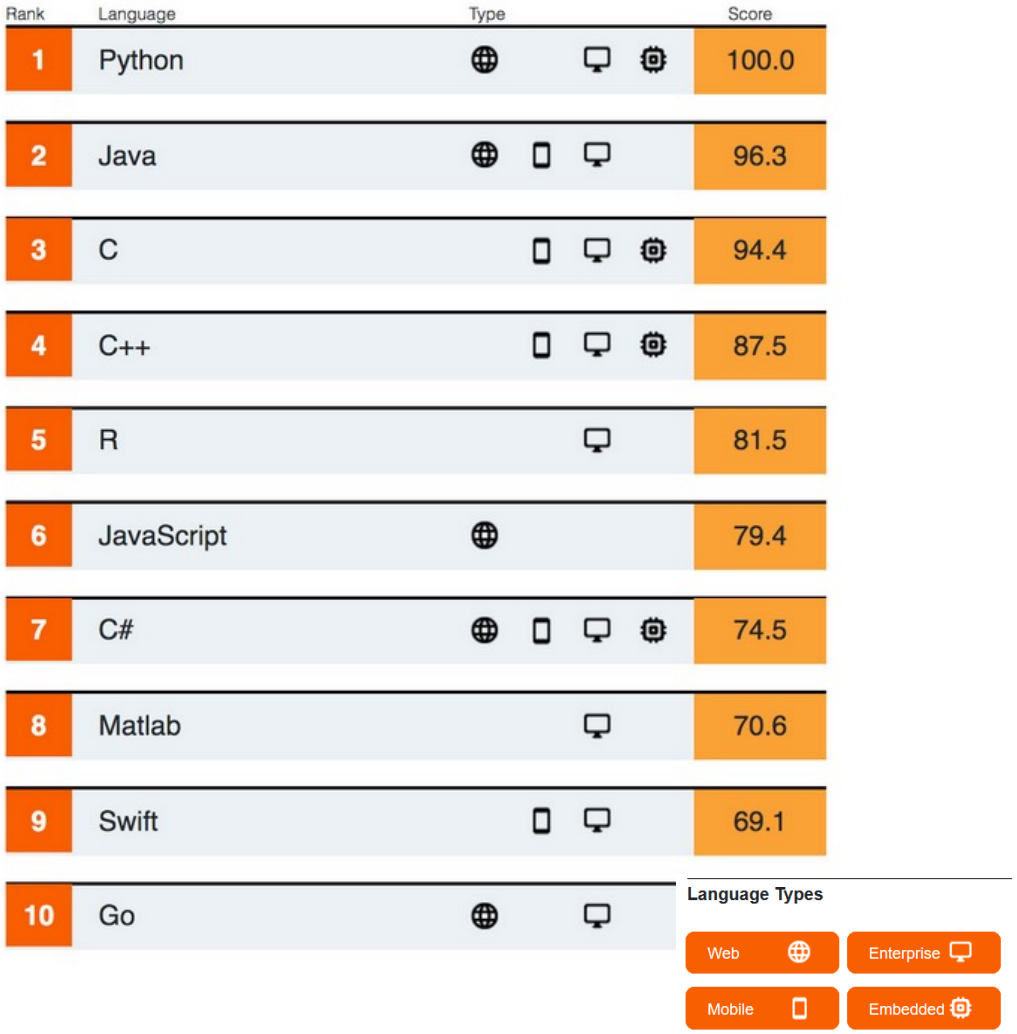
- Welche Programmiersprache…
  - … sollte man beherrschen?
  - ... ist weit verbreitet?
  - ... ist die beste?
  - ... ist modern?

- Sollte man…
  - … möglichst viele Sprachen beherrschen?
  - … die seltensten Sprachen beherrschen?
  - … die neuesten Sprachen beherrschen?

- Einzig sinnvolle Lösung:
  - Man muss in der Lage sein, sich zielgerichtet in eine neue Sprache einzuarbeiten.
  - Man sollte Vertreter wichtiger Sprachklassen aktiv beherrschen.
  - Alles andere ist abhängig davon, was Sie in Zukunft machen wollen!

Weitere Auswahlkriterien für Sprachen

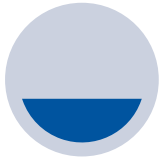
- Welche technische Umgebung ist gegeben?
  - Legacy-System? Gibt es eine Vorgabe des Unternehmens?
- Human Factor: Welche Kenntnisse/Vorlieben besitzen die ProgrammiererInnen?
- Welche Bibliotheksfunktionen/APIs/Frameworks werden benötigt?
- Wie gut ist die Werkzeugunterstützung?
  - Compiler, Debugger, IDE, ...



Weitere Beobachtungen zu Sprachen

- Fehlende Spracheigenschaften werden durch standardisierte Klassenbibliotheken abgedeckt:
  - *I/O* wurde in C durch *Bibliotheksfunktionen* definiert
  - *Threads/Nebenläufigkeit* wird in Java über die *Bibliothek* realisiert
  - *Kommunikation, Datenspeicherung* wird nicht über Sprachprimitive, sondern über Bibliotheksfunktionen angeboten (Middleware)
  - *Security* (z.B. Java Sandbox / Frameworks oder Verschlüsselung)
  - *Komponentenkonzepte* via Bibliothek und Programmiermethodik (z.B. EJBs)
- Kooperationsfähigkeiten der Sprachen erlauben die Anwendung der jeweils besten Sprache:
  - Corba, .NET, Embedded SQL (z.B. in Java)
  - Web: z.B. JavaScript + HTML + php
  - Import über API‘s (Python nutzt Java / Fortran / …)
  - Generatoren wie Antlr/Yacc: Grammatik -> Parser
- Werkzeuge für das Management der Implementierung und Wartung erlauben *flexiblere* Entwicklungsprozesse:
  - Dokumentation, Testen, Versionsverwaltung, Evolution, Generierung, Installation, ...

Was haben wir gelernt?



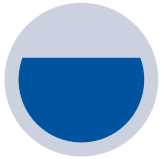
Implementierung umfasst

a) Quellprogramm einschließlich Kommentierung

b) Lauffähiges System

c) Testplanung und Testdokumentation

und d) Beschreibung des Kompiliervorganges



Programmier-sprachen

Objektorientierte

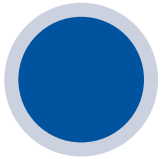
Prozedurale

Funktionale

Skriptsprachen

Spezialsprachen

Auswahl hat verschiedenste Kriterien



Auswahlkritierien

Programmier-paradigmen müssen zur Aufgabe passen

Anwendungsgebiete

Technische Umgebung (Legacy-Systeme, Vorgaben)

Benötigte Bibliotheken, APIs, Frameworks

Vorlieben des Teams

Werkzeugunterstützung

Softwaretechnik

8. Implementierung
8.3. Programmiersprachen an der RWTH

Prof. Bernhard Rumpe
Software Engineering
RWTH Aachen

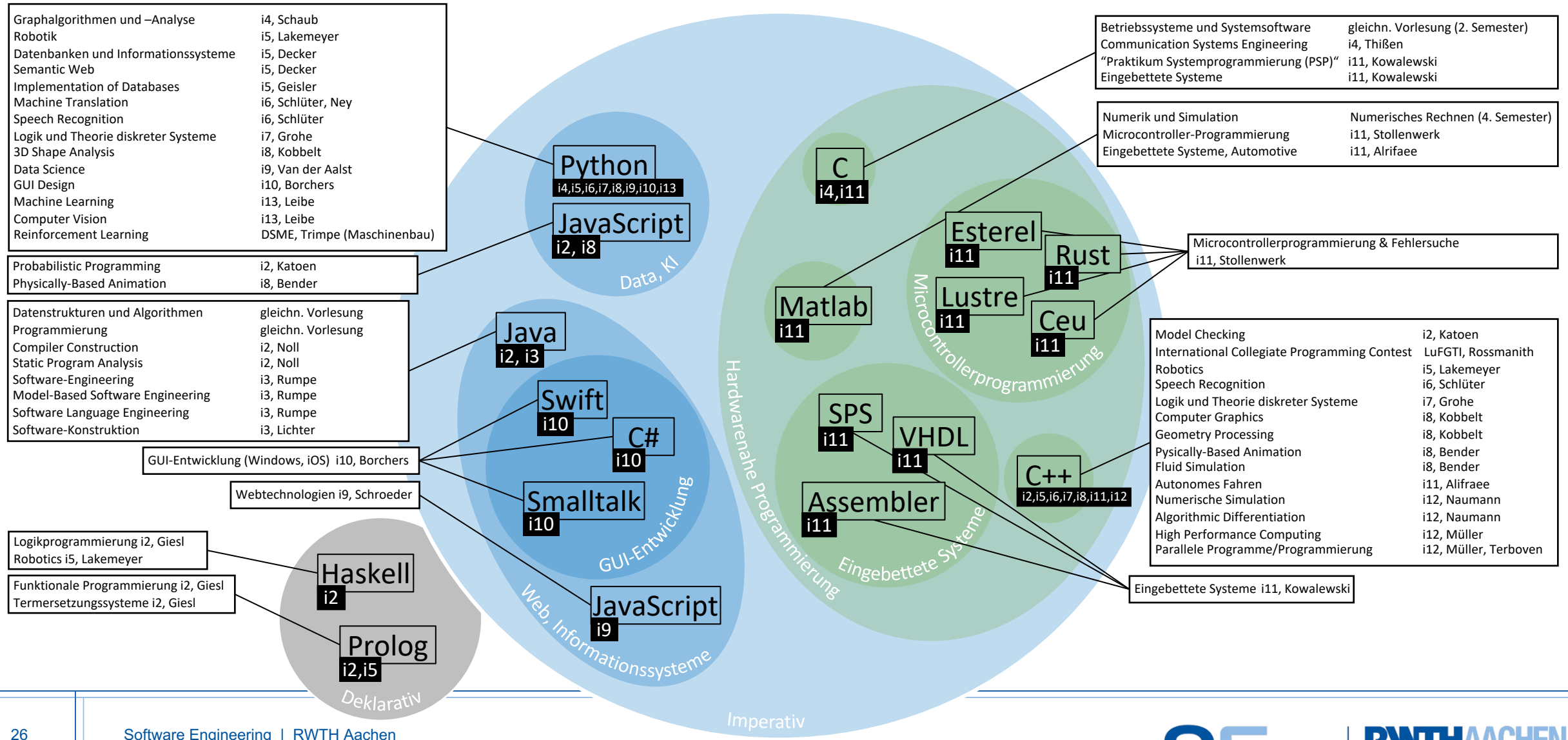
http://www.se-rwth.de/
@SE_RWTH

Literatur:

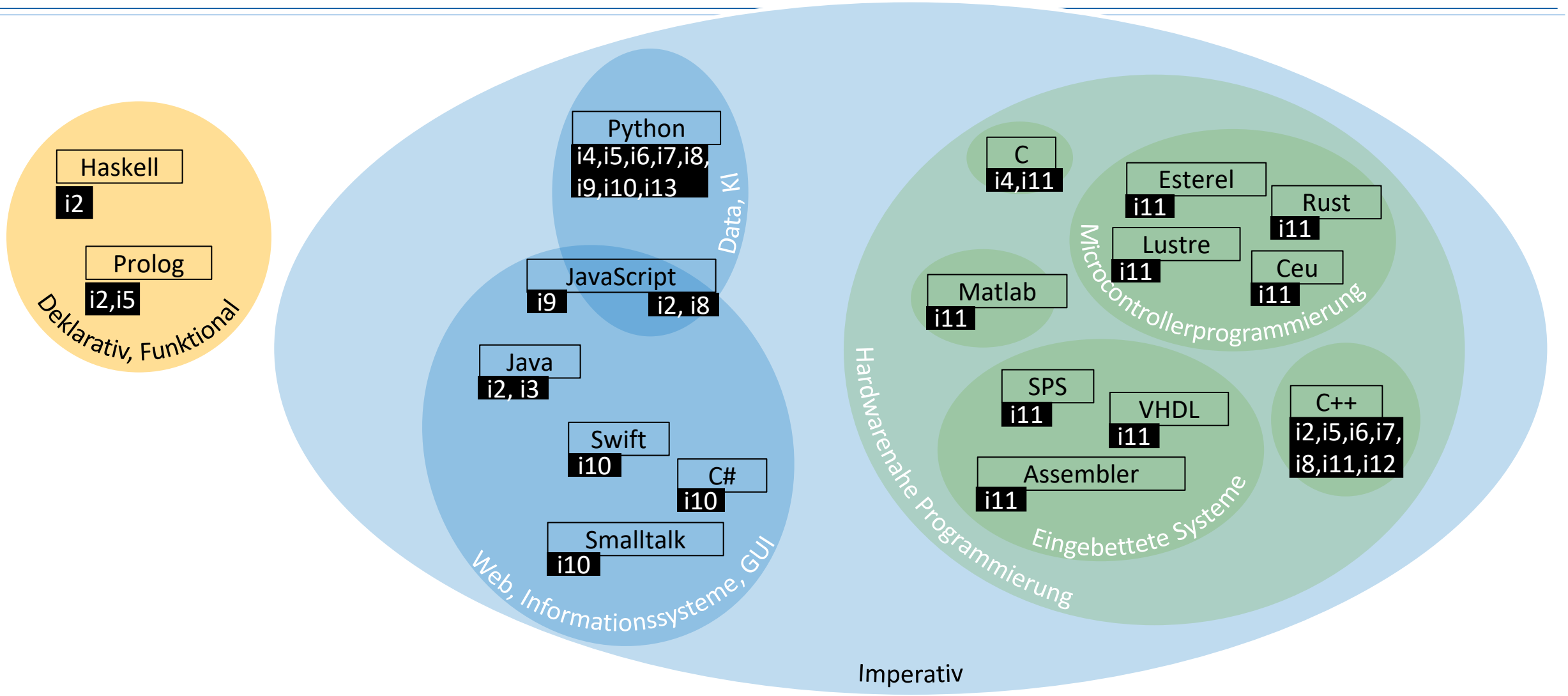
- Vermeulen et al.: The Elements of Java Style
- Scott Ambler: http://www.ambysoft.com/javaCodingStandards.html
- Balzert Bd. 1 LE 33

SE Software Engineering | RWTH AACHEN UNIVERSITY

Programmiersprachen in der Informatik der RWTH

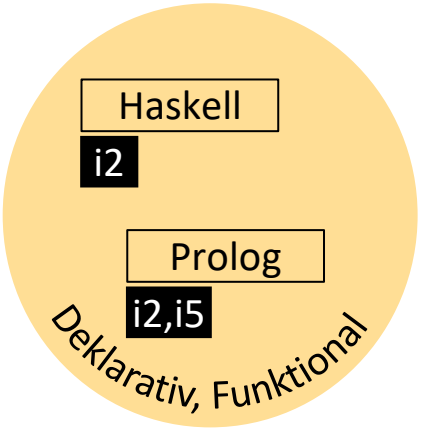


Programmiersprachen in der Informatik der RWTH



Programmiersprachen in der Informatik der RWTH Abstrakt, Logisch, Funktional

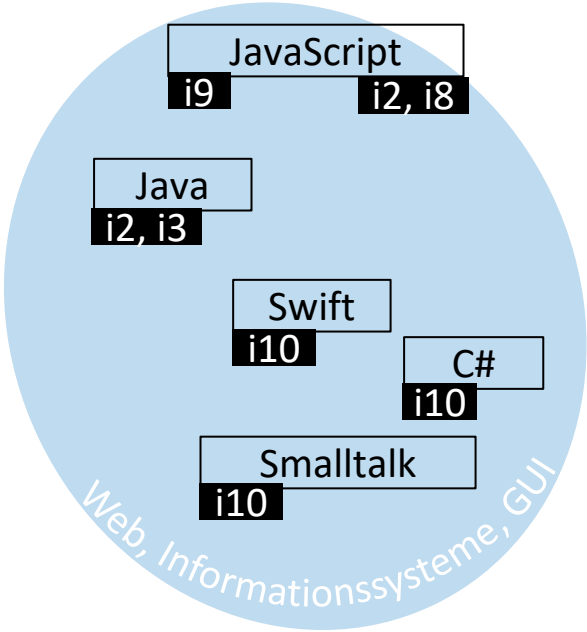
- Haskell
  - Funktionale Programmierung — i2, Prof. Giesl
  - Termersetzungssysteme — i2, Prof. Giesl



- Prolog
  - Logikprogrammierung — i2, Prof. Giesl
  - Robotics — i5, Prof. Lakemeyer

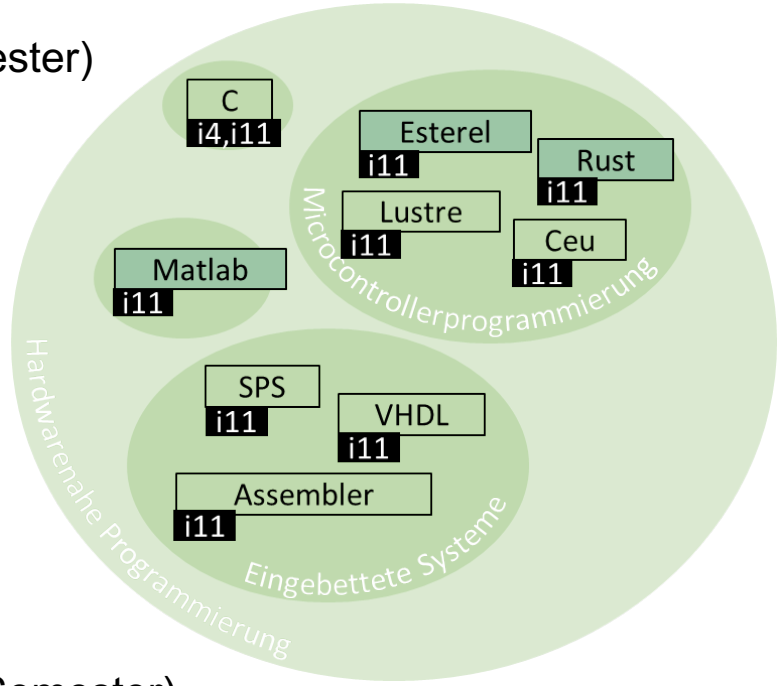
Programmiersprachen in der Informatik der RWTH Unternehmen, Web, Informationssysteme

- Java
  - Datenstrukturen und Algorithmen — gleichn. Vorlesung
  - Programmierung — gleichn. Vorlesung
  - Compiler Construction — i2, Prof. Noll
  - Static Program Analysis — i2, Prof. Noll
  - Software-Engineering — i3, Prof. Rumpe
  - Model-Based Software Engineering — i3, Prof. Rumpe
  - Software Language Engineering — i3, Prof. Rumpe
  - Software-Konstruktion — i3, Prof. Lichter
- C#, Swift, Smalltalk
  - GUI-Entwicklung (Windows, iOS) — i10, Prof. Borchers
- JavaScript
  - Webtechnologien — i9, Prof. Schroeder



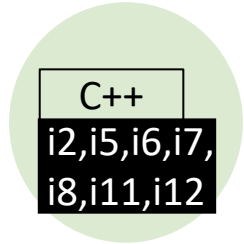
Programmiersprachen in der Informatik der RWTH Eingebettet, Hardwarenah

- C
  - Betriebssysteme und Systemsoftware — gleichn. Vorlesung (2. Semester)
  - Communication Systems Engineering — i4, Dr. Thißen
  - “Praktikum Systemprogrammierung (PSP)“ — i11, Prof. Kowalewski
  - Eingebettete Systeme — i11, Prof. Kowalewski
- Assembler, VHDL, SPS-Programmierung, …
  - Eingebettete Systeme — i11, Prof. Kowalewski
- Rust, Esterel, Lustre, Ceu
  - Microcontrollerprogrammierung & Fehlersuche — i11, Prof. Stollenwerk
- Matlab
  - Numerik und Simulation — Numerisches Rechnen (4. Semester)
  - Microcontroller-Programmierung — i11, Dr. Stollenwerk
  - Eingebettete Systeme, Automotive — i11, Dr. Alrifaee



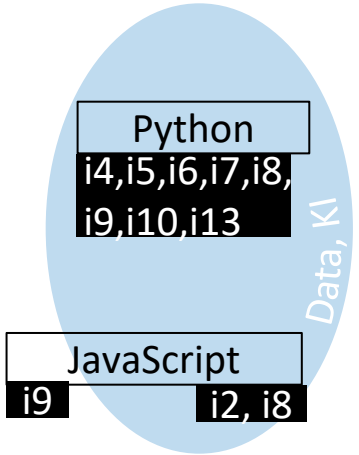
Programmiersprachen in der Informatik der RWTH Numbers, Eingebettet

- C++
  - Model Checking — i2, Prof. Katoen
  - International Collegiate Programming Contest — LuFGTI, Prof. Rossmanith
  - Robotics — i5, Prof. Lakemeyer
  - Speech Recognition — i6, Prof. Schlüter
  - Computer Graphics — i8, Prof. Kobbelt
  - Geometry Processing — i8, Prof. Kobbelt
  - Pysically-Based Animation — i8, Prof. Bender
  - Fluid Simulation — i8, Prof. Bender
  - Autonomes Fahren — i11, Dr. Alifraee
  - Numerische Simulation — i12, Prof. Naumann
  - Algorithmic Differentiation — i12, Prof. Naumann
  - High Performance Computing — i12, Prof. Müller
  - Parallele Programme/Programmierung — i12, Prof. Müller, Dr. Terboven



Programmiersprachen in der Informatik der RWTH — Data, KI

- Python
  - Data Science — i9, Prof. Van der Aalst
  - Graphalgorithmen und –Analyse — i4, Prof. Schaub
  - Robotik — i5, Prof. Lakemeyer
  - Datenbanken und Informationssysteme — i5, Prof. Decker
  - Semantic Web — i5, Prof. Decker
  - Implementation of Databases — i5, Prof. Geisler
  - Machine Translation — i6, Dr. Schlüter, Prof. Ney
  - Speech Recognition — i6, Dr. Schlüter
  - 3D Shape Analysis — i8, Prof. Kobbelt
  - GUI Design — i10, Prof. Borchers
  - Machine Learning — i13, Prof. Leibe
  - Computer Vision — i13, Prof. Leibe
  - Reinforcement Learning — DSME, Prof. Trimpe (Maschinenbau)
- JavaScript
  - Probabilistic Programming — i2, Prof. Katoen
  - Physically-Based Animation — i8, Prof. Bender



Softwaretechnik

8. Implementierung
8.3. Codingstandards: Stilfragen der Codierung

Prof. Bernhard Rumpe
Software Engineering
RWTH Aachen

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

+ Selbststudium: se-rwth.de/swt_material/

Literatur:

- Vermeulen et al.: The Elements of Java Style
- Scott Ambler: http://www.ambysoft.com/javaCodingStandards.html
- Balzert Bd. 1 LE 33

Programmierstil

- Qualität von Programmcode:
  - Funktionalität | Verständlichkeit <-> Wartbarkeit | Effizienz | Eleganz
- Im Ablauf identische Programme können in der Verständlichkeit erheblich differieren.
  - Programmierkonventionen, z.B.:
    - Verwendete Konstrukte | Reihenfolgen | Klammerung
  - Bezeichnerwahl
  - Layout
    - Einrückung
- “Style Guide”: Standard-Konventionen (z.B. für Projekt, Firma)
  - Kann unterschiedlich sein je Programmiersprache, z.B.:
    Google C++ Style Guide: https://google.github.io/styleguide/cppguide.html
    Google Java Style Guide: https://google.github.io/styleguide/javaguide.html

Was tut dieses Java-Programm?

Java

```
1 public class Z
2 {public static void main(String[] args)
3 {double x = Console.readDouble("X:"); double z =
4 Console.readDouble("Z:") / 100; int l = Console.readInt("L:");
5 double y; for (y = z - 0.01; y <= z + 0.01; y += 0.00125)
6 {double p = x * y/12/(1 - (Math.pow(1/(1 + y/12),l*12)));
7 System.out.println(100*y+" : "+p);
8 }}}
```

Künstlich verschlechtertes! Beispiel aus: Horstmann/Cornell, Core Java Vol. I, Prentice-Hall 1997

Formatierungs-Richtlinien?

Java

```
01 public class Z {
02
03   public static void main(String[] args) {
04     double x;
05     double z;
06     int l;
07     x = Console.readDouble("X:");
08     z = Console.readDouble("Z:") / 100;
09     l = Console.readInt("L:");
10     double y;
11     for (y = z - 0.01; y <= z + 0.01; y += 0.00125){
12       double p = x * y /12 /
13         (1 - (Math.pow(1/(1 + y / 12), l * 12)));
14       System.out.println(100*y+" : "+p);
15     }
16   }
17
18 }
```

Hinweise zur Formatierung (a)

- Einheitliche Formatierung verwenden!
  - Werkzeuge (“pretty printer”, “beautifier”)
  - Konfiguration der IDE
- Gemäß Schachtelungstiefe einrücken
  - Genau festgelegte Anzahl von Leerzeichen (keine Tabulatoren! Definitiv keine Tabulatoren!)
  - Formatierungsprobleme bei zu tiefer Schachtelung deuten *oft* auf Strukturprobleme des Codes hin!
- Leerzeilen verwenden (einheitlich)
  - z.B. vor und nach Methoden
  - Aber: Zusammenhängender Code soll auf einem normalen Bildschirm darstellbar bleiben!

Java

```
01 public class Z {
02
03   public static void main(String[] args) {
04     double x;
05     double z;
06     int l;
07     x = Console.readDouble("X:");
08     z = Console.readDouble("Z:") / 100;
09     l = Console.readInt("L:");
10     double y;
11     for (y = z - 0.01; y <= z + 0.01; y += 0.00125){
12       double p = x * y /12 /
13         (1 - (Math.pow(1/(1 + y / 12), l * 12)));
14       System.out.println(100*y+" : "+p);
15     }
16   }
17
18 }
```

Hinweise zur Formatierung (b)

- Einheitliche Dateistruktur verwenden
  - z.B.: Je Klasse eine .java-Datei, je Package ein Verzeichnis
  - “package” - Statement immer als erste Zeile
  - zuerst die Methoden, dann die Attribute (oder umgekehrt, aber einheitlich!)
  - einheitliche Namen für Quellcode-Verzeichnisse, Verzeichnis für kompilierten Code, Build-Dateien usw.

Java

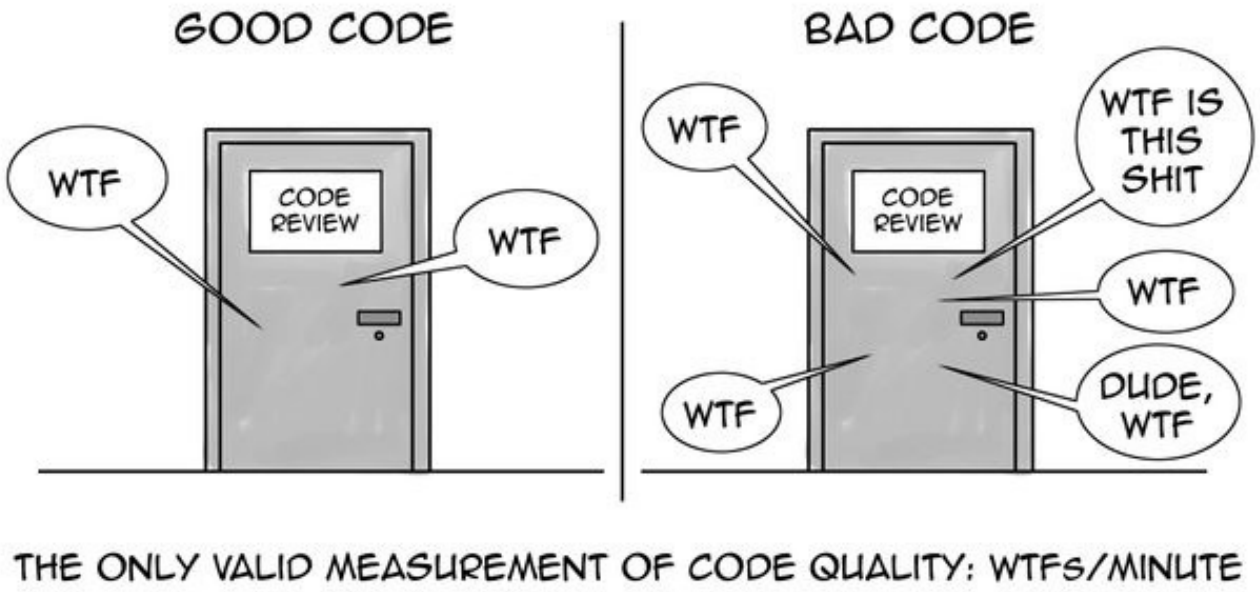
```
01 public class Z {
02
03   public static void main(String[] args) {
04     double x;
05     double z;
06     int l;
07     x = Console.readDouble("X:");
08     z = Console.readDouble("Z:") / 100;
09     l = Console.readInt("L:");
10     double y;
11     for (y = z - 0.01; y <= z + 0.01; y += 0.00125){
12       double p = x * y /12 /
13         (1 - (Math.pow(1/(1 + y / 12), l * 12)));
14       System.out.println(100*y+" : "+p);
15     }
16   }
17
18 }
```

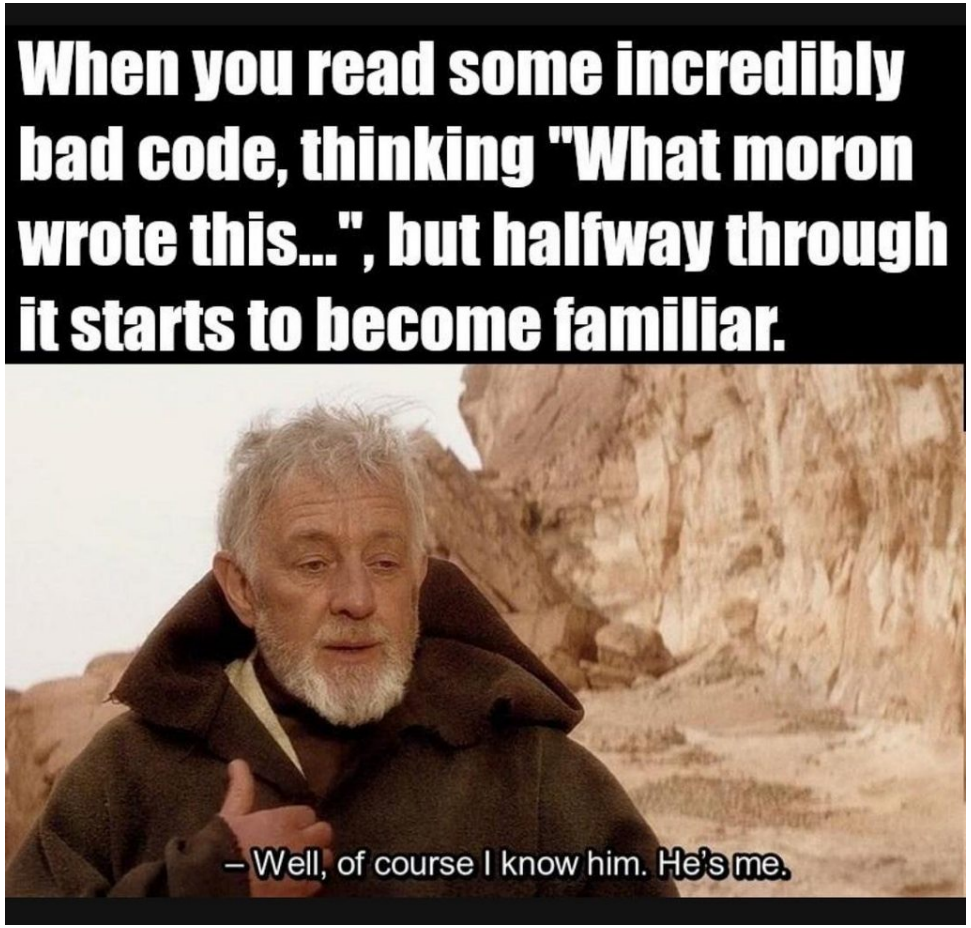
Einrückung

- SE-Konvention zu Einrückungstiefe:
  - 2 Zeichen normale Einrückung
  - 4 oder 8 Zeichen Einrückung zu Sonderzwecken

- Wichtig:
  - **Schreiben aus der Sicht des Lesers!**
    denn Code wird wesentlich(!) öfter gelesen als geschrieben.

  - Und: Leseaufwand bei schlechtem Code ist immens.





Pic Source: https://imgur.com/gallery/J1SUmYH

Weiteres Lernmaterial …

Manches lässt sich besser im **Selbststudium** lernen oder erarbeiten.

Deshalb gibt es nachfolgend Folien,
die in der Vorlesung nicht vorgestellt werden,
aber Inhalt der Vorlesung sind.
Sie eignen sich zum **Selbststudium**.

Hier finden sich diese Folien mit erklärenden Texte
und weiterem klausurrelevanten Material / Videos:

- https://se-rwth.de/swt_material/ bzw.
- https://se-rwth.github.io/swt_material/

Ggf. **Fragen** dazu bitte in der Vorlesung oder der Übung
möglichst bald stellen!

SE Software Engineering | RWTH AACHEN UNIVERSITY

Material für die Softwaretechnik Vorlesung

Material für die Softwaretechnik Vorlesung, Bernhard Rumpe

- Weil manche Inhalte besser in Ruhe (z.B zuhause) durchgearbeitet werden können, stehen hier einige Sätze kommentierter Folien aus der Vorlesung zur Verfügung.
- Diese werden in der Vorlesung nicht mehr vorgestellt, sondern bei jeweiligen Abschnitt als bekannt vorausgesetzt.

Beispiele zur Einrückung

Selbststudium!

- SE-Konvention für lange Methodenköpfe:

Java

```
01 void methode (int x, Object y, String z, Xyz v,
02               float p);  // konventionell
03
04
05 private static synchronized void etwasLang
06          (int x; Object y, String z, Xyz v,
07          float p) {
08          // Acht Zeichen Einrückung hier besser
09
10   x = y.doSomething(p); // Methodenrumpf nur zwei eingerückt
11
12   // …
13
14 }
```

Beispiele zur Einrückung

Selbststudium!

- SE-Konvention für die Fallunterscheidung:

```java
// nicht gut erfassbar
if ((bedingung1 && bedingung2 && bedingung3) ||
  bedingung4) {
  codeFürKomplexeBedingung(); …
```

Java

```java
// besser
if ( (bedingung1 && bedingung2 && bedingung3)
      || bedingung4) {
  codeFürKomplexeBedingung(); …
```

Java

Beispiele zu Klammern und Separatoren

Selbststudium!

- Relativ schlecht wartbarer Code (Java):

```Java
01 if (bedingung)
02   methode();
```

- Besser wartbarer Code, weil hinzufügen eines weiteren Statement im Rumpf nicht so leicht schief geht:

```Java
11 if (bedingung) {
12   methode();
13 }
```

- Relativ schlecht wartbarer Code (in Pascal):

```Pascal
21 if xyz then
22 begin
23   statement1;
24   statement2
25 end;
```

- Besser wartbarer Code:
  - Strichpunkt nach **`statement2`**

Wahl von Bezeichnern

- Einheitliche Namenskonvention verwenden!

- Bezeichner sollen:
  - natürlicher Sprache entnommen sein (bevorzugt Englisch)
    - Ausnahmen: Schleifenvariablen, manche Größen in Formeln
  - aussagekräftig sein
  - leicht zu merken sein
  - nicht zu lang sein, wenn häufig verwendet
  - Kurze Bezeichner nur bei sehr kleinem Scope (Schleifenvariablen)

- Beispiele: Wofür ist welcher Bezeichner gut?
  - `x1, x2, i, j, k`
  - `customername, CustomerName, customerName, cust_name`
  - `accountOpening, openAccount, isAccountOpen`
  - `FILE_EXTENSION`

Beispiele für Namenskonventionen (a)

- Klasse:
  - Substantiv, erster Buchstabe groß, Rest klein
  - Ganze Worte, Zusammensetzung durch Großschreibung
  - Bsp: **`Account, StandardTemplate`**
- Methode:
  - Verb, Imperativ (Aufforderung), erster Buchstabe klein
  - Lesen und Schreiben von Attributen mit get/set-Präfix im Namen
  - Bsp: **`checkAvailability(), doMaintenance(), getDate()`**
  - Abfragen/Queries (bool-Ergebnis, keine Änderungen):
    `isLarge(), hasFather()`

Beispiele für Namenskonventionen (b)

Selbststudium!

- Konstanten:
  - Nur Großbuchstaben, Worte mit "_" zusammengesetzt
  - Standardpräfixe: "MIN_", "MAX_", "DEFAULT_", …
  - Bsp.: **`NORTH, BLUE, MIN_WIDTH, MAX_WIDTH, DEFAULT_SIZE`**
- Generierte Bezeichner
  - Mit führendem Underscore
  - Bsp: **`_availability, _date`**

Lesbarkeit durch Bezeichnerwahl

Selbststudium!

Java

```
01 public class Zinstabelle {
02
03   public static void main(String[] args) {
04     double betrag;       // zu verzinsender Betrag
05     double zinssatzJahr; // jaehrlicher Zins als Faktor
06     int laufzeit;        // Laufzeit in Jahren
07
08     betrag = Console.readDouble("Betrag:");
09     zinssatzJahr = Console.readDouble("Zinssatz:") / 100;
10     laufzeit = Console.readInt("Laufzeit:");
11
12     double y;
13
14     for (y = zinssatzJahr - 0.01;
15             y <= zinssatzJahr + 0.01; y += 0.00125) {
16       double zinssatzMonat = y/12;
17       double zahlung = betrag * zinssatzMonat /
18               (1 - (Math.pow(1/(1 + zinssatzMonat),
19               laufzeit * 12)));
20       System.out.println(100*y+" : "+zahlung);
21     }
22   }
23 }
```

Änderungsfreundlicher Code (1)

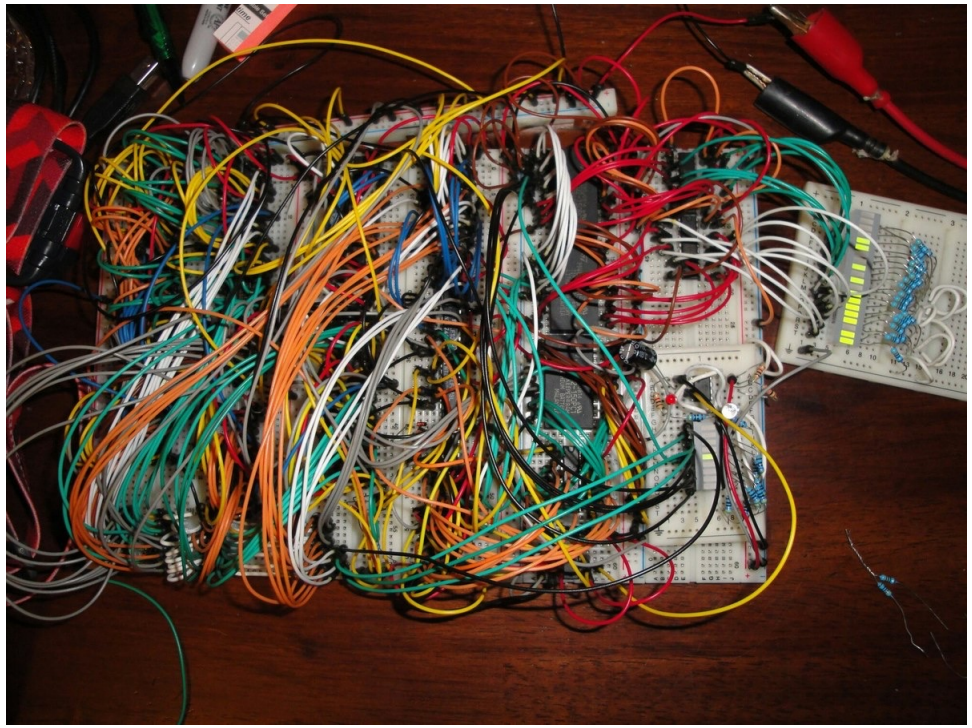
Selbststudium!

- Wahl von Variablen, Konstanten und Typen orientiert an der fachlichen Aufgabe, nicht an der Implementierung:
  - Gutes Beispiel (C):
    ```
    typedef char name [nameLength]
    typedef char firstName [firstNameLength]
    ```
  - Schlechtes Beispiel (C):
    ```
    typedef char string10 [10]
    ```
- Symbolische Konstante statt literale Werte verwenden, wenn spätere Änderung denkbar (also immer).
- Algorithmen, Formeln, Standardkonzepte in Methoden/Prozeduren kapseln.
- An den Leser denken:
  - Fließtext (z.B. Kommentare) nicht breiter als 80 Zeichen
  - Quellcode nicht breiter als Editorfenster (ca. 80 bis 160 Zeichen)
  - Zusammenhängende Einheit möglichst etwa Größe eines typischen Editorfensters (40-60 Zeilen)
  - Text probehalber vorlesen ("Telefon-Test")

Änderungsfreundlicher Code (2)

- Strukturierte Programmierung
  - Kein "goto" verwenden (auch wenn noch in der Sprache vorhanden)
  - "switch" nur mit "break"-Anweisung nach jedem Fall
  - "break" nur innnerhalb "switch"-Anweisungen verwenden
  - "continue" nicht verwenden
  - "return" nur zur Rückgabe des Werts, nicht als Rücksprung in der Mitte der Methode

- “Goto considered harmful” (E.W. Dijkstra)
  - beschreibt die Probleme der Spaghetti-Programmierung



Pic source: https://exceptionnotfound.net/spaghetti-code-the-daily-software-anti-pattern/

Änderungsfreundlicher Code (3)

Selbststudium!

- Übersichtliche Ausdrücke:
  - Seiteneffektfreie Ausdrücke
    - Schlechtes Beispiel:
      ```
      y += 12*x++;
      ```
    - Inkrementierung/Dekrementierung besser in separaten Anweisungen
      ```
      x++;
      y += 12*x;
      ```
- Sichtbarkeitsprüfungen des Compilers ausnutzen:
  - Attribute möglichst lokal und immer "private" (oder "protected") deklarieren
  - Wiederverwendung "äußerer" Namen (Verschattung) vermeiden

Stilistisch weiter verbessertes Programm

Selbststudium!

Java

```
01 public class Zinstabelle {
02
03   private static double zinsFormel(double betrag, double zinssatzMonat, double laufzeit) {
04     double faktor = 1 - Math.pow(1 / (1 + zinssatzMonat), 12 * laufzeit)
05     return betrag * zinssatzMonat / faktor;
06   }
07
08   private static final double BEREICH = 0.01;
09   private static final double SCHRITTWEITE = 0.00125;
10
11   public static void main(String[] args) {
12
13     double betrag;       // zu verzinsender Betrag
14     double zinssatzJahr; // jaehrlicher Zins als Faktor
15     int laufzeit;        // Laufzeit in Jahren
16
17     // Werte einlesen
18     betrag = Console.readDouble("Betrag:");
19     zinssatzJahr = Console.readDouble("Zinssatz:") / 100;
20     laufzeit = Console.readInt("Laufzeit:");
21
22     // Ergebnisse ausgeben
23     double y;
24     for (y = zinssatzJahr - BEREICH;   y <= zinssatzJahr + BEREICH;   y += SCHRITTWEITE) {
25       System.out.println(100 * y + " : " + zinsFormel(betrag,y / 12,laufzeit));
26     }
27   }
28 }
```

(Übrigens: es ist noch weiteres verbesserungsfähig!)

Kommentare

- Prinzip der integrierten Dokumentation:
  - Kommentare im Code sind leichter zu warten als extra Dokumente
  - Kommentare müssen parallel zum Code entstehen
    - *"Nach-Dokumentation" funktioniert in der Praxis nie!*
  - Werkzeuge zur Generierung von Dokumentation (z.B. javadoc)

- Idealzustand:
  - Kommentare zu Klassen und Methoden stellen eine kompakte Spezifikation des Codes dar

- Kommentare sollen *nicht*:
  - den Code unlesbar machen
    - *z.B. durch Verzerrung des Layouts*
  - redundante Information zum Code enthalten
  - Schlechter Kommentar:

```
i++; // i wird hochgezählt
```

- Lesbarer kommentarfreier Code ist besser als systematisch kommentierter, aber unlesbarer Code.

Typischer Einsatz von Kommentaren

Selbststudium!

- "Vorspann" von Paketen, Klassen, Methoden etc.
  - Zweck, Parameter, Ergebnisse, Exceptions
  - Vorbedingungen, Abhängigkeiten (z.B. Plattform), Seiteneffekte
  - Änderungsgeschichte, Status
- Formale Annahmen (assertions):
  - Vorbedingungen, Nachbedingungen
  - Allgemeingültige Annahmen (Invarianten)
- Lese-Erleichterung
  - Zusammenfassung komplexer Codepassagen
  - Überschriften zur Codegliederung
- Erklärung von einzelnen Besonderheiten des Codes
  - z.B. schwer verständliche Schritte, Seiteneffekte
- Arbeitsnotizen
  - Einschränkungen, bekannte Probleme
  - Offene Stellen (“!!!”,”TODO”), Anregungen, Platzhalter

Hinweise zum Verfassen von Kommentaren

Selbststudium!

- Phrasen statt Sätze: völlig ok!
  - Kürze und Übersicht zählt hier mehr als literarischer Anspruch.
- Deskriptiv (3. Person), nicht preskriptiv (2. Person)
  - Bsp: "Setzt die Kontonummer." statt: "Setze die Kontonummer."
- Unnötigen Rahmentext vermeiden:
  - Bsp.: "Setzt die Kontonummer." statt:
    "Diese Methode setzt die Kontonummer"
- Verwendung von "this" bzw. "diese/r/s"
  - Bsp: "Ermittelt die Version dieser Komponente." statt:
    "Ermittelt die Version der Komponente."

Professionell kommentierter Code / JavaDoc

Java

```
01 /*
02  * Copyright (c) 1994, 1998, Oracle and/or its affiliates. All rights reserved.
03  * DO NOT ALTER OR REMOVE COPYRIGHT NOTICES OR THIS FILE HEADER.
04  *
05  * This code is free software; you can redistribute it and/or modify it
06  * under the terms of the GNU General Public License version 2 only, as
07 ...
08 */
09 package java.util;
10
11 /**
12  * A class can implement the <code>Observer</code> interface when it
13  * wants to be informed of changes in observable objects.
14  *
15  * @author  Chris Warth
16  * @see     java.util.Observable
17  * @since   JDK1.0
18  */
19 public interface Observer {
20     /**
21      * This method is called whenever the observed object is changed. An
22      * application calls an <tt>Observable</tt> object's
23      * <code>notifyObservers</code> method to have all the object's
24      * observers notified of the change.
25      *
26      * @param   o     the observable object.
27      * @param   arg   an argument passed to the <code>notifyObservers</code>
28      *                 method.
29      */
30     void update(Observable o, Object arg);
31 }
```

viele Zeilen Erklärung für die Folie weggelassen

Einsatz von Entwurfsmustern

Beispiel:

- Factory Einsatz
  - stets Factories statt Konstruktoraufrufe verwenden
- Singleton Einsatz
  - keine statischen Methoden oder Attribute verwenden, sondern Singleton-Muster anwenden
  - wenn sichergestellt werden muss, dass es nur eine Instanz einer Klasse gibt
  - auf Testbarkeit achten
- Grund: statische Aufrufe (und Konstruktoren gehören dazu) lassen sich für Tests nicht z.B. durch simulierende Mocks ersetzen

Testfälle

Selbststudium!

- Tests mit JUnit für Java oder CppUnit für C++, …
  - für Datenbanksysteme zusätzlich dbUnit

- Vorgegebene Teststruktur

Java

```
01 public void testFoo() {
02
03     // definition of test input
04     ...
05
06     // execute tested code
07     ...
08
09     // assertions, check expected values
10     ...
11 }
```

Testfälle - Beispiel

Selbststudium!

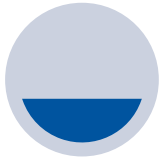
Java

```
01 public class CarTest { // naming convention: class you are testing + "Test"
02
03   @Test  // use JUnit4 annotations
04   public void testAccelerate() {  // method naming convention: "test" + method name
05
06     // define the test input. name variables with "input" prefix
07     Car inputCar = new Car();     // if you need complex objects use factories
08     int inputAccValue = 100;
09
10     // execute method and store result. name variables with "actual" prefix
11     inputCar.accelerate(inputAccValue);
12     int actualAccValue =  inputCar.getSpeed();
13
14     // define the expected values. name variables with "expected" prefix
15     int expectedAccValue = 100;
16
17     // compare actual values with expected
18     assertEquals(expectedAccValue, actualAccValue);
19
20     // note: this test is not complete (e.g. more to check)
21   }
22 }
```

Guidelines zur Projekt-Infrastruktur und Archivierung

- Einheitlicher Zeichensatz, z.B. UTF-8
  - verschiedene Zeichensätze häufig Ursache für Probleme
- Zeilenumbrüche
  - Unix-Zeilenumbrüche (LF) verwenden
- Bugtracking- oder Ticketsystem
  - Trac oder git issues verwenden
  - Tickets unmittelbar nach der Bearbeitung schließen (oder zur Abnahme markieren)
- Versionierungssystem
  - Nichts einchecken was generiert oder compiliert ist
  - Makefiles und Build-Skripte werden versioniert
  - keine Leerzeichen und Umlaute in Dateinamen
  - vor jedem Einchecken Regressionstests ausführen
- Konventionen für Dateien, die nicht SVN im liegen, aber trotzdem versioniert werden sollen
  - auf keinen Fall Zusätze wie „_final“, „_neu“
  - besser: Datum, Versionsnummer

Was haben wir gelernt?



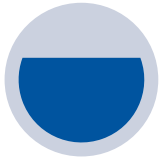
Guter Programmierstil

notwendig, da man Code deutlich öfter(!) liest als schreibt

hilft Spaghetti Code zu vermeiden und verhindert „Hacken“

(leider) kein allgemein anerkannter einheitlicher Codingstandard vorhanden

Projekt-/Firmen-interne Einigung auf einen Standard



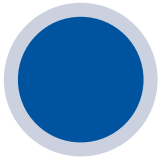
Kriterien

Aussagekräftige und kompakte Kommentierung

Namenswahl

Einrückung

Tipps: Entwurfmuster, Versionierung und Ticketing, Technische Richtlinien und Archivierung



Praxis

Oft unternehmensinterne Vorgaben

Einigung auf gemeinsamen Projektinternen Standard

Java, C++: Einige im Internet zugängliche Empfehlungen/Standards

Softwaretechnik

8. Implementierung
8.4. Datenstrukturen

Prof. Bernhard Rumpe
Software Engineering
RWTH Aachen

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

+ Selbststudium: se-rwth.de/swt_material/

Literatur:

- Einschlägige Java-Bücher,
  z.B. David Flanagan: Java in a Nutshell

SE Software Engineering | RWTH AACHEN UNIVERSITY

Bedeutung von Datenstrukturen

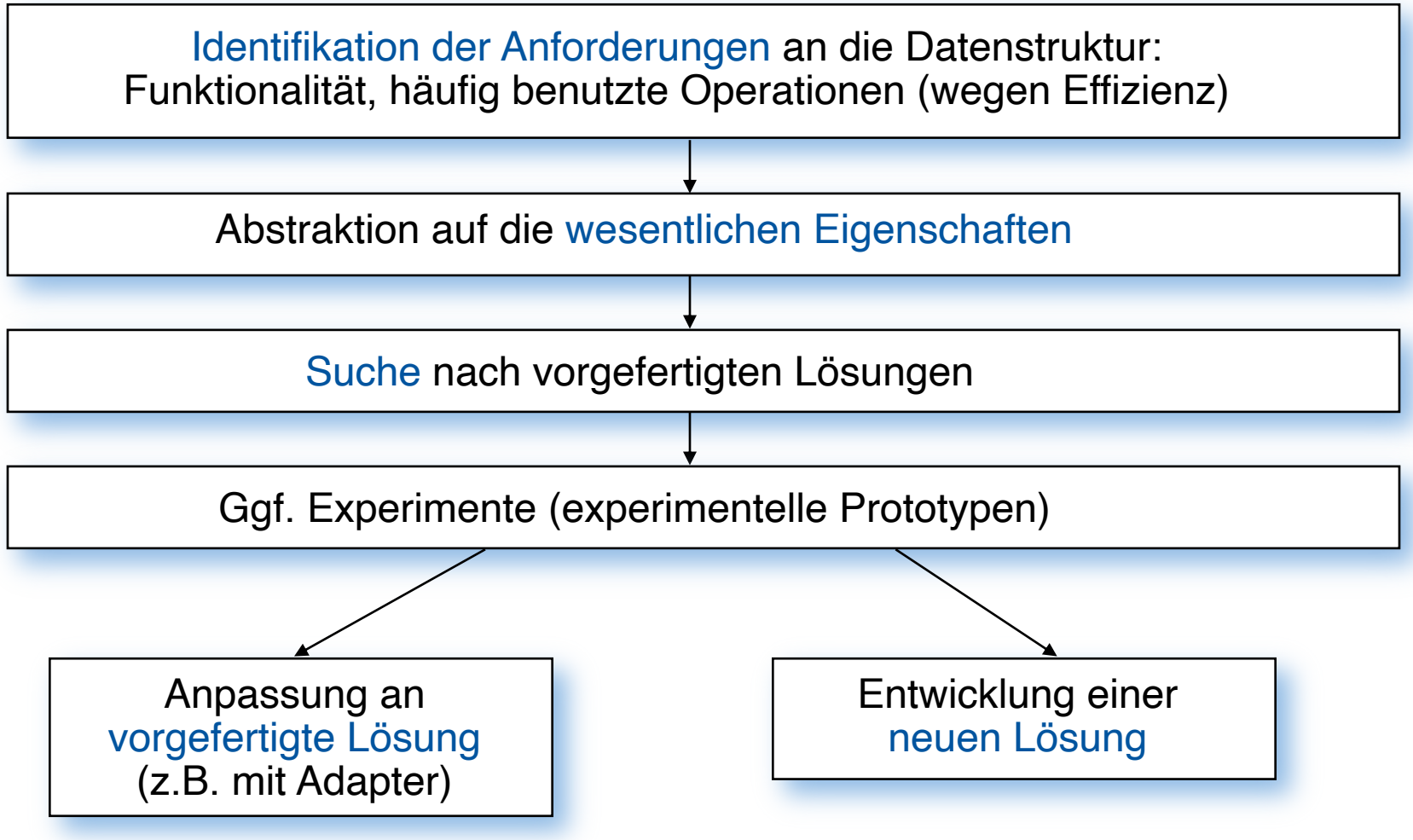
- Struktur
  - Ordnungssystem für die Daten
  - Bereitstellung von Standard-Funktionalität
- Wiederverwendung
  - Klassenbibliotheken
  - Standardalgorithmen
- Anpassbarkeit
  - Alternative Implementierungen für gleiche abstrakte Schnittstelle
- Optimierung
  - Alternativen mit verschiedener Leistungscharakteristik

- Beispiel: Implementierungen der Mengen in Java

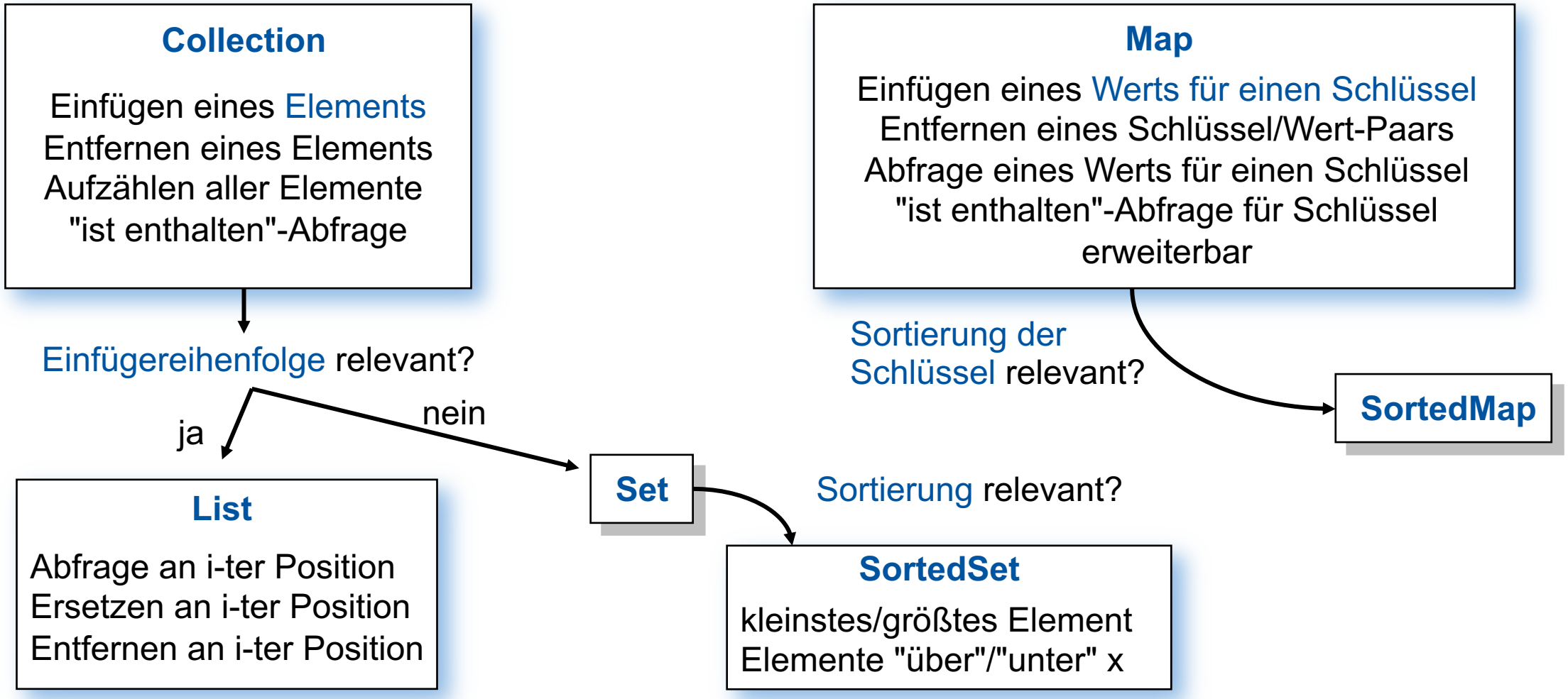
Abstrakter und konkreter Datentyp

Abstrakter Datentyp (Schnittstelle)	***Konkreter Datentyp (Implementierung)***
Abstraktion: • Operationen • Verhalten der Operationen ▪ *Theorie:* • Algebraische Spezifikationen • Axiomensysteme ▪ *Praxis*: • Abstrakte Klassen • Interfaces ▪ *Beispiel:* • Liste	Konkretisierung: • Instanziierbare Klassen • Ausführbare Operationen ▪ *Theorie:* • Datenstrukturen • Effizienzfragen ▪ *Praxis:* • Alternative Implementierungen ▪ *Beispiel:* • Verkettete Liste • Liste durch Array

Vorgehensweise beim Datenstruktur-Entwurf

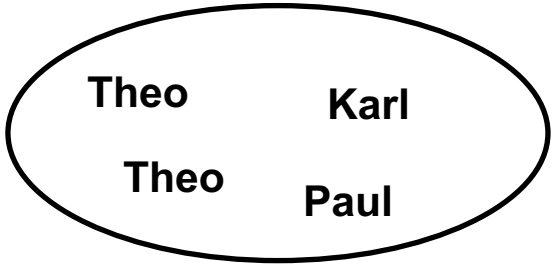


Suche nach vorgefertigten Lösungen



Collections und Maps

- Collections sind Sammlungen von Objekten
  - Mengen (ohne Duplikate)
  - Listen (geordnet)



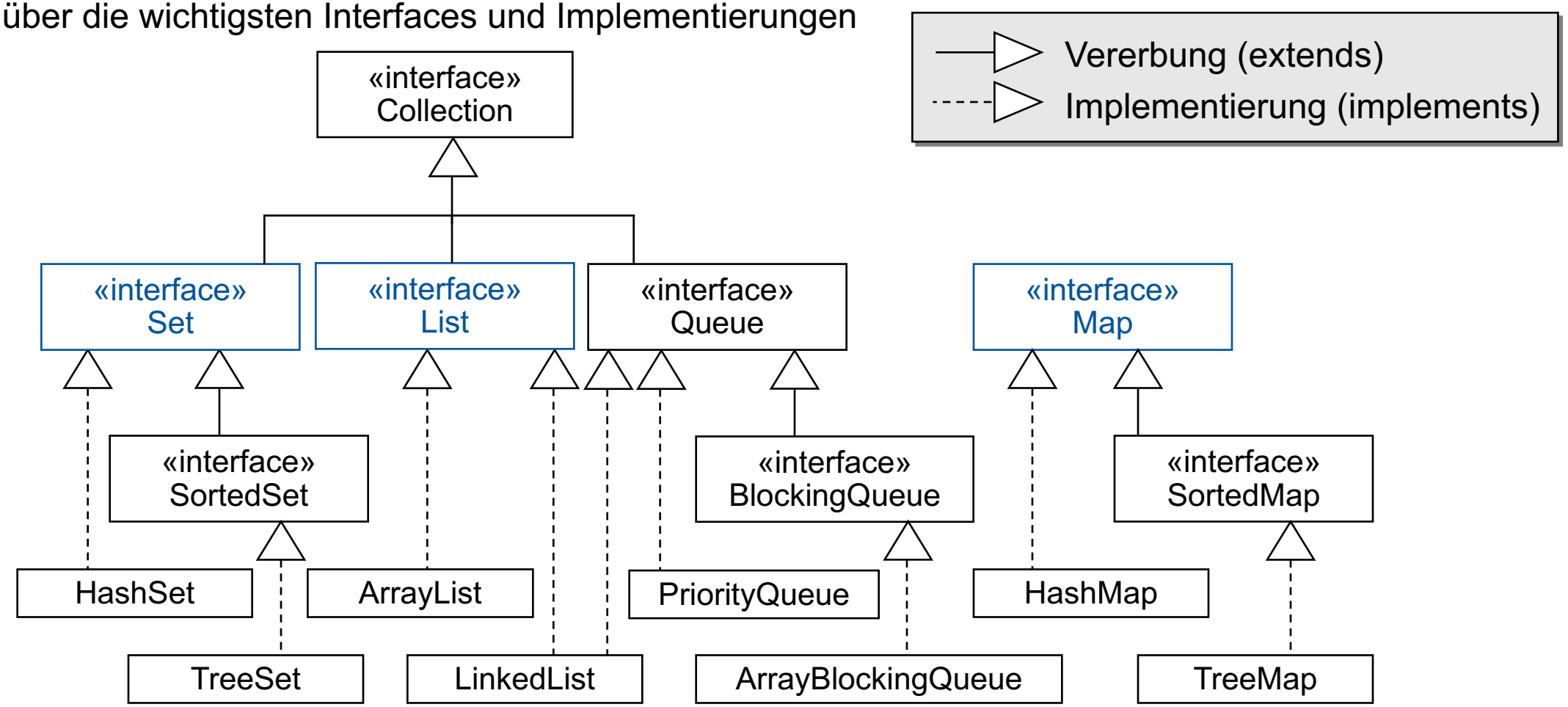
- Maps sind Abbildungen von Schlüsselobjekten auf Werte
  - Schlüssel sind selbst eine Menge
  - Werte sind eine Collection (mit Duplikation)

Theo	**Ahornstraße 55**
Karl	**Templergraben 57**
Paul	**Templergraben 57**

- Unterscheidungskriterien der Schnittstellen und Klassen (Auswahl)
  - Ordnung (keine, total)
  - Umgang mit `null`-Elementen
  - Threadsicherheit
  - Komplexität von Operationen: suchen, einfügen, löschen
  - Lösungen für Spezialfälle: `Stack, Queue`

Java Collections Framework

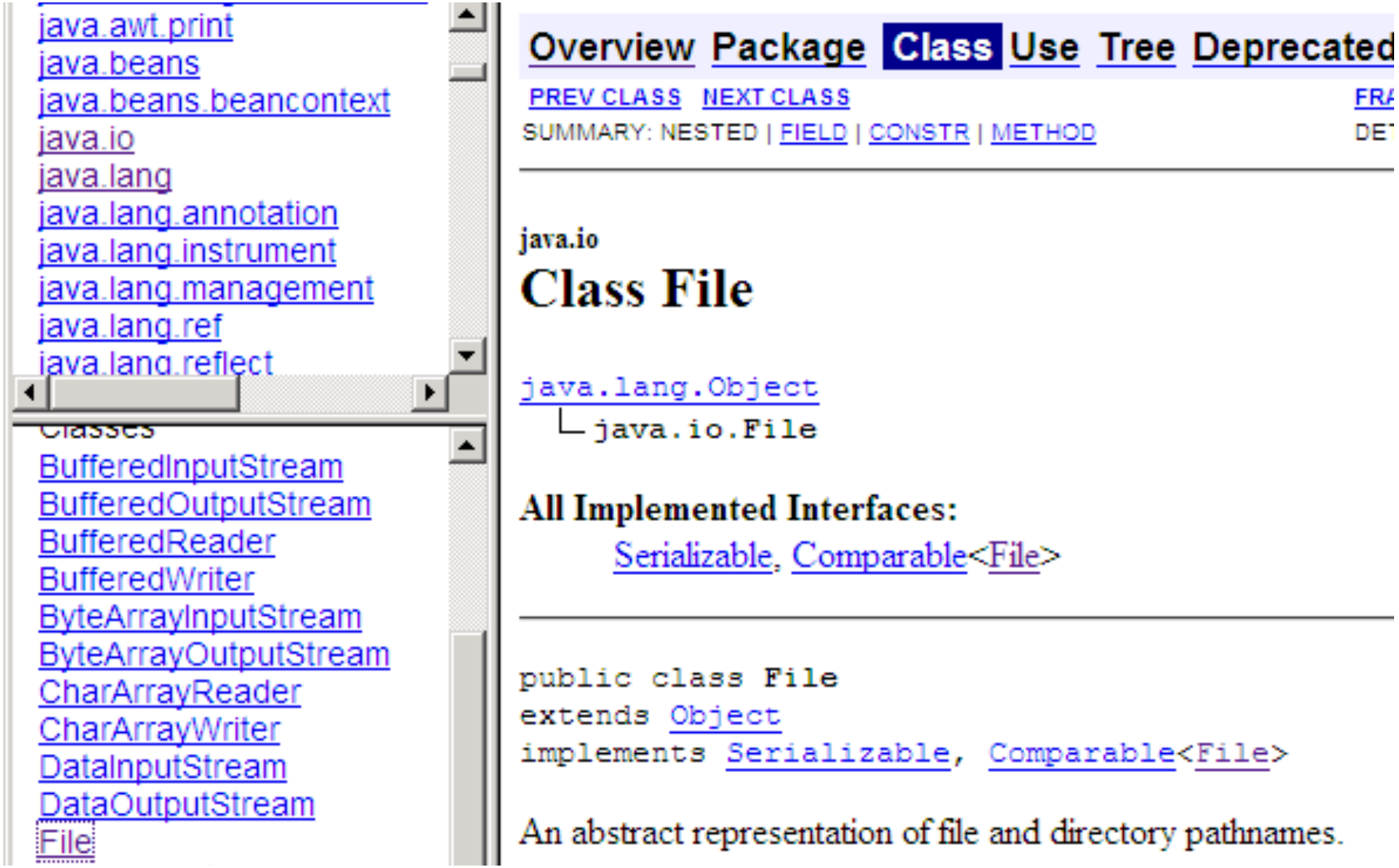
- Übersicht über die wichtigsten Interfaces und Implementierungen



Java Klassenbibliothek

- Java: Objektorientierte Sprache mit bekannten Konzepten
  (Klassen, Interfaces, Packages, Sichtbarkeiten, Vererbung, ...)
- Zusätzlich: umfangreiche Klassenbibliothek aus der Standardfunktionalität importiert werden kann
- Java 2 Standard Edition Klassenbibliothek:
  - Tausende Klassen und Interfaces in Hunderten von Packages (and growing) …
- Ziel: Wiederverwendung
  - schnellere Entwicklung | weniger Fehler | performantere Implementierung
- Problem:
  - Kennen der vorhandenen Funktionalität
  - Entscheidung zwischen Alternativen
- Wichtiges Hilfsmittel: Java API Dokumentation
  API = **A**pplication **P**rogramming **I**nterface
  http://download.oracle.com/javase/8/docs/api/

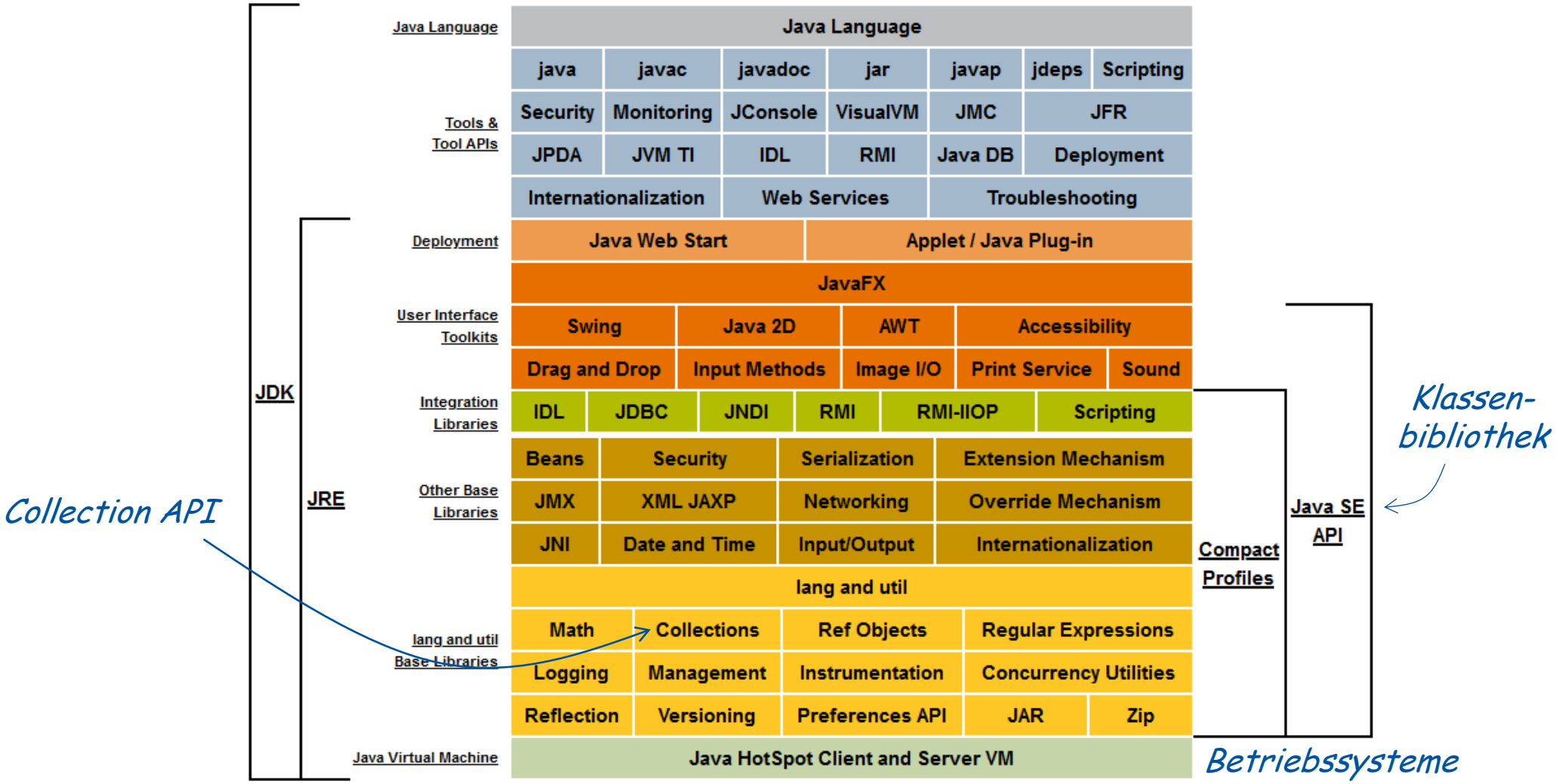
Java API Dokumentation, e.g. http://download.oracle.com/javase/6/docs/api/



Java Klassenbibliothek - Beispiele

- java.lang
  - einfache Datentypen: **`Integer`**, **`Boolean`**, **`Double`**, ...
  - Zugriff auf Umgebungsvariablen, Standard-Streams
  - nebenläufige Programmierung, Threads
- java.io
  - Ein- / Ausgabe: Kapselung in Streams, Zugriff mittels Reader und Writer
- java.net
  - Netzwerkprogrammierung: URLs, Sockets, Protokolle, ...
- java.awt, javax.swing
  - GUI-Programmierung: Fenster, Dialoge, Buttons, ...
- java.util
  - Hilfsklassen für Kalender- und Datumsfunktionalität, Zufallszahlen, ...
  - Java Collections Framework

Plattform, Klassenbibliothek und Collection API



Java Platform Standard Edition, Version 8:
https://www.oracle.com/technetwork/java/javase/tech/index-jsp-140763.html

Java Collections Framework

Selbststudium!

- Collection
  - Objekt, das eine Sammlung von Objekten repräsentiert
    - zur Strukturierung der Daten
    - entscheiden häufig über Effizienz der Operationen, Anwendung
    - vgl. Datenstrukturen der Programmierung
  - Operationen zur Abfrage, Manipulation
    - `contains(..), size(), add(..), ...`
- Framework
  - Definition (nach Pomberger/Blaschek): Ein "framework" (Rahmenwerk, Anwendungsgerüst) ist eine Menge von zusammengehörigen Klassen, die einen abstrakten Entwurf für eine Problemfamilie darstellen.

Java Collections Framework

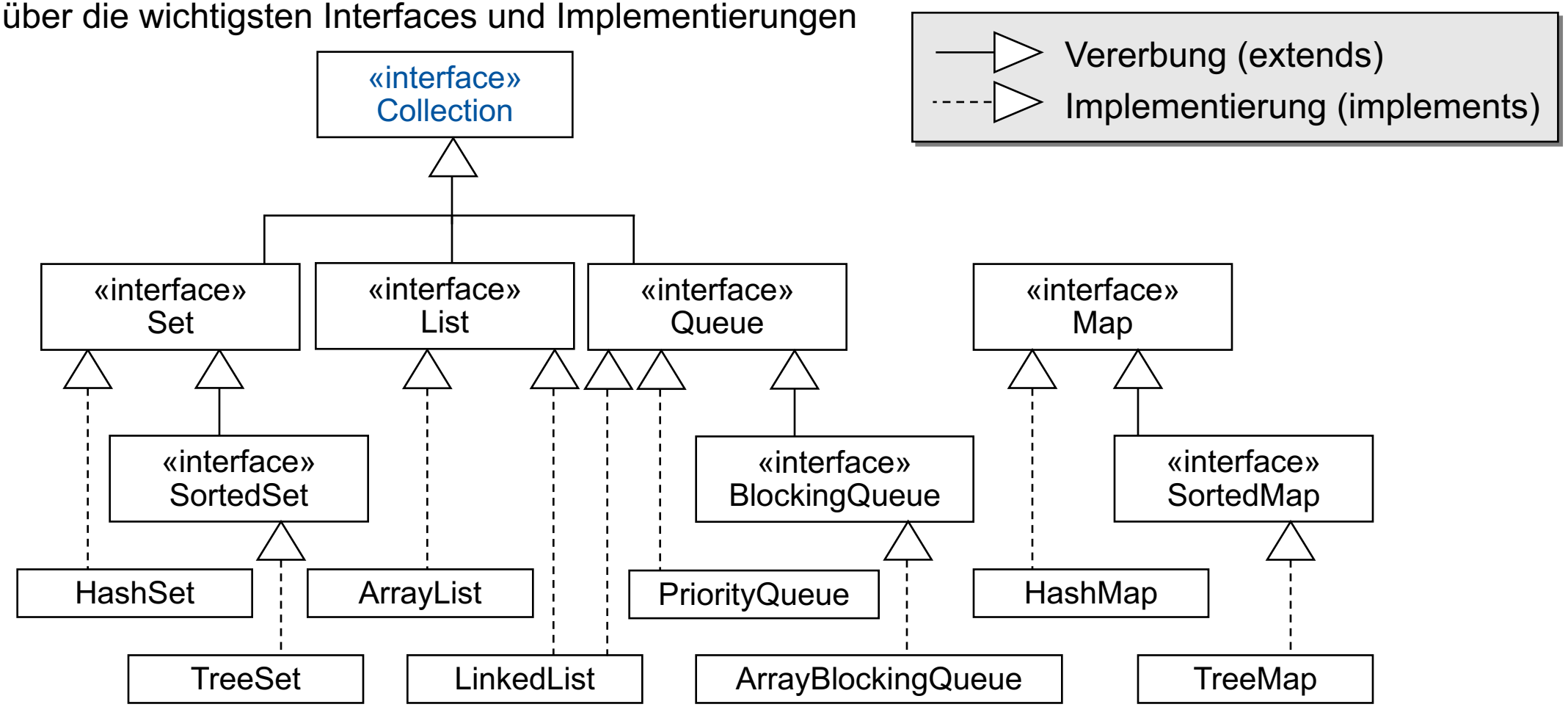
Selbststudium!

- Vorteile
  - reduzierter Programmieraufwand
  - Anpassbarkeit und Optimierbarkeit
  - erhöhte Performance
  - weniger Fehler
  - Interoperabilität von unabhängigen APIs
  - erhöhte Wiederverwendbarkeit
  - erhöhte Verständlichkeit
- „Nachteil“
  - Einarbeitungsaufwand bei „erstem Lernen“
- Umfang: 70+ Klassen/Interfaces im Package **`java.util`**

Java Collections Framework

Selbststudium!

- Übersicht über die wichtigsten Interfaces und Implementierungen



Interface `Collection`

Selbststudium!

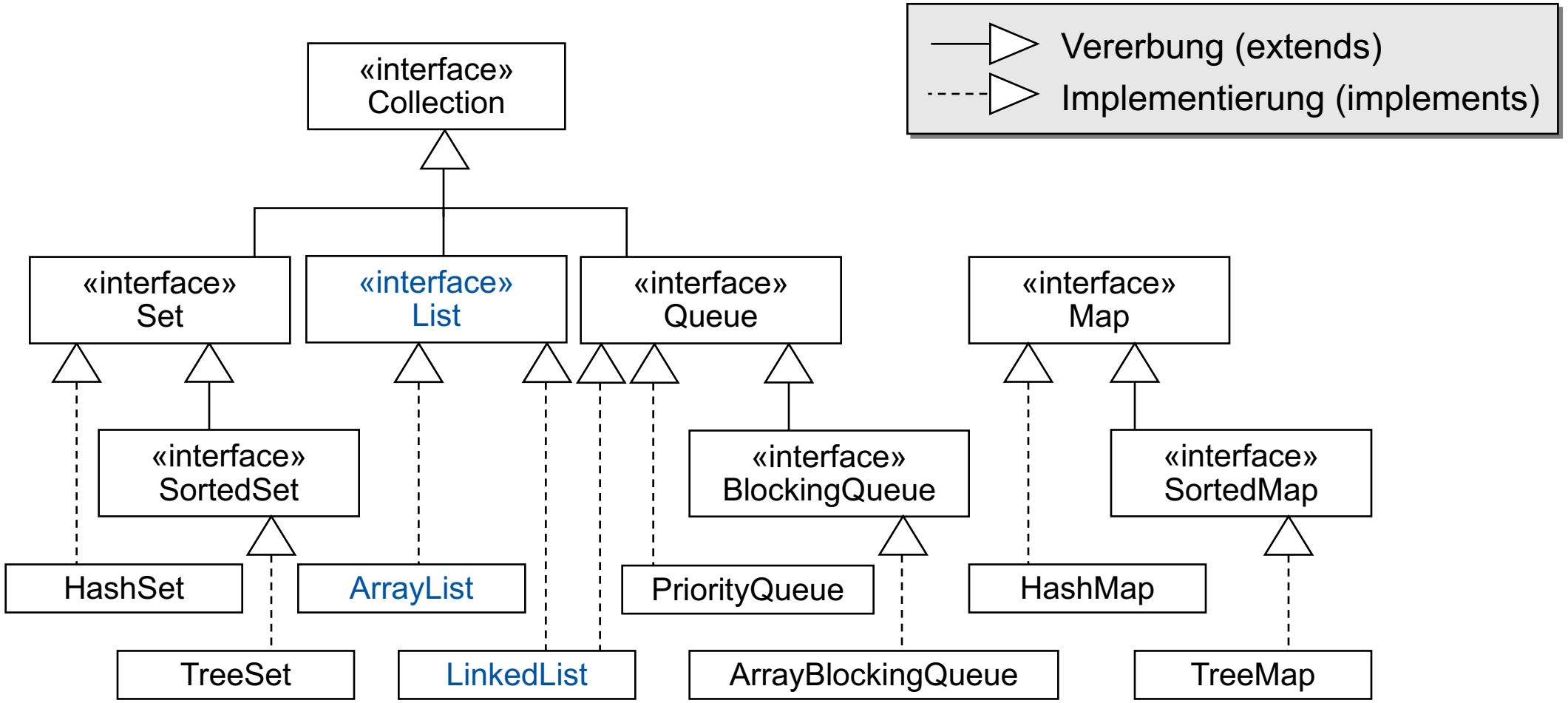
- „kleinster gemeinsamer Nenner“ der Collection Implementierungen
- schreibt Operationen für den Zugriff und die Manipulation der unterliegenden Daten vor

- Java bietet ein Schleifenkonstrukt zur Bearbeitung von Collections
  - `for (aType aVar : aCollection) { aStatement }`

- Besonderheiten:
  - optionale Operationen sind in der API Dokumentation als optional markiert und müssen nicht implementiert werden
  - lösen zur Laufzeit `UnsupportedOperationException` aus
  - aber: alle Implementierungen in `java.util` implementieren alle optionalen Methoden
  - `Map` ist Teil des Collections Framework, leitet aber nicht von `Collection` ab

Listen

Selbststudium!



Listen

Selbststudium!

- Liste oder Sequenz von Objekten, geordnet
- doppelte Elemente erlaubt
- Einfügen von `null` erlaubt
- zusätzliche verbindliche und optionale Operationen
- bieten speziellen `ListIterator` mit erweiterter Funktionalität
- Implementierungen
  - ArrayList
    - Listenimplementierung auf Array Basis
    - benötigte Arraygröße wird dynamisch angepasst
    - anfängliche Größe des Arrays kann angegeben werden
    - optimale Arraygröße kann erzwungen werden
  - LinkedList
    - Listenimplementierung als doppelt verkettete Liste
    - zusätzliche Operationen zum Einfügen/Entfernen/Zugriff auf Anfang und Ende der Liste

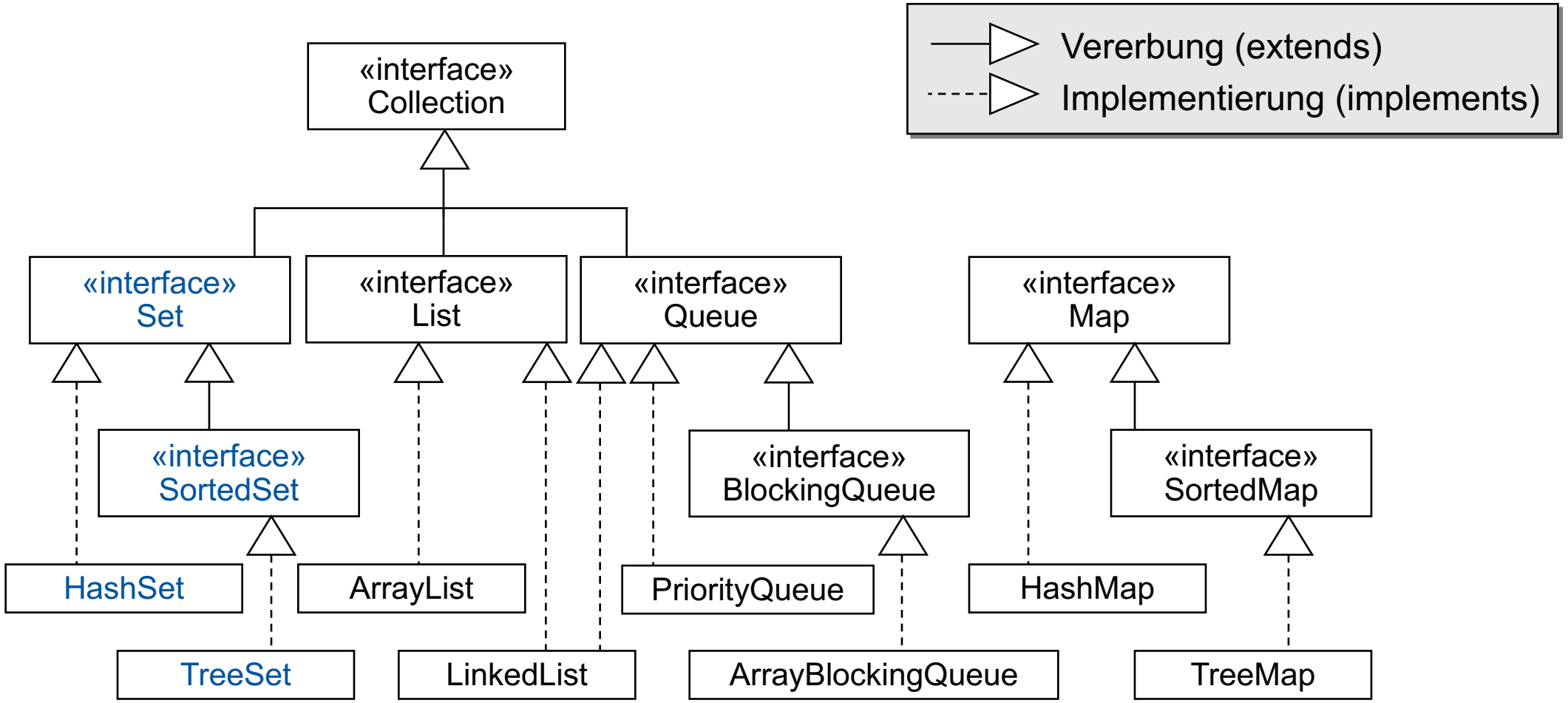
Listen Implementierungen

Selbststudium!

- **`ArrayList`** und **`LinkedList`** sind konkrete Listenimplementierungen
- Wann ist welche Liste sinnvoll?
- Performancevergleich
  - LinkedList
    - schnelles Einfügen in O(1) , Löschen O(n)
    - wahlfreier Zugriff in O(n)
  - ArrayList
    - wahlfreier Zugriff in O(1)
    - Einfügen und Löschen u.U. teuer, reicht die Kapazität des Array nicht aus, muss die gesamte Liste in ein neues Array umkopiert werden
- Faustregel:
  - Bei häufigem Zugriff und seltenen Änderungsoperationen: **`ArrayList`**
  - Bei häufigen Änderungen und seltenem wahlfreien Zugriff: **`LinkedList`**

Mengen

Selbststudium!



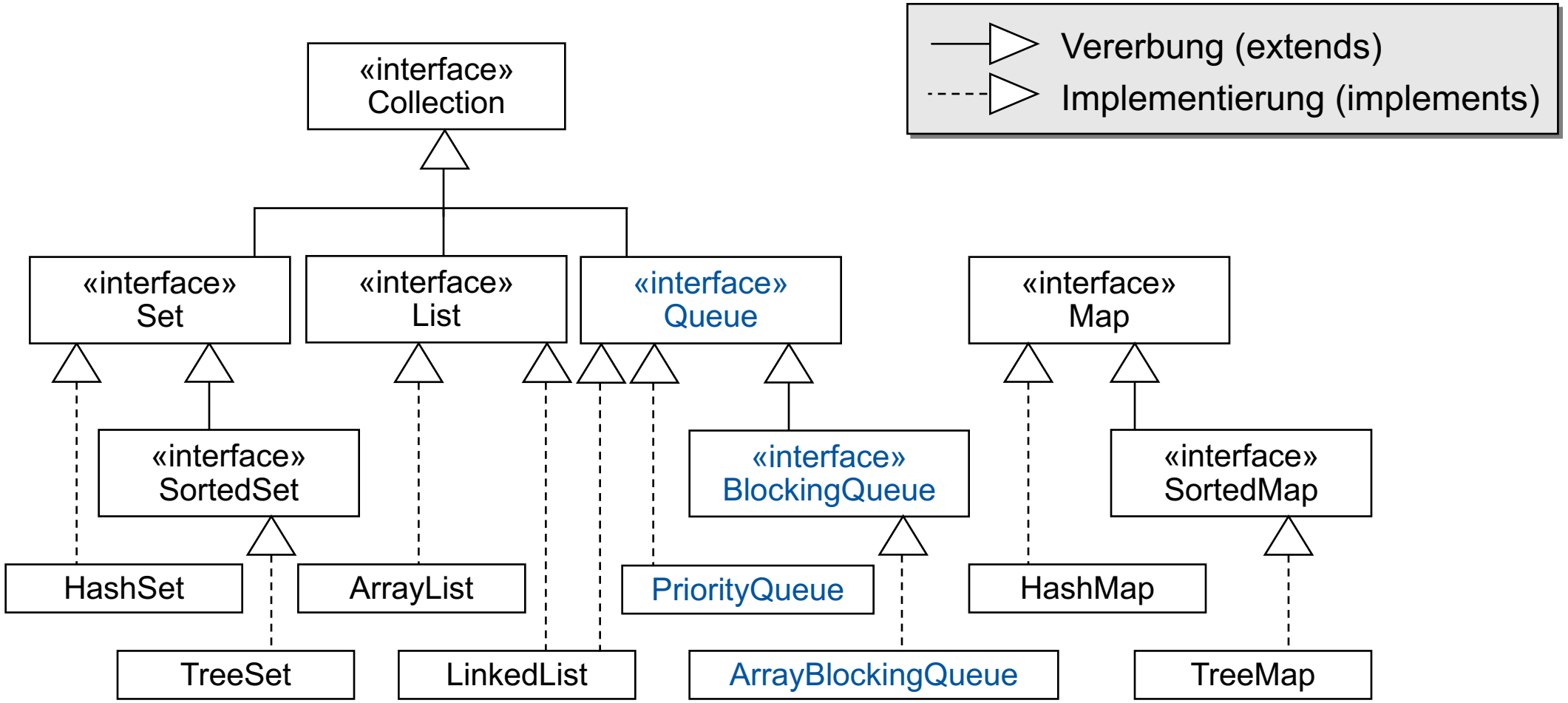
Mengen

Selbststudium!

- keine Duplikate
- keine Reihenfolge der Elemente
- mittels SortedSet kann natürliche Ordnung der Elemente hergestellt werden
- Mengenoperationen wie Vereinigung, Schnitt durch Standard Collection Operationen addAll(), retainAll()
- Implementierungen
  - HashSet
    - keine garantierte Ordnung der Elemente bei Iteration
    - basiert auf HashMap
    - daher O(1) Performance für Basisoperationen (add, contains and remove)
  - TreeSet
    - aufsteigende natürliche Ordnung der Elemente
    - basiert auf TreeMap
    - O(log(n)) für Basisoperationen (add, remove and contains)

Queue („Schlangen“)

Selbststudium!



Queue

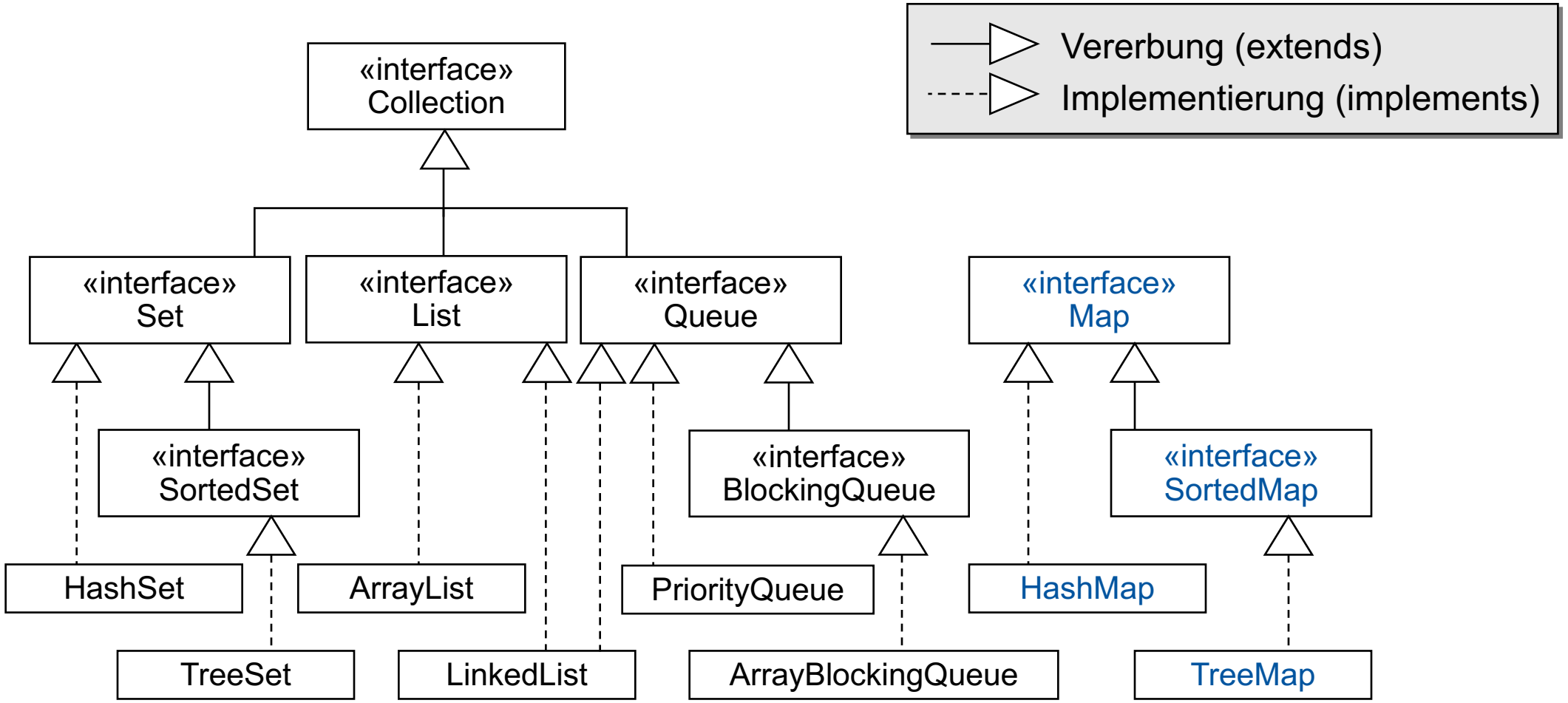
Selbststudium!

- Elemente geordnet
- Zugriff normalerweise nach dem FIFO (first in, first out) Prinzip

- Varianten:
  - BlockingQueue (eigenes Interface): blockierender Zugriff bei leerer oder voller Schlange
  - PriorityQueue: Element mit höchster Priorität wird zuerst zugegriffen

- Implementierung
  - viele Subklassen, speziell für nebenläufige Programmierung (`java.util.concurrent`)
  - **`PriorityBlockingQueue`**
  - **`ArrayBlockingQueue`**

Assoziative Speicher (Maps)



Assoziative Speicher (Map<K,V>)

- Map<K,V> speichert Key,Value-Paare
- Map ist typisiert mit 2 Typ-Parametern:
  - K – der Typ der Schlüssel (Keys)
  - V – der Typ der damit assoziierten Werte (Values)
- keine doppelten Schlüssel erlaubt
- bietet Collection „Sichten“, z.B. durch Collection der Key oder Values
- Implementierungen
  - HashMap
    - keine garantierte Ordnung der Schlüssel bei Iteration
    - O(1) Performance für Basisoperationen (containsKey, get, put, remove)
  - LinkedHashMap
    - leitet von HashMap ab, garantiert aber die Reihenfolge der Schlüssel (Einfügereihenfolge)
  - TreeMap
    - aufsteigende natürliche Ordnung der Schlüssel
    - O(log(n)) für Basisoperationen (containsKey, get, put, remove)

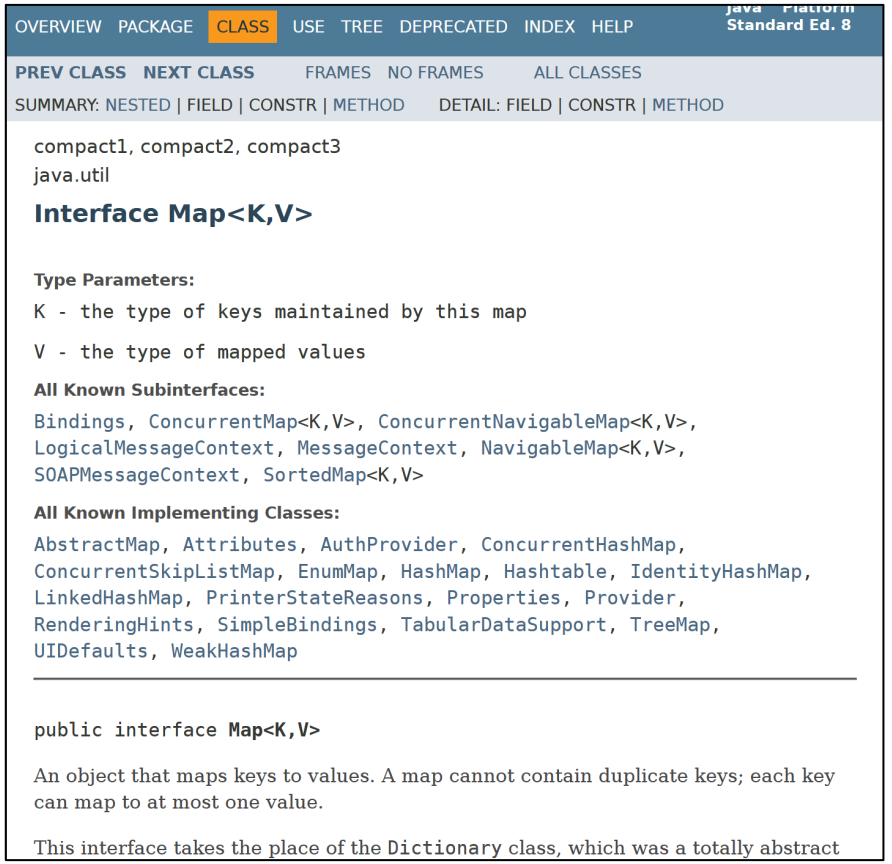
Interface des Abstrakten Datentyps `Map<K,V>` | 1

- Relevante "Lesende" Operationen
  - **`boolean containsKey(Object key)`**
    - liefert **`true`**, falls Eintrag mit Schlüssel **`key`** enthalten ist
  - **`boolean containsValue(Object value)`**
    - liefert **`true`**, falls zu **`value`** ein oder mehr Einträge vorhanden sind
  - **`Set entrySet()`**
    - liefert eine **`Set`**-Sicht auf die Paare von Einträgen
  - **`V get(Object key)`**
    - liefert den Wert zum Schlüssel **`key`**, liefert **`null`** falls Mapping nicht existiert oder **`null`** der Wert ist
  - **`boolean isEmpty()`**
    - liefert **`true`**, falls keine Schlüssel-Wert-Paare gespeichert sind
  - **`Set<K> keySet()`**
    - liefert eine **`Set`**-Sicht auf die Schlüssel
  - **`int size()`**
    - liefert die Anzahl der Schlüssel-Wert-Paare
  - **`Collection<V> values()`**
    - liefert eine **`Collection`**-Sicht auf die Werte
- Beobachtung: Argumente lesender Operationen sind oft allgemein gehalten (Object)

Interface des Abstrakten Datentyps `Map<K,V>` | 2

- Einige ändernde Operationen
  - **`void clear()`**
    - entfernt alle Schlüssel-Wert-Paare
  - **`V put(K key, V value)`**
    - legt neuen Eintrag mit Schlüssel **`key`** und Wert **`value`** an, ersetzt ggfs. vorhandene Einträge
  - **`void putAll (Map<? extends K,? extends V> m)`**
    - fügt alle Einträge aus **`m`** ein
    - **`m`** kann "spezifischer" sein als nur **`Map<K,V>`**
  - **`V remove(Object key)`**
    - entfernt den Eintrag mit Schlüssel **`key`** und liefert entsprechenden Wert zurück

- Map hat Insgesamt 25 Methoden; zu finden unter https://docs.oracle.com/javase/8/docs/api/java/util/Map.html



OVERVIEW PACKAGE CLASS USE TREE DEPRECATED INDEX HELP — Java Platform Standard Ed. 8

PREV CLASS NEXT CLASS FRAMES NO FRAMES ALL CLASSES

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

compact1, compact2, compact3

java.util

Interface Map<K,V>

Type Parameters:

K - the type of keys maintained by this map

V - the type of mapped values

All Known Subinterfaces:

Bindings, ConcurrentMap<K,V>, ConcurrentNavigableMap<K,V>, LogicalMessageContext, MessageContext, NavigableMap<K,V>, SOAPMessageContext, SortedMap<K,V>

All Known Implementing Classes:

AbstractMap, Attributes, AuthProvider, ConcurrentHashMap, ConcurrentSkipListMap, EnumMap, HashMap, Hashtable, IdentityHashMap, LinkedHashMap, PrinterStateReasons, Properties, Provider, RenderingHints, SimpleBindings, TabularDataSupport, TreeMap, UIDefaults, WeakHashMap

---

public interface **Map<K,V>**

An object that maps keys to values. A map cannot contain duplicate keys; each key can map to at most one value.

This interface takes the place of the Dictionary class, which was a totally abstract

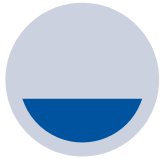
Interface `SortedMap<K,V>`

- erweitert **`Map`** z.B. mit folgenden Operationen
  - **`Comparator<...> comparator()`**
    - liefert den **`Comparator`**, der mit dem Map assoziiert ist
  - **`K firstKey()`**
    - liefert den „kleinsten“ Schlüssel
  - **`SortedMap<K,V> headMap(K toKey)`**
    - liefert ein Teilmapping mit Schlüsseleinträgen „echt kleiner“ als **`toKey`**
  - **`K lastKey()`**
    - liefert den „größten“ Schlüssel
  - **`SortedMap subMap(K fromKey, K toKey)`**
    - liefert ein Teilmapping mit Schlüsseleinträgen von **`fromKey`** (inkl.) bis **`toKey`** (exkl.)
  - **`SortedMap tailMap(K fromKey)`**
    - liefert ein Teilmapping mit Schlüsseleinträgen „größer gleich“ **`fromKey`**

Klasse `Collections`

- Sammlung von statischen Methoden für Collections
  - **`static <...> void sort(List<T> list, Comparator<...> c)`**
    - sortiert **`list`** gemäßt Comparator **`c`**
    - opt. mergesort, stabil, garantiert O(n log(n))
  - **`static int frequency(Collection<?> c, Object o)`**
    - zählt Häufigkeit von **`o`** in **`c`**
  - **`static void shuffle(List<?> list, Random rnd)`**
    - ordnet Elemente zufällig an
  - **`static Collection synchronizedCollection(Collection c)`**
    - liefert eine Thread-sichere Collection für `c`
    - entsprechende Methoden für **`List`**, **`Map`**, **`Set`**
  - ...

Was haben wir gelernt?



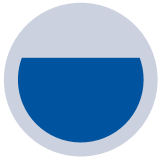
Datenstrukturen

Nicht alles selbst entwickeln

Bibliotheken in unterschiedlichen Programmiersprachen

Wichtige Konzepte: Set, List, Map

Operationen/Methoden auf den Datenstrukturen nutzen



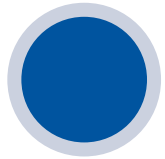
Warum?

Struktur: Ordnungssystem für die Daten und Bereitstellung von Standard-Funktionalität

Wiederverwendung

Anpassbarkeit: Alternative Implementierungen für gleiche abstrakte Schnittstelle

Optimierung: Alternativen mit verschiedener Leistungscharakteristik



Generell: gute Implementierung

Für jede Programmiersprache gibt es Richtlinien

Teamfähigkeit steigt dramatisch

Wiederverwendung vorhandener Implementierungen, z.B. für Datenstrukturen

Vorlesung Softwaretechnik

9. Generative Softwareentwicklung

Prof. Bernhard Rumpe
Software Engineering
RWTH Aachen

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

SE Software Engineering | RWTH AACHEN UNIVERSITY

Warum, was, wie und wozu betrachten wir generative Softwareentwicklung

Warum?

- Reduziert repetitive Aufgaben für EntwicklerInnen
- Reduziert Konsistenzprobleme
- In der Praxis immer weiter verbreitet

Was?

- Prinzipien generativer Entwicklung & praktisches Beispiel in einer Anwendung
- Aufbau und Funktionsweise eines Generators

Wie?

- Modelle in High-Level Sprachen
- Prinzipien der Code Generierung
- Code Templates
- Generator Framework für Informationssysteme & Anpassungen

Wozu?

- Anwendung in der Praxis
- Effizienzgewinn: Mehr Zeit für spannendere Aufgaben
- Basis für Vertiefende Vorlesungen und Abschlussarbeiten

Softwaretechnik

9. Generative Softwareentwicklung
9.1 Prinzipien generativer SE

Prof. Bernhard Rumpe
Software Engineering
RWTH Aachen

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

Literatur:

- B. Rumpe: Agile Modellierung mit UML: Code-generierung, Testfälle, Refactoring. Springer 2012.
- T. Stahl, Markus Völter: Modellgetriebene Softwareentwicklung: Techniken, Engineering, Mangement. dpunkt 2005.

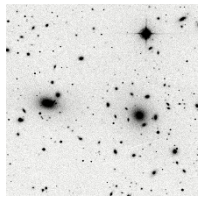
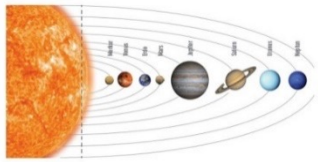
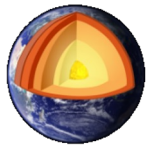
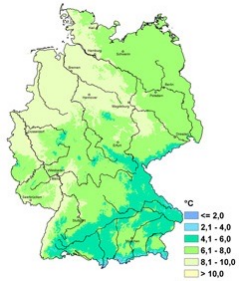
SE Software Engineering | RWTH AACHEN UNIVERSITY

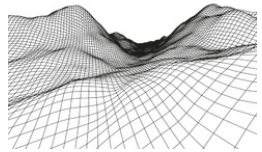
Generative Software Engineering

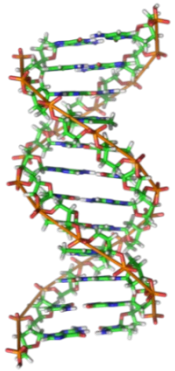
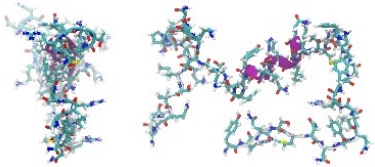
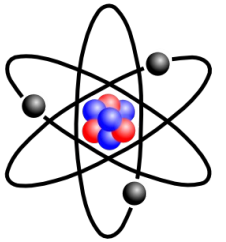
Generative Software Engineering (GSE) ist eine

- Methode der effizienten Generierung von (Teilen von) Software durch Generatoren
- auf Grundlage von Modellen
  - wie z.B. der UML oder einer DSL (domain-specific language)
- mit dem Ziel,
  - die Qualität zu erhöhen und
  - die Entwicklungszeit zu senken.

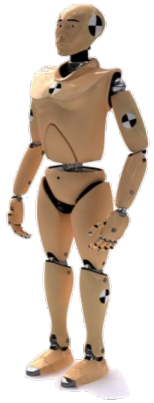
Modelle werden in allen Disziplinen verwendet

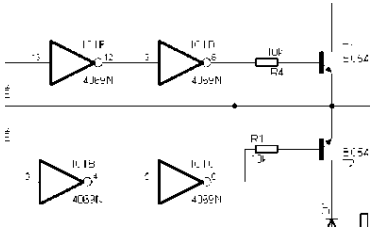
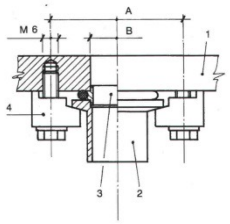
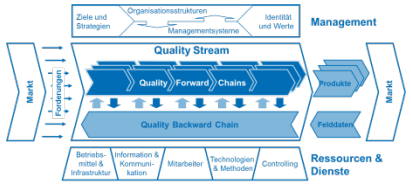





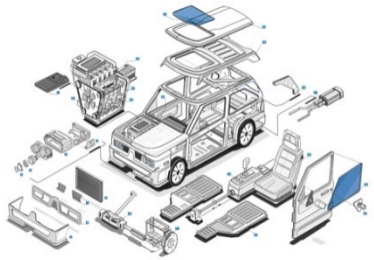


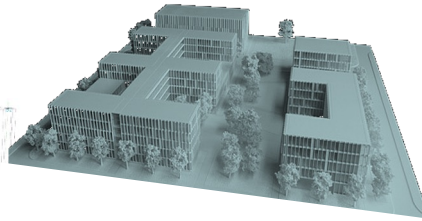




A model is an abstraction of an original made and used for a purpose.



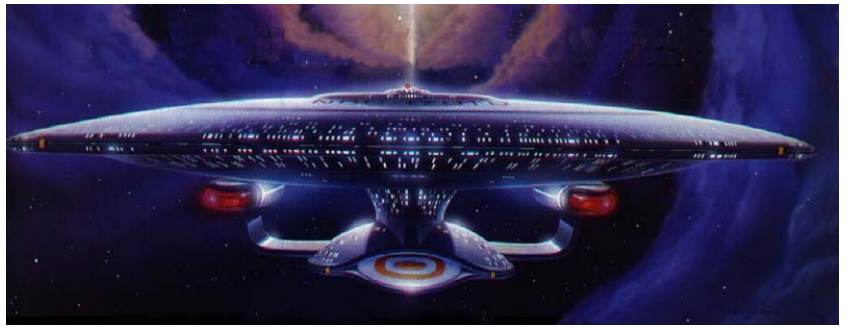


What is a Model?

> Ein Modell ist seinem Wesen nach eine in Maßstab, Detailliertheit und/oder Funktionalität verkürzte beziehungsweise abstrahierte Darstellung des originalen Systems.
>
> *Stachowiak 1973*

> A model is essentially a reduced or abstracted representation of the original system in terms of measure, precision and functionality.
>
> *Stachowiak 1973*

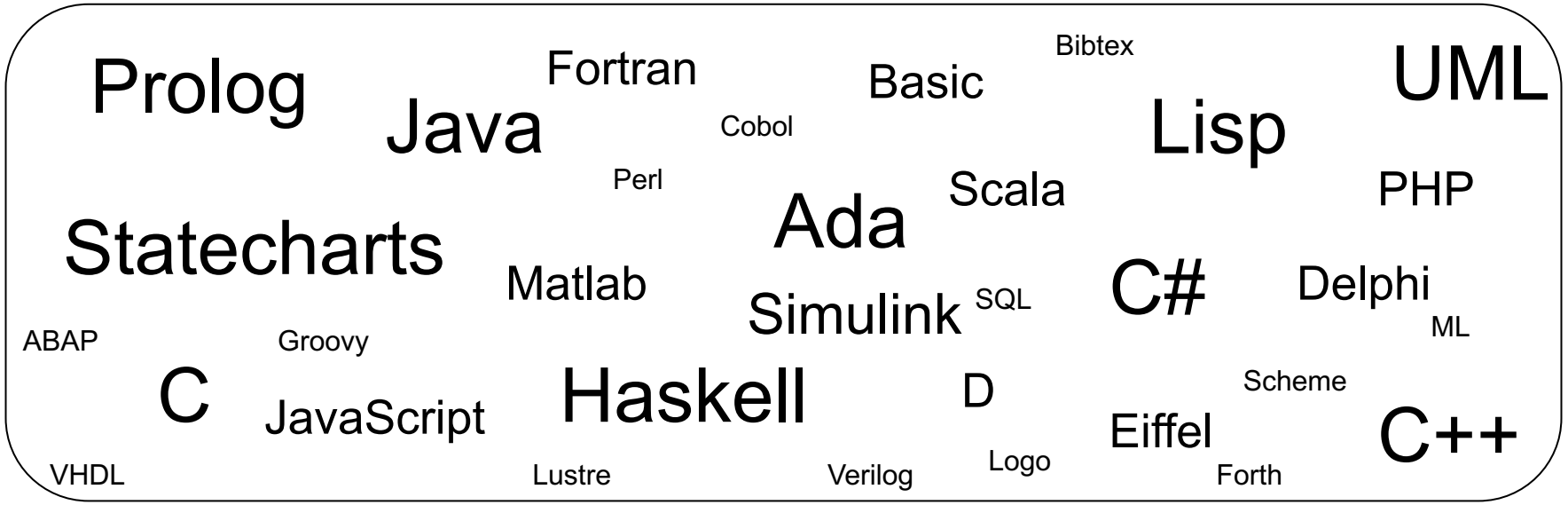
What is not a Model?



Sprachen für die Softwareentwicklung

- Sprachen sind auf unterschiedlichen Ebenen wichtig für die Softwareentwicklung
  - Modellierungssprachen wie UML
  - Programmiersprachen wie Java
  - Natürliche Sprachen wie Englisch oder Deutsch für die Dokumentation

- Die Softwareentwicklung kennt viele Sprachen:



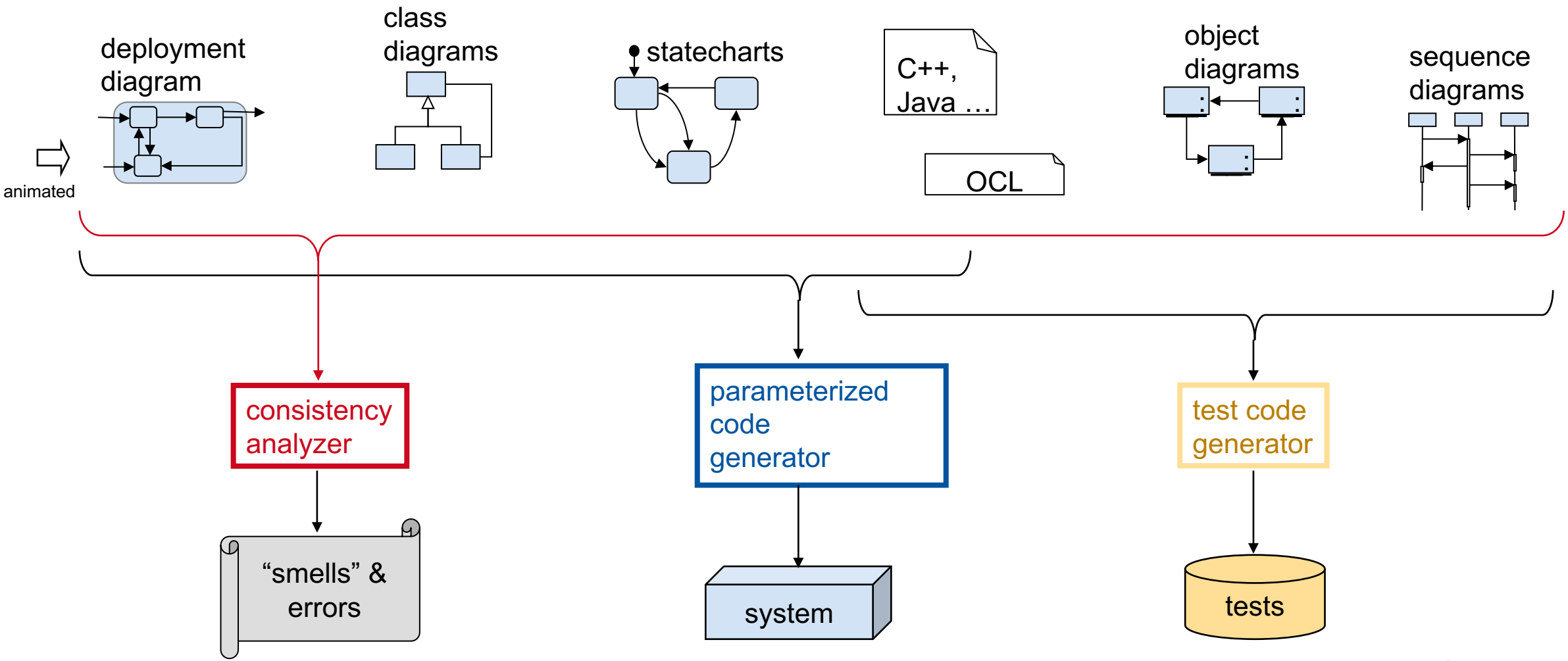
Software Languages

A software language is a human readable and computer processable language addressing a particular problem.

- Examples:
  - UML: a general purpose modeling language
  - Java: a general purpose programming language
  - XML: a format for structuring data
  - or Domain Specific Languages (DSLs)

A modeling language is a software language used for modeling software or systems.

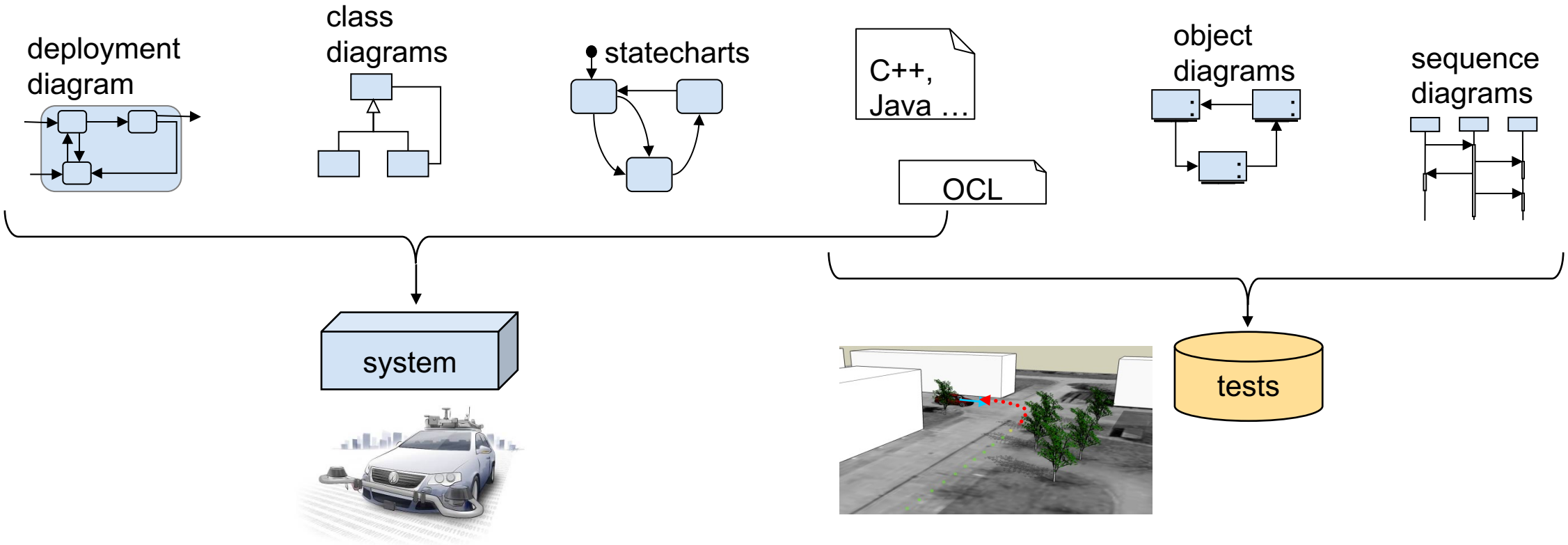
Constructive use of Models for Coding and Testing: Usage of UML-Diagrams



see: B. Rumpe: Agile Modellierung mit UML, Springer Verlag 2017

Model-based Simulation for SE

- Test-Infrastructure needs simulation of its context:
  - context can be: geographical, sociological, etc.
- Simulation helps to understand complexity



Domain Specific Language (DSL)

A domain-specific language (DSL) is a software language specialized to a particular application domain.

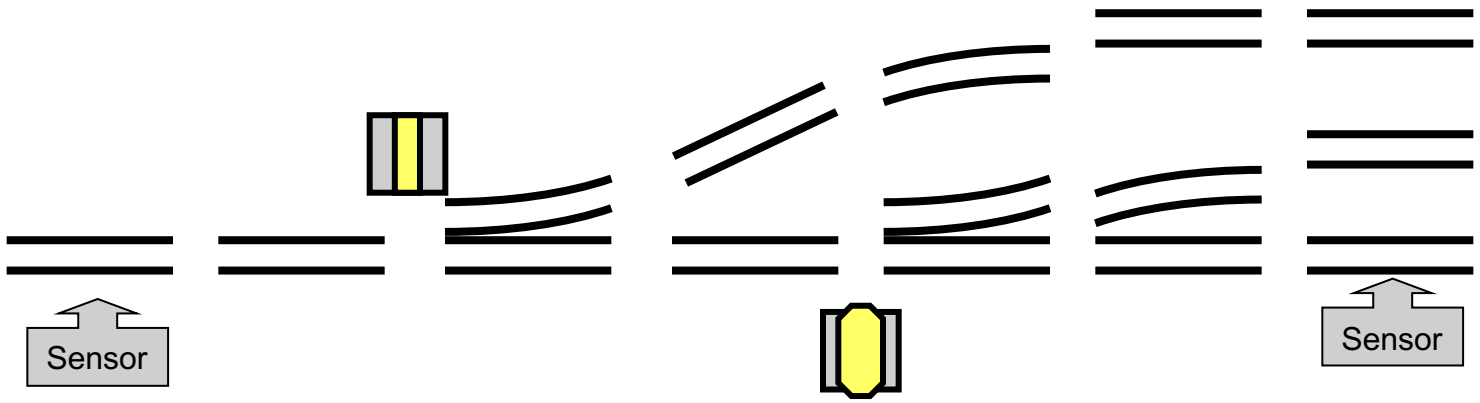
- allows us to model application domains like
  - business, telecommunication, traffic, ...
- focuses more on the application domain than the technical solution
- is not necessarily executable
  - if it is executable, it is often not Turing complete

A general-purpose language (GPL) in contrast is broadly applicable across domains and lacks specialized features for a particular domain (such as C++, Java, …).

Domänenspezifische Sprachen

- Domänenspezifische Sprachen ermöglichen die abstrakte Modellierung von Problemdomänen, wie:
  - Automobile, Geschäftsprozesse, Telekommunikation, Produktionsanlagen, Planung von Gleisnetzen ...

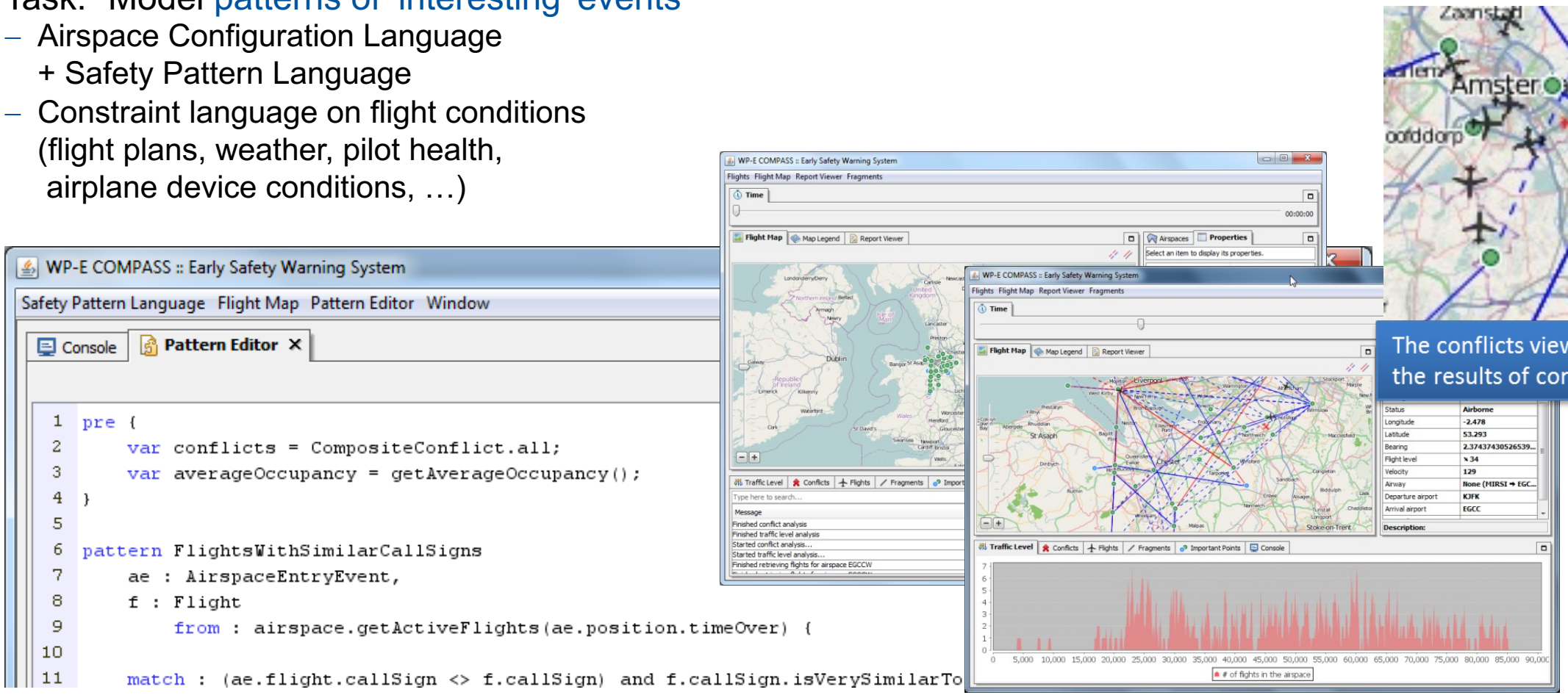
- Beispiel: Ein Gleisnetz-Steuersystem beinhaltet
  - Gleise, Weichen, Signale, Sensoren



  - Ein Framework bietet für jede Entität der Domäne eine Softwarekomponente
  - Eine grafische DSL definiert die Konfiguration der Komponenten

Example: Air Traffic Management

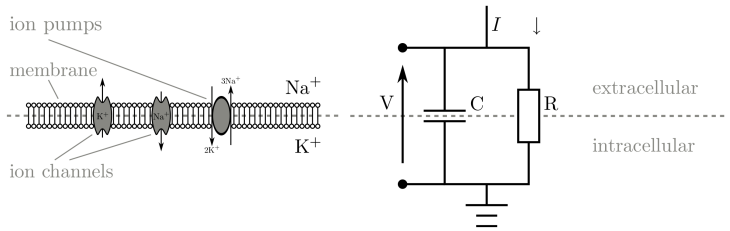
- Task: “Model patterns of ‘interesting’ events”
  - Airspace Configuration Language
    + Safety Pattern Language
  - Constraint language on flight conditions
    (flight plans, weather, pilot health,
    airplane device conditions, …)



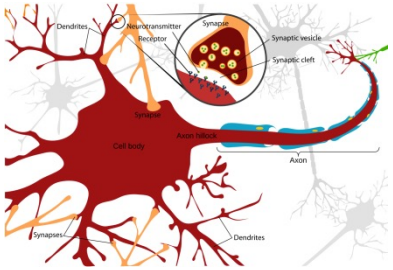
with A. Horst

Example: NESTML – Neuron ML + Simulator

- Modeling of
  - spikes



  - dendrites, axons, neurons



  - networks with 10^7 – 10^9 neurons

- Part of EU – Human Brain Project
- Languages: specific DSL

```
package nestml:
  import units.unitless.*

  component Math:
    internal:
      PI real = atan(1)*4
```

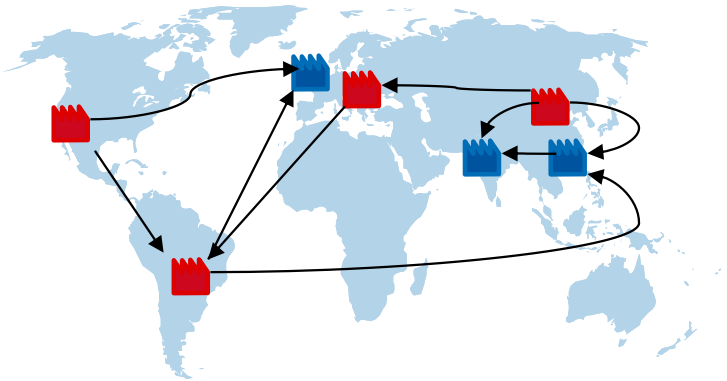
```
neuron MyNeuron:

  dynamics timestep (t ms):
    // check refractory
    if not Refr.isRefractory():
      V_m = P33 * V_m  // leaky
      V_m = V_m + P30 * (y0 + I_e) + P31 * y1 + P32 * y2
    else:
      Refr.decreaseSteps()
    end
    // alpha shape synapse
    y2 = P21 * y1 + P22 * y2
    y1 = y1 * P11
    Logging.debug
    y1 = y1 + PSC
```

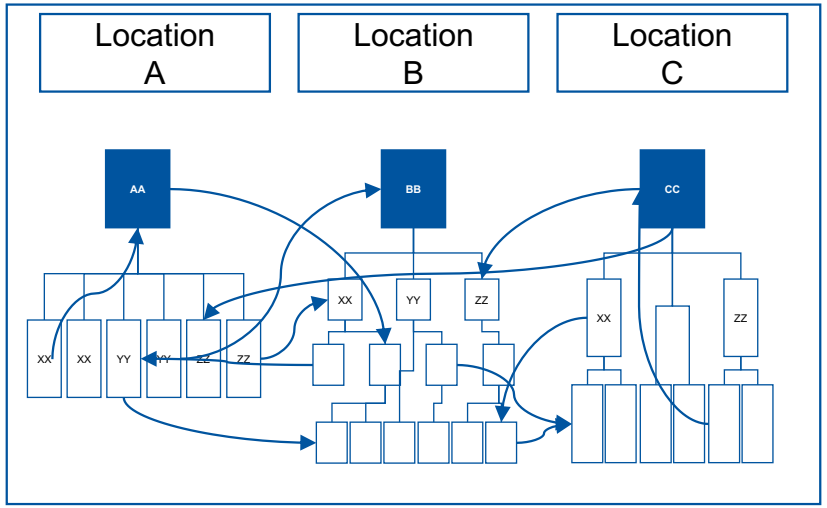
```
unit foo Integer [ 0    ... inf    )
unit bar Real    ( -inf ... 21.3e3 ]
```

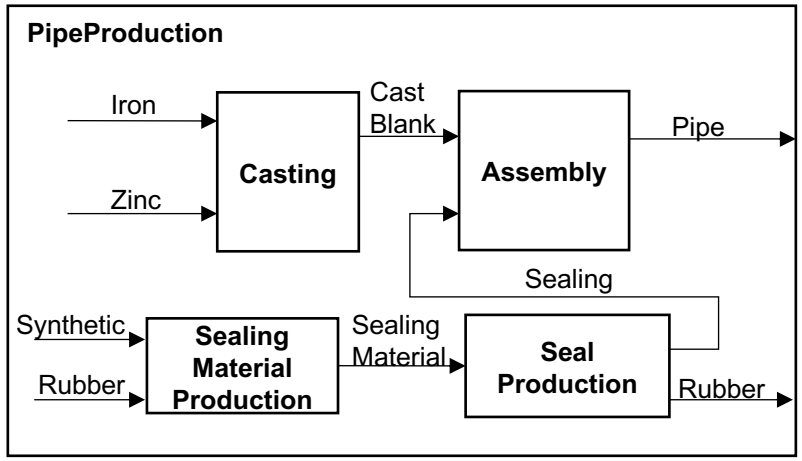
with M. Diesmann, A. Morrison, I. Blundell, M. Look, D. Plotnikov

Example: ProNet^{sim} for Logistics

- Modeling and simulation of process chains and production nets
  - Globally distributed
  - Hierarchy: factories, production units, …
- Analysis through simulation of scenarios
  - Variance analysis
  - Critical paths
- Modeling of process chains as architectural component nets
- DSL based on:
  - MontiArc + Statecharts + Java-Stmts.





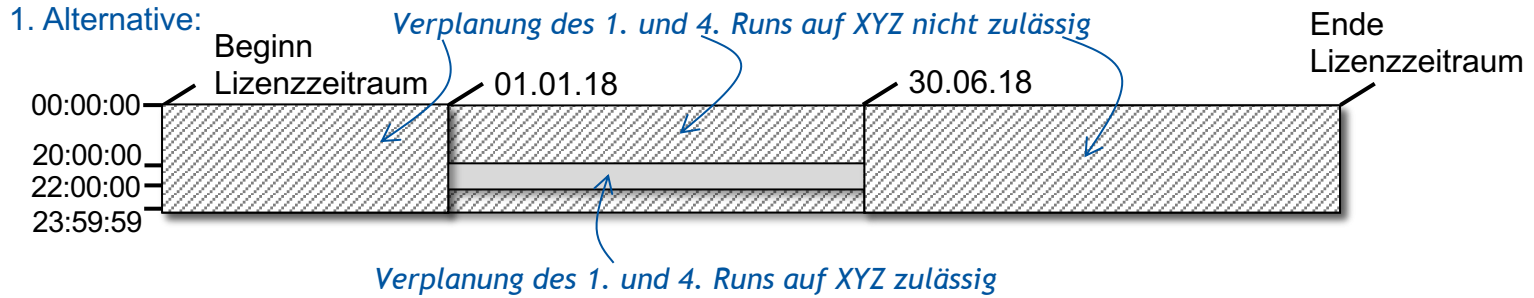


with A. Haber, WZL

Example: DSL for planning of the TV program

- Planning audio-visual offers such as TV program and video-on-demand
- Restrictions in licensing contracts
- Risks: Misinterpretation and resulting planning errors
- DSL for
  - Verification of plans
  - Calculation of allowed planning periods

```
1 Alternative 1:
2 Der 1. & 4. Run innerhalb von 01.01.2018 bis 30.06.2018 turnus immer von 20:00 bis
3 22:00 auf Nutzer XYZ
4 Alternative 2:
5 Alle Run innerhalb von 01.01.2018 bis 30.06.2018 turnus ohne auf Nutzer XYZ
```

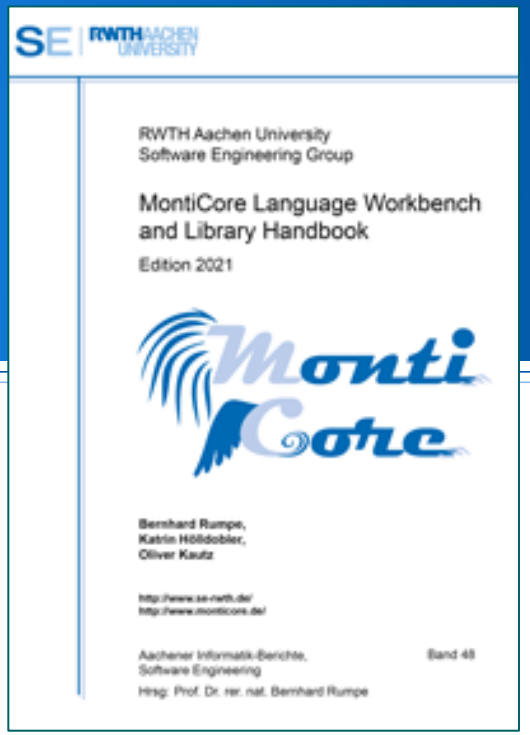


with I. Drave, K. Hölldobler, O. Kautz

Mit passendem Tooling ist eine Modellierungssprache im Entwicklungsprozess effizient einsetzbar

Softwaretechnik

9. Generative Softwareentwicklung
9.2 Aufbau eines Generators am Beispiel von MontiCore



Prof. Bernhard Rumpe
Software Engineering
RWTH Aachen

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

Literatur:

- [HKR21] K. Hölldobler, O. Kautz, B. Rumpe: MontiCore Language Workbench and Library Handbook: Edition 2021. Aachener Informatik-Berichte, Software Engineering Band 48. Shaker Verlag, May 2021.

- https://monticore.de/download/handbook.pdf

SE Software Engineering | RWTH AACHEN UNIVERSITY

Generative Software Engineering

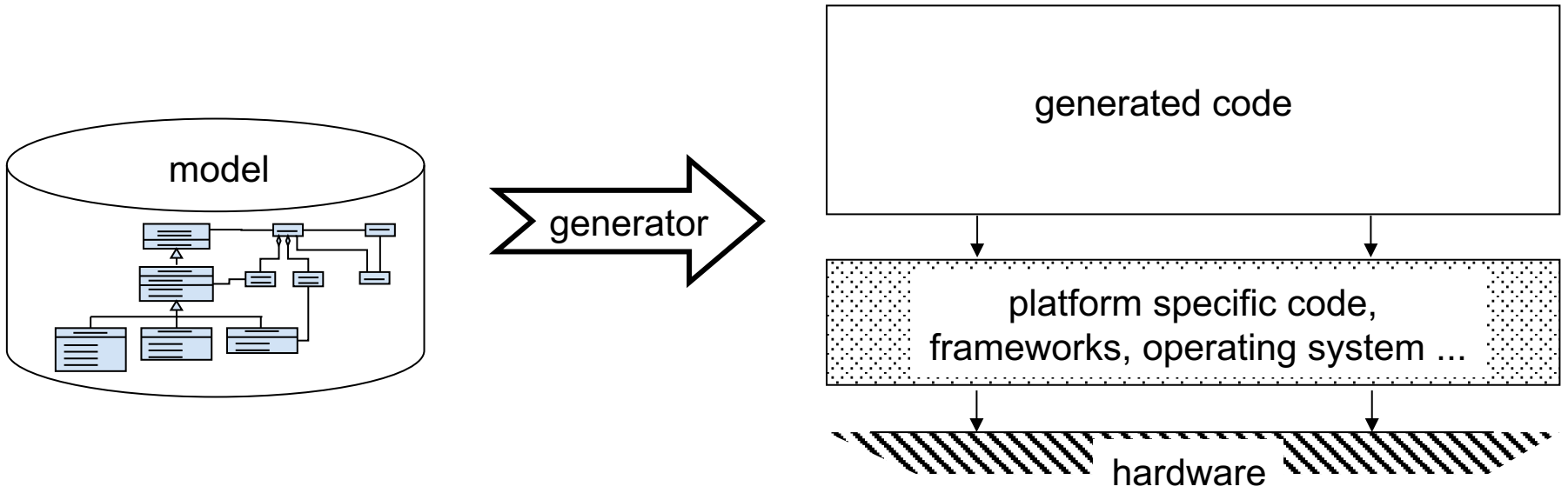
Generative Software Engineering (GSE) ist eine

- Methode der effizienten Generierung von (Teilen von) Software durch Generatoren
- auf Grundlage von Modellen
  - wie z.B. der UML oder einer DSL (domain-specific language)
- mit dem Ziel,
  - die Qualität zu erhöhen und
  - die Entwicklungszeit zu senken.

- Mit DSLs können Domänenexperten das Softwaresystem direkt verstehen, validieren und bearbeiten.
- UML- bzw. DSL-Modelle beschreiben das Softwaresystem in einer verständlichen, intuitiven Weise.
- Vorgefertigte bzw. selbstentwickelte Generatoren generieren Code und Tests auf Grundlage der Modelle.

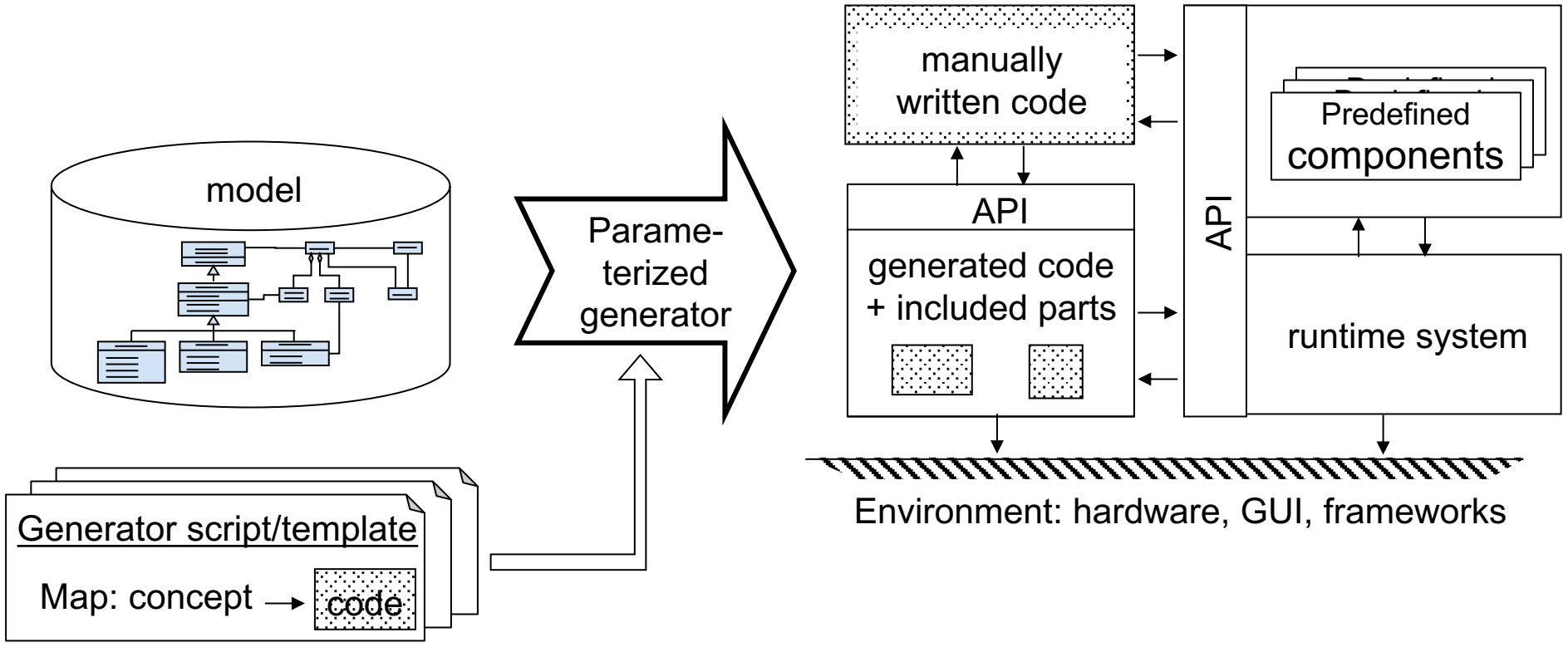
Prinzipien der Code-Generierung

- Grundsätzlich: Abbildung von Modell auf Code

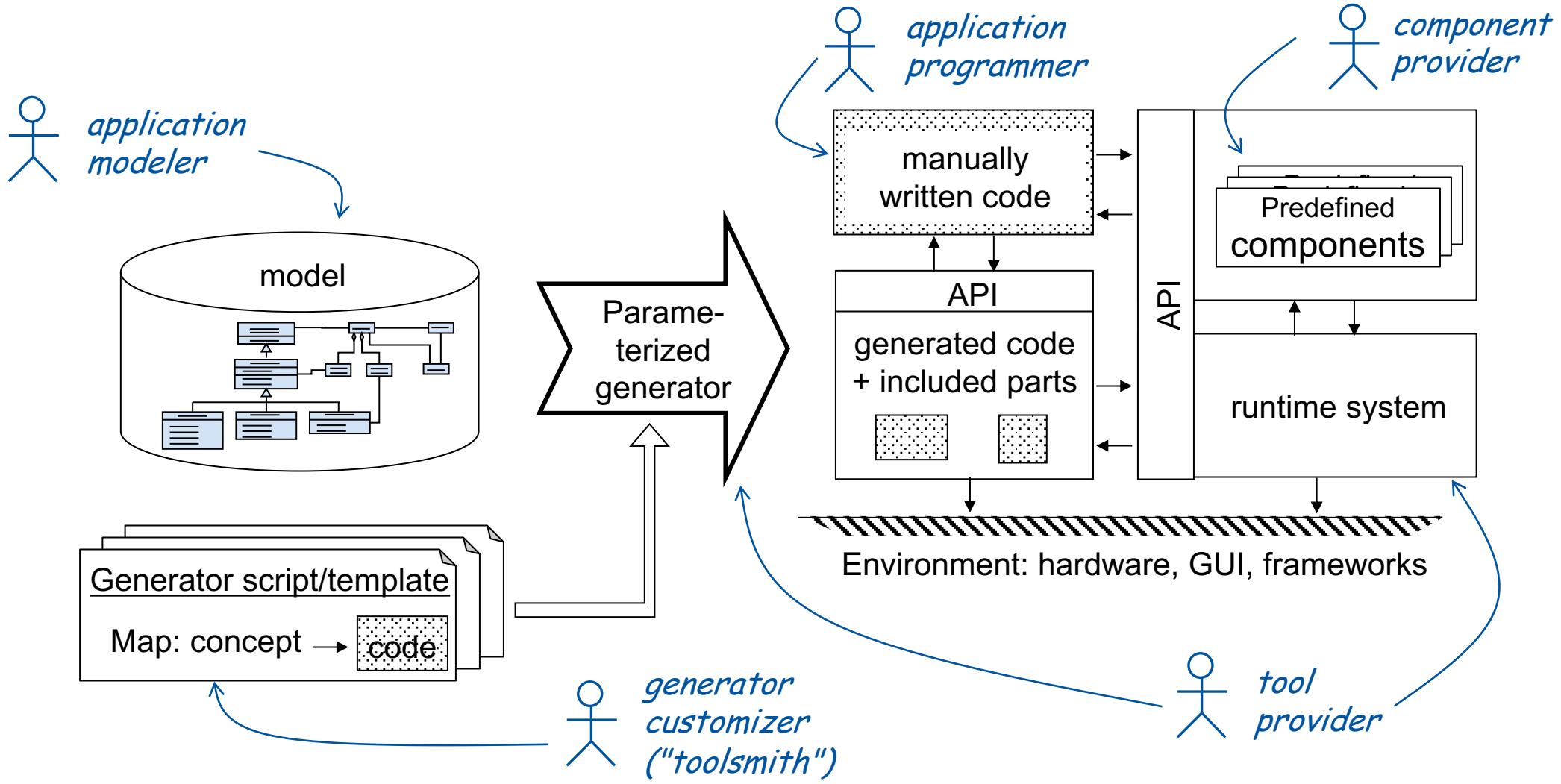


Generator bildet Modelle auf den daraus erzeugten Code ab
Problem 1: kann der Code komplett generiert werden?
Problem 2: der generierte Code ist plattformspezifisch?

Funktion eines Generators

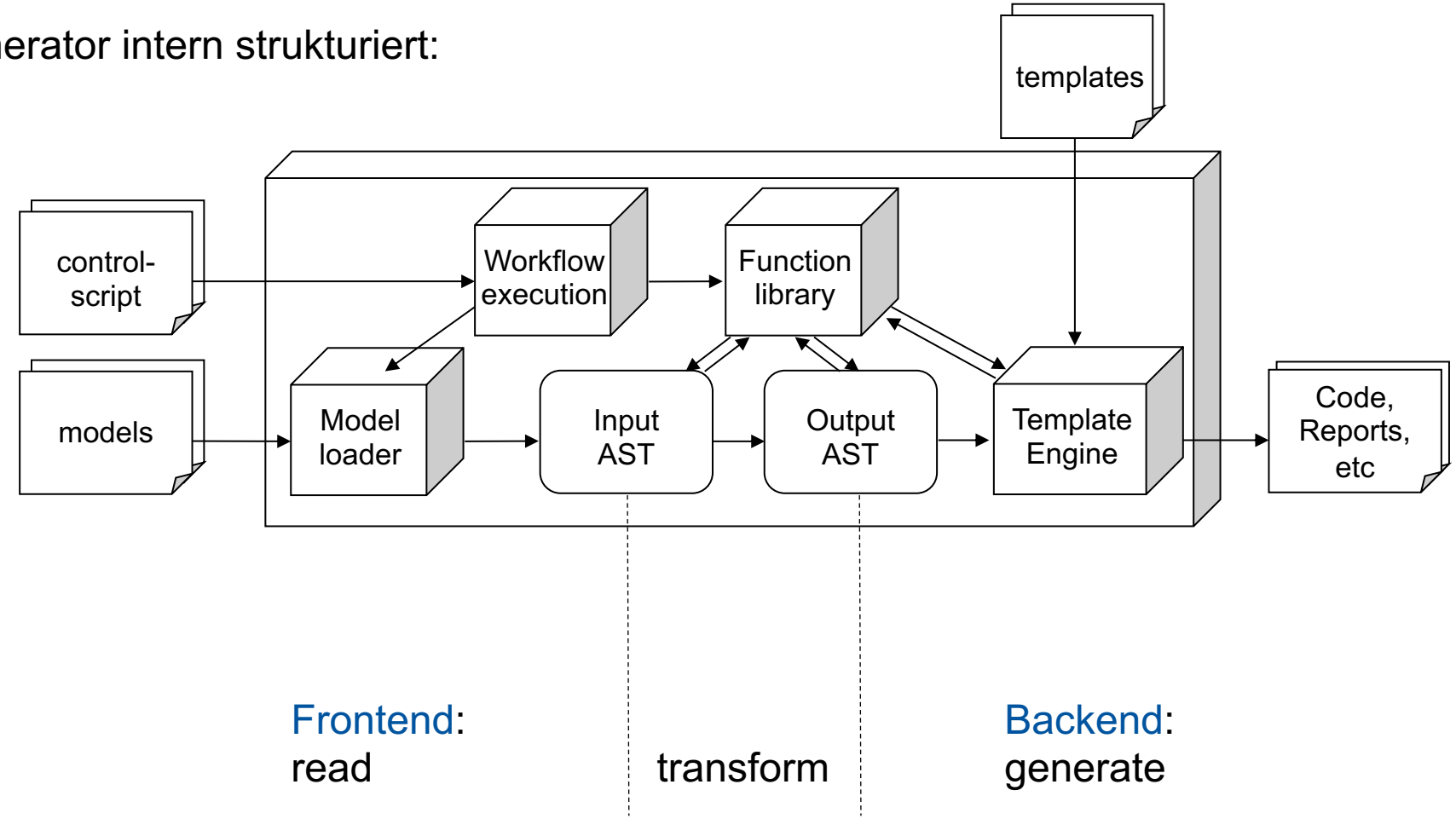


Rollen bei generativer Softwareentwicklung



Generator Architektur

- So ist ein Generator intern strukturiert:



Language & Tooling Workbench MontiCore

- Definition von modularen Sprachfragmenten
- Interfaces zwischen Modellen/Sprachfragmenten
  - Namespaces, typing (~ Java, UML)
  - “kinds” + Signaturen
- Unterstützung bei der Analyse
- Unterstützung von Transformationen
- Pretty printing, editors (graphical + textual)

- Sprachkomposition:
  - unabhängige Sprachentwicklung
  - Komposition von Sprachen und Tools
  - Spracherweiterung
  - Sprachvererbung (ermöglicht Ersetzungen)

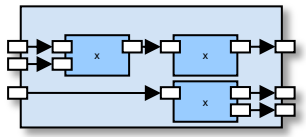
https://monticore.de/
https://monticore.github.io/

- Einfache und schnelle Entwicklung von Domänenspezifischen Sprachen (DSLs)
  - durch Wiederverwendung existierender
  - Variabilität in Syntax, Context Conditions, Generierung, Semantik

MontiCore: Auswahl an Sprachen

- MontiCore
  - using Bootstrapping

- UML / SysML
  - Class diagrams
  - Object diagrams
  - Statecharts
  - Sequence diagrams
  - OCL, logic



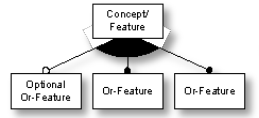
- MontiArc
  - Architectural models / ADL, function nets
  - + automata + Java + views
  - Embedded & Internet Of Things Versions

- Java, C++

- Artificial Neural Network Architectures (MontiANNA)

- MontiCore transformations
  - Pattern matching | Extended by Java

- FeatureDSL
  - Feature diagram & config.



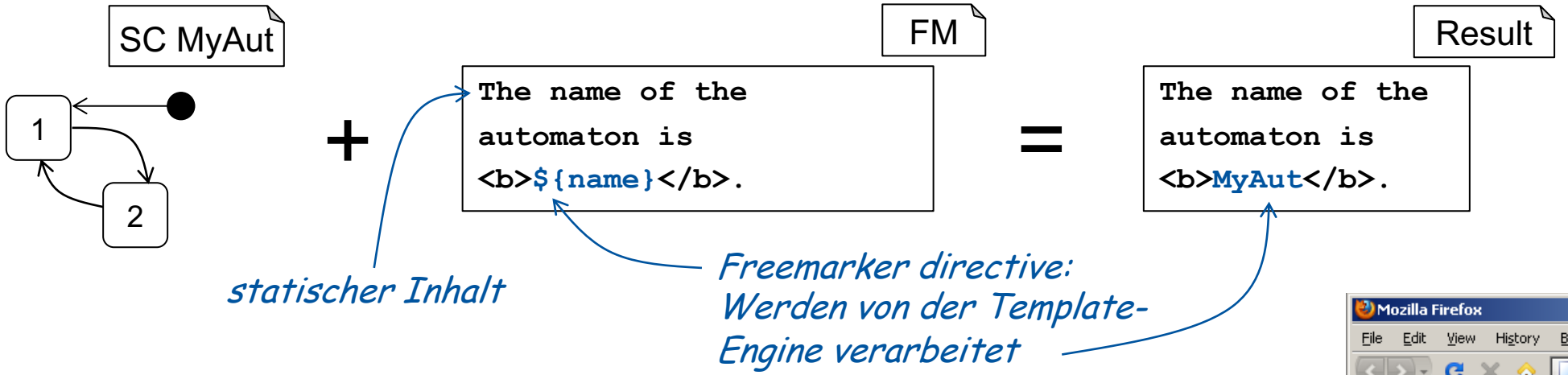
- AutosarDSL
  - Components, deployment, interfaces

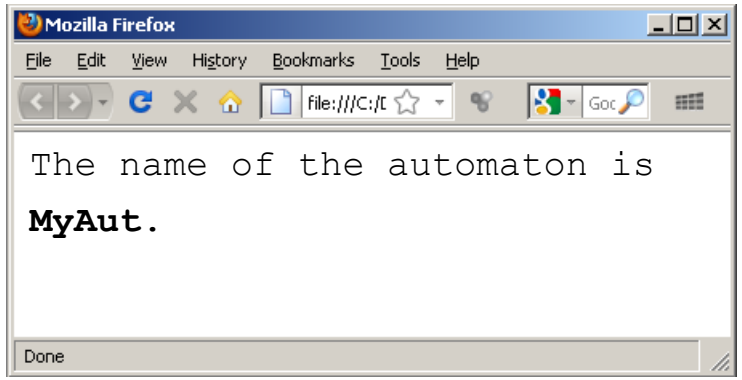
- Flight control: constraint language
- Physical SI Units: for scientific models
- TV Contract; Tax Laws

- Cloud Service Configurator
  - Management of Services
- BPMN

Freemarker Templates (für das Generator-Backend)

- Codegenerierung ist gekennzeichnet durch eine Kombination aus AST und Template
- In unserem Fall: Template wird mit einem AST als Datenmodell aufgerufen





- Interpretation des Ergebnisses
  - z.B., als HTML

FreeMarker Beispiel: Generierung einer Factory

FM Factory.ftl

```
01 package ${ast.getPackage()};
02
03 <#assign cname = ${ast.name}>
04 public class ${cname}Factory {
05
06   protected static
07     ${cname}Factory f = null;
08
09   protected ${cname}Factory() {}
10
11   public static ${cname} create() {
12     if (f == null) {
13     …
```

Generated Java

```
21 package a.b;
22
23
24 public class PersonFactory {
25
26   protected static
27     PersonFactory f = null;
28
29   protected PersonFactory() {}
30
31   public static Person create() {
32     if (f == null) {
33     …
```

- Basierend auf UML Klassendiagramm mit Klasse `Person` in Packet `a.b`

- `${ast.getPackage()}`
  - Java-like Expressions ergeben einen String (**`ast`** ist ein Objekt der abstrakten Syntax des CD)
- `<#assign cname = ${ast.name}>`
  - Definition einer lokalen Variable

Freemarker – Template Languages

- Freemarker templates are stored as files with extension .ftl

A template consists of three languages that are interwoven

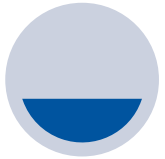
- The target language:
  - Can be any language; Freemarker knows nothing about it
  - (e.g. html, latex, Java, etc.)
- The FM directives language:
  - Controls the output (provides loops, conditionals, template calls etc.): `<#if …>, <#assign …>`
- Expression language with callbacks to the underlying data model
  - looks like `${java-like expression}`

FM

```
1 The name of the
2 automaton is
3 <b>${ast.getName()}</b>.
```

- Ftl is a typical example for interweaving of languages.
- Don't confuse the sub-languages: e.g. Java as desired target vs. Java-like callbacks to the tool-data structures in the FM expression language!

Was haben wir gelernt?



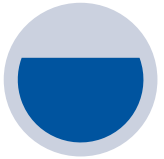
Generatives Software Engineering

… ist die effiziente Generierung von (Teilen von) Software durch Generatoren

Modelle als Basis für die Generierung: GPL wie z.B. UML, SysML oder eine DSL

Generierung in eine oder mehrere Zielsprachen

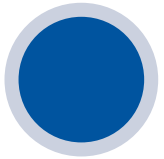
Ziel: Mehr Qualität & Effizienz



Software Sprachen

Menschen lesbare und Computer verarbeitbare Sprache für ein konkretes Problem

- Programmiersprachen wie C++ oder Java
- Modellierungssprachen zur Modellierung von Software oder Systemen
- Domänen-spezifische Sprache (DSL) entwickelt für einen speziellen Anwendungsbereich (vs. general purpose languages (GPL))



Generator

Funktionsweise

- Modelle laden -> Abstrakter Syntaxbaum (AST)
- AST transformieren
- Generierung von Code durch Nutzung von Templates
- z.B. Freemarker Template Sprache

Softwaretechnik

9. Generative Softwareentwicklung
9.3. Generierung von Informationssystemen

Prof. Bernhard Rumpe
Software Engineering
RWTH Aachen

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

Software Engineering | RWTH AACHEN UNIVERSITY

Information System

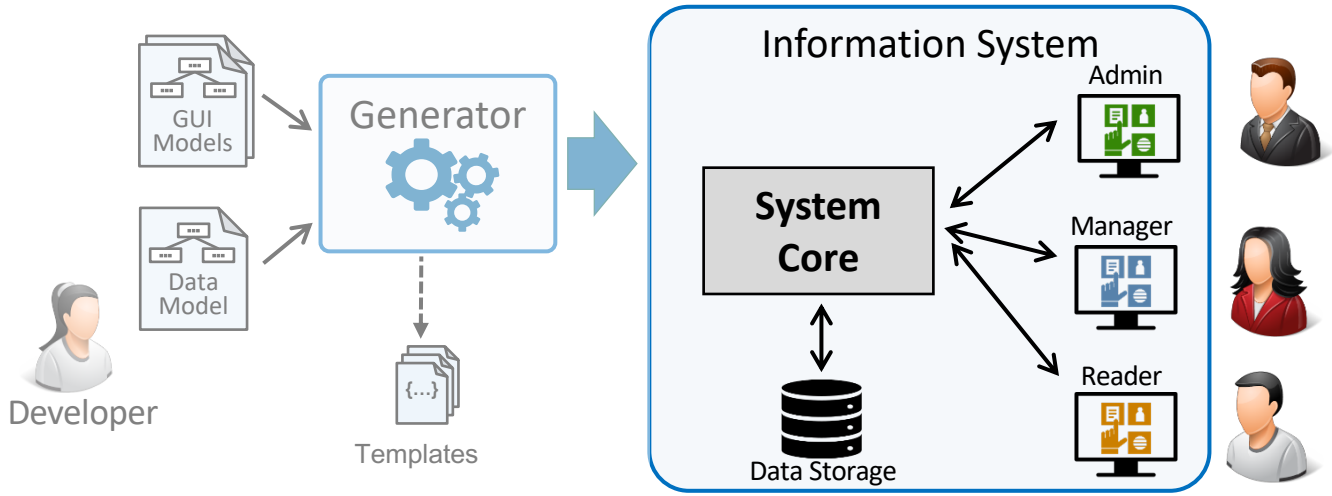
Information systems (**IS**) are formal, sociotechnical, organizational systems designed to collect, process, store, and distribute information. In a sociotechnical perspective, information systems are composed by four components: task, people, structure (or roles), and technology.

Main system goal

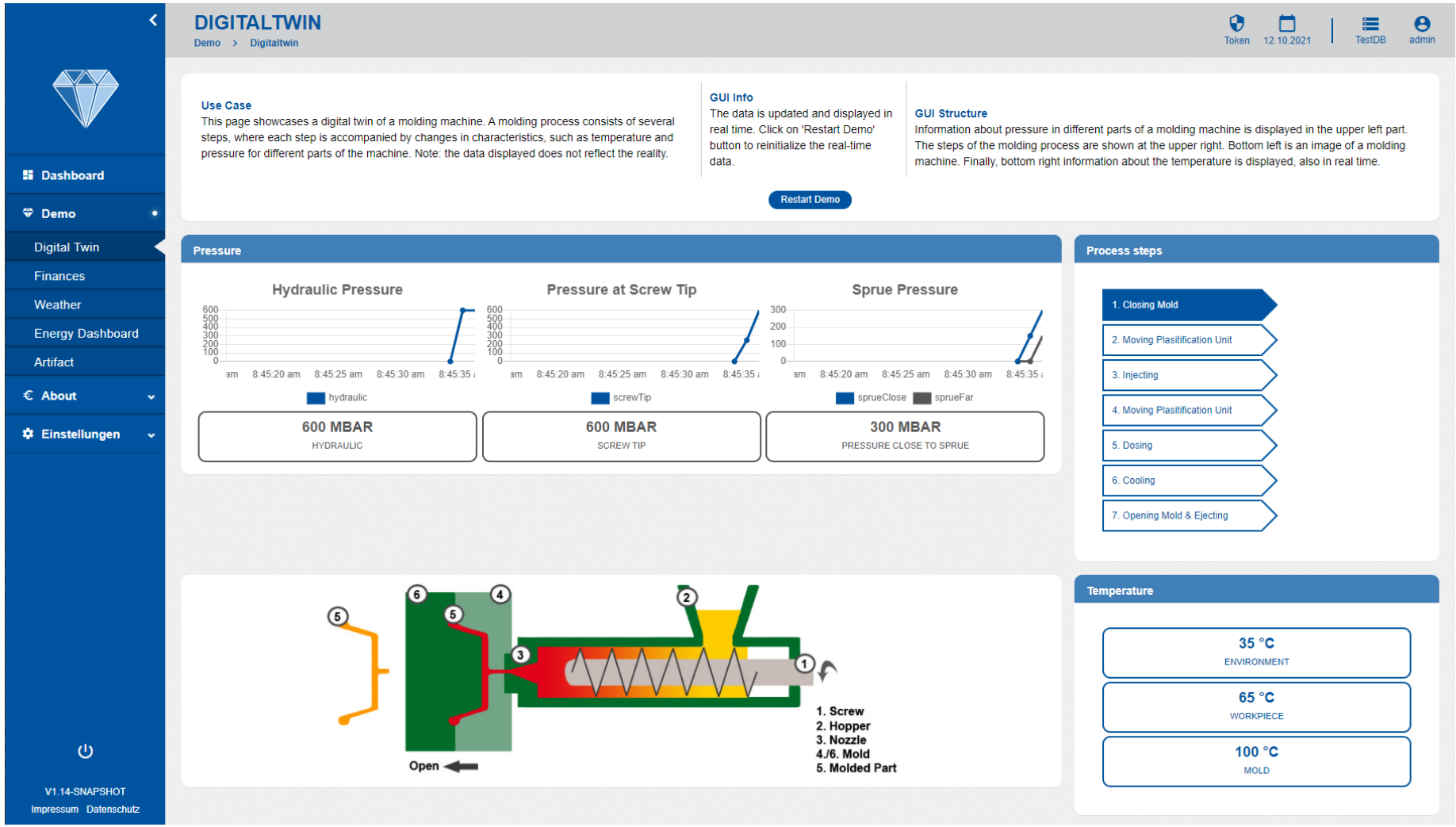
- **Provide information**
- More precise: **create**, **read**, **update**, **delete** data (CRUD)

MBSE Solution

- using a Generator Framework, like MontiGem



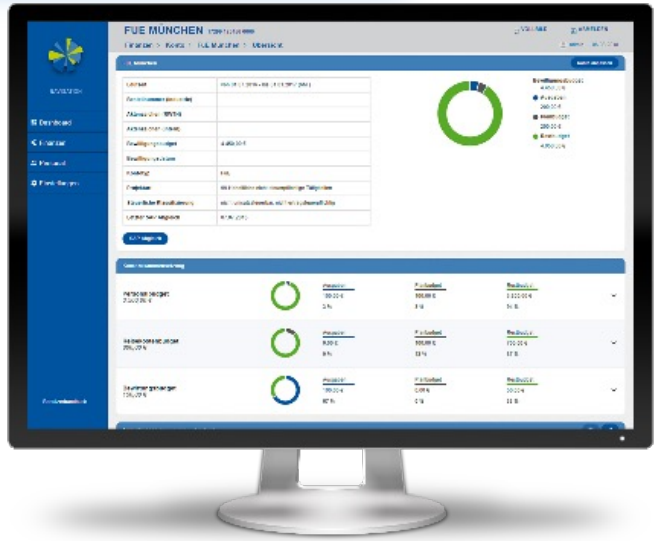
Beispiel: MontiGem generiert einen Digitalen Zwilling



Digital Twin Beispiel: Anwendungsfall Spritzguss
(JM, MD, AW, SVa, LN, AG, MH)

MontiGem

MontiGEM is a framework that provides support to generate large parts of a web application for a domain specific Enterprise Information System consisting of data base, persistence layer, command infrastructure and graphical user interface based on the input of models.

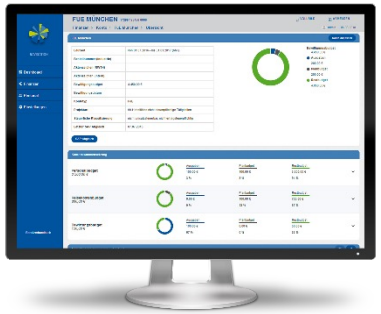


Architecture of the (Generated) Enterprise Information System

CLIENT

Angular
TypeScript-based open-source web application framework …
In the client's browser

Presentation
Views, Forms



FRONTEND

Apache HTTP
Open-source cross-platform web server


Server

BACKEND

Apache TomEE
Java Enterprise Edition of Apache Tomcat

Apache Shiro
security framework for authentication, authorization, sessions

Hibernate
classes to database tables, loading/storing objects

Application Core

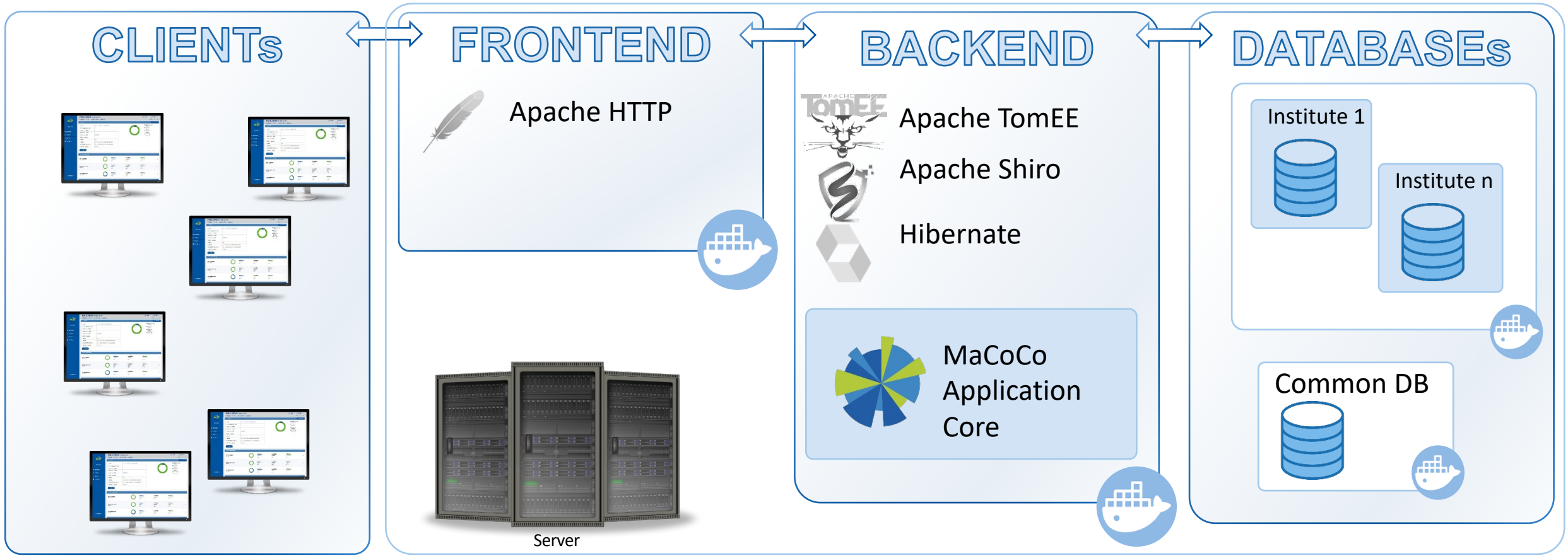
DATABASE

MySQL
Open-source relational database management system

Used Technologies and Software Stack

MontiGem (AG, MH, JM, LN, SVa, GV)

Architecture of the MaCoCo instances

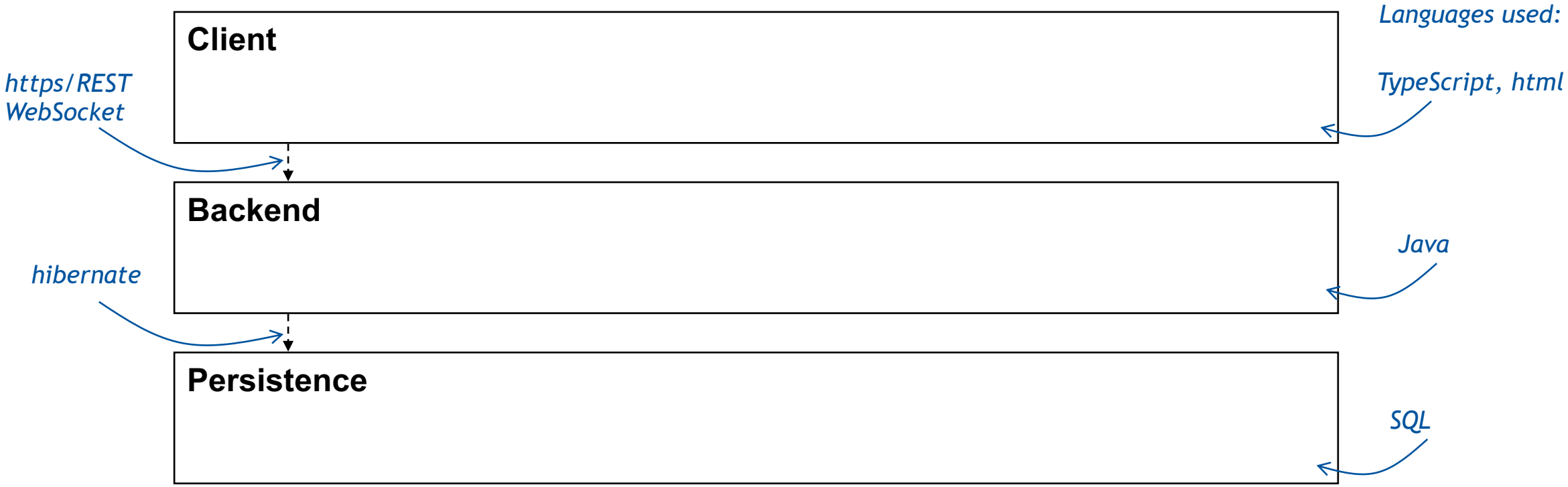


Institutes have separated Data Spaces (that they can maintain themselves)

MaCoCo (AG, MH, JM, LN, SVa, GV, PH, PL)

Generated Product

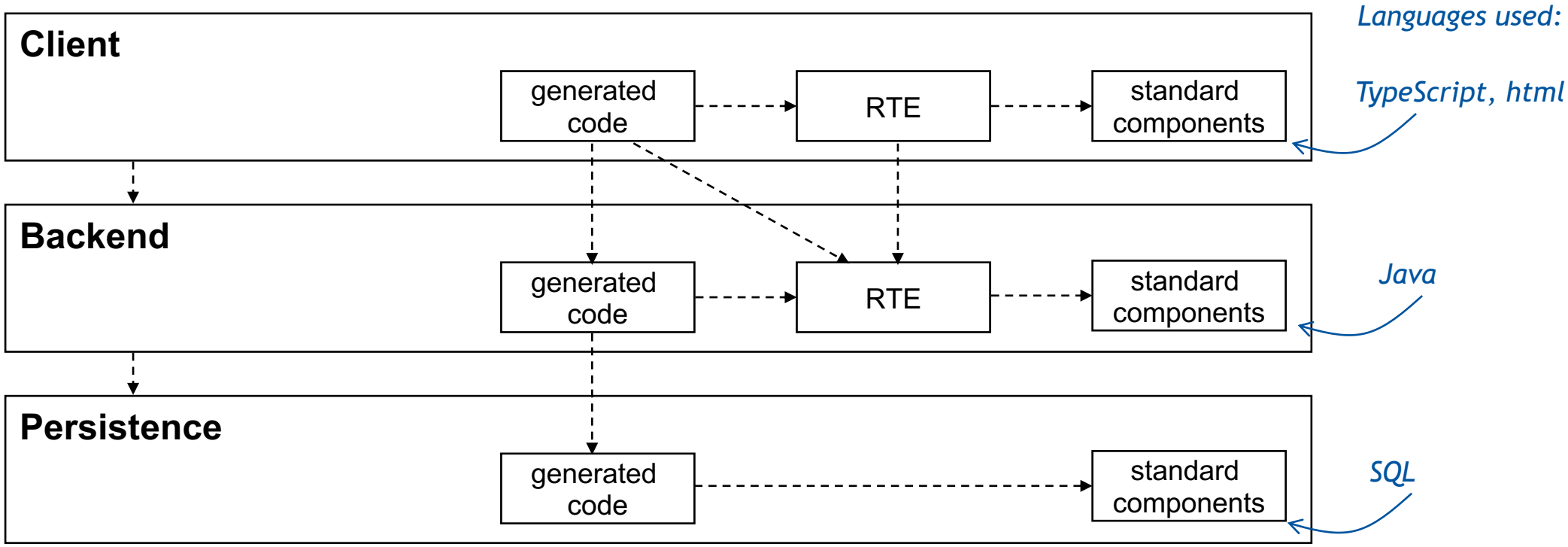
- MontiGEM product architecture has three layers



Legend: ----► static knowledge (e.g. import)

Generated Product

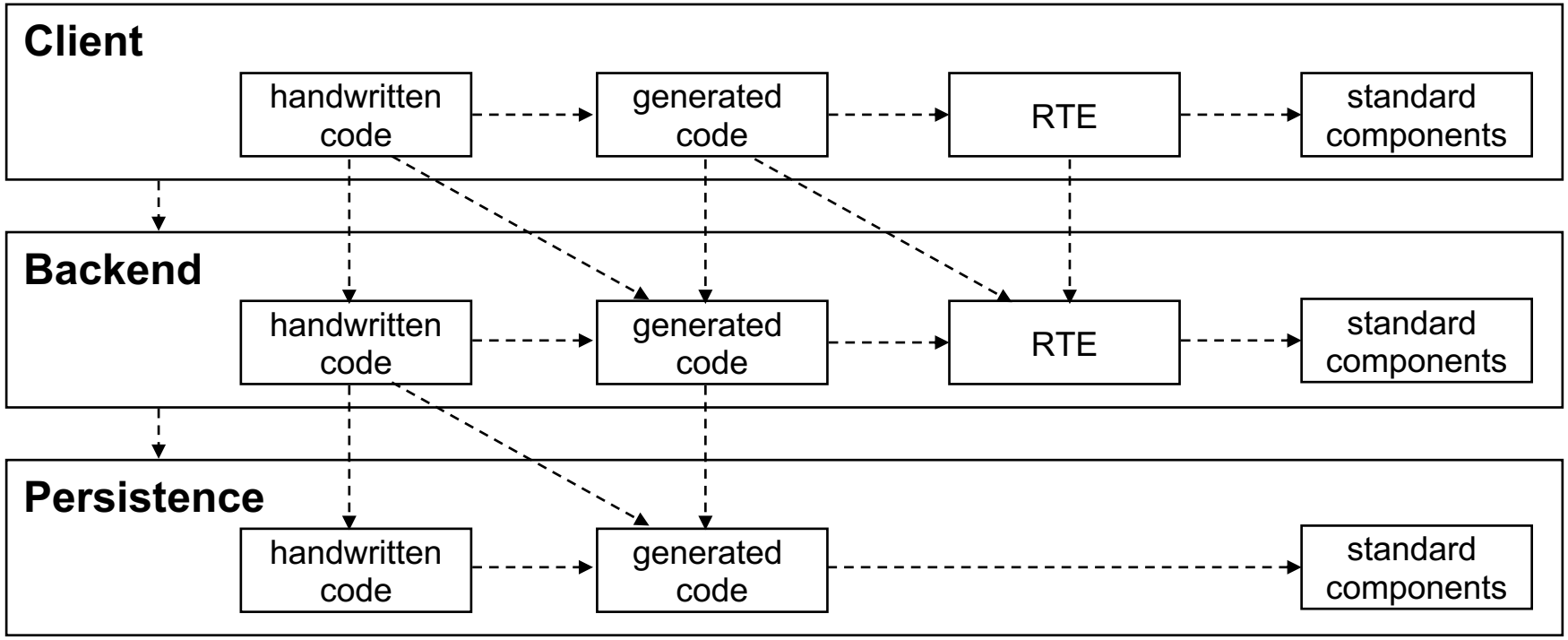
- Generated code: depends on the models describing the domain (e.g. class diagram)
- RTE: is shipped together with the generator (but independent of the model)



Legend: ----→ static knowledge (e.g. import)
RTE: runtime environment

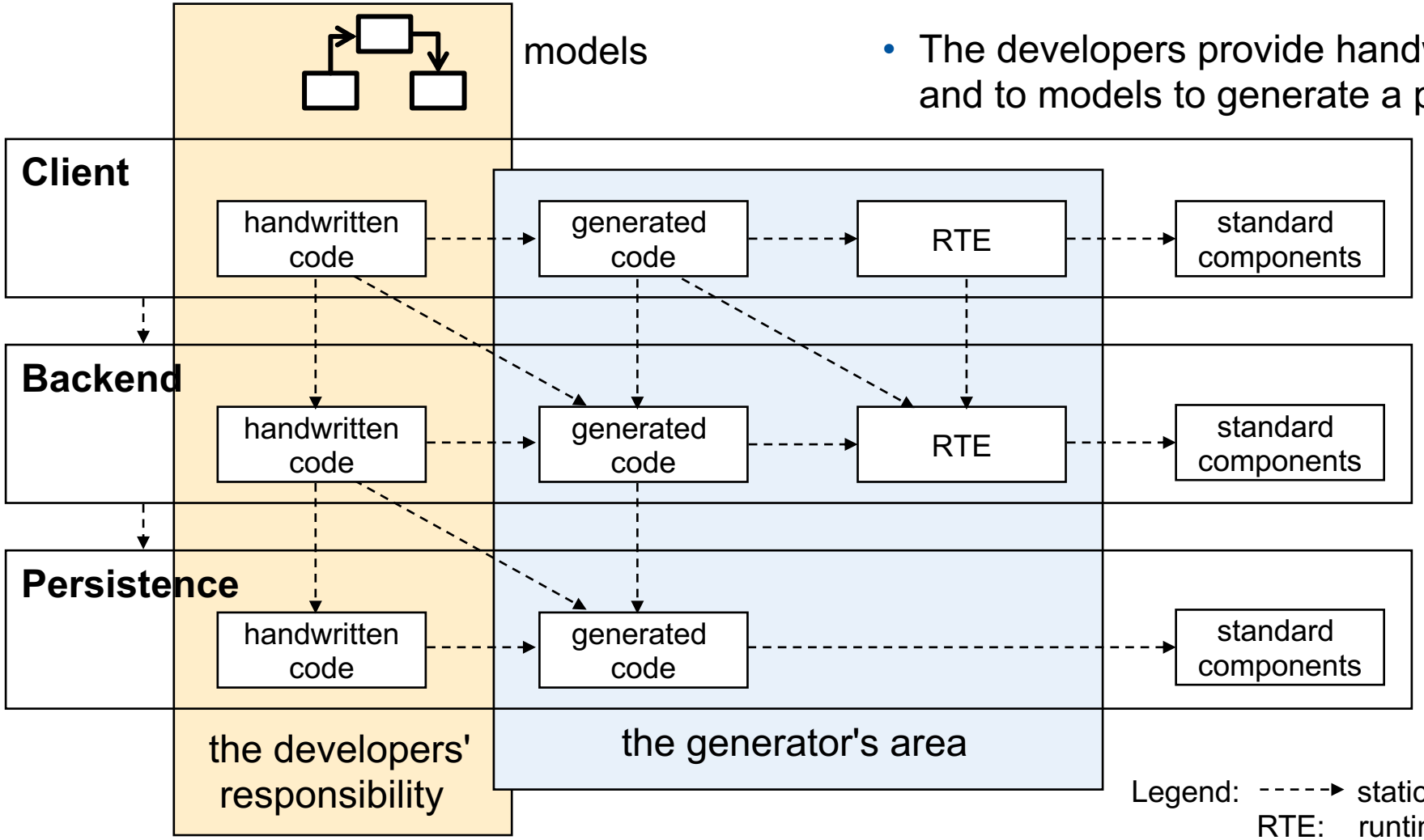
Generated Product

- The developers also add handwritten code: not everything can/should be generated



Legend: ----→ static knowledge (e.g. import)
RTE: runtime environment

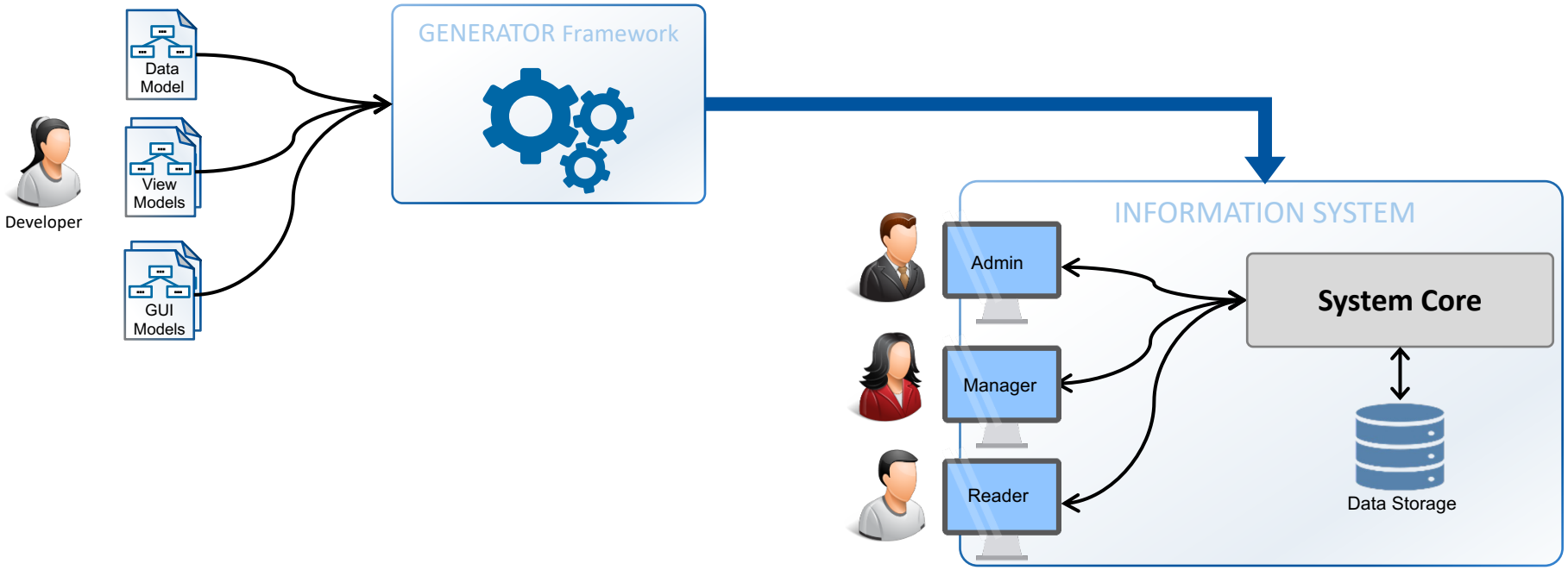
Generated Product



- The developers provide handwritten code and to models to generate a product

Legend: ----→ static knowledge (e.g. import)
RTE: runtime environment

Model-Driven Software Engineering of Information Systems



1 Define models

- Required: Class Diagrams (Data structure), GUI Models (Views)
- Optional: OCL (Input data validation), BPMN (Workflows)

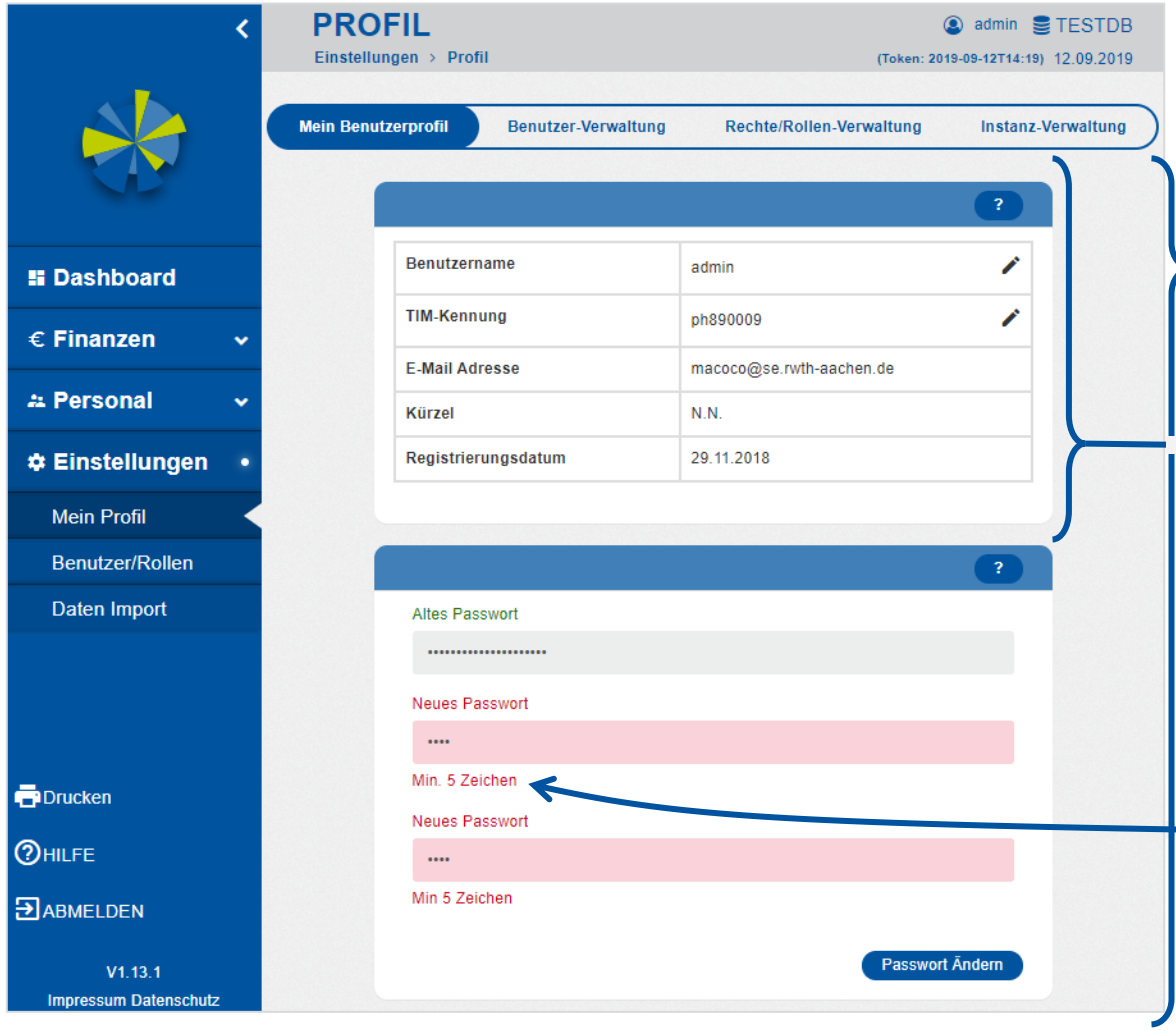
2 Generate

- Database
- Communication Infrastructure between Frontend and Backend
- Graphical User Interfaces
- All code comes with hookpoints

3 Add Hand-Written Extensions

- … through hookpoints
  - Allows for re-generation
- Define the business logic
- Customize Appearance
- Optimize the generated code

DSLs in MontiGem – MaCoCo



CD4A

```
1  class User {
2    String            username;
3    Optional<String>  encodedPassword;
4    ZonedDateTime     registrationDate;
5    Optional<String>  initials;
6    String            email;
7    boolean           authenticated;
8    Optional<String>  timID;
9  }
```

GUI-DSL

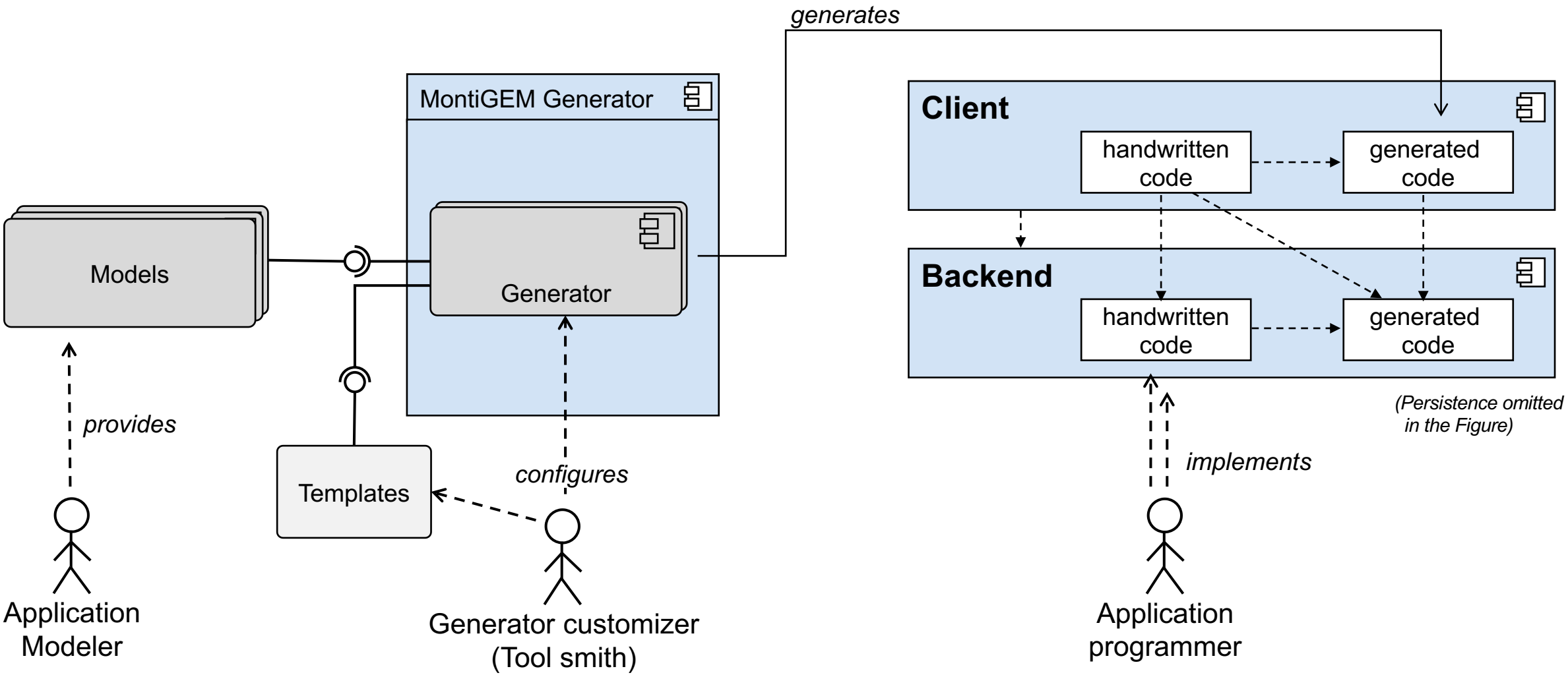
```
1  datatable "meinBenutzerInfoTabelle" {
2    columns < it {
3      row "Benutzername"         , <username (editable)
4      row "TIM-Kennung"          , <tim (editable)
5      row "E-Mail Adresse"       , <email
6      row "Kürzel"               , <initials
7      row "Registrierungsdatum"  , date(<registrationDate)
8    }
9  }
```

OCL/P

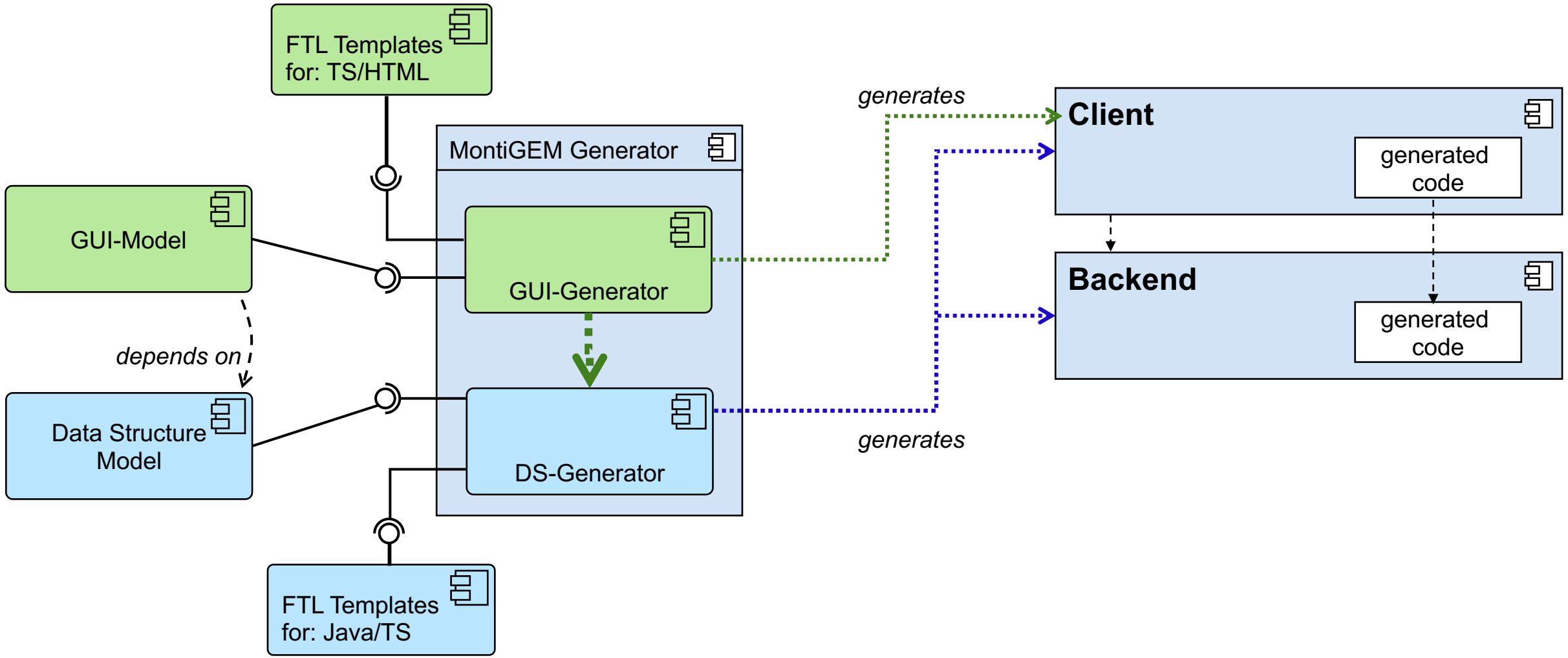
```
1  context User inv isPasswordValid:
2    password.length() >= 5;
3    shortError: "Min. 5 Zeichen";
4    error: "Das Passwort muss aus mindestens 5
5      Zeichen bestehen, hat aber nur " +
6      passwort.length() + " Zeichen.";
```

Data structure | User interface | Constraints

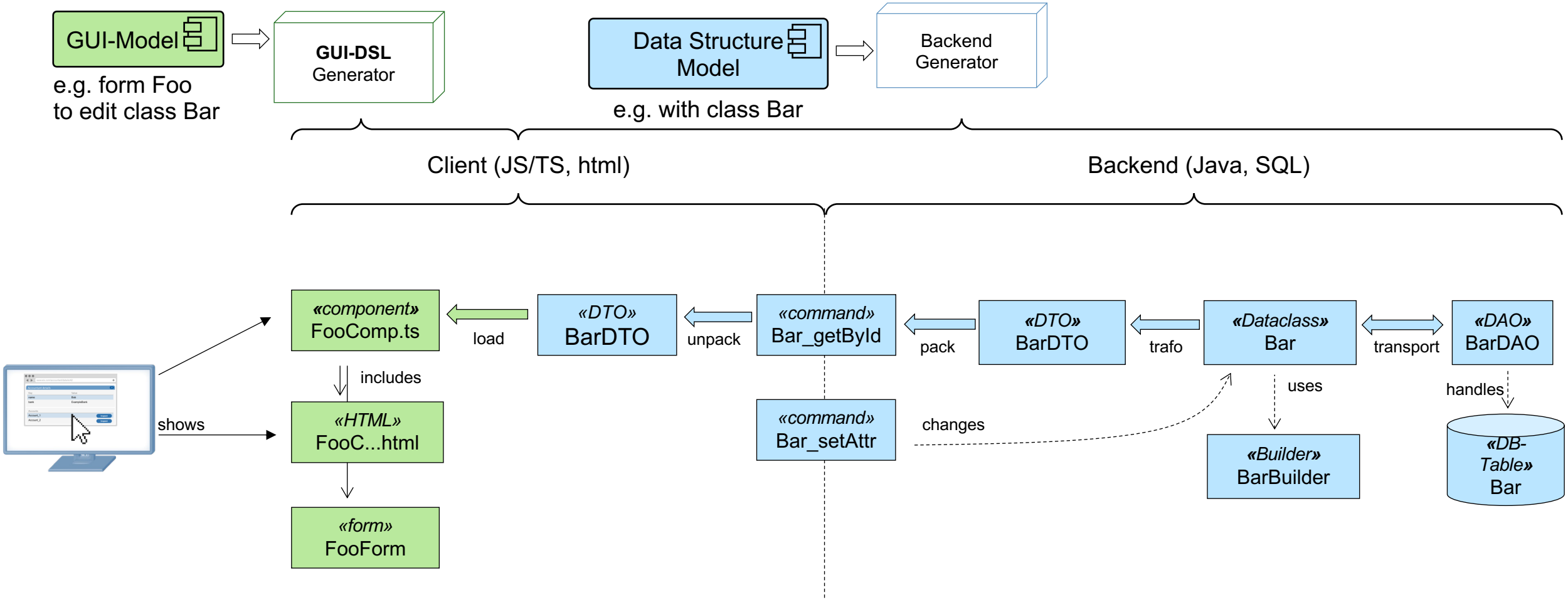
Generator Architecture, Principle



Generator Architecture, level 2

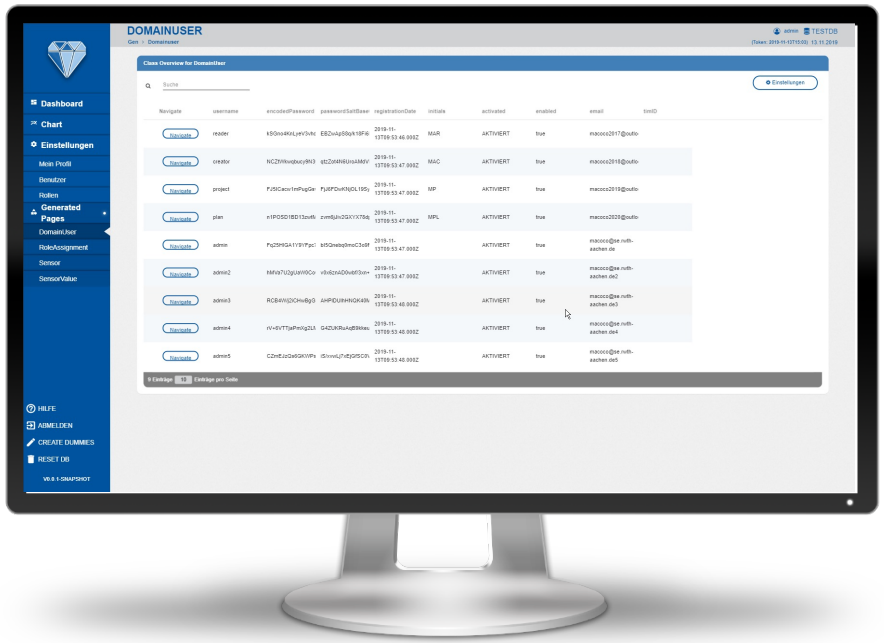


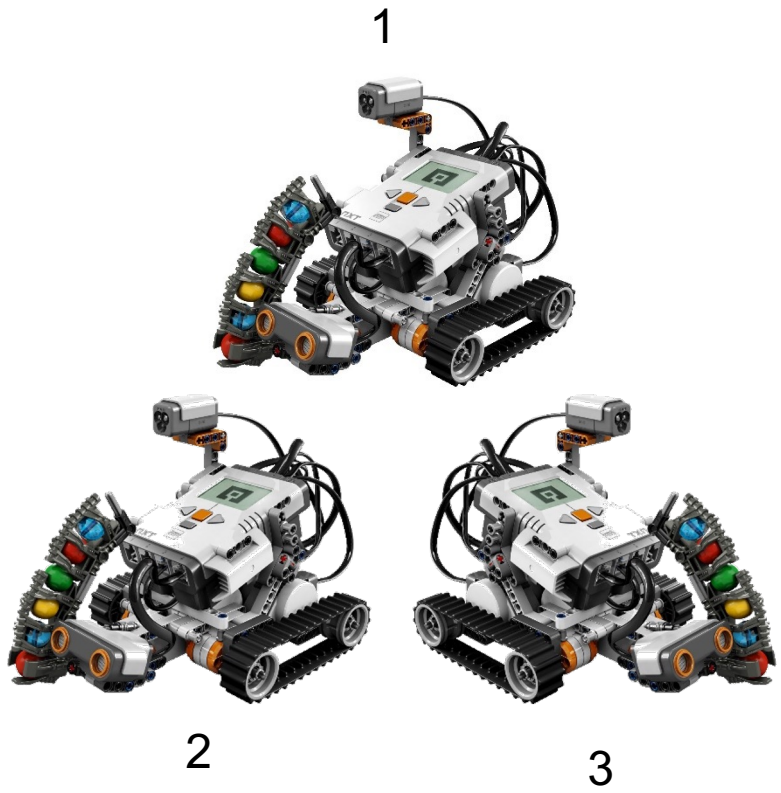
Communication Architecture and its generated Parts



Use Case

- GOAL : Display Battery Status information of Robot Fleet in MontiGem



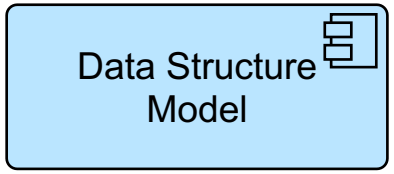


Domain Model

- Definition of data structure used to store and communicate information about robot.

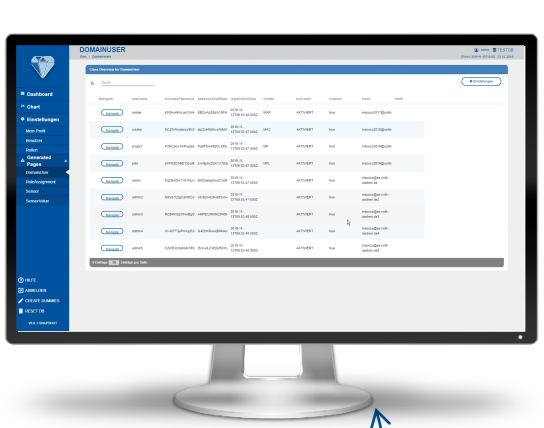
CD4A

```
01  classdiagram RobotManagement {

02    class Robot {
03      String name;
04      String macAddress
05      long batteryLoad;
06    }

07  }
```



Use Case

- DS-Generator Generates Database based on data structure specification from DS-Model
- Information from Robots can be stored in Database



Defines UI

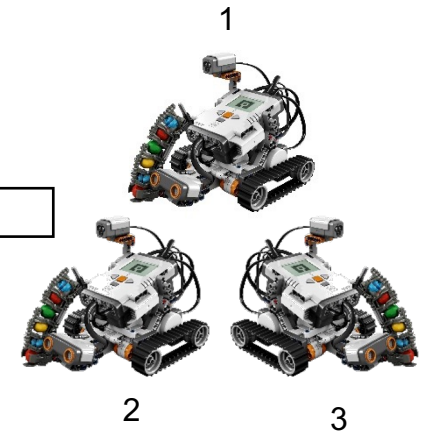
GUI-Model

BackEnd

«DB-Table»

API

Robot1; 45 %
Robot2; 15%
Robot3; 81%



Data structure and transport mainly defined by DS Model

Data base defined by DS Model

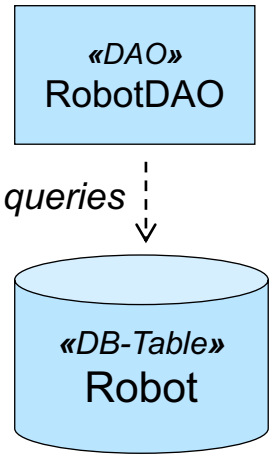
Data Structure Model

Backend – DAO classes

- **Data Access Object:** Represent the connection to the database

- Classes are generated (DS-Modell)
  - can be adapted for optimization

- **Hibernate** infrastructure is generated and can be used to load the data objects.
  - Every object in database has unique ID
  - Default generated queries retrieve on object by ID



RobotDAO.java

Java «Gen»

```
01 public class RobotDAO extends AbstractDAO {
02   // ... Infrastructure for Singleton
03
04   public Robot _getRobotbyId(Long robotId) {
05     TypedQuery<Robot> dataQuery =
06         em.createQuery("SELECT batteryLoad FROM Robot r
07               WHERE r.id = : identification", Robot.class)
08          .setParameter(„identification", robotId);
09     return dataQuery.getSingleResult();
10   }
11   public static Robot getRobotbyId() ...
12 }
```

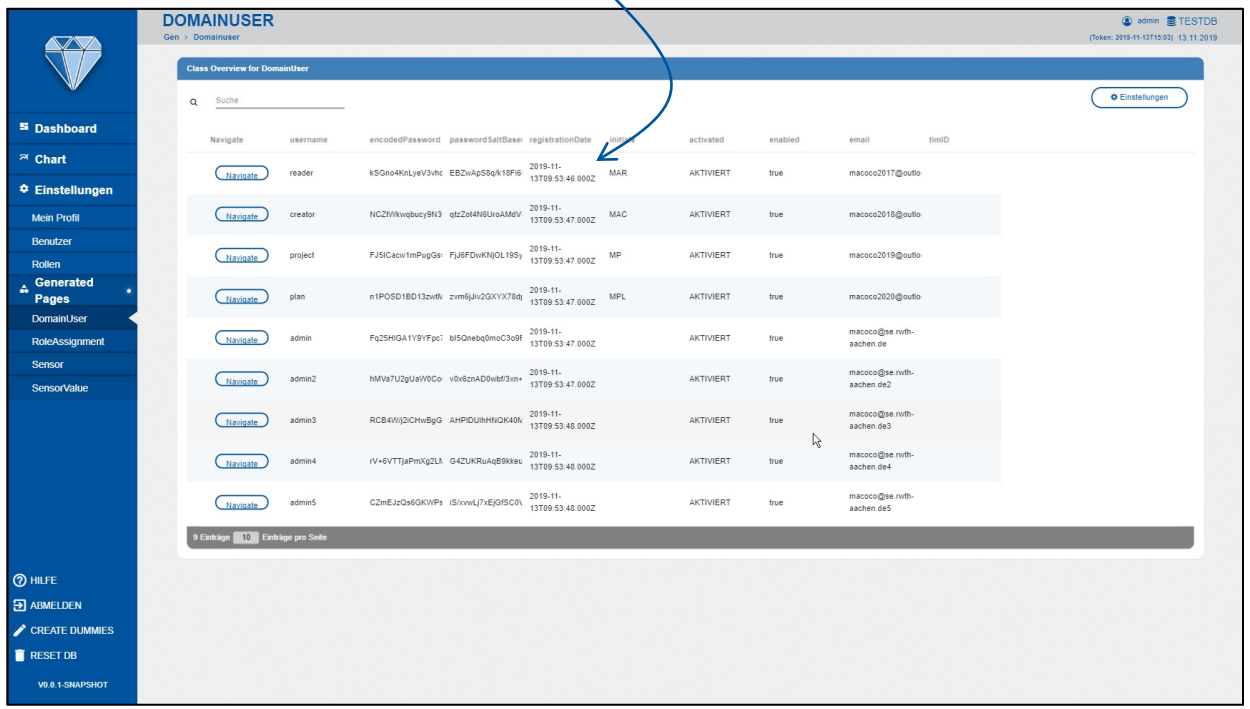
Example loading Robot from Database (in BackEnd):

```
Robot r = RobotDao.getRobotById(id)
```

Singleton

Use Case

- Visualize Data classes From Back End in GUI-Components of Front End



Data Class

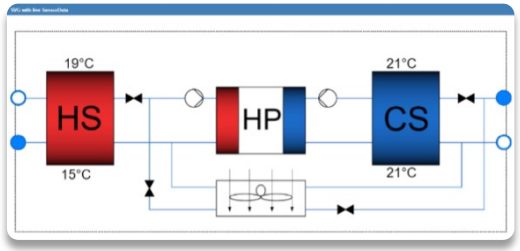
GUI-DSL «HW»

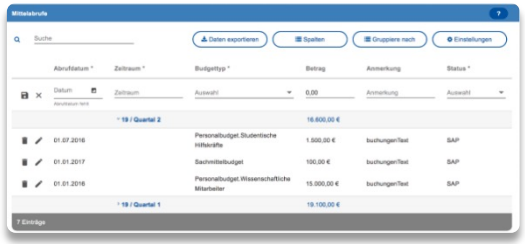
```
 1 webpage DataDashboard(all Robot robots) {
 2   card {
 3     head {
 4       row (stretch) {
 5         label "Battery Overview"
 6       }
 7     }
 8     body {
 9       datatable "BatteryInfo" {
10         rows <robots.entries {
11           column "Name", < name
12           column "Battery", < batteryLoad
13           ...
14         }
15       }
16     }
17   }
18 }
```

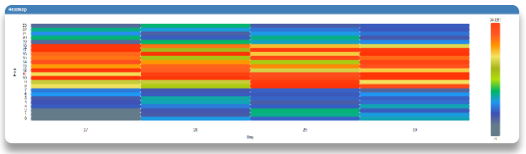
Layout of GUI

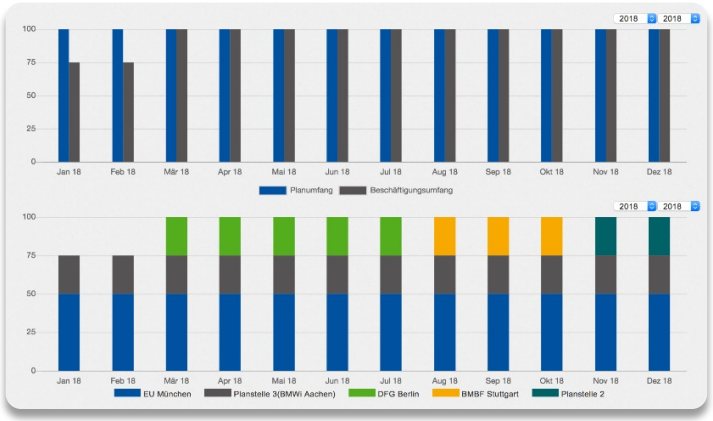
Attributes

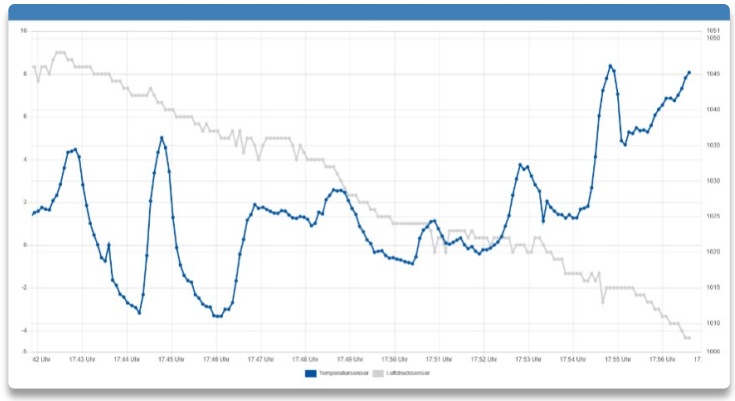
Frontend: GUI Component Examples

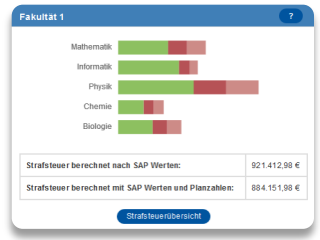


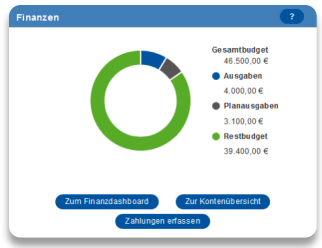












Example: MaCoCo: Full-size real world MDSE application

MaCoCo – Management Cockpit für Controlling
Ein Projekt der RWTH

Goal:
Provide management, planning, control and monitoring of organizational processes for the operative chairs and institutes of a university (RWTH).

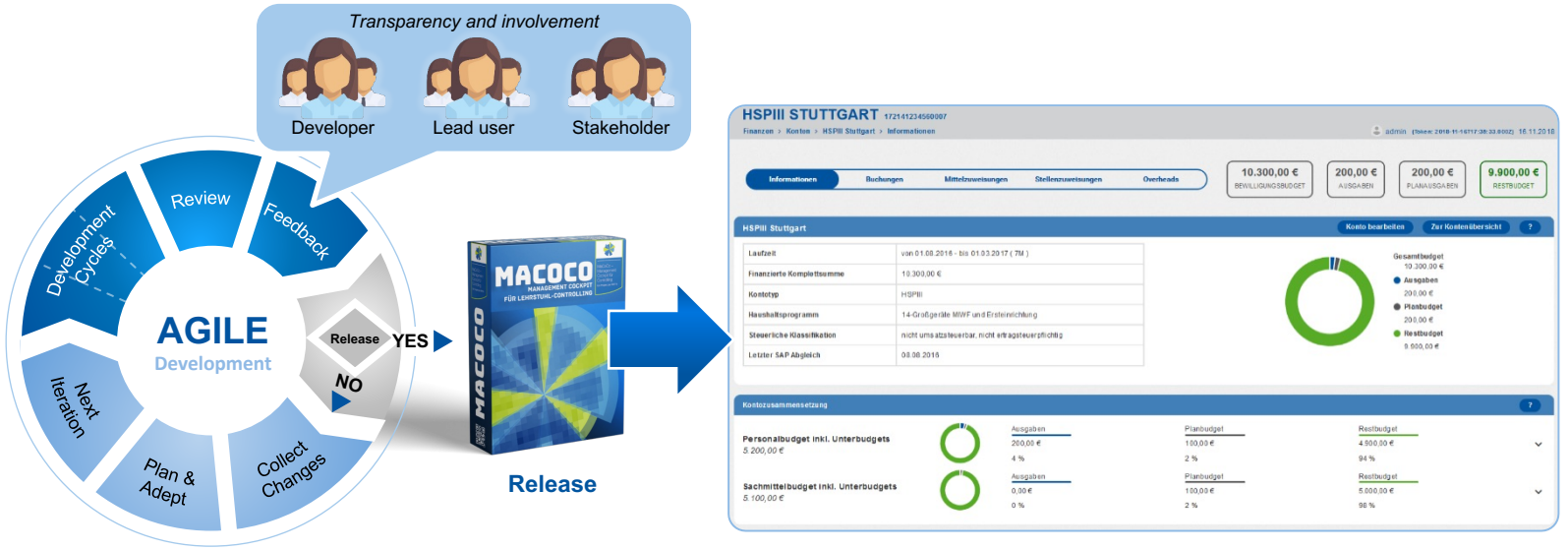
Functionalities
- Finances
  - Accounts, budgets, bookings
  - SAP connection
- Staff
  - Contracts, vacation
- Projects
  - Time sheets, effort planning

160+ instances (=databases) @RWTH Aachen
- Individual instances per chair/department
- One for *each* chair of Faculty 1 (Mathematics, Computer Science and Natural Sciences)

2020: 9.000 LOC in Models

Generated code (Java, TS+HTML)
- 250.000 + 140.000 = 390.000 LOC

Handwritten code (Java, TS+HTML)
- 55.000 + 60.000 = 115.000 LOC
- app. 18% backend, 30% frontend



www.se-rwth.de/projects/MaCoCo.php

Anwendungsbeispiel: Projekt MaCoCo

- **Softwarelösung** zur Professionalisierung des Controllings an Lehr- und Forschungseinheiten der RWTH

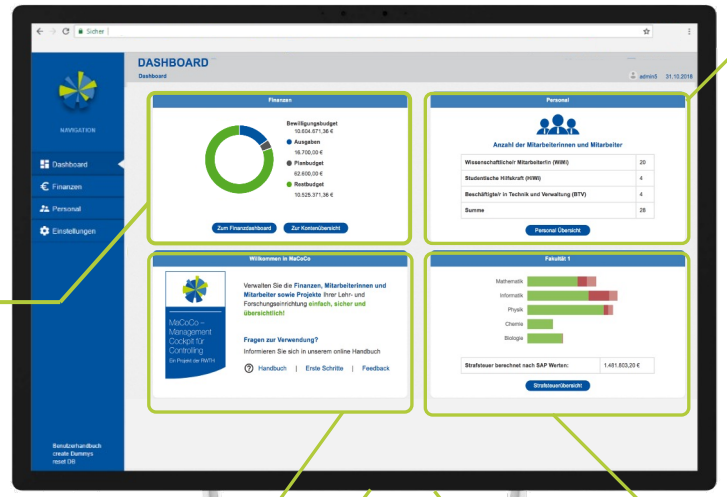
160+ instances (=databases) @RWTH Aachen
- Individual instances per chair/department
- One for *all* chairs of Faculty 1 (Mathematics, Computer Science and Natural Sciences)

2020: 9.000 LOC in Models
Handwritten code (Java, TS+HTML)
- 55.000 + 60.000 = 115.000 LOC
- app. 18% backend, 30% frontend

Generated code (Java, TS+HTML)
- 250.000 + 140.000 = 390.000 LOC

Finanzen
- Unterstützung in der Haushaltsverwaltung
- Planung von Budgets und Ausgaben
- Finanzieller Überblick

Personal
- Personal und Vertragsverwaltung
- Personalfinanzierung



Projekte
- Projekt Management
- Stundenzettel
- Personalplanung

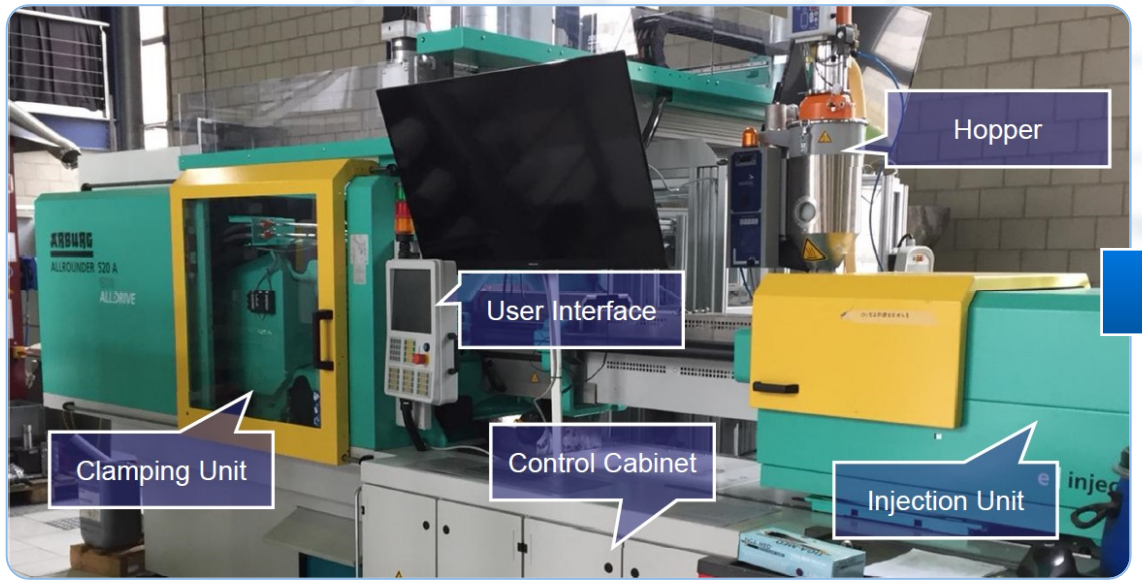
- Benutzerhandbuch
- SAP Schnittstelle
- Rechte/Rollen System

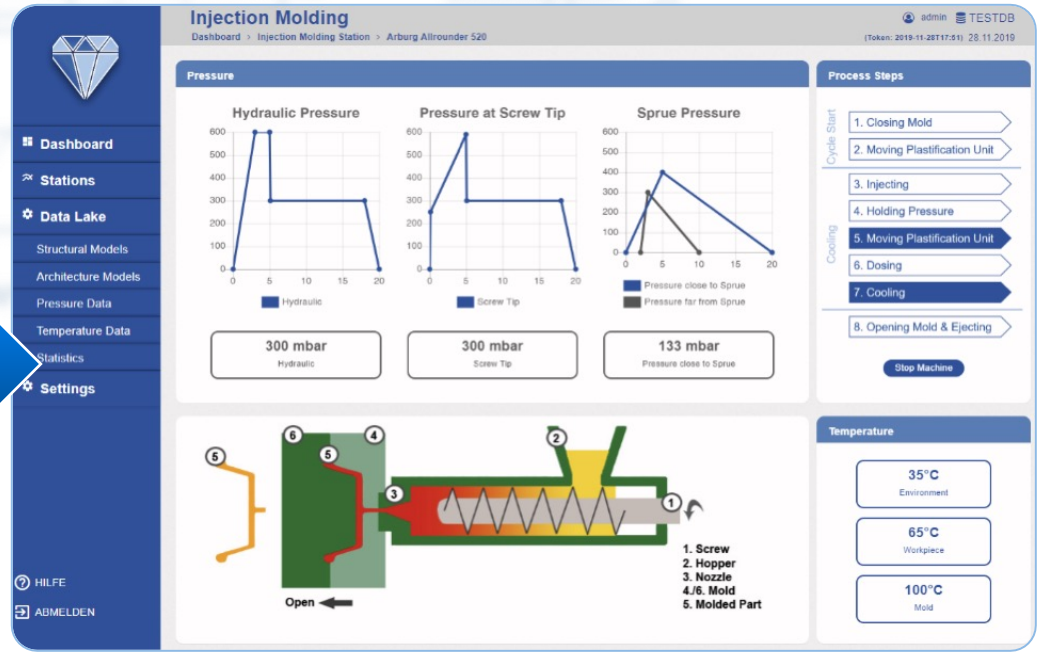
www.se-rwth.de/projects/MaCoCo.php

Anwendungsbeispiel: Management Cockpits for Smart Manufacturing

Problems for creating Digital Twins:
Integration of different technologies & heterogeneous data sources

Solution:
Agile, evolutionary development of Interactive Digital Twins Generating infrastructure from common data structures





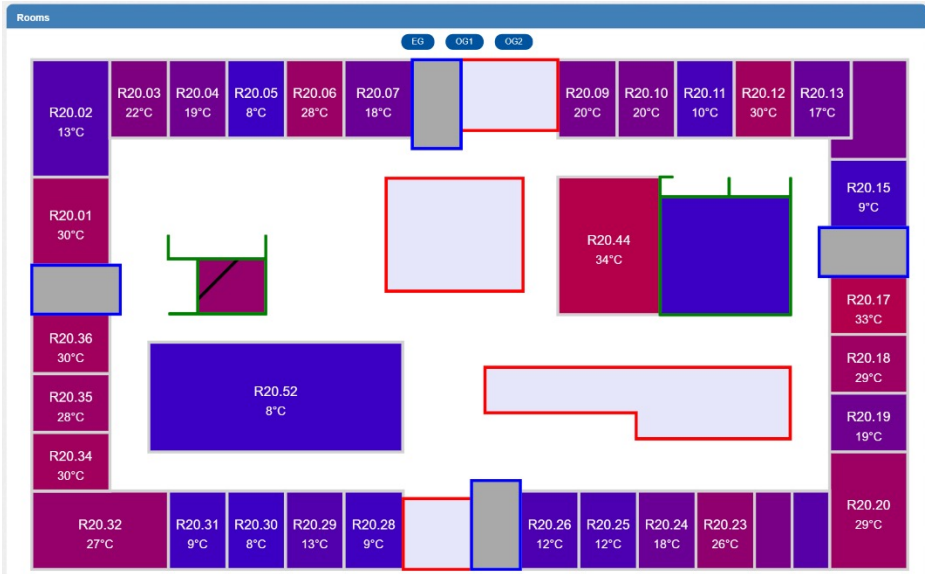
with M. Dalibor, J. Michael, S. Varga, A. Wortmann

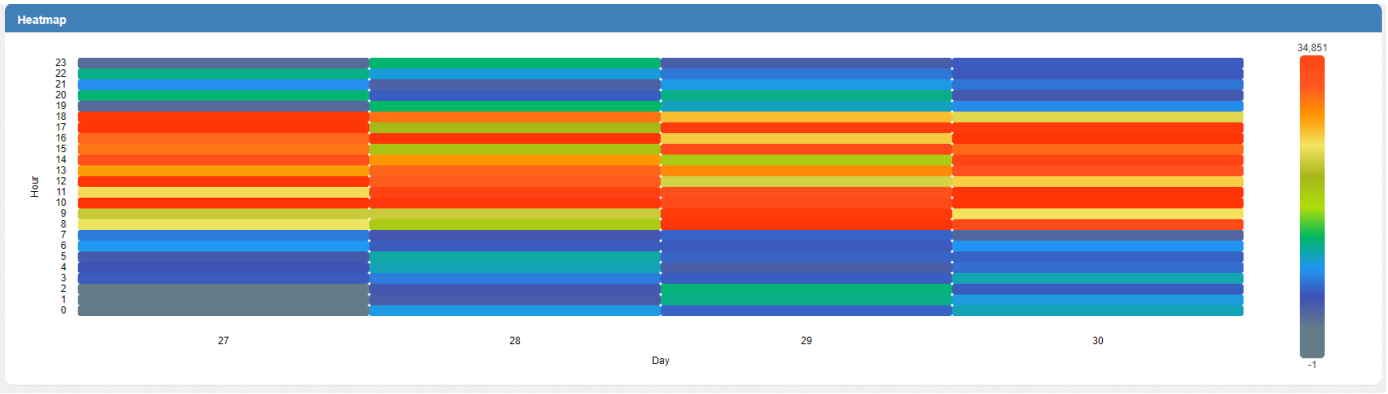
Anwendungsbeispiel: Energy Information System*

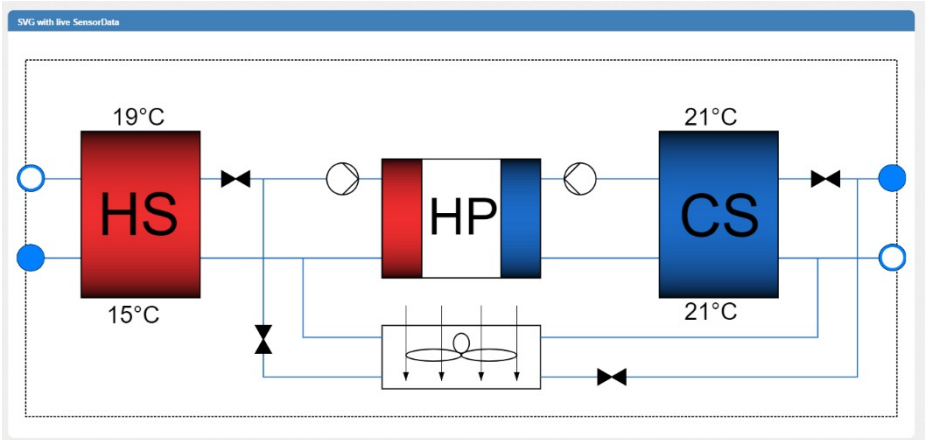
Problem:
Database of sensor data without visualization to inform user

Solution:
Display sensor data in an information system

- Room-temperature shown in floorplan
- History visualized in Heatmap and Line chart







*Part of the N5GEH-Project

Anwendungsbeispiel: Workflow System

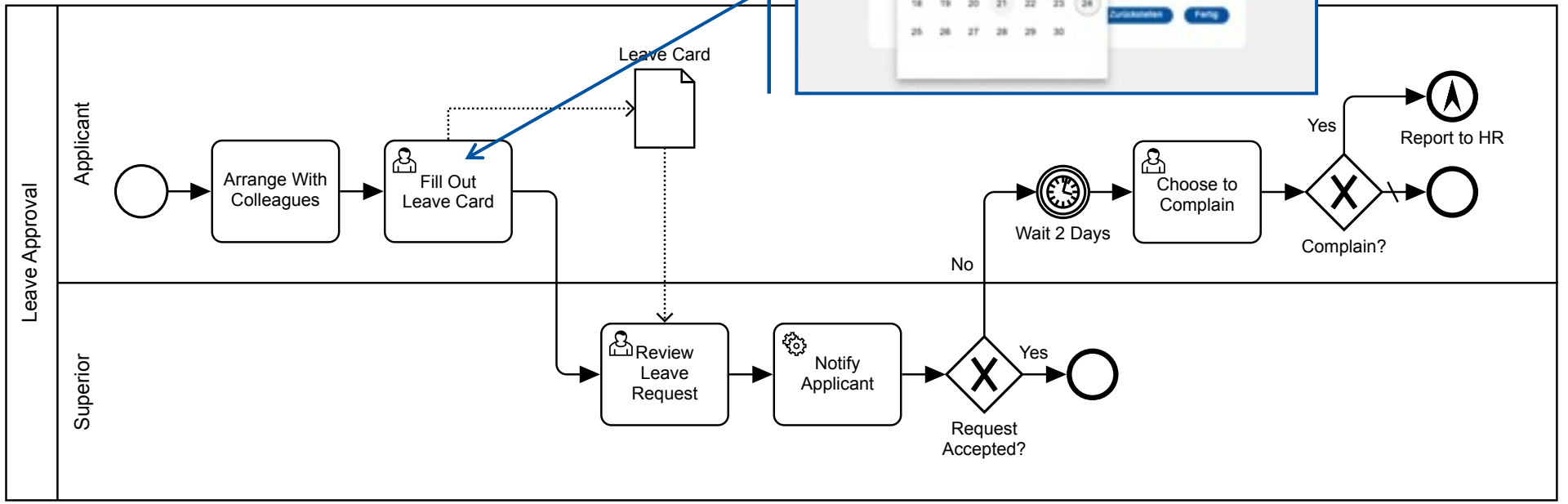
Challenge for workflow handling

Allow specific processes and interaction model-driven

Solution:

- Define BPMN models & generate forms for data input
- Use workflow engine

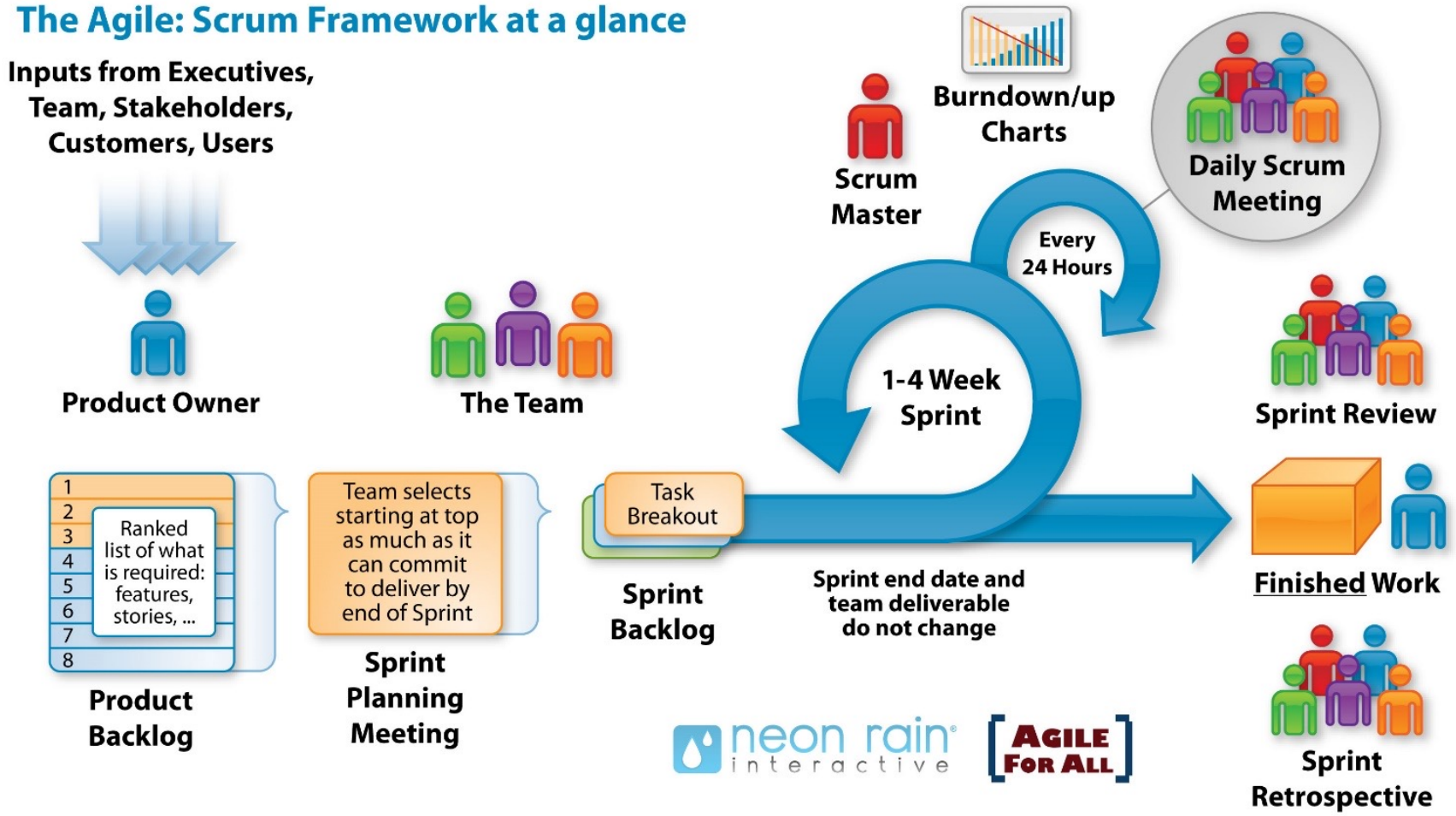
BPMN



with IDr, JM, SVa

Scrum: Perfekt für Generative SE

- Modelle
  - als schnell erweiterbare, ausbaubare Artefakte
- Evolutionärer Prozess
  - Änderbarkeit
  - Schnelles Feedback
- Kurze Zyklen
- Think and Plan on Models:
  - Design Models
  - Generate Code / Prototypes
  - Test it



Quelle: https://www.neonrain.com/agile-scrum-web-development/

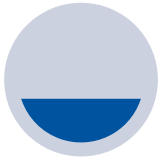
MontiGem Projects and Application Areas

- Management Cockpit for Chair Controlling (MaCoCo)
  www.se-rwth.de/projects/MaCoCo.php
- Digital twin cockpits
  DFG Excellence Cluster „Internet of Production", https://www.iop.rwth-aachen.de
- Sovereign decisions regarding personal data from wearables
  BMBF InviDas: Interaktive, visuelle Datenräume zur souveränen, datenschutzrechtlichen Entscheidungsfindung, https://invidas.gi.de

- Further projects:
  - IoT for vehicle fleets (Industry project, OEM)
  - Building energy management system, Engineering of wind turbines

- *BA/MA Theses* in MDSE of information systems for
  - weather stations | maintenance tasks of chemical plants | meta-analysis of publication data | diary for Parkinson's disease | include BPMN models to realize workflows | privacy checkpoints

Was haben wir gelernt?

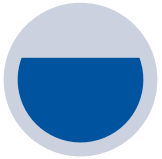


Generierung

Infrastruktur

Prinzip eines Generators

Vorteile der Generierung

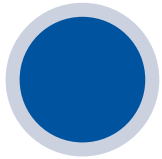


Generierung von Informations-systemen

Bestandteile:
Persistenz (Datenbank),
Oberfläche (Frontend),
Anwendungskern (Backend)

Funktionalität: CRUD auf Daten, Suche, Filter

Typischer Softwarestack
- Angular Framework im Frontend (ts, HTML),
- Java Backend



MontiGem

Modelle
- Klassendiagramme für Datenstruktur und Kommunikation
- GUI-Modelle für Oberflächen
- OCL für Validatoren

Agiler modellbasierter Entwicklungsprozesse durch Generierung

Vorlesung Softwaretechnik

10. Qualitätsmanagement

Prof. Bernhard Rumpe
Software Engineering
RWTH Aachen

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

SE Software Engineering | RWTH AACHEN UNIVERSITY

Warum, was, wie und wozu? Qualitätsmanagement

Warum?

- Software muss Qualitätsanforderungen erfüllen: Korrekt, schnell genug, verlässlich,…
- Fehler in der Software können sehr teuer werden

Was?

- Entwicklungsprozesse
- Software

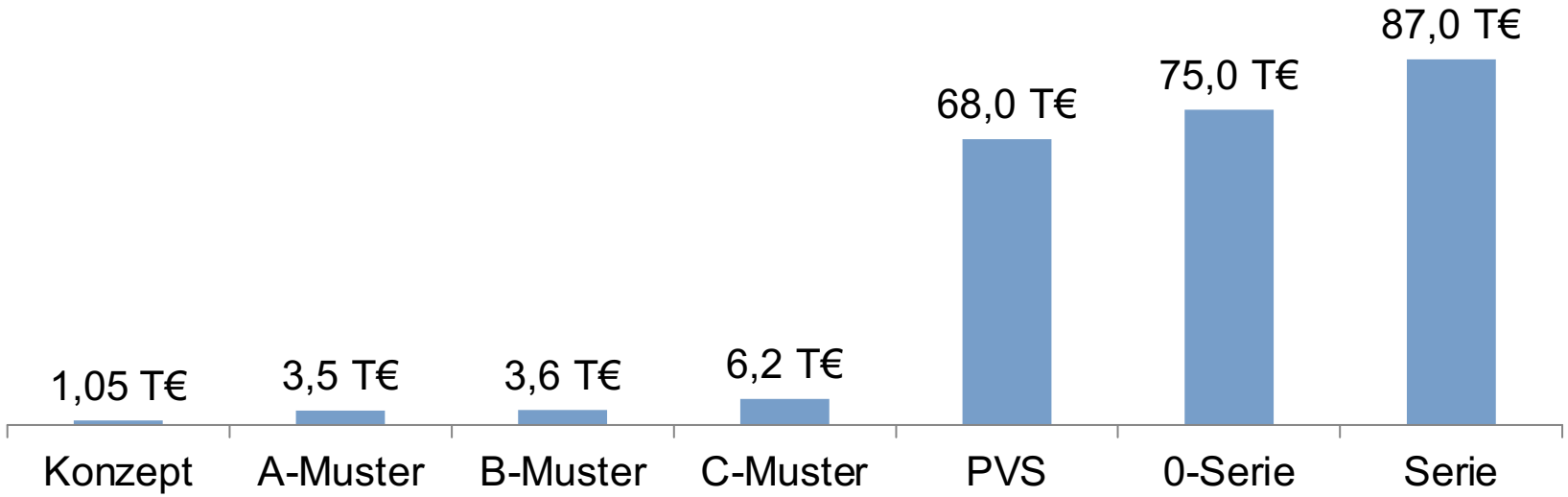
Wie?

- Dokument- und Code-Reviews
  Qualitätsmanagement
  Prozess Standards
- Testen, z.B.
  Funktionale Tests (black box)
  Strukturtests (white box)

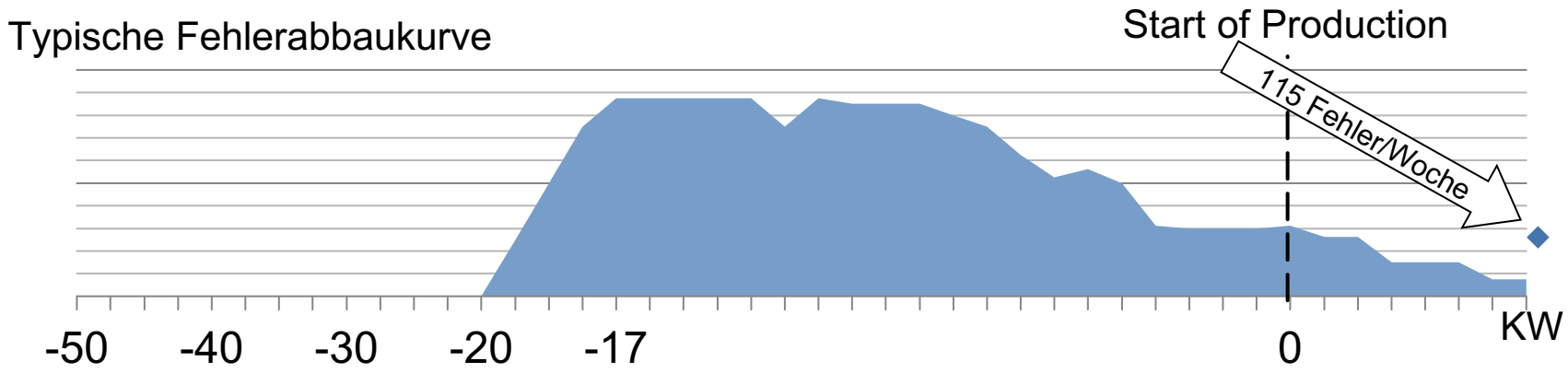
Wozu?

- Software Tests sind gelebte Praxis
- QM Prozesse in der SW-Entwicklung in den meisten großen Unternehmen etabliert

Geschätzte relative Abstellkosten je Software-Fehler bei der Fahrzeugentwicklung



Quelle: HIS, Stand 1999

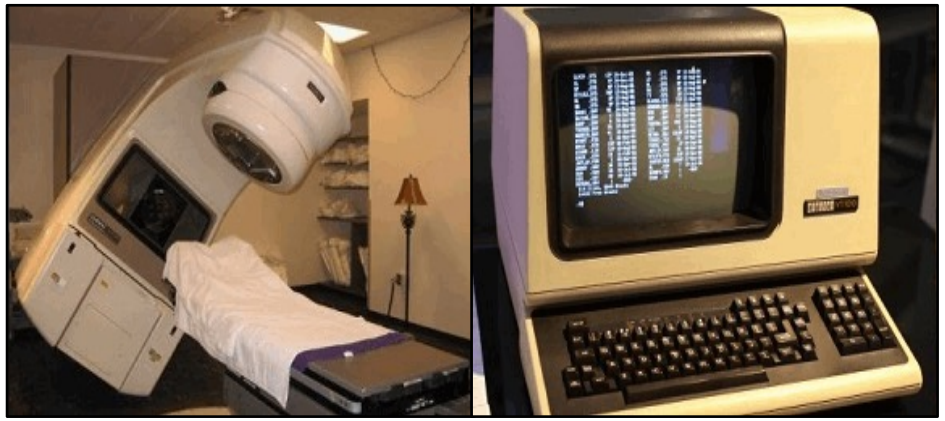


Quelle: G.Wagner, M. Liebler, 2007

Warum Qualitätssicherung zB. durch Testen?

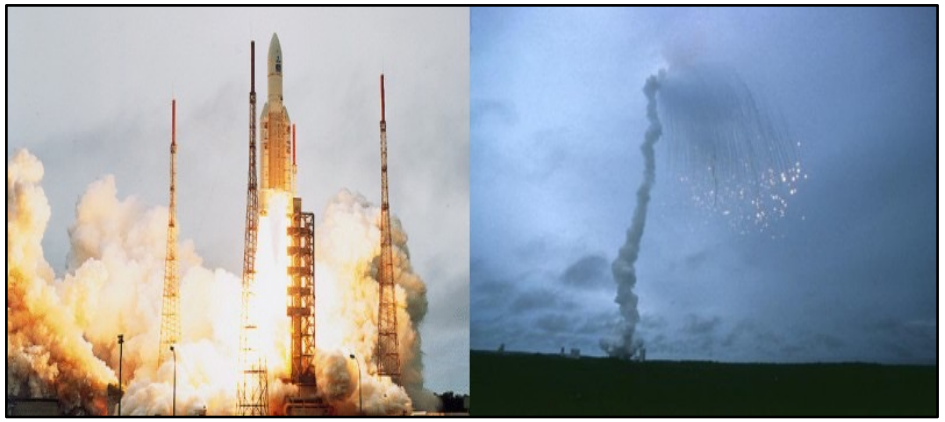
Therac-25-Unglück (1982-85)

- Strahlentherapiegerät zur Krebsbekämpfung
- Vorgänger rein mechanisch/elektrisch
- Ursachen:
  1. Software fängt fehlerhafte Eingaben nicht ab
  2. Überlauf → Kontrollflussfehler → Bestrahlung fehlerhaft
- Schaden: 6x Strahlenüberdosierung: 3 Patienten verstorben



ESA Ariane-5 Erstflug (1996)

- Rakete sprengte sich selbst unmittelbar nach dem Start
- Ursache:
  - Code der Ariane 4 wiederverwendet, Ariane 5 beschleunigt stärker.
  - Überlauf bei Umwandlung von 64-Bit-Zahl in 16-Bit-Zahl
  - Rakete steuert gegen → unvorhergesehene Belastungen → zerbricht
- Schaden: Rakete, Nutzlast, Image - ca. 370 Millionen $



Details

https://iansommerville.com/software-engineering-book/case-studies/ariane5/

Softwaretechnik

10. Qualitätsmanagement
10.1. Prozessqualität

Prof. Bernhard Rumpe
Software Engineering
RWTH Aachen

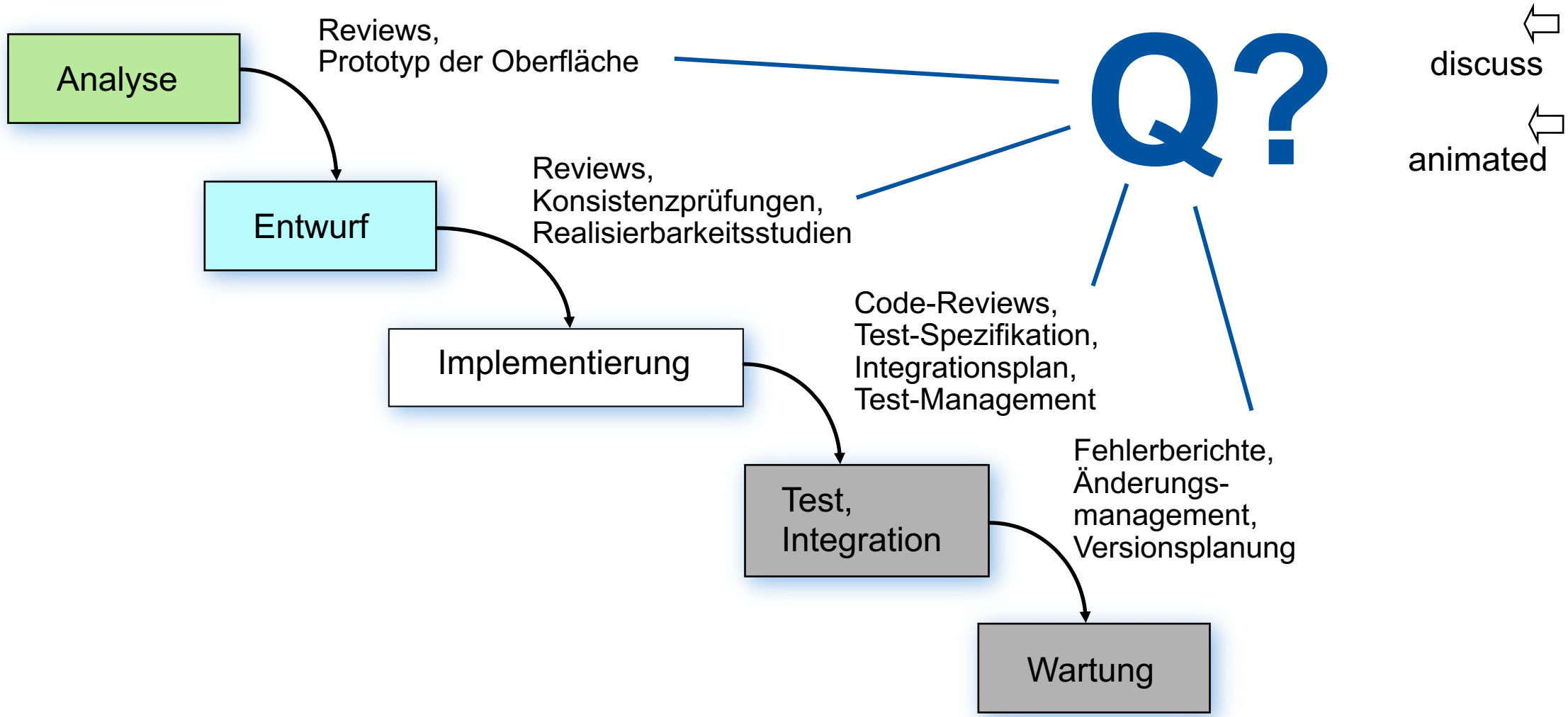
http://www.se-rwth.de/
@SE_RWTH

Literatur:

- Sommerville 24-25
- Balzert Band II, LE 12-13

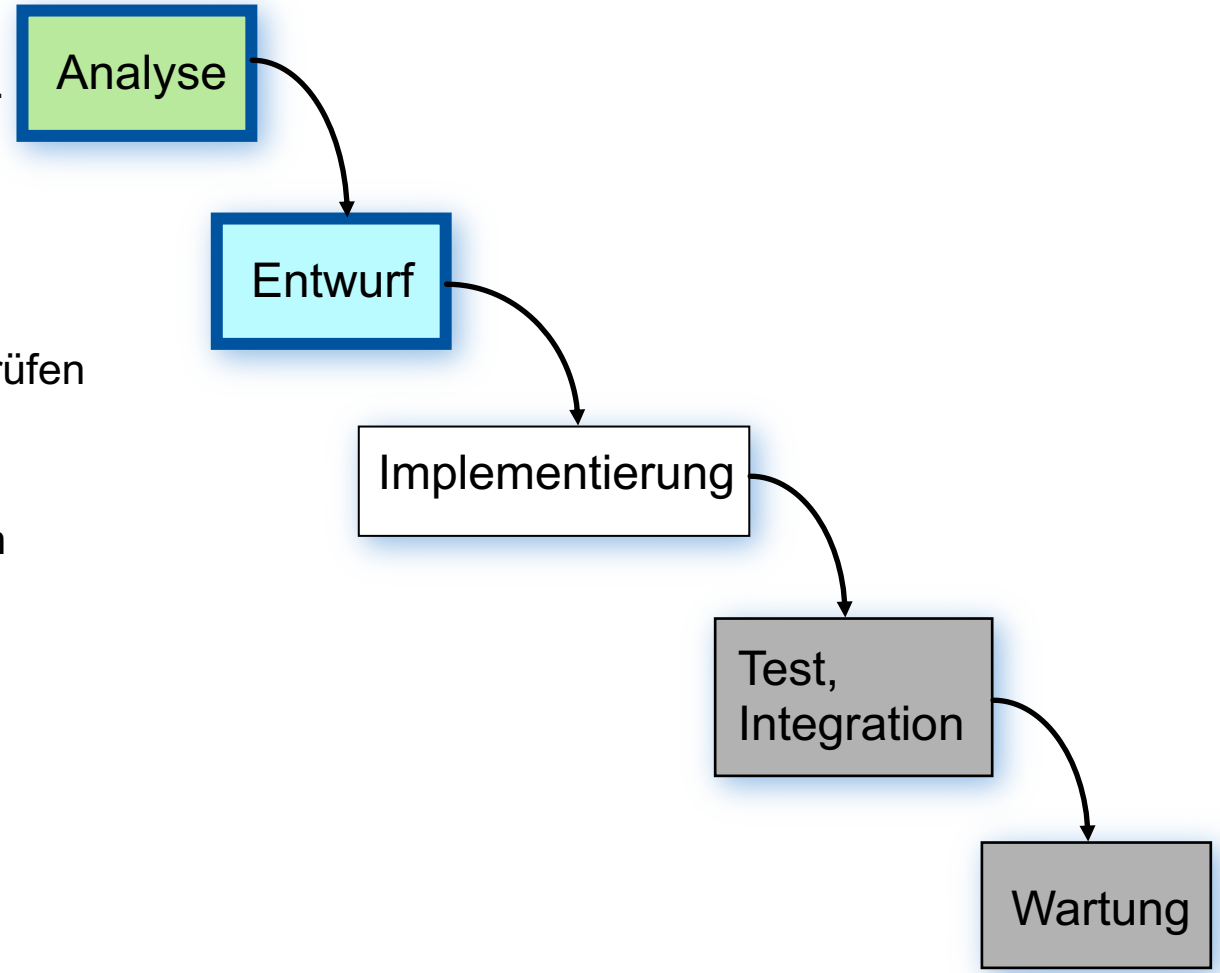
SE Software Engineering | RWTH AACHEN UNIVERSITY

Prozessintegrierte Qualitätssicherung



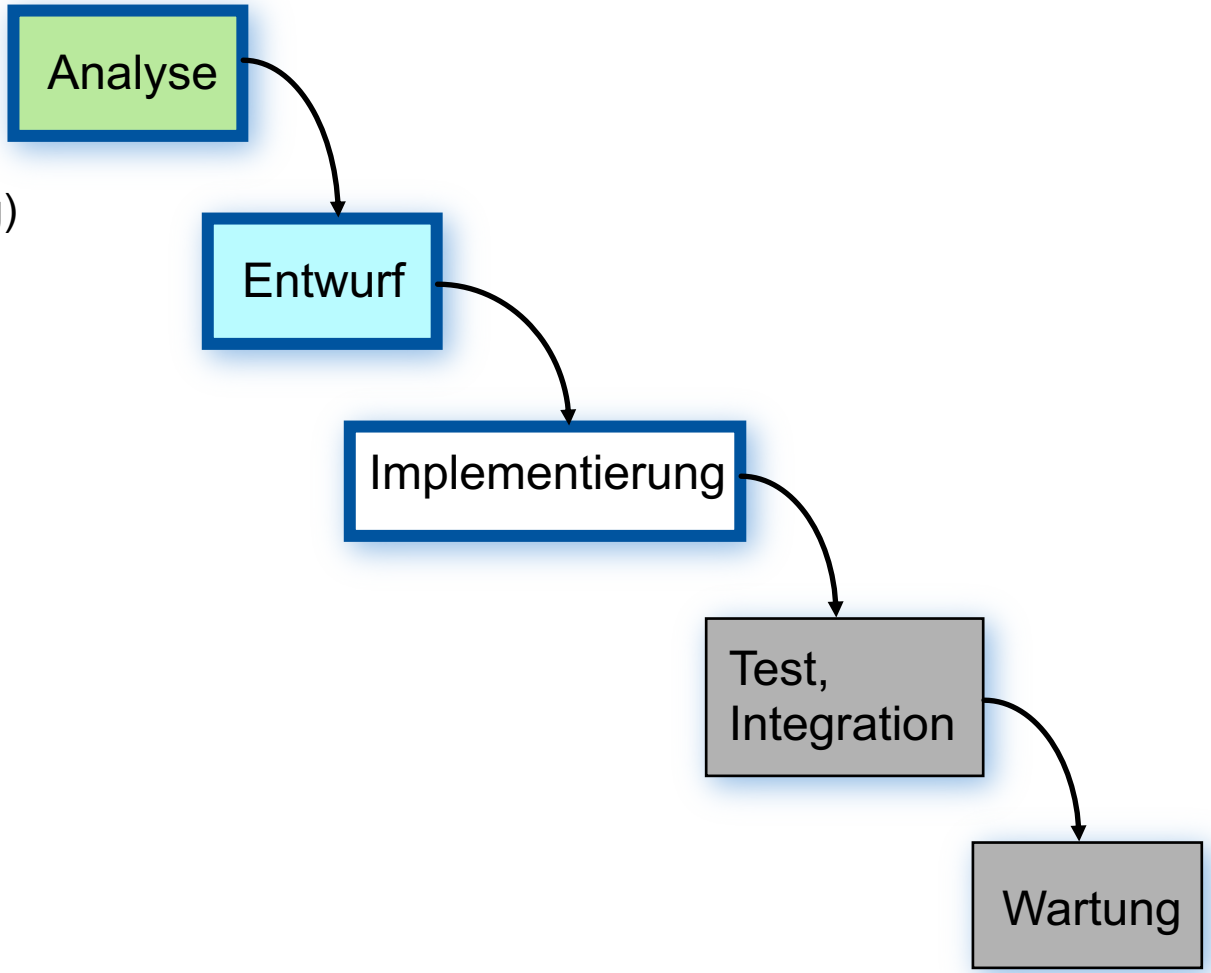
Qualitätssicherung für Analyse und Entwurf

- Hohe Bedeutung früher Phasen für Produktqualität !
  - Deshalb: Alle Dokumente frühzeitig überprüfen ("Validation").
- Techniken:
  - Anforderungskatalog-Begutachtung
    - Echte Benutzer einbeziehen
    - Anforderungskatalog auf Vollständigkeit und Korrektheit prüfen
  - Use-Case-Szenarien
    - Echte Benutzer einbeziehen
    - "Funktionsfähigkeit" der abstrakten Modelle demonstrieren
  - Prototyping
    - Prototyp auf der Basis der Analyse/Entwurfs-Dokumente
    - Echte Benutzer einbeziehen
    - Vorgezogener Akzeptanztest
  - Abgleich des Entwurfs mit Use-Cases/Anforderungskatalog
    - Erste verifizierende Tätigkeiten



Begutachtung (Review)

- Produkt wird von Expertengremium begutachtet
  - Anwendbar auf fast alle Artefakte in der Entwicklung
- Was wird begutachtet?
  - Genau definiertes Dokument (Art, Status, Prozesseinbindung)
  - Teil der Gesamtplanung des Projekts (Termin)
- Wer begutachtet?
  - Teammitglieder (Peer-Review)
  - Externe Spezialisten
  - Echte Benutzer
  - Moderator, Manager
- Wie läuft die Begutachtung ab?
  - Einladung
  - Vorbereitung, Sammlung von Kommentaren
  - Begutachtungssitzung(en), Protokolle
  - Auswertung und Konsequenzen (Wiederholung, Statusänderung)

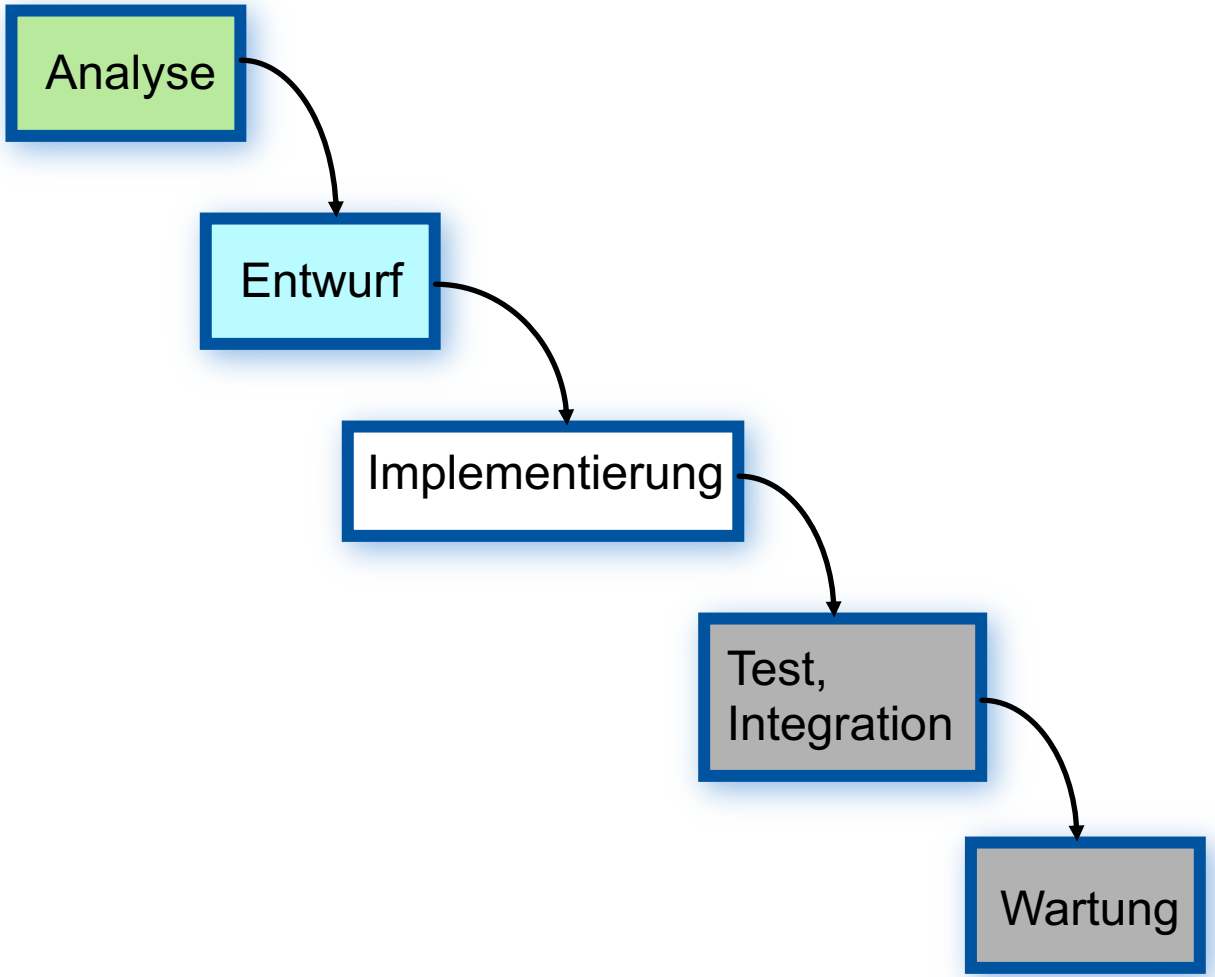


Regeln für wirkungsvolle Begutachtung (Review)

- "Checklisten" für die Gutachter vorbereiten
  - z.B. "Enthält das Dokument alle laut Firmenstandard vorgesehenen Informationen?"
  - "Gibt es für jede Klasse eine informelle Beschreibung?"
  - "Sind Kardinalitäten im Klassendiagramm korrekt?„
- Richtige Vorkenntnisse der Gutachter sind wesentlich.
- Die Rolle des Moderators ist anspruchsvoll:
  - Vermittlung in persönlichen Konflikten und Gruppenkonflikten
  - Fachlicher Gesamtüberblick
- Das Dokument wird begutachtet, nicht die Autoren!
- Ergebnisse von Begutachtungen müssen Auswirkungen haben.
- Begutachtung ist auch anwendbar auf Programm-Code (Code-Inspektion).

Produktqualität und Prozessqualität

- Software:
  - Keine Qualitätsmängel durch Massenproduktion
  - Qualitätsmängel bei Entwicklung begründet
- Qualitätsmanagement:
  - Organisatorische Maßnahmen zur Prüfung und Verbesserung der Prozessqualität
  - Beachtung von internen Kunden-/Lieferantenbeziehungen
- Qualität des Entwicklungsprozesses!
- Ansätze:
  - ISO 9000
  - Total Quality Management (TQM)
  - Business (Process) Re-Engineering
  - Capability Maturity Model (CMM)
  - Software Process Improvement and Capability Determination (SPICE), ISO/IEC 15504

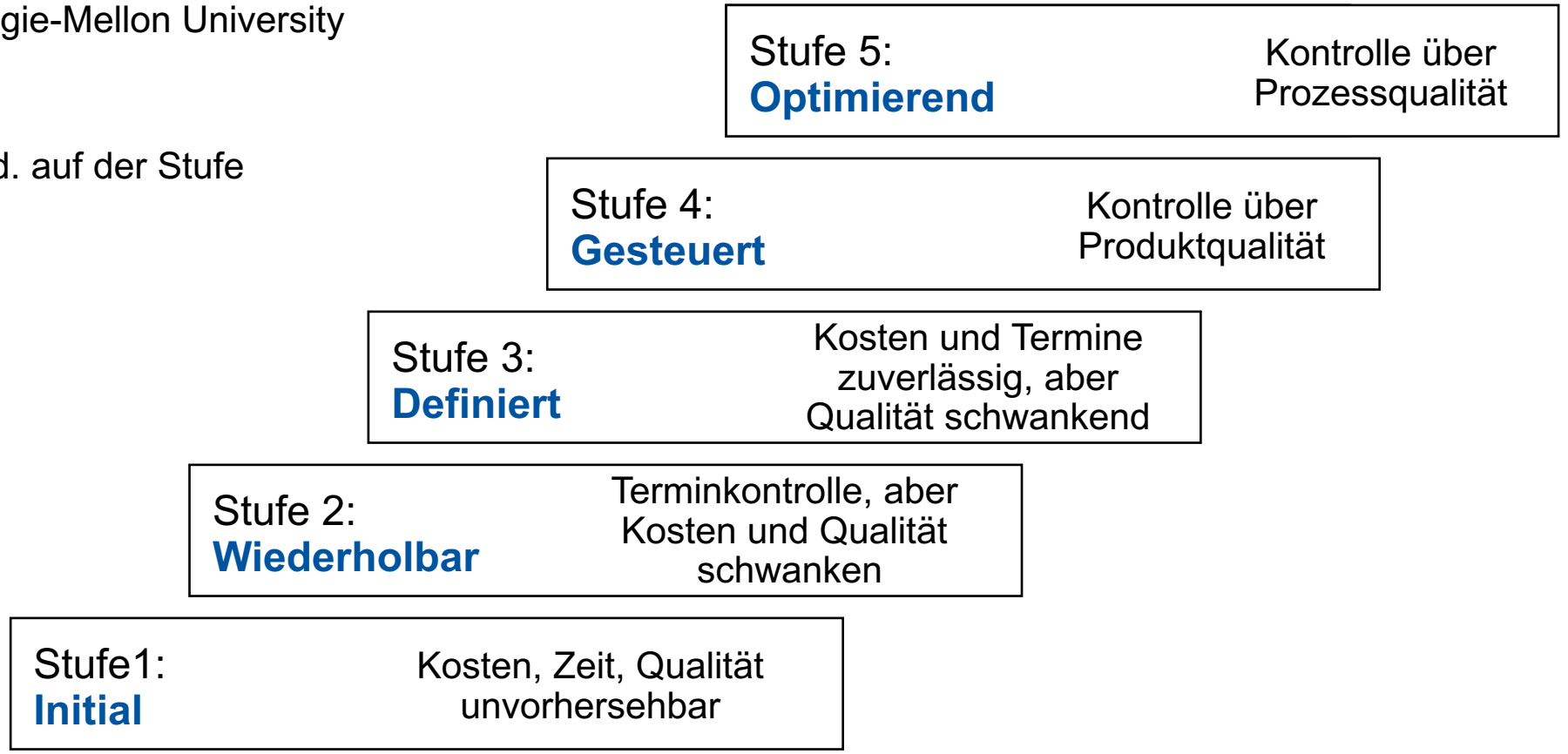


ISO 9000

- Internationaler Standard für Qualitätsmanagement
- Normreihe mit Grundsätzen für Maßnahmen zum Qualitätsmanagement
  - ISO 9001 weite Verbreitung (ISO 9001:2015)
  - Verantwortlichkeiten und Zuständigkeiten, Strukturen und Arbeitsabläufe werden in Prozess- und Verfahrensanweisungen geregelt und dokumentiert
- Anwendbar auf alle Organisationen und Branchen
- Auditierbar, zertifizierbar
  - auf freiwilliger Basis durch eine für diese Aufgabe akkreditierte Stelle
- Gründe für Zertifizierung
  - Marketing
  - Zukunftssicherung
  - (Rechtliche)

Capability Maturity Model (CMM)

- Einstufungsverfahren für Reifegrad der Organisation
  - entwickelt von der Carnegie-Mellon University
- Stufenwechsel
  - Alle Prozessgebiete mind. auf der Stufe



Weiterentwicklung: Capability Maturity Model Integration (CMMI)

- Familie von Referenzmodellen für unterschiedliche Anwendungsgebiete
  - *for Development (CMMI-DEV)*
    - Organisation entwickelt Software, Systeme oder Hardware
  - *for Supplier Management (CMMI-SPM), ehemals Acquisition CMMI-ACQ*
    - Organisation kauft Software, Systeme oder Hardware ein, entwickelt aber nicht selbst
  - *for Services (CMMI-SVC)*
    - Organisation erbringt Dienstleistungen
  - *People Capability Maturity Model (PCMM)*
    - Reife der Belegschaft entwickeln
  - *Data Management Maturity (DMM)*
    - Datenmanagement der Organisation

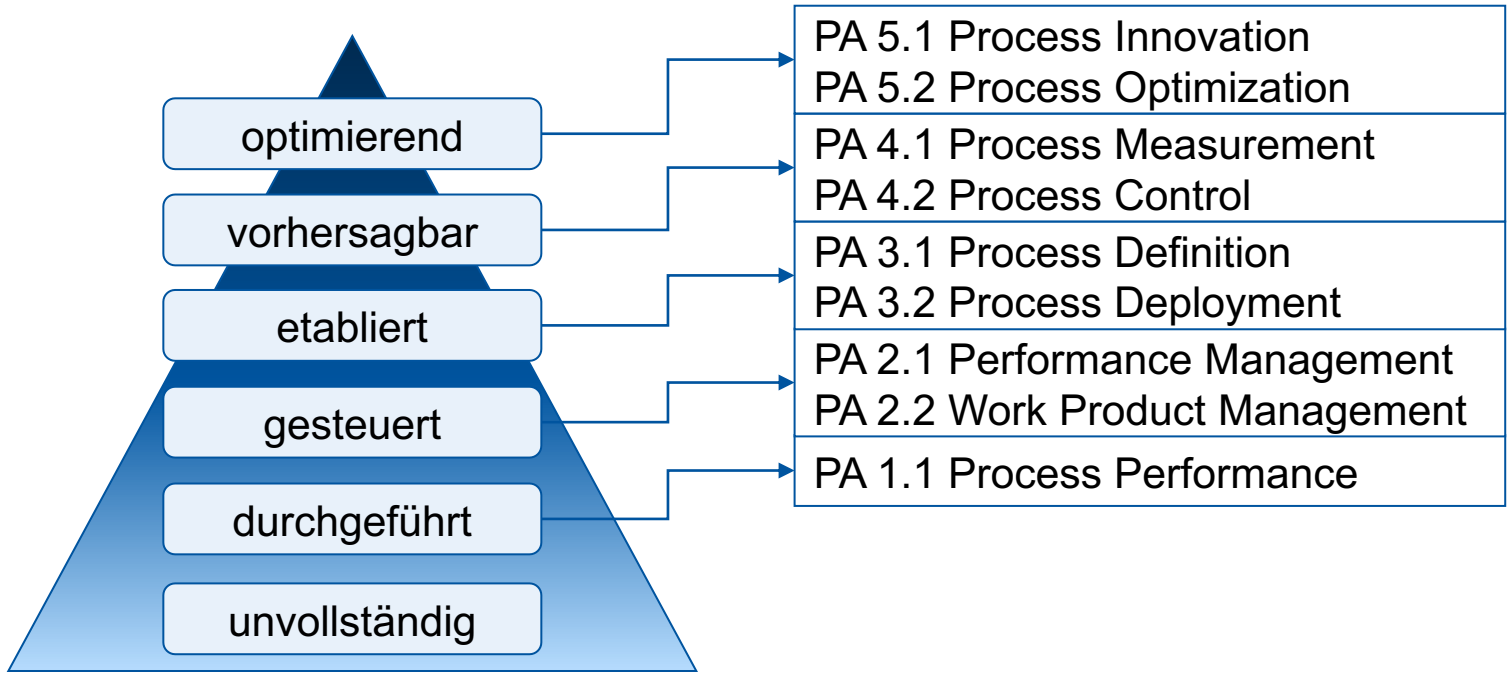
Quelle: https://cmmiinstitute.com/cmmi

Software Process Improvement and Capability Determination (SPICE)

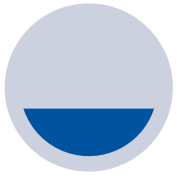
- ISO/IEC 15504
- Schwerpunkt auf Softwareentwicklung

- 6 Gradstufen
  - vergl. CMM

- 9 Prozessattribute
  - nicht erfüllt: 0 % – 15 %
  - teilweise erfüllt: >15 % – 50 %
  - weitgehend erfüllt: >50 % – 85 %
  - vollständig erfüllt: >85 % – 100 %



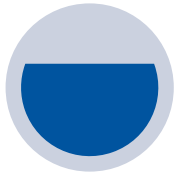
Was haben wir gelernt?



Prozessqualität

Produktqualität oft schwer zu messen: daher Fokus auf Prozess

Wenn der Prozess gut ist hat man auch gute Chancen, dass das Produkt gut ist

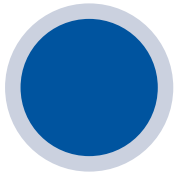


Qualitäts-management Prozesse

Zertifizierung

Standards

- ISO 9000
- Total Quality Management
- Capability Maturity Model (CMM)
- Software Process Improvement and Capability Determination (SPICE), ISO/IEC 15504



Reviews

Code Reviews durch Teammitglieder

ABER: Code und Tests gibt es erst nach der Analyse und Design Phase

Frühzeitige Betrachtung von Dokumenten auf allen Entwicklungsstufen

… erfordern gute Soft-Skills von allen Beteiligten und sind konstruktiv

Softwaretechnik

10. Qualitätsmanagement
10.2. Test und Integration

Prof. Bernhard Rumpe
Software Engineering
RWTH Aachen

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

Literatur:

- Sommerville 19-20
- Balzert Band II, LE 14-15

Speziell:

- Robert V. Binder: Testing object-oriented systems, Addison-Wesley 2000

SE Software Engineering | RWTH AACHEN UNIVERSITY

Was ist Testen?

> *„Testen ist der Prozess, ein Programm mit der Absicht auszuführen, Fehler zu finden.“*
> - Glenford J. Myers (1979) -

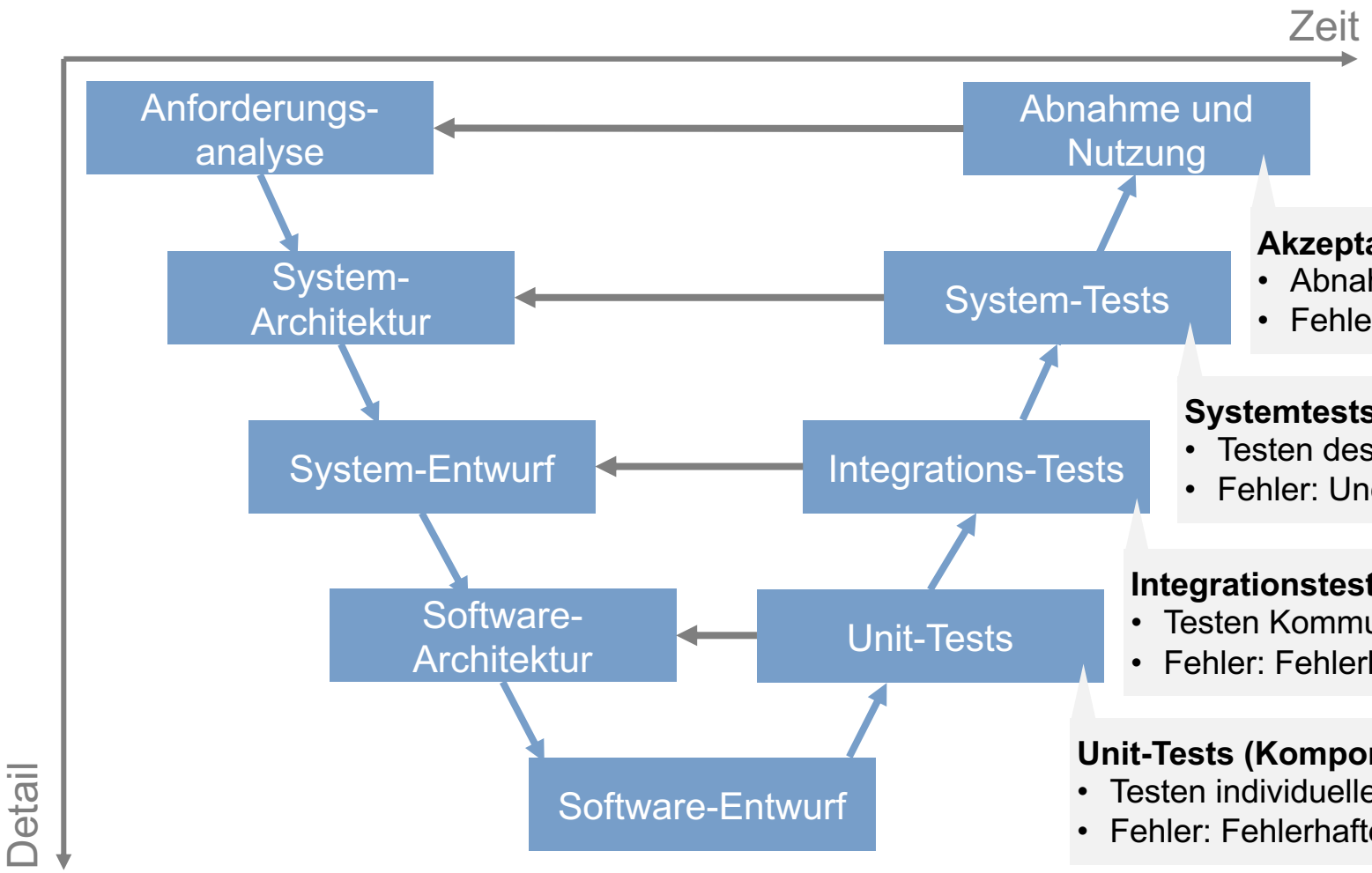
- Testen ist eine Form der dynamischen Qualitätssicherung: das Programm wird ausgeführt

> *„Program testing can be used to show the presence of bugs, but never to show their absence!“*
> - Edsger W. Dijkstra (1970) -

- Vollständiges Testen ist unmöglich
- Testen kann die Korrektheit eines Programms nicht beweisen

⇨ Ein Test führt das Programm aus und ist erfolgreich, wenn Fehler gefunden werden

Testen entlang des V-Modells



Akzeptanztests
- Abnahmetests durch Anwender bzw. Kunden
- Fehler: Verfehlte Anforderungen, schlechte Umsetzung, …

Systemtests
- Testen des Gesamtsystems
- Fehler: Unerfüllte Funktionalitäten, schlechtes Systemverhalten, …

Integrationstest
- Testen Kommunikation zwischen Komponenten
- Fehler: Fehlerhafte Interfaces, falsche Abläufe, …

Unit-Tests (Komponententests)
- Testen individueller Funktionen, Unterprogramme, Klassen
- Fehler: Fehlerhafte Algorithmen oder Berechnungen, …

Was ist ein (guter) Testfall?

Grundlegende Definitionen

- Testobjekt: Zu testende Software(komponente)
- Testdaten: Daten, die vor Test-Ausführung existieren und Testausführung beeinflussen bzw. davon beeinflusst werden
  - Platzhalter (stub, dummy) ist Ersatz für bzw. Simulation von einem (noch) nicht vorhandenes Unterprogramm
- Testfall:
  1. Vorbedingungen
  2. Menge der Testdaten, die (vollständige) Ausführung des Testobjekts bewirken
  3. Menge der erwarteten Ergebnisse
  4. Erwartete Nachbedingungen

Ein guter Testfall ist…

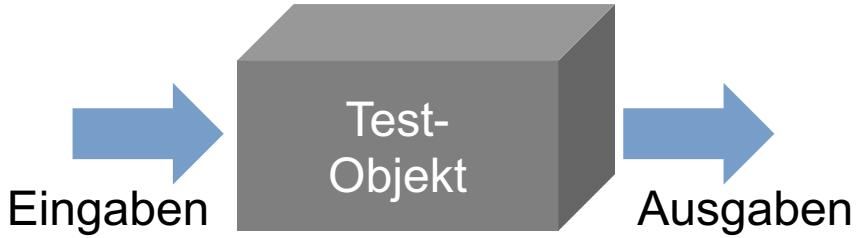
- Repräsentativ
  Stellvertretend für viele verschiedene Eingaben
- Fehlersensitiv
  Hohe Wahrscheinlichkeit Fehler zu produzieren
- Redundanzarm
  Prüft nicht, was andere bereits prüfen
- Ökonomisch
  Deckt maximal viele Fehler auf

Quelle: ISTQB/GTB Standardglossar der Testbegriffe, Version 2.1, Stand 09/2011

Zwei grundlegende, systematische Testverfahren

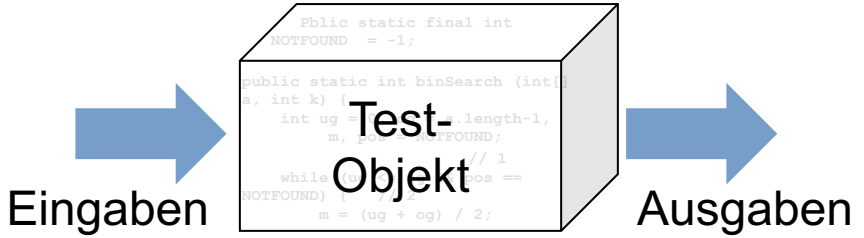
Funktionaler Test (“black-box test”)

- Außensicht auf Testobjekt
- Testfallauswahl beruht auf Spezifikation
  - Funktionsüberdeckung
  - Eingabeüberdeckung
  - Ausgabeüberdeckung



Strukturtest („white-box test“)

- Innensicht auf Testobjekt
- Testfallauswahl beruht auf Programmstruktur
  - Kontrollflussorientierter Test
    - Anweisungsüberdeckung
    - Zweigüberdeckung
    - Pfadüberdeckung
  - Datenflussorientierter Test
    - Definition/Verwendung von Variablen

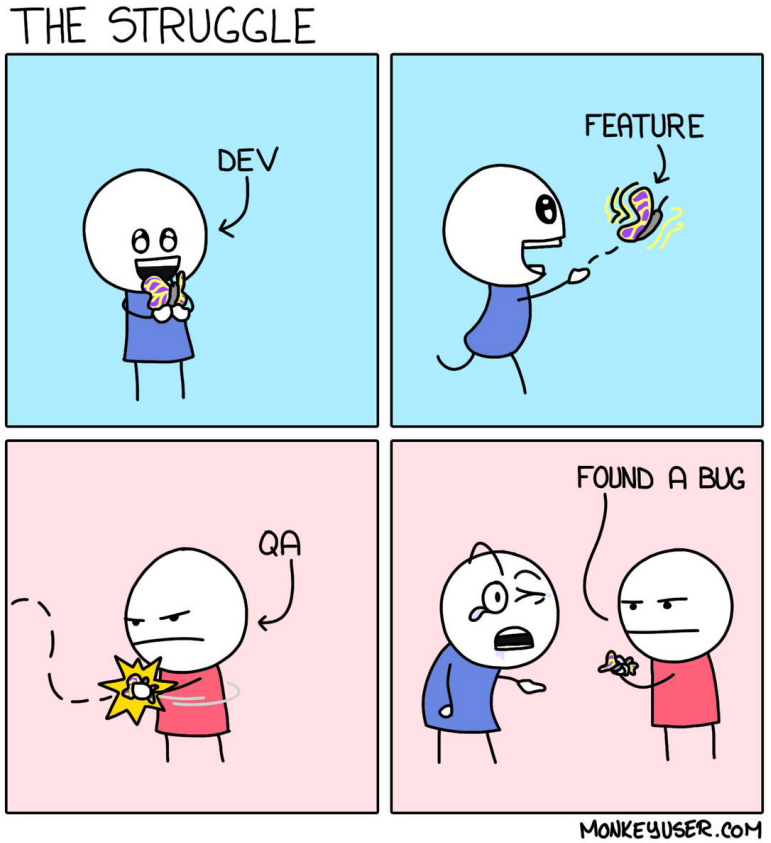


Fehlfunktionen

- Fehlfunktion (*fault*):
  Unerwartete Reaktion des Testlings

- Unterscheidung zwischen dem fehlerhaften Code (*error, bug*) und dem Auftreten der Fehlfunktion (*fault*):
  - Ein „fault“ kann aufgrund mehrerer „bugs“ auftreten.
  - Ein „bug“ kann mehrere „faults“ hervorrufen.

- Arten von Fehlfunktionen:
  - Falsches oder fehlendes Ergebnis
  - Nichtterminierung
  - Unerwartete oder falsche Fehlermeldung
  - Inkonsistenter Speicherzustand
  - Unnötige Ressourcenbelastung (z.B. von Speicher)
  - Unerwartetes Auslösen einer Ausnahme, "Abstürze"
  - Falsches Abfangen einer Ausnahme

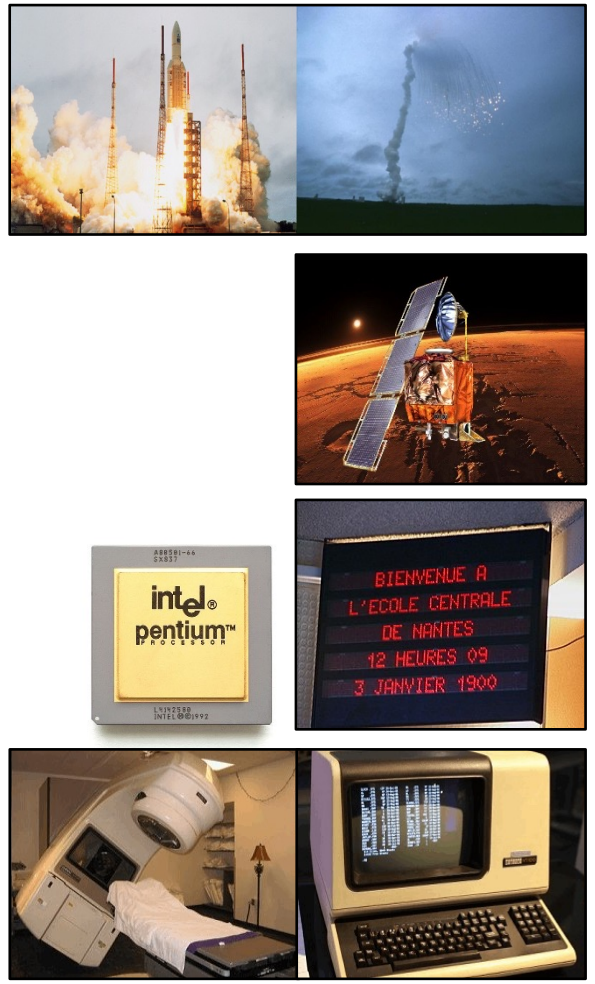


https://pbs.twimg.com/media/DYVLghYW0AE8dw2.jpg

- Testen kann nur die Anwesenheit von Fehlern nachweisen, nicht deren Abwesenheit!

Zusammenfassung – Grundlagen

- Ein Test ist erfolgreich, wenn Fehler gefunden werden
- Vollständiges Testen unmöglich → kann Programmkorrektheit nicht beweisen
- Systematisches, reproduzierbares Testen ist essentiell
- Fehlerfolgekosten: bis zu Milliarden Euro, Personenschäden
- Je früher man Fehler findet, desto günstiger ist es diese zu beheben
- Im V-Modell: Unit-Test → Integrations-Test → System-Test → Akzeptanztest
- Gute Testfälle finden ist eine wesentliche Herausforderung:
  - Funktionale Tests („black-box“)
  - Strukturtests („white-box“)



Softwaretechnik

10. Qualitätsmanagement
10.3. Funktionaler Test

Prof. Bernhard Rumpe
Software Engineering
RWTH Aachen

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

Literatur:

- Sommerville 19-20
- Balzert Band II, LE 14-15

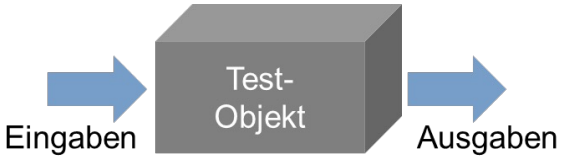
Speziell:

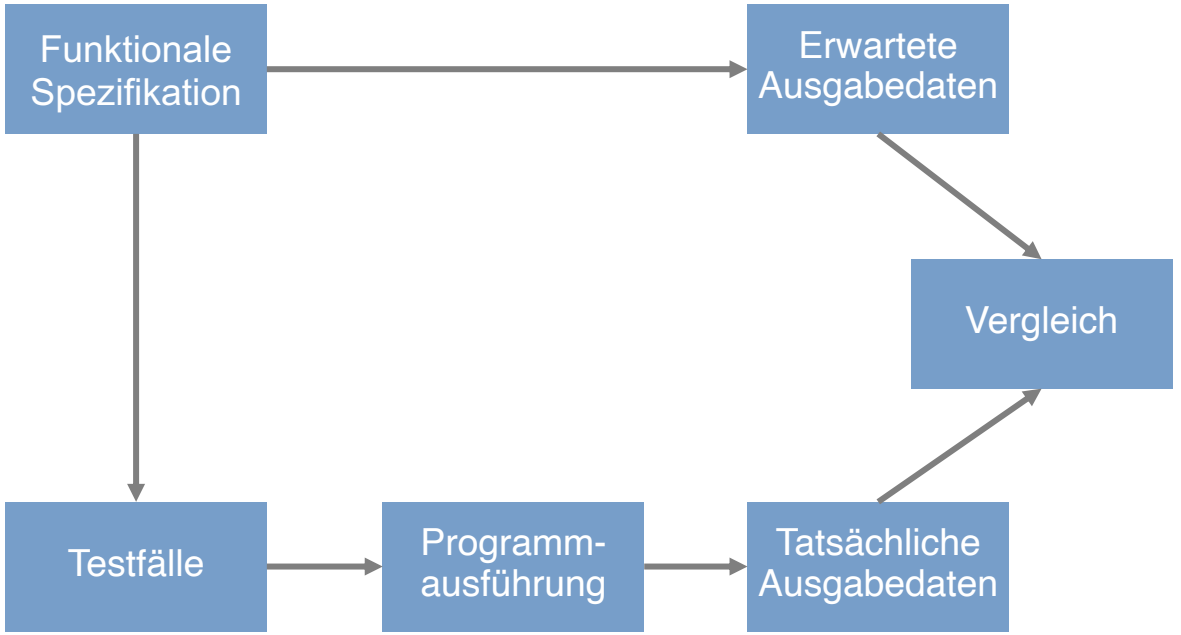
- Robert V. Binder: Testing object-oriented systems, Addison-Wesley 2000

SE Software Engineering | RWTH AACHEN UNIVERSITY

Funktionaler Test („black box“)

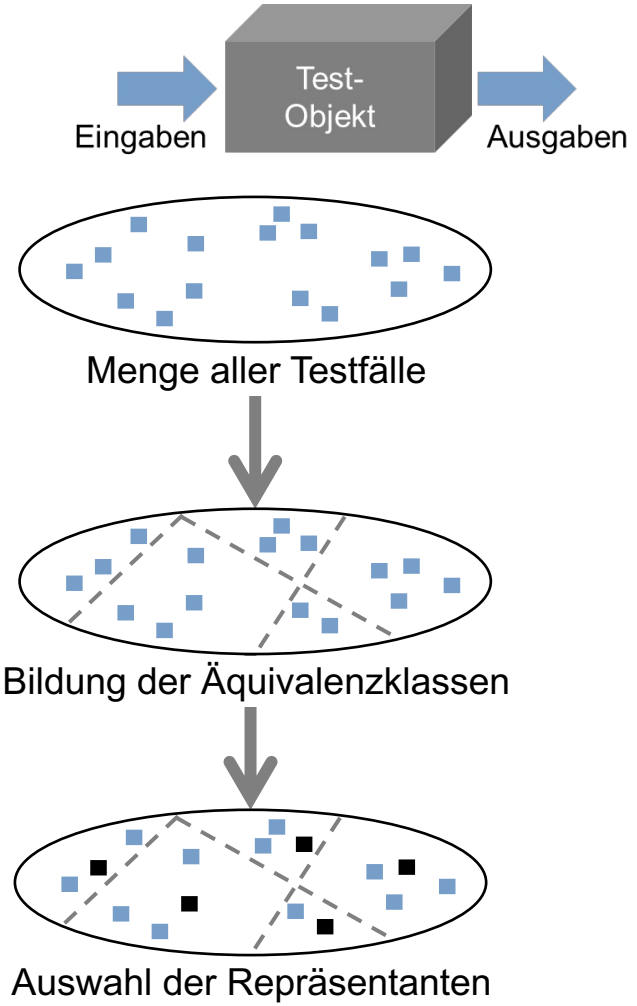
- Testen von Funktionen nach Spezifikation
- Unabhängig von konkreter Implementierung
- Möglichkeiten zur Auswahl der Testfälle:
  - Funktionsüberdeckung: Jede spezifizierte Funktion wird ausgeführt
  - Eingabeüberdeckung: Jede Eingabemöglichkeit wird mindestens in einem Testfall verwendet
  - Ausgabeüberdeckung: Jede Ausgabesituation wird mindestens einmal erzeugt
- Nachteile:
  - Zum Teil sehr aufwendig (→ alle Eingaben/Ausgaben)
  - Zusammenspiel verschiedener Funktionen unberücksichtigt





Identifikation von Testfällen mittels Äquivalenzklassen

- Ein guter Testfall ist repräsentativ, fehlersensitiv, redundanzarm, ökonomisch
- Ziel: Mit wenigen Testfällen möglichst große Wirkung erzielen
- Idee: Aufteilung der Eingabe- und Ausgabemöglichkeiten in Äquivalenzklassen
- Äquivalenzklasse:
  - Annahme: Testobjekt reagiert für alle Werte aus der Äquivalenzklasse prinzipiell gleich
  - Teilmenge der Eingaben & erwartete Ausgaben
  - Test eines Repräsentanten jeder Äquivalenzklasse
- Finden von Äquivalenzklassen:
  - Kriterien für Werte entwickeln und diese wo sinnvoll kombinieren
  - Zulässige und unzulässige Teilbereiche der Datenwerte



Beispiel für Äquivalenzklassen

- Funktion: `public Boolean check(Postleitzahl p, Stadt s);`
- Spezifikation: Postleitzahl… Positiv, 5-stellig, 01067 ≤ p ≤ 99998

in Dresden *zwei Städte in der Nähe von Erfurt*

Stadtname… String der Länge 2 bis 32 Zeichen, nur Buchstaben, „-“ und Leerzeichen

mehrere “Au” in BW, BY *Gemeinde Hellschen-Heringsand-Unterschaar in SH*

- Äquivalenzklassen

Zu testende Eigenschaft	Gültige Äquivalenzklassen	Ungültige Äquivalenzklassen
Postleitzahl: Länge	5-stellige Zahl-Strings	0-4-stellig, 6+-stellig
Postleitzahl: Wertebereich	01067 ≤ p ≤ 99998	p < 1067 ; p > 99998
Postleitzahl: String	String der ganze Zahlen darstellt	Reelle Zahlen
Stadtname: Länge		
Stadtname: Sonderzeichen		

Beispiel für Äquivalenzklassen

- Funktion: `public Boolean check(Postleitzahl p, Stadt s);`
- Spezifikation: Postleitzahl… Positiv, 5-stellig, 01067 ≤ p ≤ 99998

  Stadtname… String der Länge 2 bis 32 Zeichen, nur Buchstaben, „-“ und Leerzeichen
- Äquivalenzklassen

Zu testende Eigenschaft	Gültige Äquivalenzklassen	Ungültige Äquivalenzklassen
Postleitzahl: Länge	5-stellige Zahl-Strings	0-4-stellig, 6+-stellig
Postleitzahl: Wertebereich	01067 ≤ p ≤ 99998	p < 1067 ; p > 99998
Postleitzahl: String	String der ganze Zahlen darstellt	Reelle Zahlen
Stadtname: Länge		
Stadtname: Sonderzeichen		

Beispiel für Äquivalenzklassen

- Funktion: `public Boolean check(Postleitzahl p, Stadt s);`
- Spezifikation: Postleitzahl… Positiv, 5-stellig, 01067 ≤ p ≤ 99998

  Stadtname… String der Länge 2 bis 32 Zeichen, nur Buchstaben, „-“ und Leerzeichen
- Äquivalenzklassen

Zu testende Eigenschaft	Gültige Äquivalenzklassen	Ungültige Äquivalenzklassen
Postleitzahl: Länge	5-stellige Zahl-Strings	0-4-stellig, 6+-stellig
Postleitzahl: Wertebereich	01067 ≤ p ≤ 99998	p < 1067 ; p > 99998
Postleitzahl: String	String der ganze Zahlen darstellt	Reelle Zahlen
Stadtname: Länge	2-32 Zeichen	|s| ≤ zwei Zeichen; |s| ≥ 32 Zeichen
Stadtname: Sonderzeichen	Buchstaben, „-“ und Leerzeichen	Sonderzeichen („$“, „#“, „+“, „&“, „~“, …)

Testfälle verwenden Repräsentanten basierend auf Äquivalenzklassen

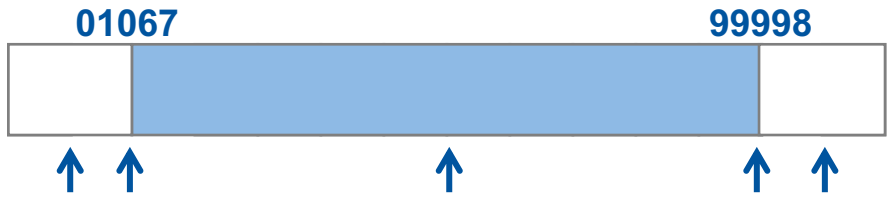
Zu testende Eigenschaft	Gültige Äquivalenzklassen	Ungültige Äquivalenzklassen
Postleitzahl: Länge	5-stellige Zahl-Strings	0-4-stellig, 6+-stellig
Postleitzahl: Wertebereich	01067 ≤ p ≤ 99998	p < 1067 ; p > 99998
Postleitzahl: String	String der ganze Zahlen darstellt	Reelle Zahlen
Stadtname: Länge	2-32 Zeichen	|s| ≤ zwei Zeichen; |s| ≥ 32 Zeichen
Stadtname: Sonderzeichen	Buchstaben, „-“ und Leerzeichen	Sonderzeichen („$“, „#“, „+“, „&“, „~“, …)

- Testfall für gültige Äquivalenzklassen (Ziel: Möglichst viele abdecken):
  - Eingabe: (`„52074“, „Aachen“`), Erwartetes Ergebnis: `true`
- Testfälle für ungültige Äquivalenzklassen (Ziel: Jeweils nur eine abdecken)
  - Eingabe: (`„5207“, „Aachen“`), Erwartetes Ergebnis: `false` // Postleitzahl zu kurz
  - Eingabe: (`„520740“, „Aachen“`), Erwartetes Ergebnis: `false` // Postleitzahl zu lang
  - Eingabe: (`„52076“, „Aach3n“`), Erwartetes Ergebnis: `false` // Stadtname beinhaltet ungültige Sonderzeichen

Identifikation von Repräsentanten: Grenzwertanalyse und Verwendung spezieller Werte

- Annahme: Besonderes Verhalten ist am ehesten in besonderen Bereichen / an Grenzen zu erwarten
- Grenzwerte: Randfälle der Spezifikation
  - Ränder von Zahl-Intervallen
  - Schwellenwerte
- Spezielle Werte:
  - Zahlenwert 0
  - Leere Felder, Sequenzen und Zeichenreihen
  - Einelementige Felder, Sequenzen und Zeichenreihen
  - Null-Referenzen
  - Sonderzeichen (Steuerzeichen, Tabulator, „\r\n“)

Beispiel: Postleitzahl soll zwischen 01067 und 99998 liegen



Repräsentanten:

- Gültig: 01067; 02048; 99998
- Ungültig: 1066,9; 99999

- Analyse von Grenzwerte und spezielle Werten erleichtert Identifikation hervorragender Repräsentanten

Äquivalenzklassen: Beispiel

- Kriterien für Äquivalenzklassen:

```
int search (int[] a, int k)
Vorbedingung:
        a.length >= 0
Nachbedingung:
        (result ≥ 0 ∧ a[result] == k) ∨
        (result == -1 ∧ (¬∃ i . 0 ≤ i < a.length ∧ a[i] == k))
```

- Äquivalenzklassen: / Testfälle:

discuss

Grenzwertanalyse: Beispiel

- … auf der Basis der vorher definierten drei Äquivalenzklassen (i.e. drei Tests)
- Weitere Kriterien für Tests:

```
int search (int[] a, int k)
```

Vorbedingung:

a.length >= 0

Nachbedingung:

(result ≥ 0 ∧ a[result] == k) ∨

(result == -1 ∧ (¬∃ i . 0 ≤ i < a.length ∧ a[i] == k))

discuss

Neue Testfälle:

Grenzwertanalyse: Beispiel

- … auf der Basis der vorher definierten drei Äquivalenzklassen (i.e. drei Tests)
- Weitere Kriterien für Tests:

3A: Element am linken Rand	a[0]==k
3B: Element am rechten Rand	a.last==k
3C: Element an keinem Rand	...
4A: a einelementig	a.length==1
4B: a nicht einelementig	a.length!=1
5A: k ist 0	k==0
5B: k ist nicht 0	k!=0

```
int search (int[] a, int k)
```

Vorbedingung:

a.length >= 0

Nachbedingung:

(result ≥ 0 ∧ a[result] == k) ∨

(result == -1 ∧ (¬∃ i . 0 ≤ i < a.length ∧ a[i] == k))

⇦ discuss -- a result

Neue Testfälle:

a == [17], k == 17, result == 0	(Kriterien 1B, 2B, 3A+B, 4A, 5B)
a == [11, 23, 0], k == 0, result == 2	(Kriterien 1B, 2B, 3B, 4B, 5A)

Softwaretechnik

10. Qualitätsmanagement
10.4. Strukturtest

Prof. Bernhard Rumpe
Software Engineering
RWTH Aachen

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

Literatur:

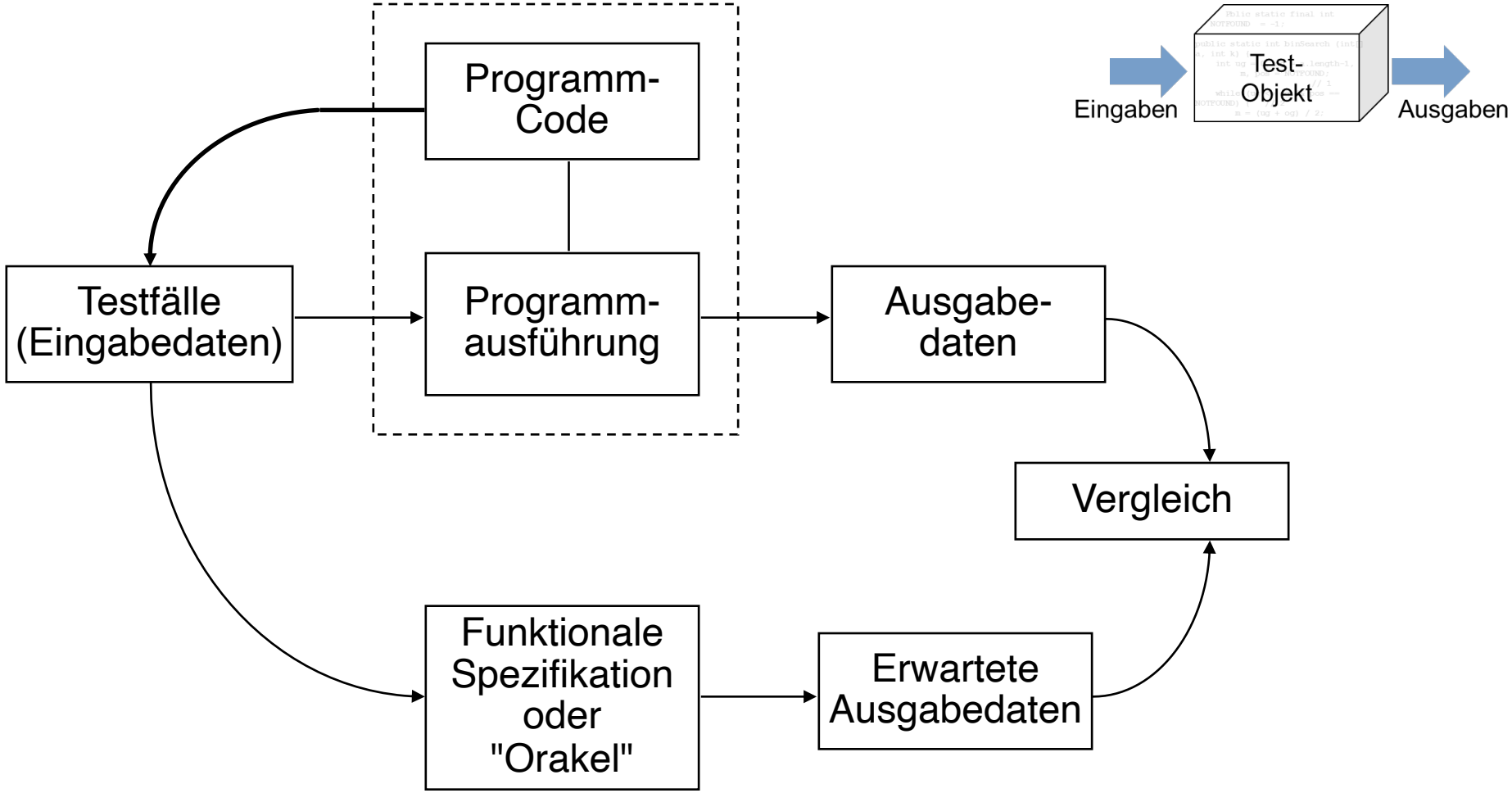
- Sommerville 19-20
- Balzert Band II, LE 14-15

Speziell:

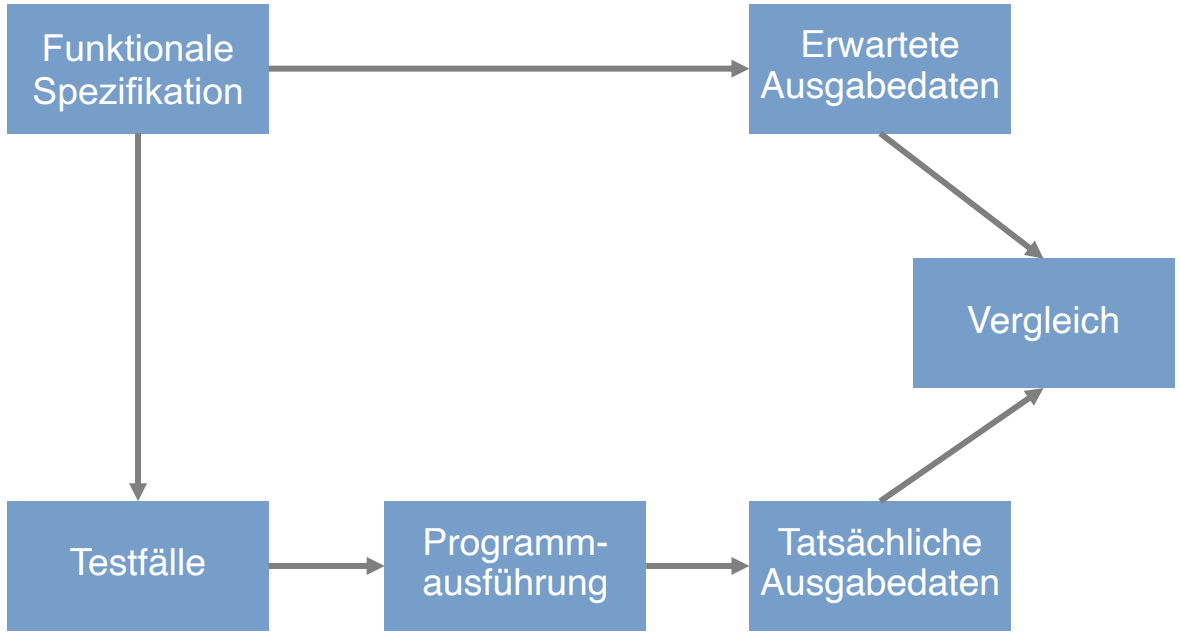
- Robert V. Binder: Testing object-oriented systems, Addison-Wesley 2000

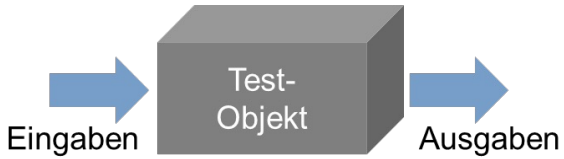
SE Software Engineering | RWTH AACHEN UNIVERSITY

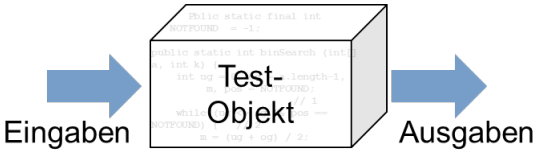
Strukturtest (*white box, glass-box*)

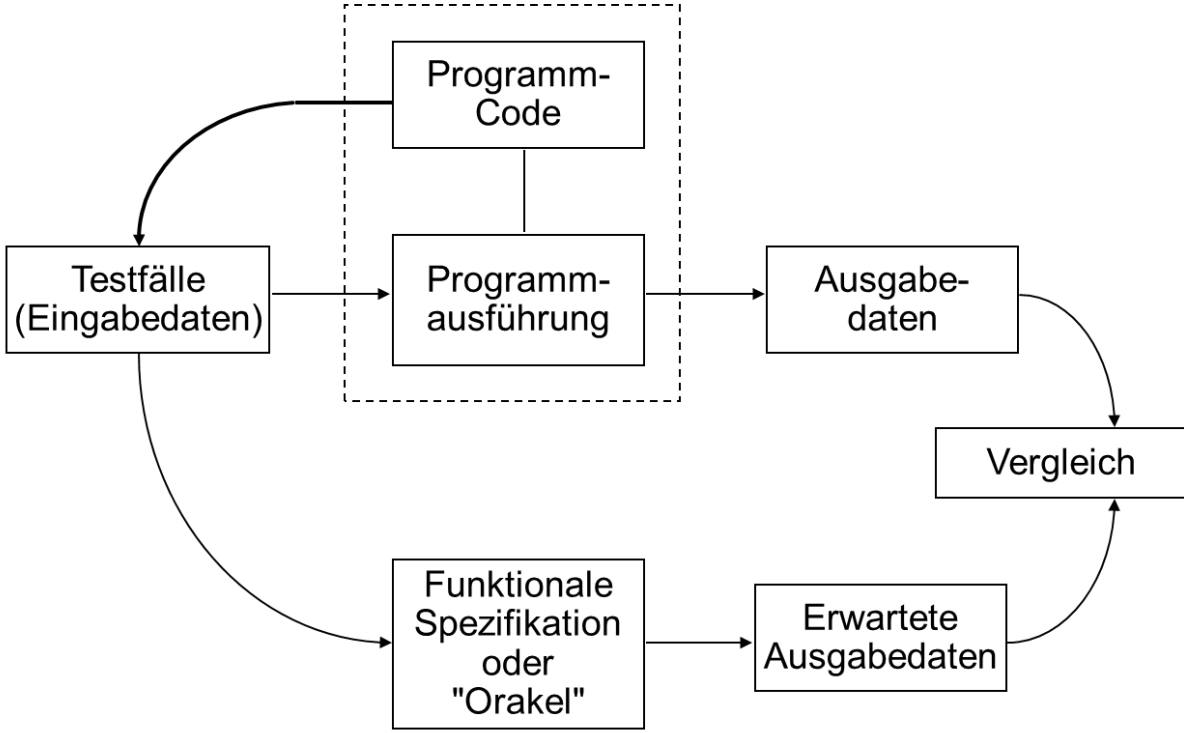


Funktionaler Test (Black Box) vs. Strukturtest (White Box)









Überdeckungsgrad (*coverage*)

- Maß für die Vollständigkeit eines Tests
- Welcher Anteil des Programmtexts wurde ausgetestet?
- Messung der Überdeckung:
  - Instrumentierung des Programmcodes
  - Ausgabe statistischer Informationen
- Planung der Überdeckung:
  - gezielte Anlage der Tests auf volle Überdeckung
- Überdeckungsarten:
  - Anweisungsüberdeckung: Anteil ausgeführter Anweisungen
  - Zweigüberdeckung: Anteil ausgeführter Anweisungen und Verzweigungen
  - Pfadüberdeckung: Anteil ausgeführter Programmablaufpfade
- Hilfsmittel:
  - Kontrollflussgraph

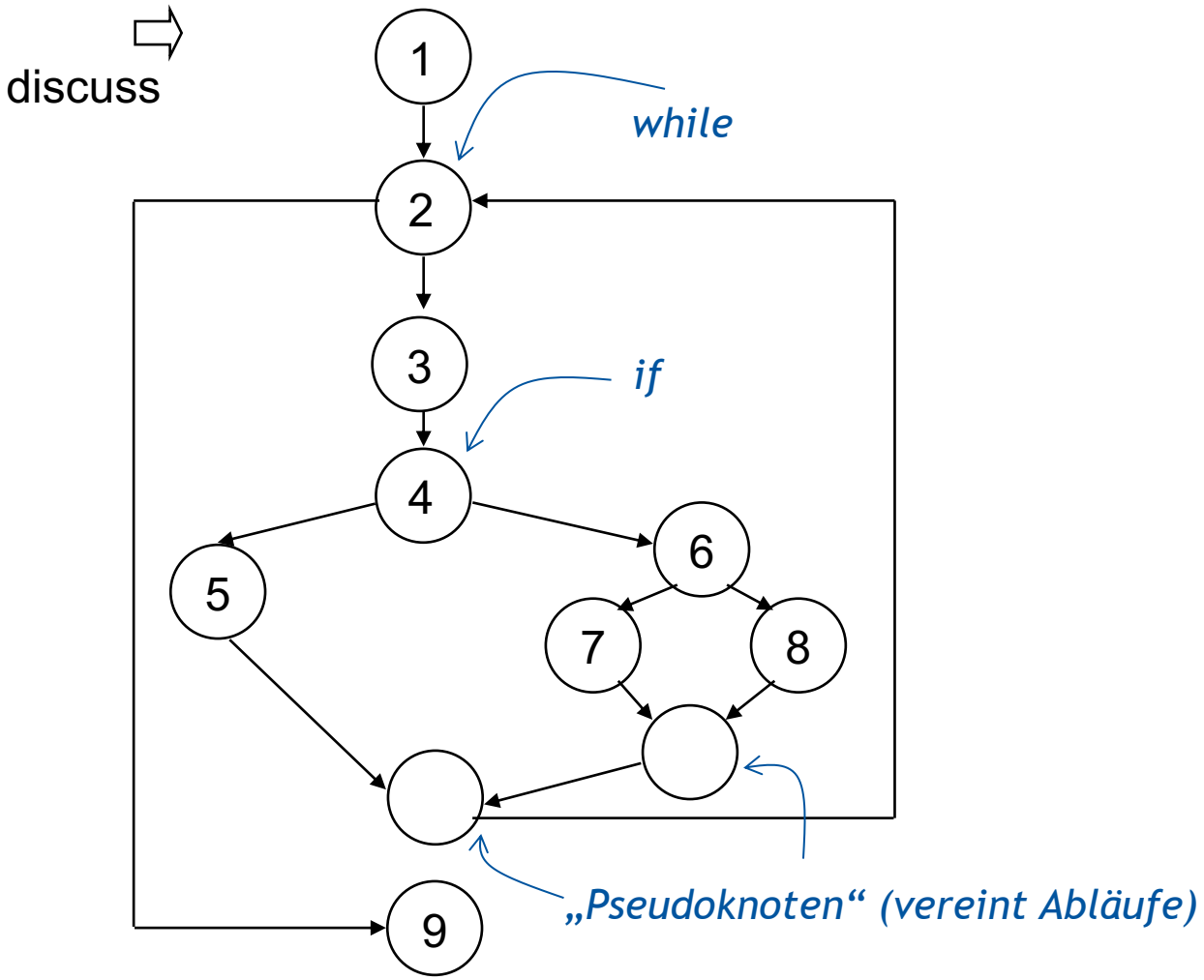
Beispielprogramm: Binäre Suche | Kontrollflussgraph

Java

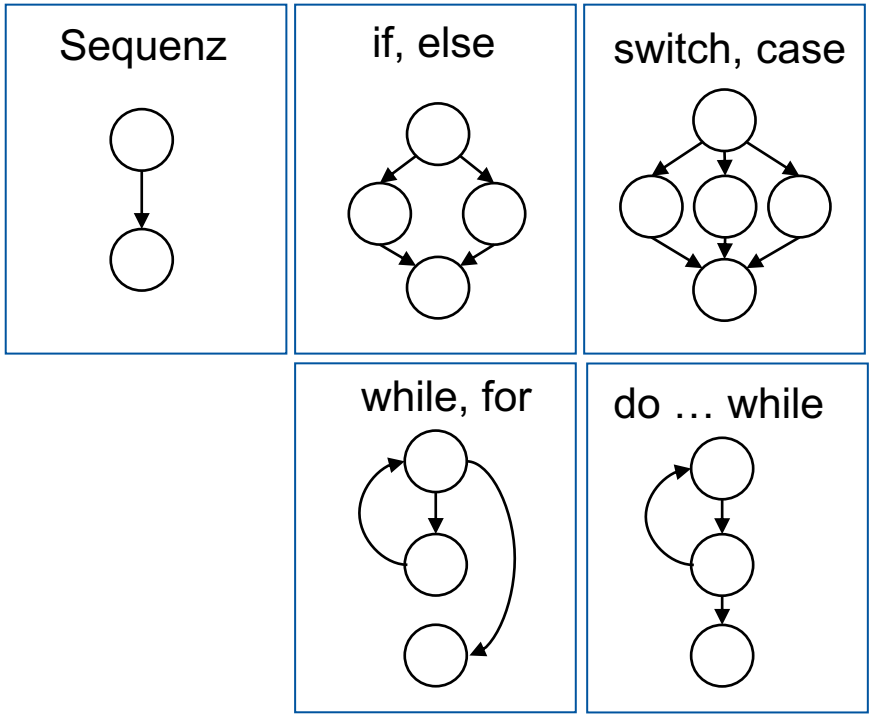
```
1  public static final int NOTFOUND = -1;
2
3  public static int binSearch (int[] a, int k) {
4    int ug = 0, og = a.length-1,
5              m, pos = NOTFOUND;
6    while (ug <= og && pos == NOTFOUND) {
7      m = (ug + og) / 2;
8      if (a[m] == k)
9        pos = m;
10     else
11       if (a[m] < k)
12         ug = m + 1;
13       else
14         og = m - 1;
15   }
16   return pos;
17 }
```

discuss

Beispiel Ergebnis: Kontrollflussgraph



Kontrollstrukturen



Anweisungsüberdeckender Test

- Überdeckung aller Anweisungen: *C_0-Test*
  - Jede Anweisung (nummerierte Zeile) des Programms wird mindestens einmal ausgeführt.

Beispiel:

  - `a = {11, 22, 33, 44, 55}, k = 33`
    - überdeckt Anweisungen:
      `1, 2, 3, 4, 5, 9`
  - `a = {11, 22, 33, 44, 55}, k = 15`
    - überdeckt Anweisungen:
      `1, 2, 3, 4, 6, 7, 8, 9`
  - Beide Testfälle zusammen erreichen volle Anweisungsüberdeckung.

Java

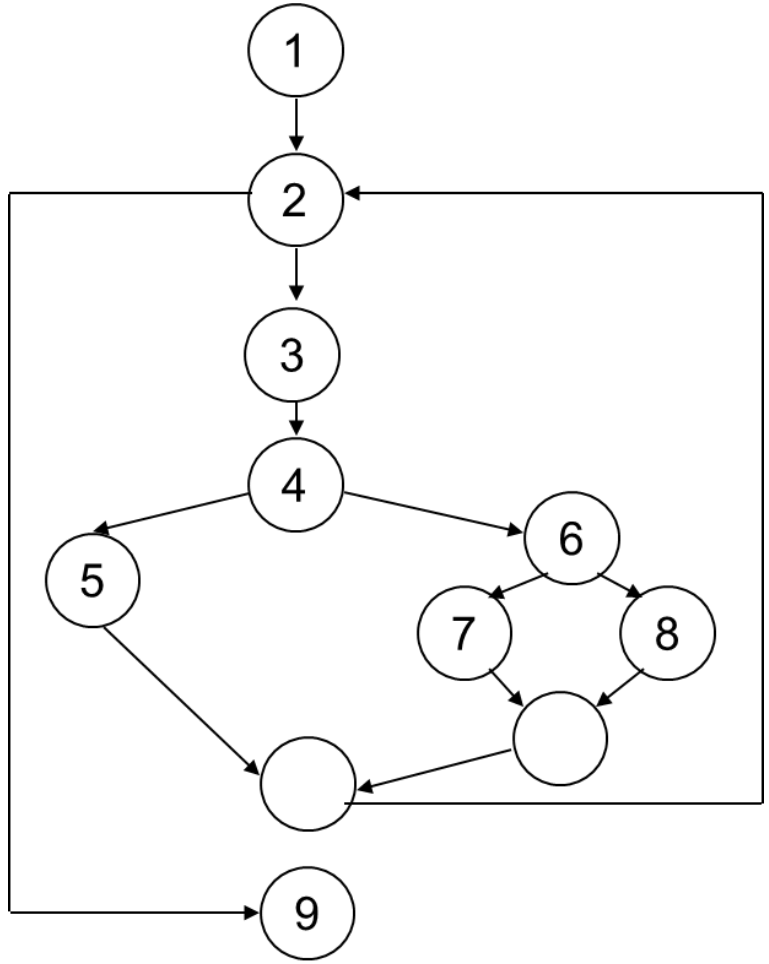
```
 1 public static final int NOTFOUND = -1;
 2
 3 public static int binSearch (int[] a, int k) {
 4   int ug = 0, og = a.length-1,
 5             m, pos = NOTFOUND;              // 1
 6   while (ug <= og && pos == NOTFOUND) {     // 2
 7     m = (ug + og) / 2;                      // 3
 8     if (a[m] == k)                          // 4
 9       pos = m;                              // 5
10     else
11       if (a[m] < k)                         // 6
12         ug = m + 1;                         // 7
13       else
14         og = m - 1;                         // 8
15   }
16   return pos;                               // 9
17 }
```

Zweigüberdeckender Test

- Überdeckung aller Zweige: *C_1-Test*
  - Bei jeder Fallunterscheidung (einschließlich Schleifen) werden beide Zweige ausgeführt (Bedingung=true und Bedingung=false).
  - Zweigtest zwingt auch zur Untersuchung "leerer" Alternativen:

```
1  if (x >= 1)
2    y = true;   // kein else-Fall
```

- *Beispiel:*
  - Die beiden Testfälle der letzten Folie überdecken alle Zweige.



Messung/Instrumentierung von Anweisungen

- Messen der Überdeckung:
  - "Instrumentieren" des Codes (durch Werkzeuge)
  - Einfügen von Zählanweisungen bei jeder Anweisung
  - Arrays für Anweisungen, oder „Bedingung True“, die hochgezählt werden:

- Messung/Instrumentierung von

- Anweisung Nummer s:

```
1 anweisungsZaehler[s]++;
2 ...
```

- Fallunterscheidung:

```
11 if (...) {                          Java
12   bedingungTrue[s]++; ...
13 } else {
14   bedingungFalse[s]++; ...
15 }
```

- While-Schleife:

```
21 while (...) {                       Java
22  bedingungTrue[s]++; ...
23 }
24 bedingungFalse[s]++;
```

Pfadüberdeckung

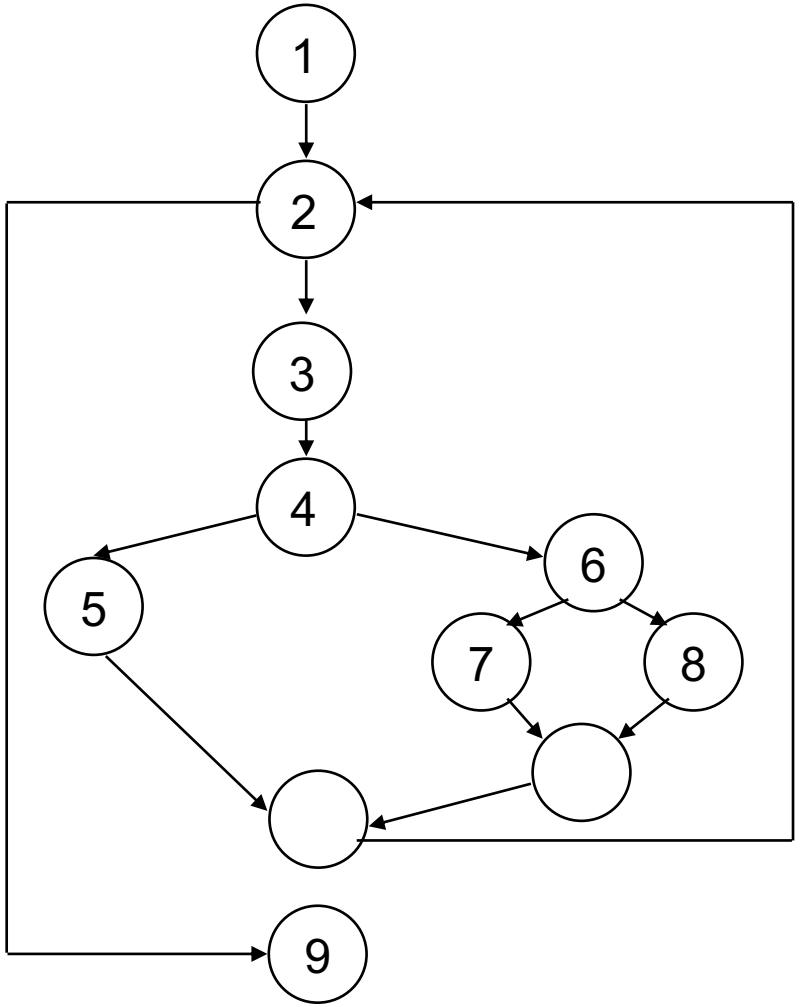
- Pfad =
  Sequenz von Knoten im Kontrollflussgraphen, so dass in der Sequenz aufeinanderfolgende Knoten im Graphen direkt miteinander verbunden sind.

  Anfang = Startknoten, Ende = Endknoten.

- Theoretisch optimales Testverfahren

- Explosion der Zahl von möglichen Pfaden, deshalb **nicht praktikabel**.
  (Schleifen haben unendliche Pfad-Mengen)

- Praktikablere Varianten, z.B. *bounded-interior-Pfadtest*: Alle Schleifenrümpfe höchstens n-mal (z.B. einmal) wiederholen

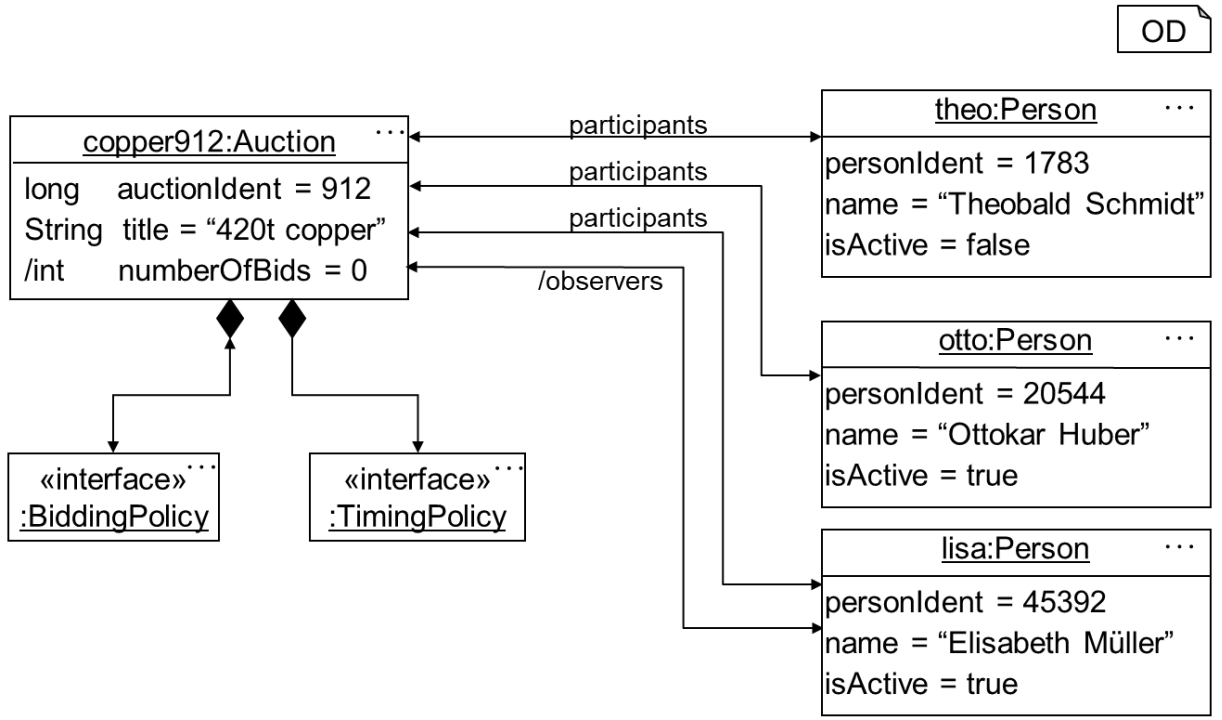


Testen objektorientierter Programme

- Klassische Algorithmen-Testverfahren sind anwendbar für einzelne Methoden
  - aber: OO-Methoden sind oft kurz und einfach
- Komplexität liegt zum großen Teil in der *Kooperation*.

- Weitere Probleme mit Substituierbarkeit durch Vererbung: Tests müssen für Unterklassen wiederholt werden:
  - Beeinflussung von Variablen der Oberklasse
  - Redefinition von Methoden der Oberklasse

- Spezielle Techniken für objektorientiertes Testen: Zustandstests, Sequenztests, etc.



Zustandstests

- Spezifikationsbezogen ("black box"):
  - Verwendung von Zustandsdiagrammen (z.B. UML) aus Analyse und Entwurf
  - Abdeckungskriterien:
    - alle Zustände
    - alle Übergänge
    - für jeden Übergang alle Folgeübergänge der Länge n
  - Praxistauglich

- Programmbezogen ("glass box"):
  - Automatische Berechnung eines Zustandsdiagramms
  - Zustand = Abstraktion einer Klasse zulässiger Attributwerte
  - Bestimmung der Übergänge erfordert symbolische Auswertung von Methoden
    - problematisch bei Schleifen und Rekursion
  - Im Forschungsstadium

Softwaretechnik

10. Qualitätsmanagement
10.5. Testfälle mit JUnit

Prof. Bernhard Rumpe
Software Engineering
RWTH Aachen

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

Inkrementelles Testen

- Programmierer testen meist nicht gerne.
  - „Testen ist zerstörerische Tätigkeit.“
  - Zu spätes Testen führt zu komplizierten Fehlersuchen!

- Inkrementelles Testen / Test-First-Ansatz:
  Tests entstehen parallel zum Code (oder sogar vor dem Code)

- Programmierer schreiben Tests für praktischen Nutzen:
  - Klare Spezifikation von Schnittstellen
  - Beschreibung kritischer Ausführungsbedingungen
  - Dokumentation von Fehlerbeseitigung
  - Festhalten von notwendigem Verhalten vor größerem Umbau ("Refaktorisierung")

- Tests archiviert und automatisch ausführbar
- Weitere Information:
  - K. Beck, E. Gamma: Programmers love writing tests (JUnit)
  - K. Beck: Extreme programming explained, Addison-Wesley 2000

Testfallerzeugung mit JUnit

- Testen ist wichtige Qualitätssicherungsmaßnahme
- Wichtig: wiederholbare, automatisierbare Testfallausführung

- Unit Test
  - Testen von (kleinen) Einheiten in Isolation von anderen (seiteneffektfrei)
  - Unit Test in OO ist typischerweise ein Methodentest

- White Box Test
  - Implementierungsdetails des Prüflings sind bekannt

- JUnit
  - kleines Test Framework
  - hat für Java schnell weite Verbreitung gefunden
  - Testfälle werden in Java kodiert (kein Sprachwechsel für Tests)
  - Framework kümmert sich um automatisierte Testfallausführung

JUnit & Test-First

- Test-First ist Grundsatz von XP (eXtreme Programming)
- „Test a little, code a little“

  - Wir überlegen uns erste Testfälle für die geforderte Funktionalität.
  - Wiederholung der Schritte 1-6:

    1. Auswahl des nächsten Testfalls.
    2. Wir entwerfen einen Test, der zunächst fehlschlagen sollte.
    3. Wir schreiben gerade soviel Code, dass sich der Test übersetzen lässt (Signatur).
    4. Wir prüfen, ob der Test fehlschlägt.
    5. Wir schreiben gerade soviel Code, dass der Test erfüllt sein sollte.
    6. Wir prüfen, ob der Test durchläuft.

  - Die Entwicklung ist abgeschlossen, wenn uns keine weiteren Tests mehr einfallen, die fehlschlagen können.

Anwendungsbeispiel: Konto

- Wir überlegen uns erste Testfälle
  - Erzeuge neues Konto (Account) für Kunden
  - Mache eine Einzahlung (deposit)
  - Mache eine Abhebung (withdraw)
  - Überweisung zwischen zwei Konten, ...

1. Auswahl des nächsten Testfalls:
   Erzeuge Konto

2. Wir entwerfen einen Test

- Ausführung der Tests durch Eclipse Plugins, CI-System oder Kommandozeile möglich:

- Kommandozeile:
  - `java -cp junit.jar org.junit.runner.JUnitCore AccountTest`

- Konventionen, z.B.:
  - Testmethoden beginnen mit „test“:
  - Sie führen die getesteten Methoden aus und prüfen das Ergebnis

```
21 // file AccountTest.java                                    Java
22 import org.junit.*;
23 public class AccountTest{
24
25  @Test
26  public void testCreateAccount() {
27   Account account = new Account("Anna");
28   assertEquals("Anna", account.getCustomer());
29   assertEquals(0, account.getBalance());
30  }
31 }
```

Annotation markiert Testmethode d.h. wird automatisch von JUnit aufgerufen

assert-Methoden prüfen Ergebnisse*

3. Wir schreiben gerade soviel Code, dass sich der Test übersetzen lässt

```java
// file Account.java
public class Account {
  public Account(String customer) {
  }
  public String getCustomer() {
    return null;
  }
  public int getBalance() {
    return 42;
  }
}
```

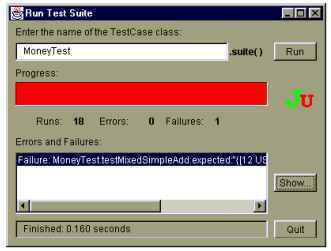
Java

```java
// file AccountTest.java
import org.junit.*;
public class AccountTest{

 @Test
 public void testCreateAccount() {
  Account account = new Account("Anna");
  assertEquals("Anna", account.getCustomer());
  assertEquals(0, account.getBalance());
 }
}
```

Java

- Die zu testende Klasse kennt die Testklasse nicht und kann daher auch ohne Tests eingesetzt werden.
- Übersetzbarkeit bedeutet: Signaturen + Default-Return-Werte sind vorhanden

4. Wir prüfen, ob der Test fehlschlägt:



5. Wir schreiben gerade soviel Code, dass der Test erfüllt sein sollte

```java
public class Account {
  private String customer;
  public Account(String customer) {
    this.customer = customer;
  }
  public String getCustomer() {
    return customer;
  }
  public int getBalance() {
    return 42;
  }
}
```

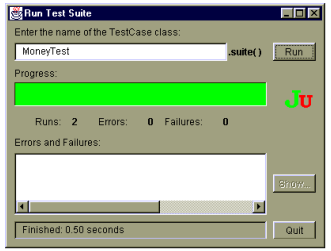
Java

```java
// file AccountTest.java
import org.junit.*;
public class AccountTest{

 @Test
 public void testCreateAccount() {
  Account account = new Account("Anna");
  assertEquals("Anna", account.getCustomer());
  assertEquals(0, account.getBalance());
 }
}
```

Java

- Im Beispiel werden deshalb Konstruktor, getCustomer und das notwendige Attribut (String customer) definiert.

6. Wir prüfen, ob der Test durchläuft:



Weiterer Test: Abheben

Java

```
1  public class AccountTest{
2   ...
3   private Account account;
4
5   @Before
6   protected void setUp() {
7    account = new Account("Anna");
8   }
9   @Test
10  public void testWithdraw() throws Exception {
11   account.deposit(100);
12   account.withdraw(60);
13   assertEquals(40, account.getBalance());
14   try {
15    account.withdraw(42);
16    fail("Missing AmountNotCoveredException");
17   } catch (AmountNotCoveredException expected) {}
18  }
19 }
```

Initialisieren: @Before - Methode wird vor jedem Test aufgerufen
Aufräumen: @After - Methode wird nach jedem Test aufgerufen

Mehrere „tests“ in einer Testklasse sind möglich

Achtung: Dieser eine Test reicht nicht aus, um Randfälle abzudecken!

Auch erwartete Exceptions können getestet werden

TestSuite – Testfälle organisieren

Java

```
1  import org.junit.runners.Suite;
2  import org.junit.runner.RunWith;
3
4  @Suite.SuiteClasses({AccountTest.class, ...<weitere>})
5  public class AllMyTests {
6
7          @BeforeClass
8          public static void myInitTestSuite(){
9                  // Daten für Testsuite vorbereiten
10         }
11         @AfterClass
12         public static void myTearDowntestSuite(){
13                 // Daten nach Ausführung der Testsuite aufräumen
14         }
15 }
```

Testklassen können in TestSuiten organisiert werden

Wird einmalig vor Ausführung der Testfälle in den Testklassen ausgeführt

Wird nach Ausführung aller Testfälle ausgeführt

Übersicht Junit Annotationen

- Annotationen für Klassen
  - `@RunWith(Suite.class)` `// deklariert die Klasse als Testsuite`
  - `@Suite.SuiteClasses({..})` `// deklariert die Testklassen der Suite`
- Annotationen für Methoden
  - `@Test` `// kennzeichnet einen Test`

    ```
    public void testWithdrawCash(){..}
    ```
  - `@Before` `// wird vor jedem Test ausgeführt`

    ```
    public void initDataset(){..}
    ```
  - `@After` `// wird nach jedem Test ausgeführt`

    ```
    public void resetDataset(){..}
    ```
  - `@BeforeClass` `// wird einmalig, vor dem ersten Test ausgeführt`

    ```
    public static void initTestDatabase() {..}
    ```
  - `@AfterClass` `// wird einmalig, nach dem letzten Test ausgeführt`

    ```
    public static void cleanupTestDatabase(){..}
    ```

Softwaretechnik

10. Qualitätsmanagement
10.6. Test Management

Prof. Bernhard Rumpe
Software Engineering
RWTH Aachen

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

Literatur:

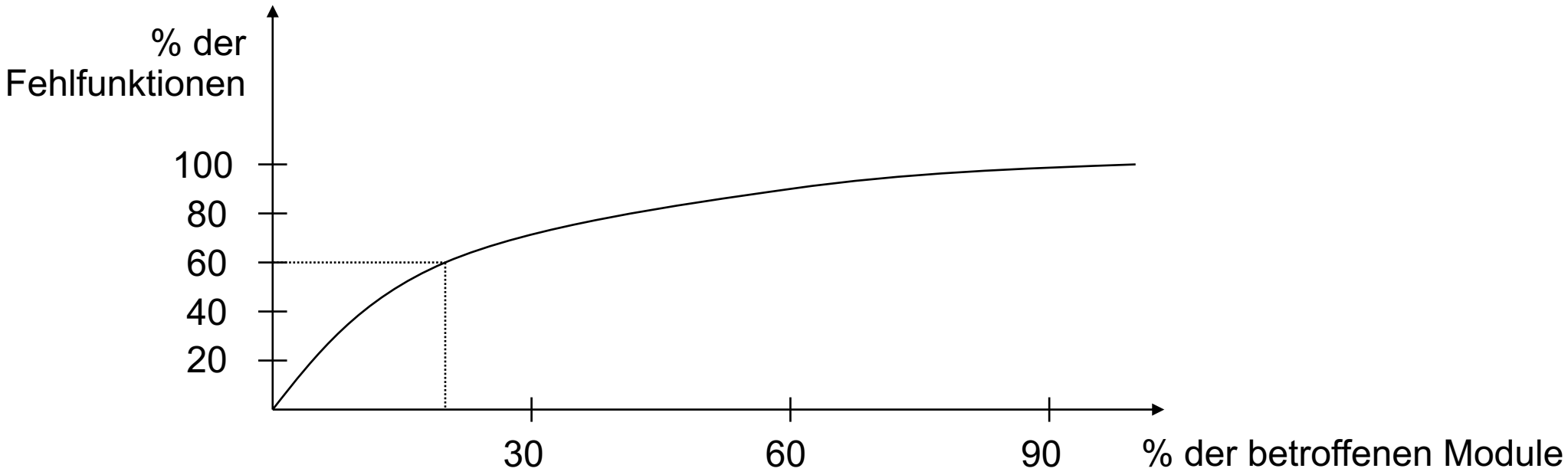
- Lwakatare, Lucy Ellen: DevOps adoption and implementation in software development practice: concept, practices, benefits and challenges, Dissertation, University of Oulu, 2017. URL: http://jultika.oulu.fi/files/isbn9789526217116.pdf
- Jez Humble and David Farley: Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation. Addison-Wesley. 2010

SE Software Engineering | RWTH AACHEN UNIVERSITY

Wohin mit dem Test-Code?

- Zur Durchführung von Tests entsteht zusätzlicher Code:
  - Testtreiber | Testattrappen (Stubs) | Testsuiten
- Testcode muss *aufbewahrt*, Tests nach allen größeren Änderungen *wiederholt* werden
- Alternativen:
  - "Spiegelbildliche" Struktur der Testklassen
    - - Redundanzproblem
    - + Erleichtert Verwendung von Test-Frameworks (z.B. JUnit)
  - Testskripte in eigenen Sprachen
    - klassischer Ansatz, - keine Vererbung von Testfällen möglich
  - Test-Unterklassen
    - -problematisch wegen Mehrfachvererbung
- Junit Ansatz: eigene Substruktur für Testklassen

Wo besonders intensiv testen? Pareto-Prinzip



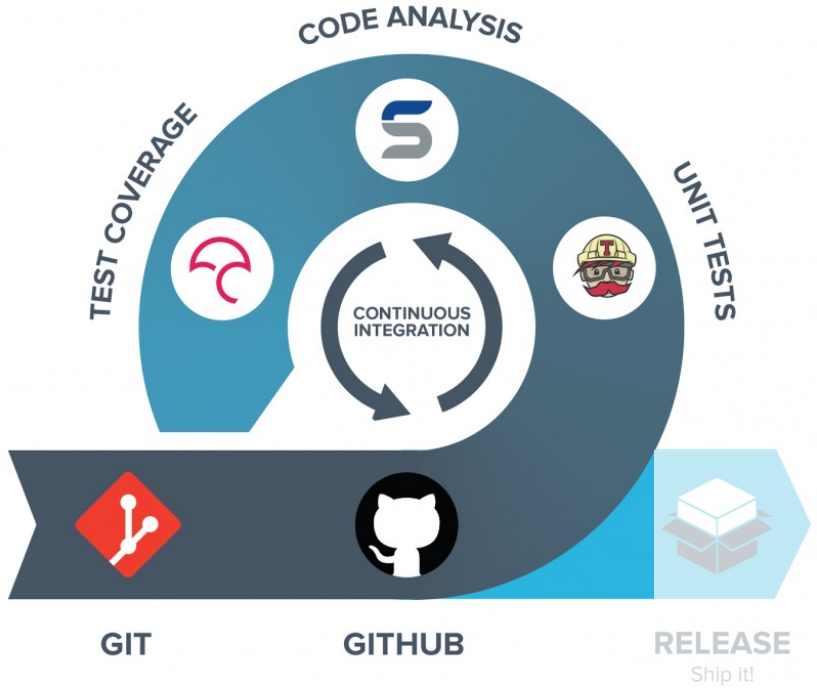
- Fenton/Ohlsson 2000 und andere empirische Untersuchungen:
  - 20 % der Module sind verantwortlich für 60 % der Fehler
  - Diese 20 % der Module entsprechen 30 % des Codes
- Testmuster: "Scratch'n sniff"
  - Fehlerhafte Stellen deuten oft auf weitere Fehler in der Nähe hin

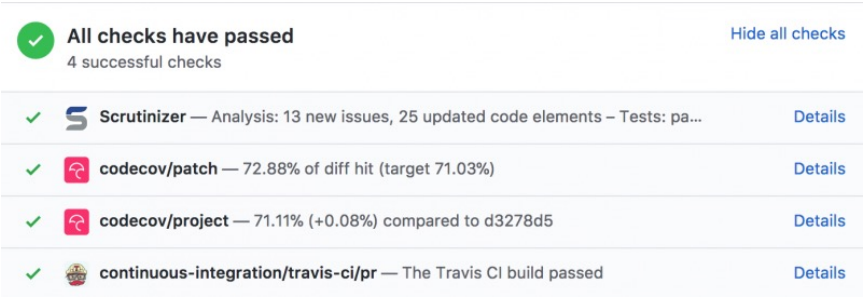
Continuous Integration

- Ständiges, automatisiertes Integrieren und Testen des Systems
- Zerteilung der großen Integrationsaufgabe in kleine Einheiten

- Möglich dank
  - Versionsverwaltung
  - Build-Automatisierung
  - Automatischer Testfallausführung

- Regelmäßiges Einchecken in kurzen Schritten
- System muss nach jedem Einchecken
  - zu kompilieren sein
  - Tests bestehen

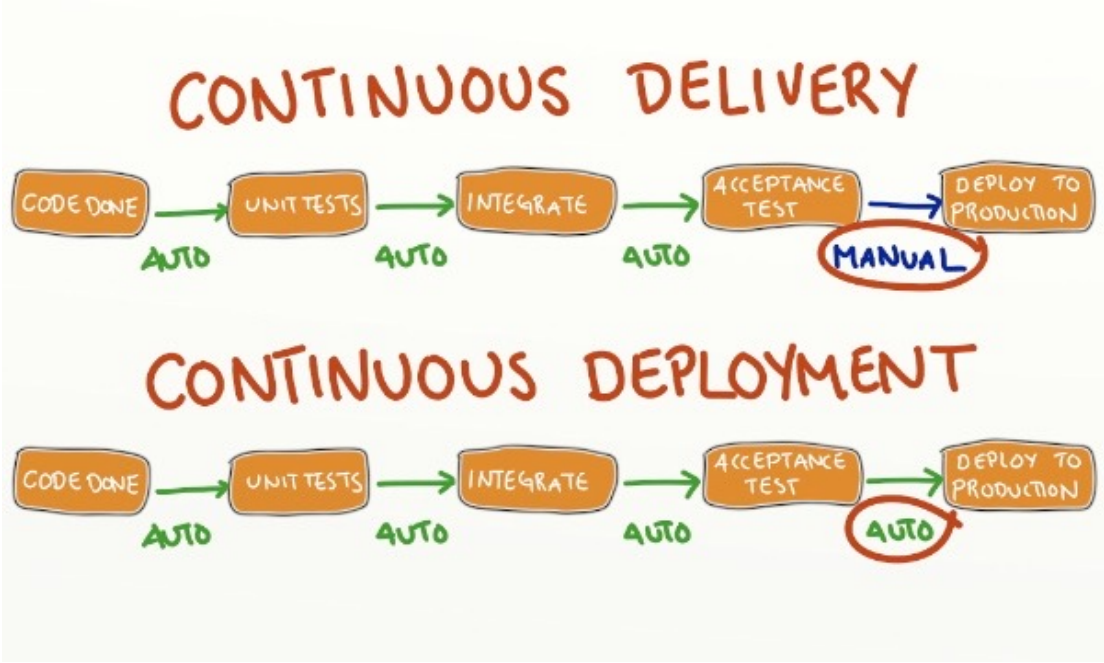




Quelle: https://www.silverstripe.org/blog/developers-how-we-use-continuous-integration-at-silverstripe/

Continuous Delivery/ Continuous Deployment

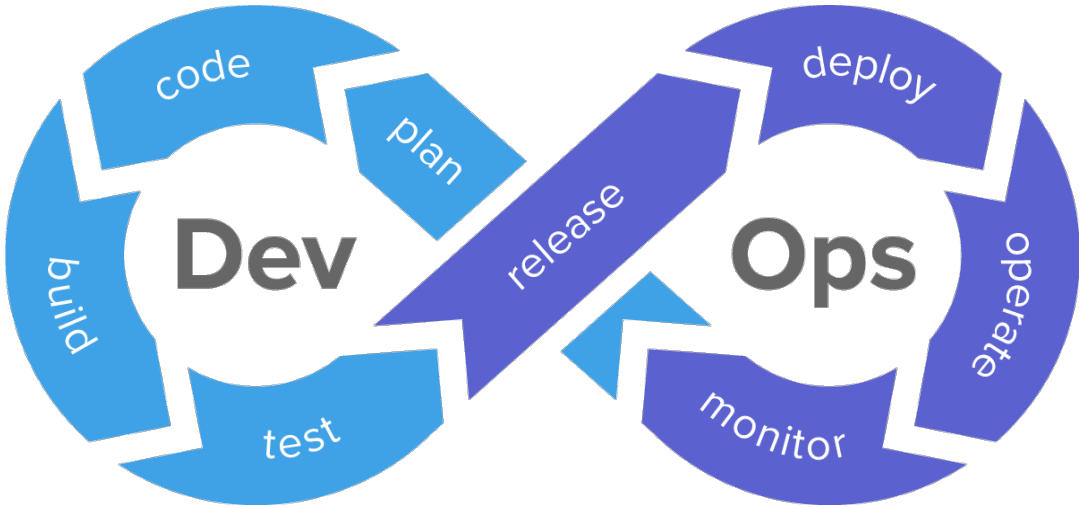
- Continuous Delivery (CDE)
  - Ziel: Anwendung muss immer in auslieferbarem Zustand sein
  - automatisierte Tests und Qualitätsüberprüfungen
  - Auslieferung an Kunden: Manueller Schritt (pull-based)
- Continuous Deployment (CD)
  - Voll automatische und kontinuierliche Auslieferung der Anwendung (push-based)
  - Für CD ist CDE notwendig, aber nicht umgekehrt
  - Nicht für alle Systeme und Organisationsstrukturen geeignet
    - z.B. Eingebettete Systeme mit Kopplung der Software an Hardware, Organisatorische Einschränkungen



Quelle: https://sdtimes.com/automation/guest-view-continuous-delivery-vs-continuous-deployment-whats-difference/

DevOps

- Development and Operations
  - Software Development/ Engineering (Dev)
  - Software Operations (Ops)
    - *“A process of putting into use, and supporting end user(s) in the use of software production in operational environment. Its activities include: installation, upgrade, migration, operational control, monitoring, configuration management, alerting, availability and support.”*[Lwakatare 17]
  - Getrennten Rollen wie Entwicklung, IT-Betrieb, Qualitätstechnik und Sicherheit, müssen zusammenarbeiten
  - Methoden:
    - CI/CD
    - Versionskontrolle
    - Agile Softwareentwicklung
    - Infrastruktur als Code (IaC)
    - Konfigurationsverwaltung
    - Continuous Monitoring



Was haben wir gelernt?

Qualitäts-management

… beginnt mit ersten Aktivitäten und begleitet das gesamte Projekt

beinhaltet
- Reviews, Inspektionen
- Tests

aber auch:
- Bildung von Prototypen,
- Einbindung von Kunden, intensive Kommunikation, gutes Arbeitsumfeld, etc.

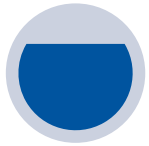
Test und Integration

…sehr komplexe Tätigkeit, die sehr viel Erfahrung erfordert

Tests sind gut zu planen und frühzeitig umzusetzen

Test-First-Ansatz

Stark eingebettet in SE Prozesse: CI/CD, DevOps

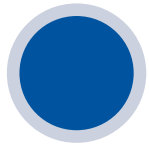


Arten von Tests

Funktionale Tests (black box)
- Äquivalenzklassen
- Grenzwertanalyse

Strukturtests (white box, glass-box)
- Überdeckungs-grad durch Kontrollflusspfad
- Anweisungs-, Zweig-, Pfadüberdeckung

OO: Spezifikations- oder Programm-bezogene Zustandstests



Testfälle mit JUnit

Testfallerzeugung

wiederholbare, automatisierbare Testfallausführung

Unit Tests: in OO Methodentest

White Box Test

Vorlesung Softwaretechnik

11. Werkzeuge

Prof. Bernhard Rumpe
Software Engineering
RWTH Aachen

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

SE Software Engineering | RWTH AACHEN UNIVERSITY

Warum, was, wie und wozu? Tools & Werkzeuge

Warum?

- Werkzeuge zur Unterstützung des Software Engineering Prozesses notwendig & nutzbar
- Vereinfachen Verwaltung, reduzieren Komplexität, verbessern Kommunikation…

Was?

- Versionierung
- Automatisierung
- Qualitätsmanagement
- Projektmanagement Plattformen

Wie?

- Wichtige Vertreter kennenlernen (es gibt noch viel mehr)
- Grundprinzipien verstehen
- Übersetzung Maven - Gradle

Wozu?

- Kennenlernen des Spektrums an Möglichkeiten
- In der Praxis: Verwendung der bereits vorhandenen Infrastrukturen
- Einführung neuer Werkzeuge

Werkzeuge für Software

- Software ist ein Werkstoff
- Zu ihrer Bearbeitung benötigt man Werkzeuge

Software tool –

A computer program used in the development, testing, analysis or maintenance of a program or its documentation. Examples include comparator, cross-reference generator, decompiler, driver, editor, flowcharter, monitor, test case generator, timing analyzer.

(IEEE Std 610.12)

Warum Werkzeuge?

- Vorteile
  - Verwaltung/Management vereinfacht
  - Monotone Aufgaben reduziert
  - Komplexität besser beherrschbar
  - Kommunikation zwischen Entwicklern verbessert
  - Dokumentation erleichtert
  - Produktivität gesteigert
  - Qualität gesteigert
- Nachteile
  - „A fool with a tool is still a fool“
  - Werkzeugauswahl schwierig
  - Anschaffung und Schulung teuer
  - Akzeptanz des Werkzeuges muss etabliert werden
  - Werkzeugintegration und –konfiguration nicht trivial

Werkzeuge für welche Zwecke?

- Editieren
  - Editoren für textuelle Notation (Bsp. Code)
  - Editoren für graphische Notationen (Bsp. UML)
  - Editoren für spezielle Dokumente (Bsp. Anforderungen)
- Transformieren
  - Compiler
  - Code-Generatoren
  - Code-Restrukturierung (Refactoring)
- Automatisieren
  - Build-Werkzeuge
  - Continuous Integration / Continuous Delivery (Micro Services)
- Prüfen und Messen
  - Testwerkzeuge
  - Metriken (Bsp. Analyse der Performance, Testabdeckung, …)
- Kollaboration & Kommunikation
  - Verteiltes Arbeiten
  - Wissensverwaltung (Bsp. Wiki)
  - Dokumentation
- Verwalten
  - von Versionen
  - von Änderungsanforderungen (Bugs, Requests)
  - des Projektes (Termine, Zuständigkeiten, Kosten)
  - von Dokumenten und ihren Abhängigkeiten

Weiteres Lernmaterial …

Manches lässt sich besser im **Selbststudium** lernen oder erarbeiten.

Deshalb gibt es nachfolgend Folien,
die in der Vorlesung nicht vorgestellt werden,
aber Inhalt der Vorlesung sind.
Sie eignen sich zum **Selbststudium**.

Hier finden sich diese Folien mit erklärenden Texte
und weiterem klausurrelevanten Material / Videos:

- https://se-rwth.de/swt_material/ bzw.
- https://se-rwth.github.io/swt_material/

Ggf. **Fragen** dazu bitte in der Vorlesung oder der Übung
möglichst bald stellen!

SE Software Engineering | RWTH AACHEN UNIVERSITY

Material für die Softwaretechnik Vorlesung

Material für die Softwaretechnik Vorlesung, Bernhard Rumpe

- Weil manche Inhalte besser in Ruhe (z.B zuhause) durchgearbeitet werden können, stehen hier einige Sätze kommentierter Folien aus der Vorlesung zur Verfügung.
- Diese werden in der Vorlesung nicht mehr vorgestellt, sondern bei jeweiligen Abschnitt als bekannt vorausgesetzt.

Softwaretechnik

11. Werkzeuge
11.1. Versionierung

Prof. Bernhard Rumpe
Software Engineering
RWTH Aachen

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

+ Selbststudium: se-rwth.de/swt_material/

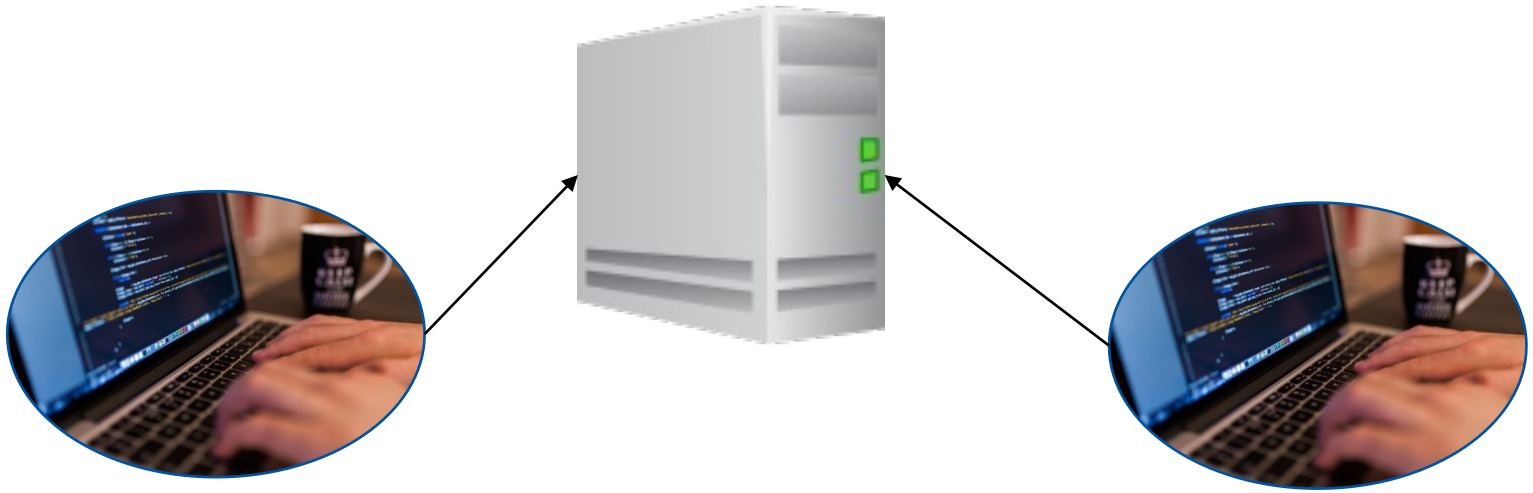
Literatur:
- Webseiten

Versionsverwaltung

- Versionsverwaltung von Software
  - Verwaltung von Versionen
  - Verwaltung von Varianten
  - Automatische Kennzeichnung mit Versionsnummer
- Gründe für die Versionsverwaltung von Software
  - Parallele Bearbeitung von Software durch mehrere Personen
  - Protokollierung durchgeführter Änderungen
  - Rücknahme von Änderungen möglich
  - Datensicherung des Quelltextes

Versionsverwaltungsserver

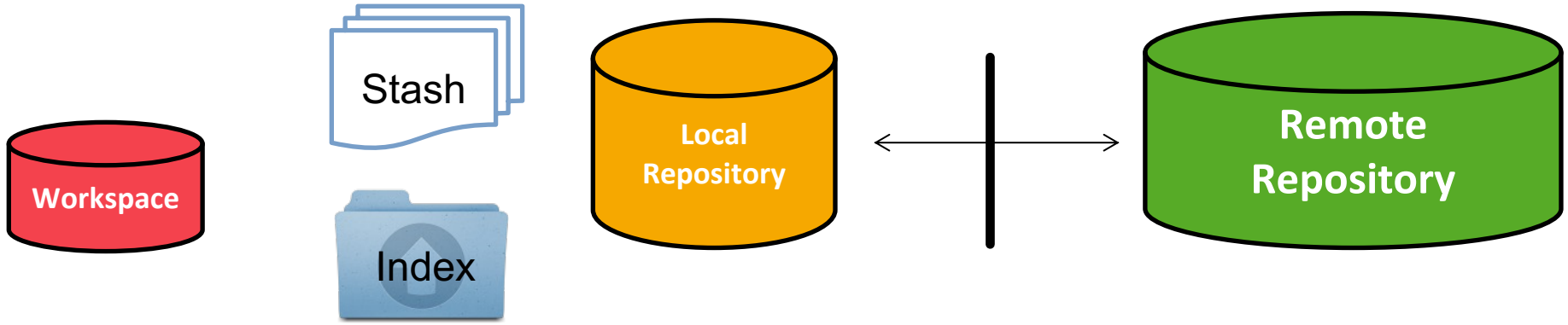
- Server
  - Hat Adresse
    - z.B. https://git.rwth-aachen.de/
  - Hat ein oder mehrere Repositories
  - Authentifizierung erfolgt
    - z.B. über Name und Passwort oder SSH-Key



git

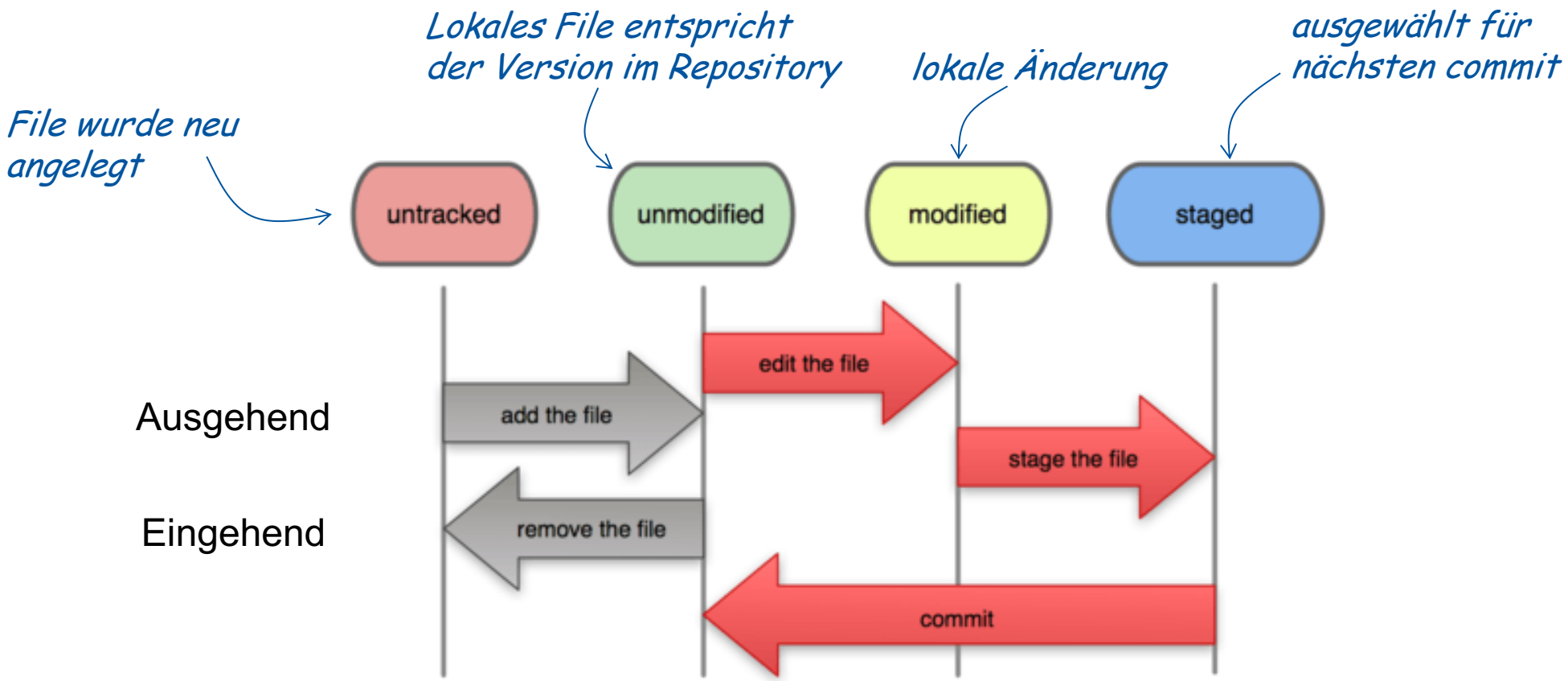
- http://git-scm.com/
- Entwickelt ursprünglich von Linus Torvalds (2005) zur Entwicklung des Linux Kernels
- Dezentrales Versionsmanagement
  - Jeder besitzt vollständige Kopie des Repositories inkl. History
  - Gemeinsame Verwaltung, aber auch
    - lokale Commits und lokale History (zur lokalen Verwaltung alter Versionen)
  - Varianten-Entwicklung in parallelen und verschmelzbaren (merge) Branches
- Git ist mächtig, aber auch komplex und bedarf entsprechender Vereinbarngen von Vorgehensweisen

Ablageorte bei Git



- Remote Repository
  - gemeinsames Repository (zentraler Zugriffspunkt, ggf. wird von das Continous Integration System gestartet)
- Local Repository
  - eigenes Repository (lokale Versionshistorie und Ablage für Teständerungen)
- Workspace
  - momentane Version an der gearbeitet wird (Verzeichnisinhalt!)
- Index / Staging Area / Stash
  - Liste vorbereitete Änderungen, die noch nicht durchgeführt wurden (z.B. neue Dateien added aber noch nicht committed)
  - Ablage für Änderungen und vorbereitete Änderungen, die nicht versioniert wurden (Stapel)

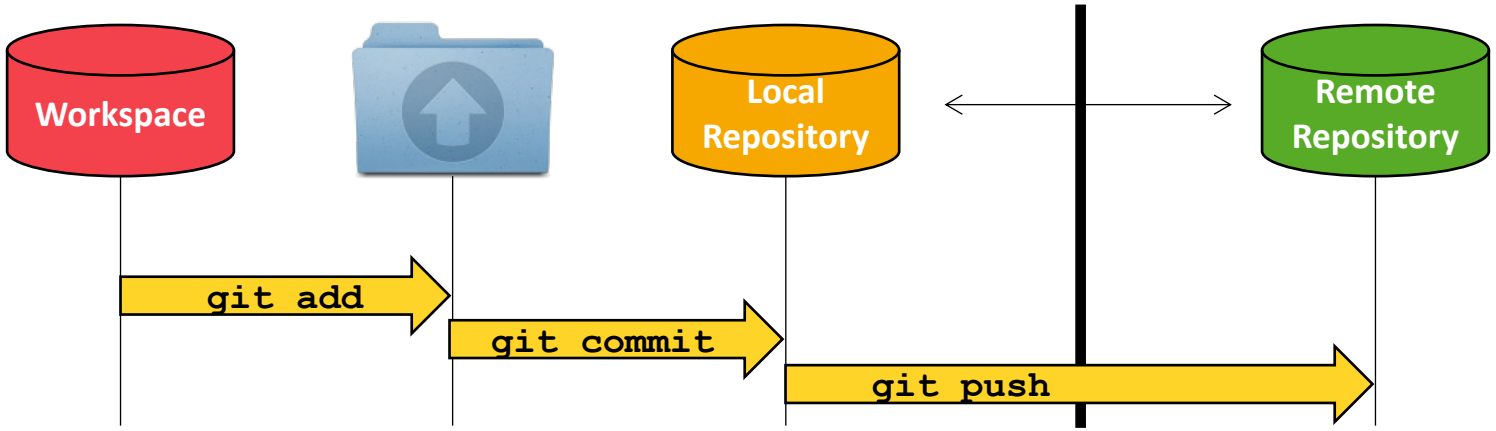
git: Status von Dateien (Zustandsmodell)



Quelle: http://git-scm.com/book/en/Git-Basics-Recording-Changes-to-the-Repository

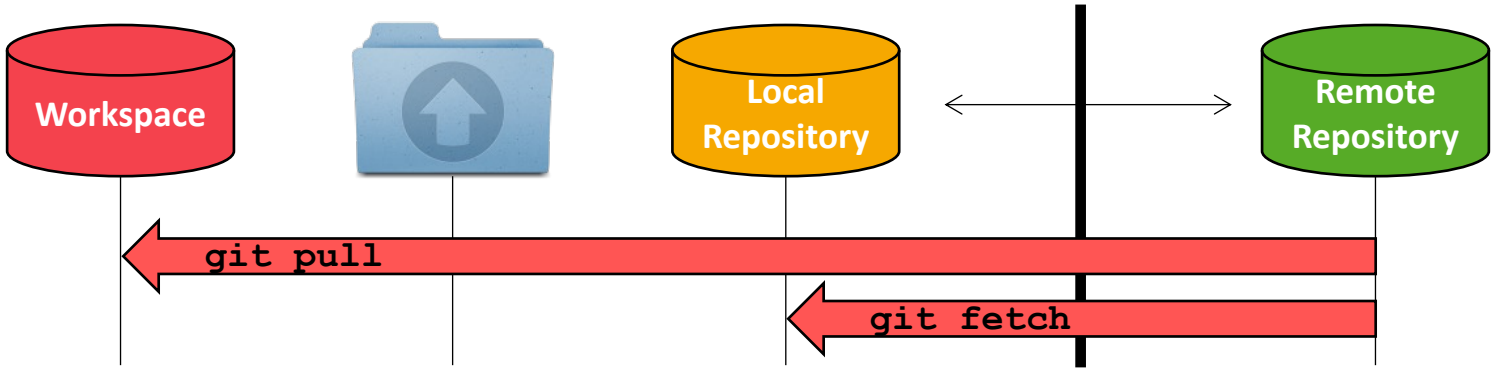
git: Änderungen nach remote überspielen

Annahme: Bisher alle Änderungen nur im Workspace



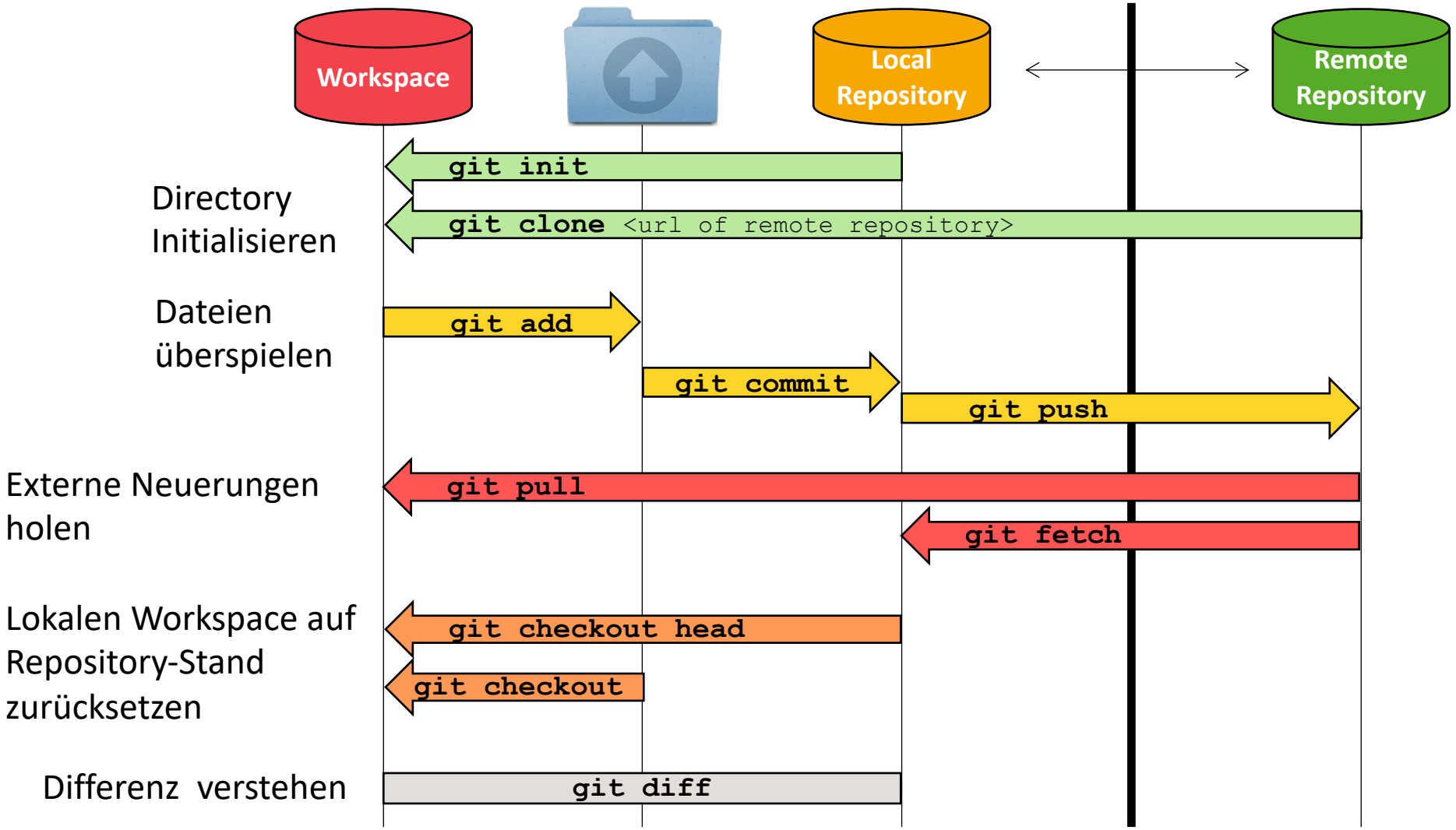
- **git add** <Dateiname>
  - Hinzufügen von Dateien und Verzeichnissen, die bisher nicht versionsverwaltet sind, in den Index
- **git commit**
  - Überspielen der lokalen Änderungen auf lokales Repository
- **git push**
  - Überspielen der Version im lokalen Repository auf entfernten Repository

git: von remote aktualisieren

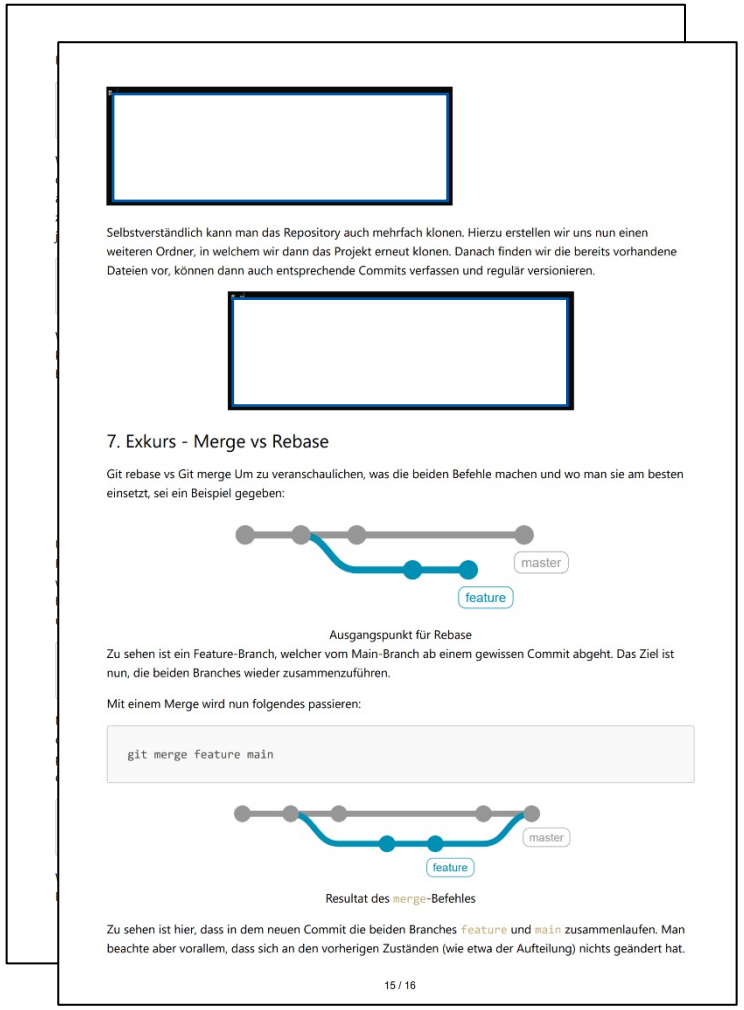


- **git pull**
  - Die aktuelle Version vom entfernten Repository laden und mit dem lokalen Zweig verschmelzen (merge)
- **git fetch**
  - Die aktuelle Version vom entfernten Repository laden und mit dem lokalen Repository verschmelzen

Git Workflows



Git zum Selbststudium



Selbstverständlich kann man das Repository auch mehrfach klonen. Hierzu erstellen wir uns nun einen weiteren Ordner, in welchem wir dann das Projekt erneut klonen. Danach finden wir die bereits vorhandene Dateien vor, können dann auch entsprechende Commits verfassen und regulär versionieren.

7. Exkurs - Merge vs Rebase

Git rebase vs Git merge Um zu veranschaulichen, was die beiden Befehle machen und wo man sie am besten einsetzt, sei ein Beispiel gegeben:

Ausgangspunkt für Rebase

Zu sehen ist ein Feature-Branch, welcher vom Main-Branch ab einem gewissen Commit abgeht. Das Ziel ist nun, die beiden Branches wieder zusammenzuführen.

Mit einem Merge wird nun folgendes passieren:

```
git merge feature main
```


Resultat des `merge`-Befehles

Zu sehen ist hier, dass in dem neuen Commit die beiden Branches `feature` und `main` zusammenlaufen. Man beachte aber vorallem, dass sich an den vorherigen Zuständen (wie etwa der Aufteilung) nichts geändert hat.

15 / 16

- Git ist für die Softwareentwicklung und für Ihre Klausur relevant
- Selbststudium ist hier aber die optimale Wahl.
- Wir bieten:
  - Nachschlagewerk mit praxisrelevanten Hinweisen
  - Übersichtsvideos auf moodle
  - Siehe dazu auch
    - https://se-rwth.de/swt_material/
- Immer möglich: das Internet

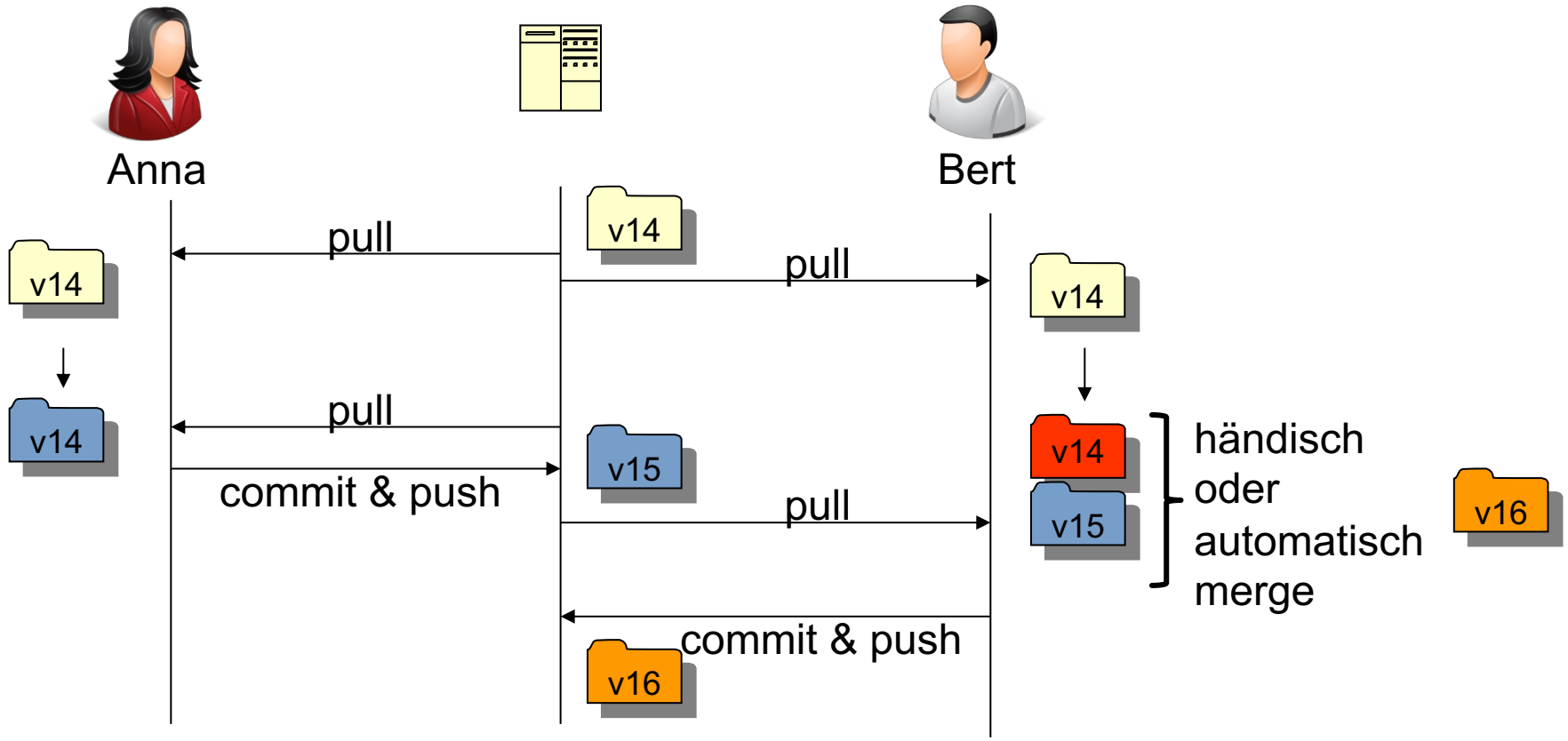
Regeln für die Nutzung einer Versionsverwaltung am Beispiel von Git

- Start der täglichen Arbeit:
  git pull
- Während der Arbeit:
  - Vor jedem Commit ein
    git fetch und
    git pull
    damit aktueller Stand
    - ggf. Konflikte beheben
- Commits vernünftig kommentieren

- Nicht machen:
- Nur lauffähigen Code committen
  - kompilierbar & getestet
  - das verhindert Ärger bei Kolleg:innen
- Keine generierten Dateien ablegen
  - (z.B. .class/jar-Dateien, pdfs)
- Keine personenspezifischen Konfigurationsdateien
  - Sonst treten ständig Konflikte auf

Parallele Bearbeitung von Quellcode im Team

- Parallele Bearbeitung von Quellcode wird problematisch, wenn zwei Personen gleichzeitig dieselbe Datei ändern.



git: Automatisches Merge

Repository

```
int x = 7;
x = x + x;
System.out.println(x);
```

pull/
clone

pull/
clone

```
int x = 555;
x = x + x;
System.out.println(x);
```

```
int x = 7;
x = x + x;
System.out.print(x);
```

commit
& push

```
int x = 555;
x = x + x;
System.out.println(x);
```

pull
(mit automatischem merge)

```
int x = 555;
x = x + x;
System.out.print(x);
```

```
int x = 555;
x = x + x;
System.out.print(x);
```

commit
& push

dabei ist sinnvoll eingestellt:
`git config --global pull.rebase true`

git: Konflikte, wenn im sleben File an selber Stelle

Repository

```
int x = 7;
x = x + x;
System.out.println(x);
```

pull/
clone

pull/
clone

```
int x = 7;
x = x * 2;
System.out.println(x);
```

```
int x = 7;
x = 2 * x;
System.out.println(x);
```

commit
& push

```
int x = 7;
x = x * 2;
System.out.println(x);
```

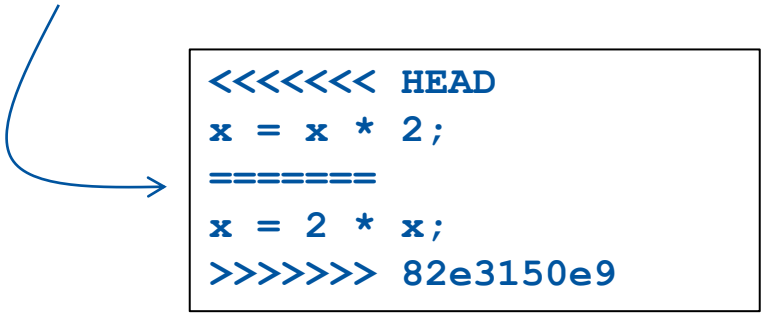
pull

```
int x = 7;
<<<<<<< HEAD
x = x * 2;
=======
x = 2 * x;
>>>>>>> 82e3150e9
System.out.println(x);
```

manuell zu korrigieren, um merge auszuführen

Merge und Konfliktbehebung

- Bei Textdateien (z.B. reiner Text, Java, HTML, etc.)
  - Automatisches Merge, falls kein Konflikt vorliegt
  - Dateien enthalten im Konfliktfall den relevanten Abschnitt aus beiden Versionen
  - müssen im Konfliktfall mit einem Editor geöffnet und manuell wieder konsistent gemacht werden

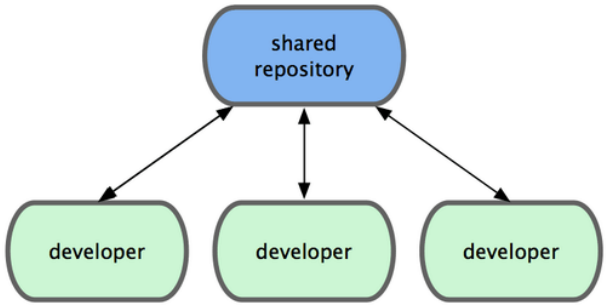


- Bei Binärdateien (z.B. Word, Excel, Grafikformate)
  - Kein automatisches Merge
  - Werkzeugunterstützung zur Konfliktbehebung oft nicht gegeben
    - Stattdessen liegen mehrere Varianten im Workspace
  - Es bleibt nur: Auswahl einer der Varianten + ggf. manuelle Integration von Änderungen

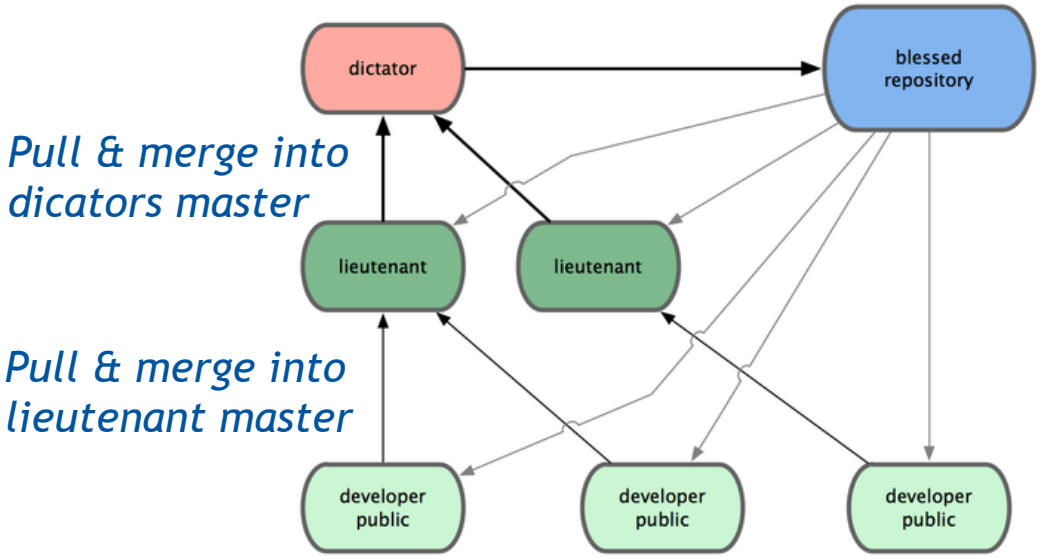
Repository Management Workflows | Schreibrechte auf Branches/Master

Zentralisiert
git: push für alle erlaubt

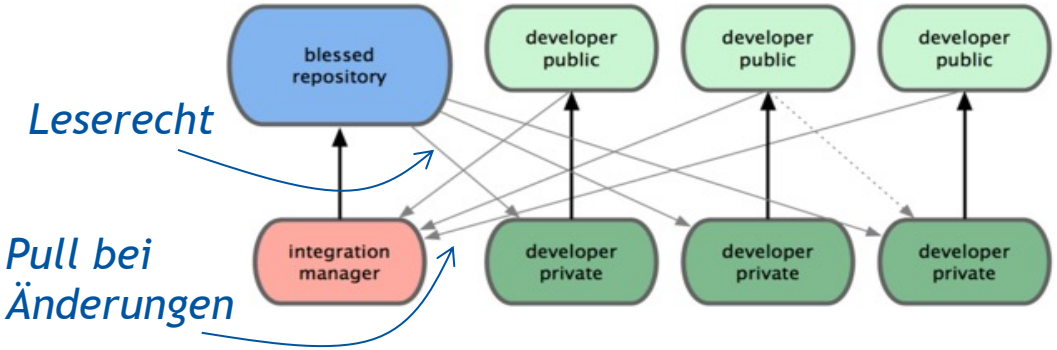
SVN kann das auch



Diktator und Lieutenant
Riesige Projekte mit vielen Beitragenden
z.B. Linux Kernel



Integration-Manager
git: Integration mehrerer remote Repositories.
Alternative: branching!



Quelle: http://git-scm.com/book/en/Distributed-Git-Distributed-Workflows

git: Weitere Dokumentation

git ist für SW Entwickler in der Praxis mittlerweile essentiell

- `git status` hilft oft
- `git log -n <#>` zeigt die letzten commits

- https://git-scm.com/book

- Git Troubleshooting: **http://ohshitgit.com/**

- *Literatur*
  - *Daily Git: Wie ein kompetenter Kollege Ihnen Git erklären würde*
    Martin Dilg, CreateSpace Independent Publishing Platform, 2014
    (auch als Kindle e-Book)

Softwaretechnik

11. Werkzeuge
11.2. Automatisierung

Prof. Bernhard Rumpe
Software Engineering
RWTH Aachen

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

+ Selbststudium: se-rwth.de/swt_material/

Literatur:
- Webseiten

Build-Automatisierung

- Aufgaben bei der Softwareerzeugung
  - Code kompilieren, binden und testen
  - Distributionen zusammenstellen
  - Tests durchführen
  - Testergebnisse kommunizieren
  - Dokumente generieren (Dokumentation, Protokolle, …)
  - Dokumente wieder aufräumen
  - Deployment der Software
  - Datenbank aufsetzen
  - Webserver konfigurieren
  - etc.

- Viele Aufgaben bei der Softwareerzeugung lassen sich automatisieren
  - Eintönige, sich wiederholende Arbeit vermeiden
  - Fehler vermeiden
  - Abhängigkeiten automatisch berücksichtigen
  - Inkrementell genau geänderte Teile neu erzeugen
    - nichts vergessen & effizient
  - Einheitlichen Erzeugungsprozess gewährleisten
- Automatisierung ist essentiell für effiziente Entwicklung!

Alternativen: Make, (Ant, Maven), Gradle

- Make
  - inkrementelle (und parallelisierbare) Build-Prozesse
  - Traditionelles Tool unter UNIX & Teil des POSIX-Standards
  - Ähnlichkeit mit UNIX Shellscripts
  - Derivate für viele Plattformen

GNU make

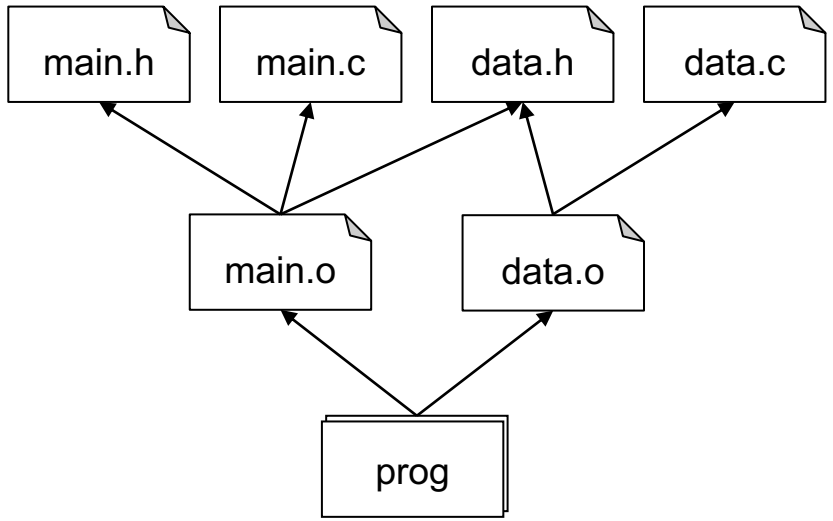
- Ant, Maven
  - Basiert auf Java
  - Konfigurierbar über XML-Dateien
  - Projekte der Apache Foundation (Open Source)
  - Aus Effizienzgründen sehr suboptimal

- Gradle
  - Verwendet Groovy basierte Skripte
  - inkrementelle und parallel ablaufende Build-Prozesse

Make

- Definiert Abhängigkeiten zwischen Artefakten im Projekt und
- Aktionen sie aktuell zu halten

creation dependency graph:
“A wird aus B erzeugt”



target (file)

dependencies (source files)

make

```
01 prog:    data.o main.o
02          cc data.o main.o -o prog
03
04 main.o: data.h main.h main.c
05          cc -c main.c
06
07 data.o: data.c data.h
08          cc -c data.c
09
10 clean:   rm *.o prog
```

actions to produce the target (shell commands)

- Aufruf: make prog
  - Erstellung des makefiles oft automatisierbar (automake)

Make

- Make bietet Variablen
  - für Lesbarkeit und leichtere Anpassbarkeit
- Fortgeschrittene Regeln für systematische Abhängigkeiten
- Beliebige Unix-Kommandos können als Kommandos eingesetzt werden
- Make ist ausgereift, parallelisierbar

make

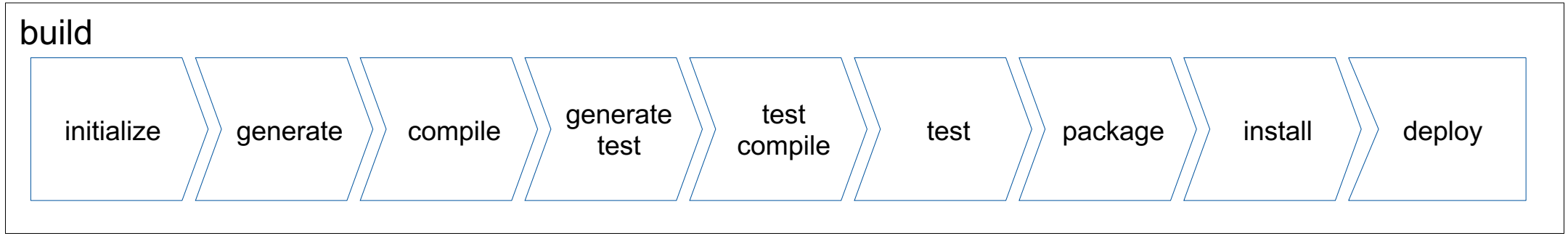
```
01 CP    = -classpath ...
02 JAVAC = /usr/bin/javac $(CP)
03
04
05
06
07 %.class: %.java
08       $(JAVAC) $<
```

Parametrisierte Regel: Target X.class hängt von Quelle X.java ab (für jede Klasse X)

$< steht für Quelle (.java-File)

Maven

- Projekt-Build besteht aus fest definierter Sequenz von Phasen
- Also keine Datei-Abhängigkeiten, sondern vorgegebene Phasen



- Aufruf eines Builds mit der in diesem Build zu erreichenden Phase (bspw. „mvn package“)
- Viele Konventionen eingebaut, aber Konfiguration ist auch möglich

- Gravierender Nachteil: Stupide Wiederholung der Aktionen, statt intelligente, effiziente Erkennung minimal notwendiger Aktivität

Maven

- Konfigurationsdatei: pom.xml (Auszug)
- Aufruf z.B. mvn test

```
01 <project xmlns="http://maven.apache.org/POM/4.0.0"
02   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
03     xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
04       http://maven.apache.org/xsd/maven-4.0.0.xsd">
05     <modelVersion>4.0.0</modelVersion>
06
07     <groupId>de.projectname</groupId>
08     <artifactId>parent</artifactId>
09     <version>2.0.1</version>
10     <packaging>pom</packaging>
11
12     <properties>
13       <hibernate.version>5.2.4.Final</hibernate.version>
14       ...
15     </properties>
16
17     <dependencies>
18       <dependency>
19           <groupId>junit</groupId>
20           <artifactId>junit</artifactId>
21           <version>4.12</version>
22           <scope>test</scope>
23       </dependency>
24     </dependencies>
25   ...
26 </project>
```

Eindeutige Identifikation z.B. Java Package

.jar Name

Programmversion

Eigenschaften & Versionen

Abhängigkeiten

Phase

Maven™

Maven

- Maven organisiert Abhängigkeiten zwischen Projekten
  - auch für importierte Repositories (also in Richtung anderer Projekte)

- Dependency Management
  - zwischen Repositories (pro Maven-Projekt eines)

- In öffentlichen Repositories (Maven Central, Nexus) sind annähernd alle bekannten Java Libraries enthalten
  + Unternehmen haben lokale Repositories
  + Snapshot-Verwaltung

Gradle

- Konfigurationsdatei: build.gradle (Auszug)

- Phasen
- 1) Konfiguration:
  Abhängigkeitsgraph zwischen Artefakten und zwischen Tasks erzeugen
- 2) Ausführung:
  Notwendige Tasks gemäß Graph abarbeiten
  und Dateien produzieren

Fügt spezielle Tasks hinzu

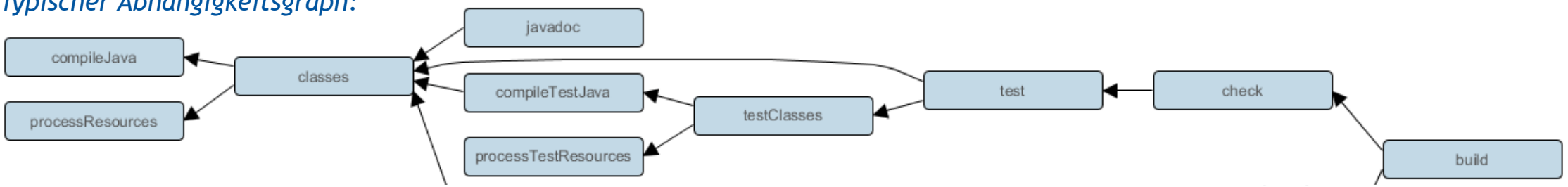
Definiert extra Variablen

Abhängigkeiten

Gradle

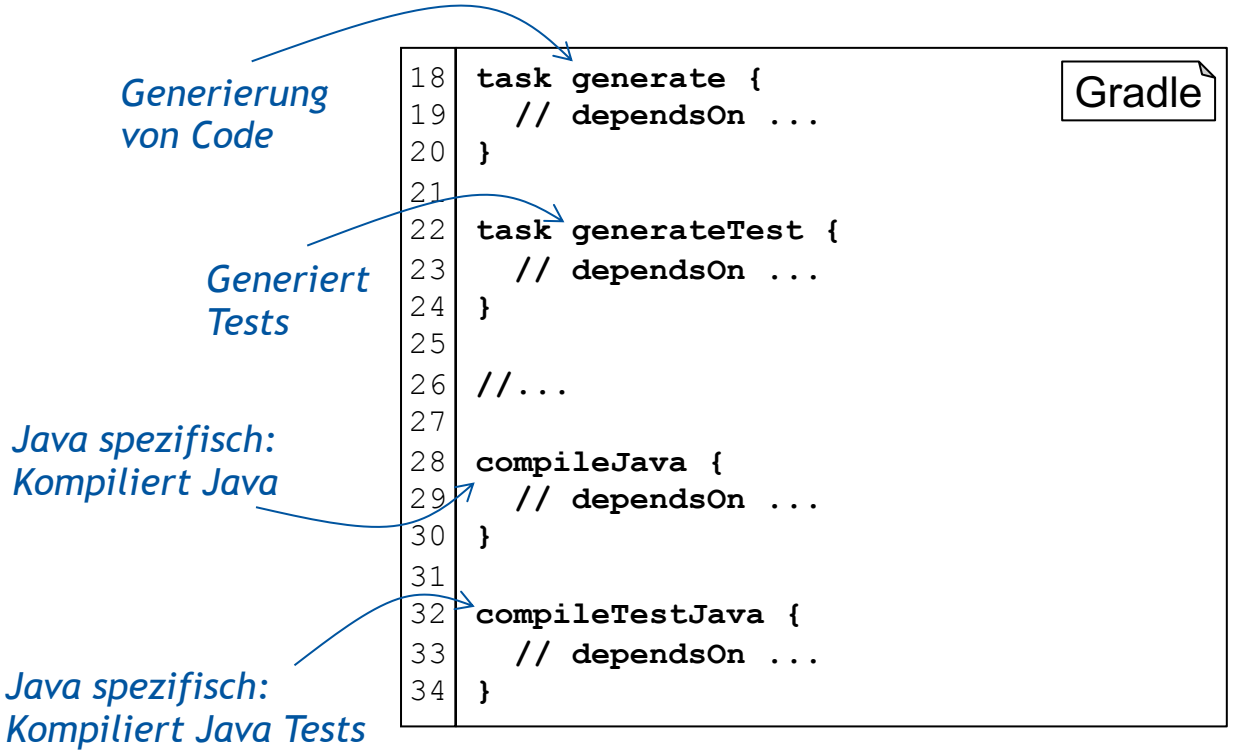
```
01 plugins {
02   id "java"
03   //...
04 }
05
06 ext {
07   grammarDir = "src/main/grammars"
08   guava_version = "23.0"
09   junit_version = "4.12"
10   //...
11 }
12
13 dependencies {
14   //...
15   implementation "com.google.guava:guava:$guava_version"
16   testImplementation "junit:junit:$junit_version"
17 }
```

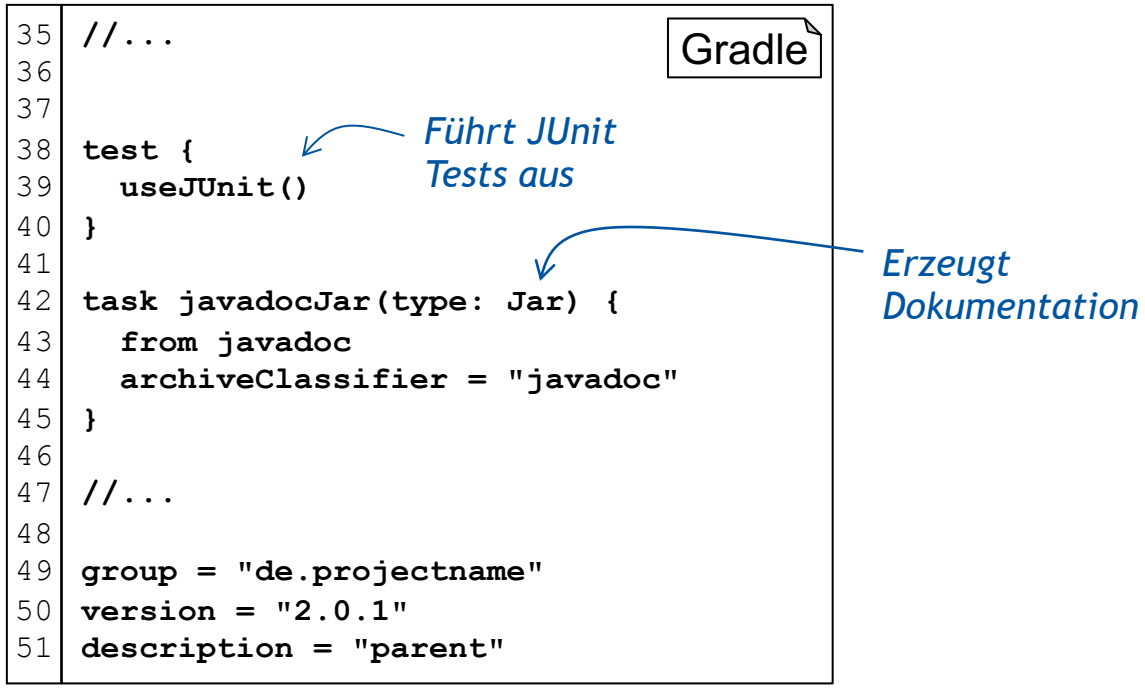
Typischer Abhängigkeitsgraph:



Gradle

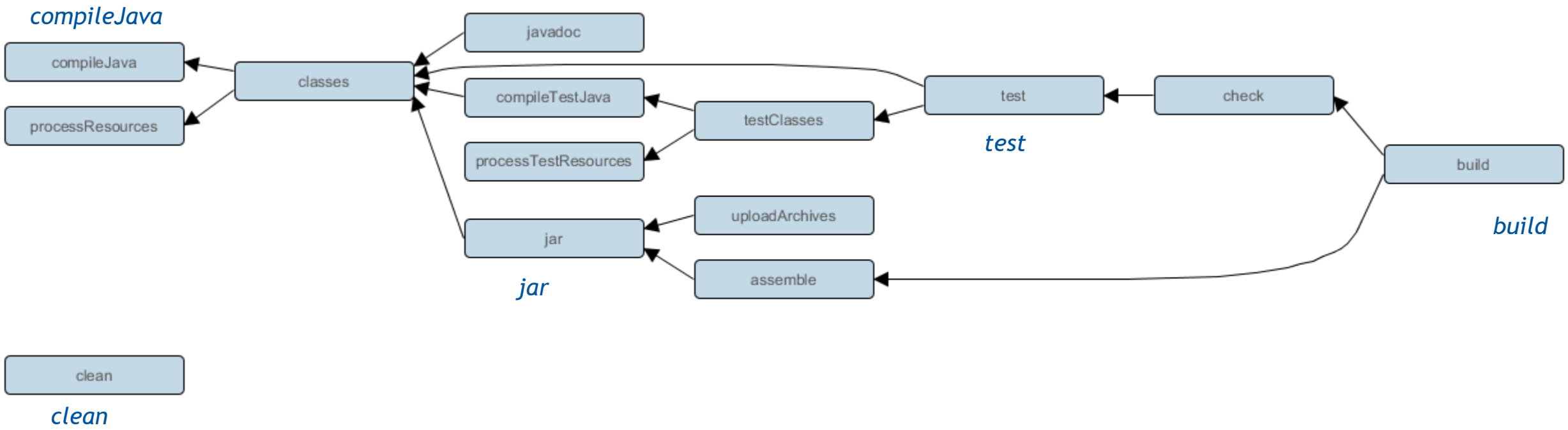
- Konfigurationsdatei: build.gradle (Auszug)





Gradle's Java Plugin Tasks

- Abhängigkeiten zwischen den Tasks mit denen Gradle Java behandelt:



Quelle: https://docs.gradle.org/current/userguide/java_plugin.html

Vergleich Gradle, Maven & Make

- Gradle & Maven: Abhängigkeiten werden zwischen Aktivitäten festgelegt
- Make: Abhängigkeiten werden zwischen Dateien festgelegt
- Alle besitzen viele Tasks (mkdir, cp, javac, etc.)
  - Maven: Library Java-programmierter „tasks“
  - Gradle: Library Groovy- und Java-programmierter „tasks“
  - Make: Shell Kommandos
- Alle erweiterbar
  - make: via shell
  - gradle: via plugins in Java

GNU make

Fazit:

- Dateiabhängigkeit (make) ist klüger, weil re-generate nur im Bedarfsfall aufgerufen wird (inkrementell)
  - Gradle kann Dateiabhängigkeiten, wenn klug definiert
- Maven kann zusätzlich projektübergreifende Pakete/Module
- Make dominiert eher bei C, C++ u.ä., Gradle/Maven im Java Umfeld (weil Java compiler selbst einen Teil der dependencies managed)
- Make (mit shell) ist für Java Programmierer ungewohnt
- Gradle/make ist flexibler; make schneller; maven untauglich

Continuous Integration (CI)

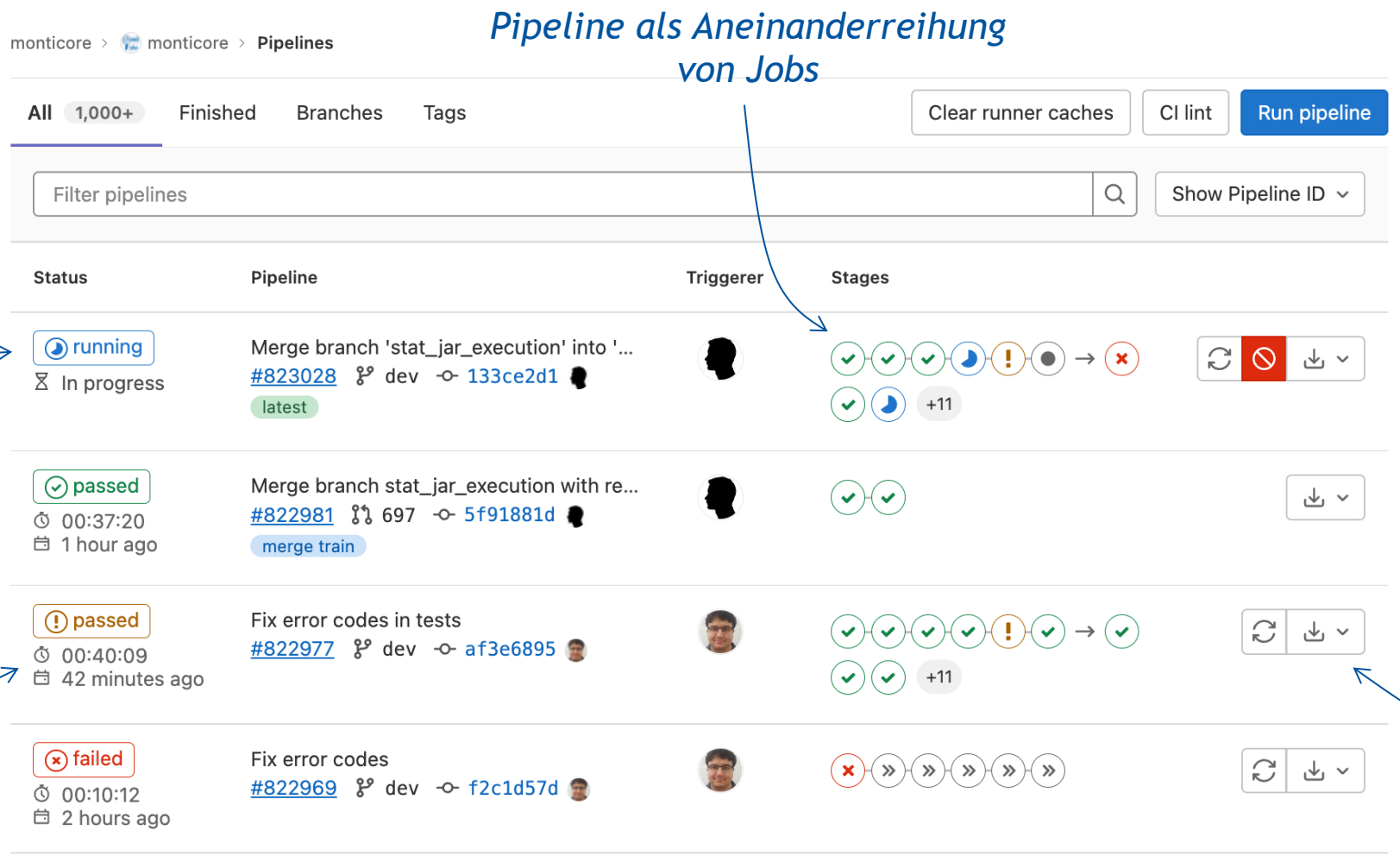
- Ziel: regelmäßige, automatisierte Integration von Softwareprojekten
  - Build-Stabilität
  - Modul-Integration
  - Automatisiertes Testen → kurze Test-Zyklen
  - Qualitäts-Analysen: Code-Smells, Commit-Historie
  - Reporting – Benachrichtigung im Fehlerfall
  - Automatisierte Verteilung auf Testsysteme
- Turnus
  - Nightly build
  - Build on commit
  - manuell
- Tools
  - Hudson, Jenkins, Gitlab CI pipelines
  - Apache Continuum etc.

CI/CD

Jenkins

continuum

Gitlab CI: Übersicht



Pipeline als Aneinanderreihung von Jobs

Status

Laufzeit

Buildprozess starten

Download von Artefakten

Zum Selbststudium: https://moodle.rwth-aachen.de/course/view.php?id=24730

Softwaretechnik

11. Werkzeuge
11.3. Qualitätsmanagement

Prof. Bernhard Rumpe
Software Engineering
RWTH Aachen

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

SE Software Engineering | RWTH AACHEN UNIVERSITY

Qualitätsmanagement

> „You can't control what you can't measure"
>
> *(DeMarco, 1982)*

- Softwarequalität kann nur verbessert werden, wenn sie gemessen wird
- Statische und dynamische Analyse führt zu mehreren Maßzahlen (Metriken)
  - Statische Analyse: Keine Codeausführung
  - Dynamische Analyse: Codeausführung (z.B. Testen)
- Codequalität in größeren Projekten mit mehreren Mitarbeitern ist nur *schwer überschaubar* OHNE Tool-Unterstützung

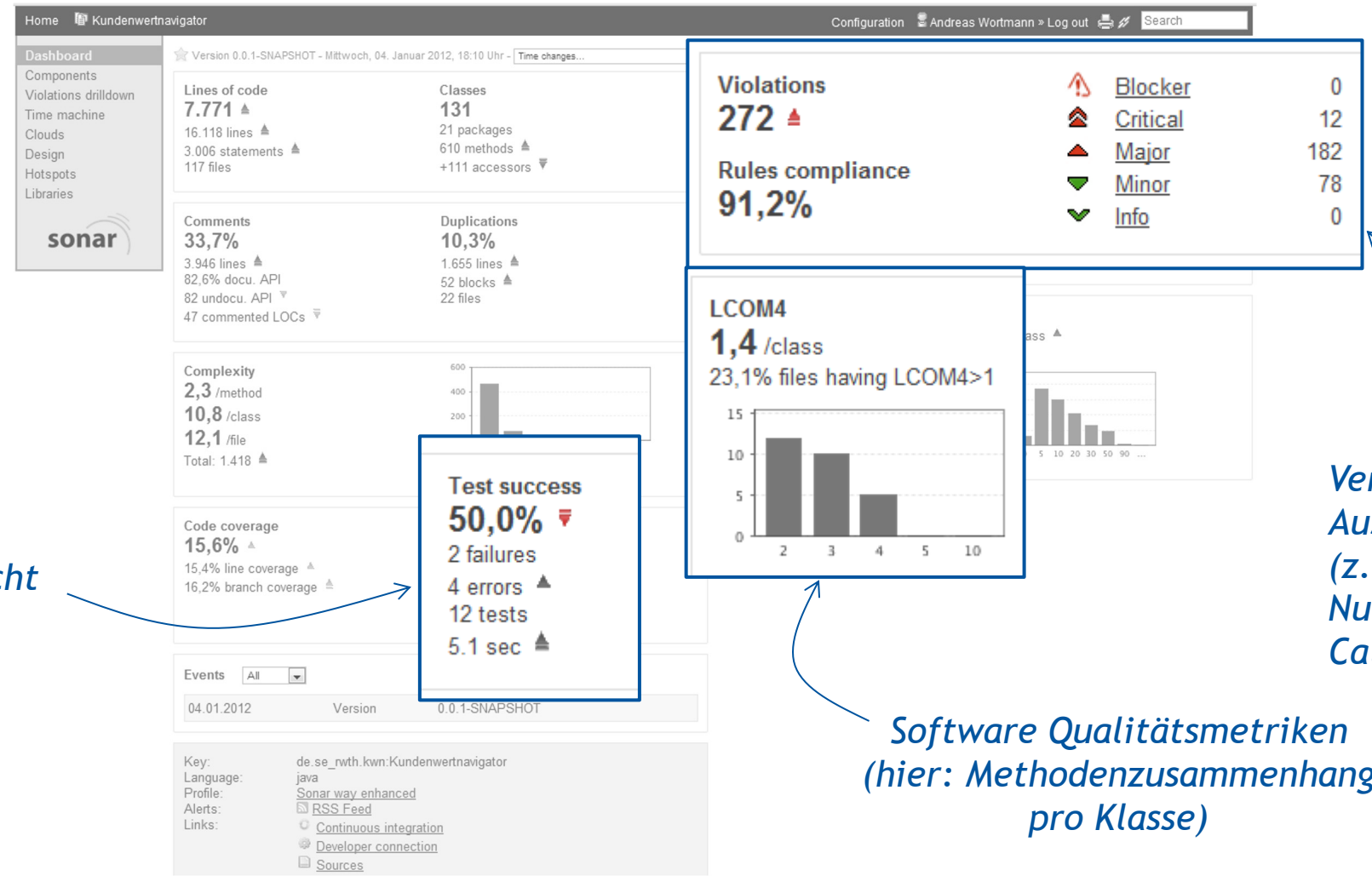
Sonar

- Webanwendung zur Anzeige von qualitätsrelevanten Projektkennzahlen
- Kennzahlen werden von Plugins erstellt und in Sonar dargestellt

- Beispielsweise Software-Metriken
  - Allgemeine Metriken
    - Lines of Code (LOC)
  - Objektorientierte Metriken
    - Zusammengehörigkeit von Methoden einer Klasse (LCOM)
    - Tiefe im Vererbungshierarchie (DIT)

- Weitere Kennzahlen
  - Testabdeckung, Testerfolg
  - Verletzungen von Code Conventions
  - Abhängigkeiten, Duplikate

Sonar: Dashboard



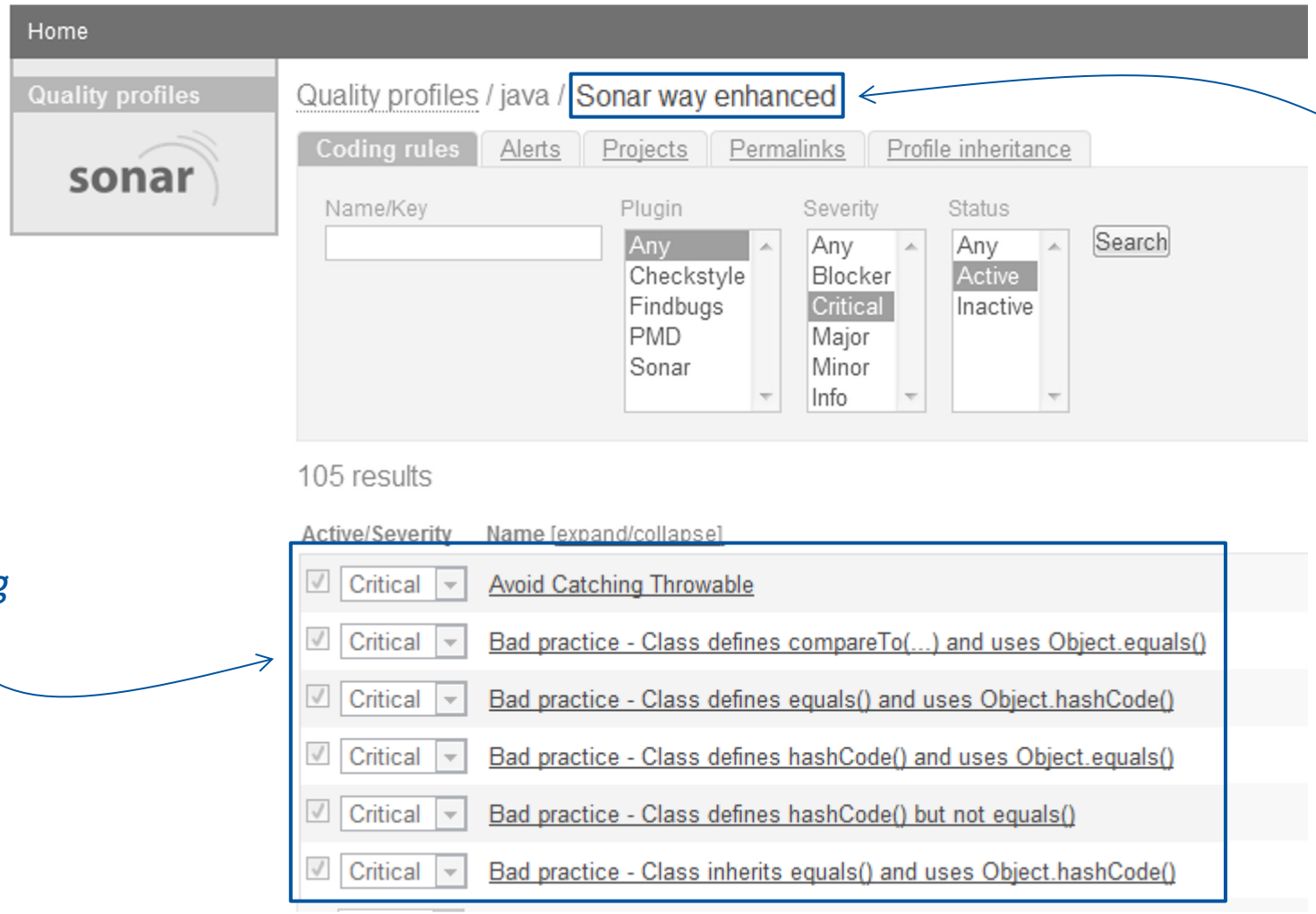
sonarqube

Testübersicht

Verstöße gegen das Auswertungsprofil (z.B. mögliche Nullpointer, leere Catch-Blöcke, …)

Software Qualitätsmetriken (hier: Methodenzusammenhang pro Klasse)

Sonar: Qualitätsprofile

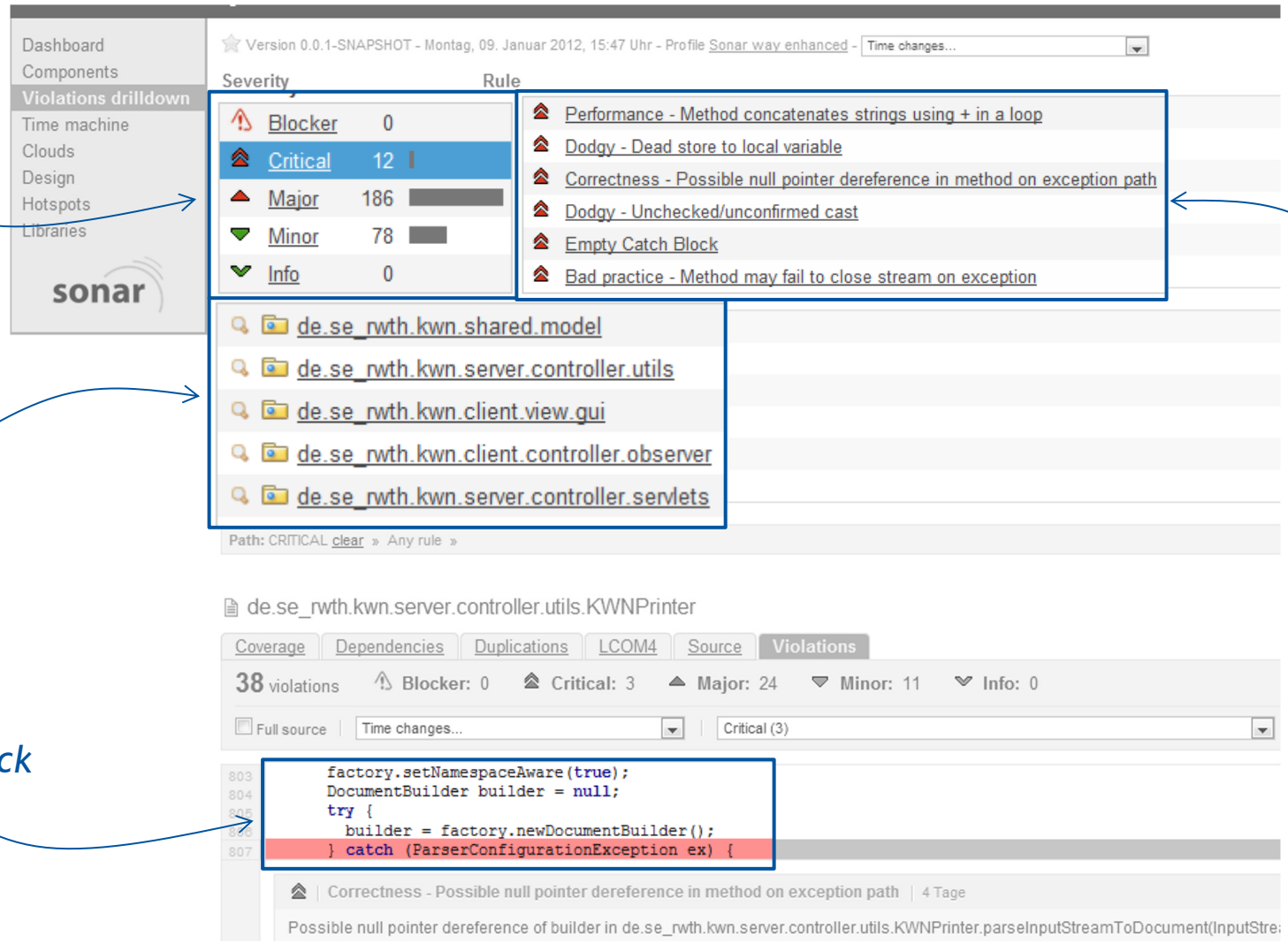


Gewähltes Auswertungsprofil

Code Violations des Profils und deren Bewertung

Sonar: Qualitätsverstöße

Problemarten in diesem Projekt

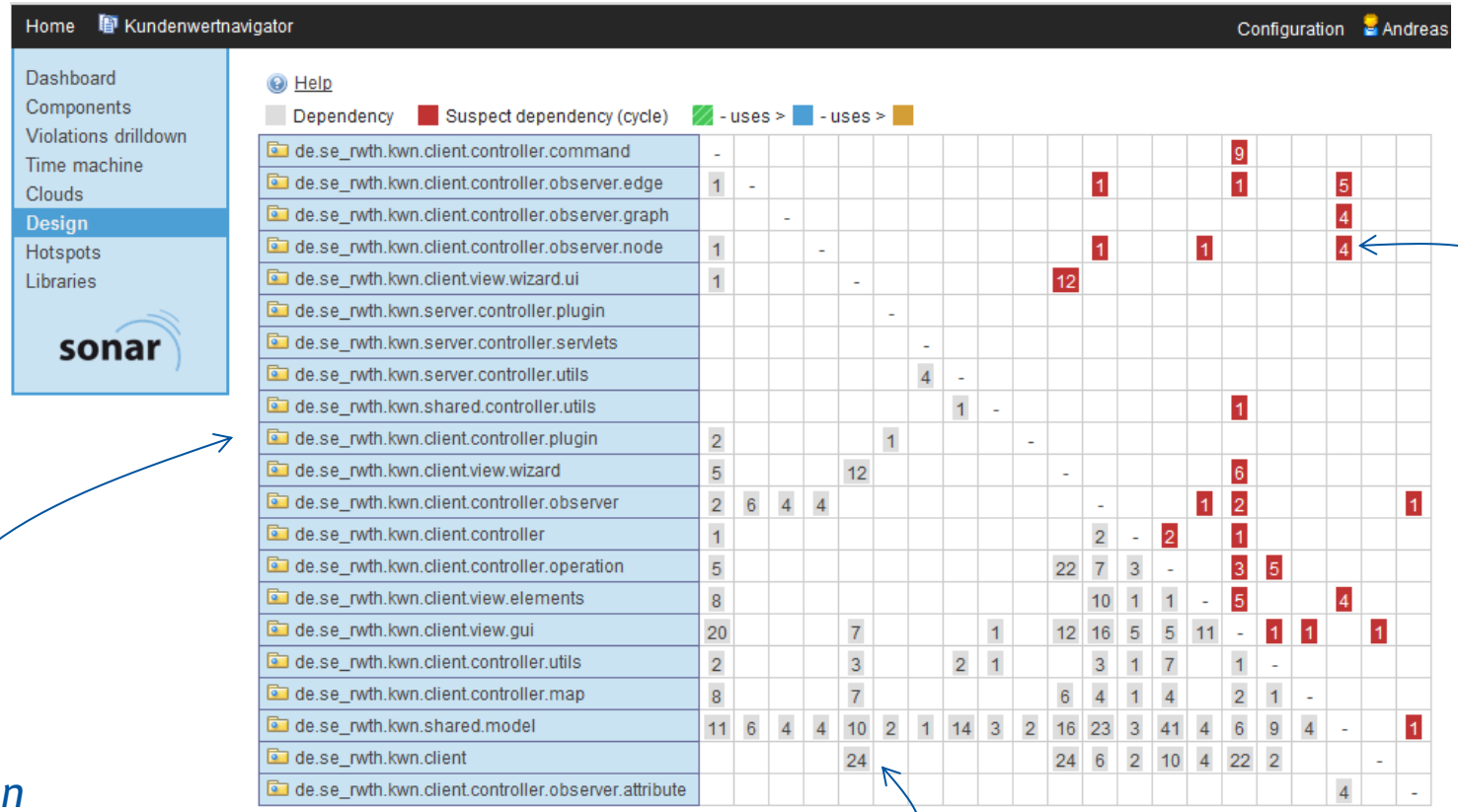


Betroffene Dateien

Problematischer Quellcode: hier leerer Catch-Block

Regeln gegen die verstoßen wurde

Sonar: Paket Abhängigkeiten



sonarqube

Zyklen

Übersicht der Abhängigkeiten

Normale Abhängigkeiten („kennt“)

Softwaretechnik

11. Werkzeuge
11.4. Projektmanagement Plattformen

Prof. Bernhard Rumpe
Software Engineering
RWTH Aachen

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

SE Software Engineering | RWTH AACHEN UNIVERSITY

GitLab: the DevOps platform

- Webbasiertes Projektmanagement-Werkzeug
- Integriert viele Dienste
  - Git als Versionskontrolle/Repository
  - Wiki
  - Bugreporting
  - Repository-Browser
  - Timeline / Roadmap / Meilensteinplanung
  - Automatisierung der Toolchain (Continuous Integration, CI pipelines)
    - Generieren, compilieren, testen, prüfen, …
- Eigene Dienste als Plugins möglich
- Varianten
  - In der cloud gehostet: github.com
  - Als freie Software verfügbar (community edition): gitlab.com
  - Gitlab/github dominieren heute im open source Bereich über frühere Plattformen (z.B. svn im sselab, etc.)

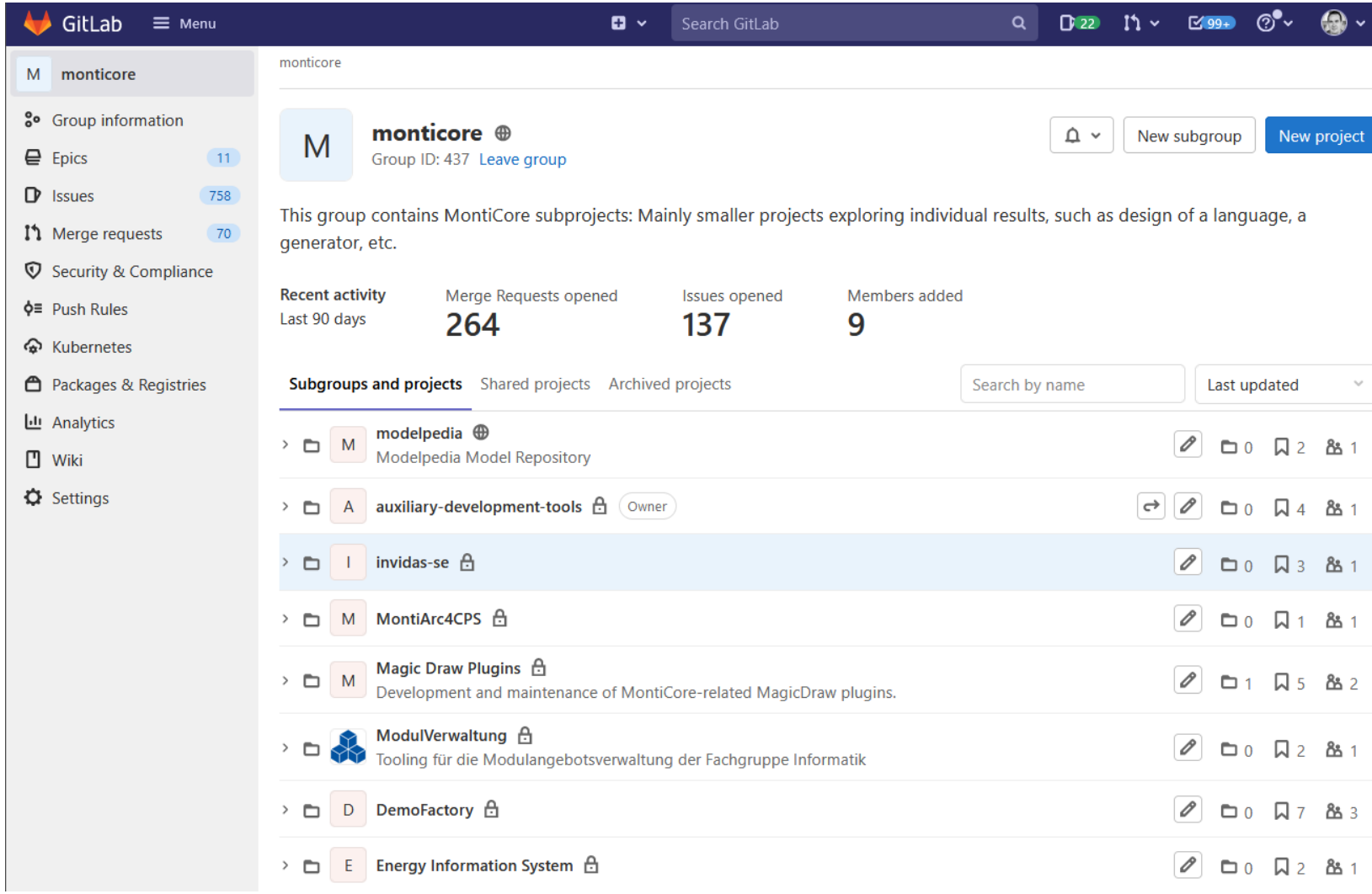
GitLab: Verwalten von Projekten und Entwicklerteams

- git Repository verwaltet
  - Versionierung
  - Verwaltung Merge Requests
  - Branches

- Kern ist das Git-Projekt
- git-Projekt hat
  - Branches (unabhängige Entwicklungslinien, die gemergt werden können)
  - Versioniert seine Inhalte
    - + Datum, wer?, commit-messages

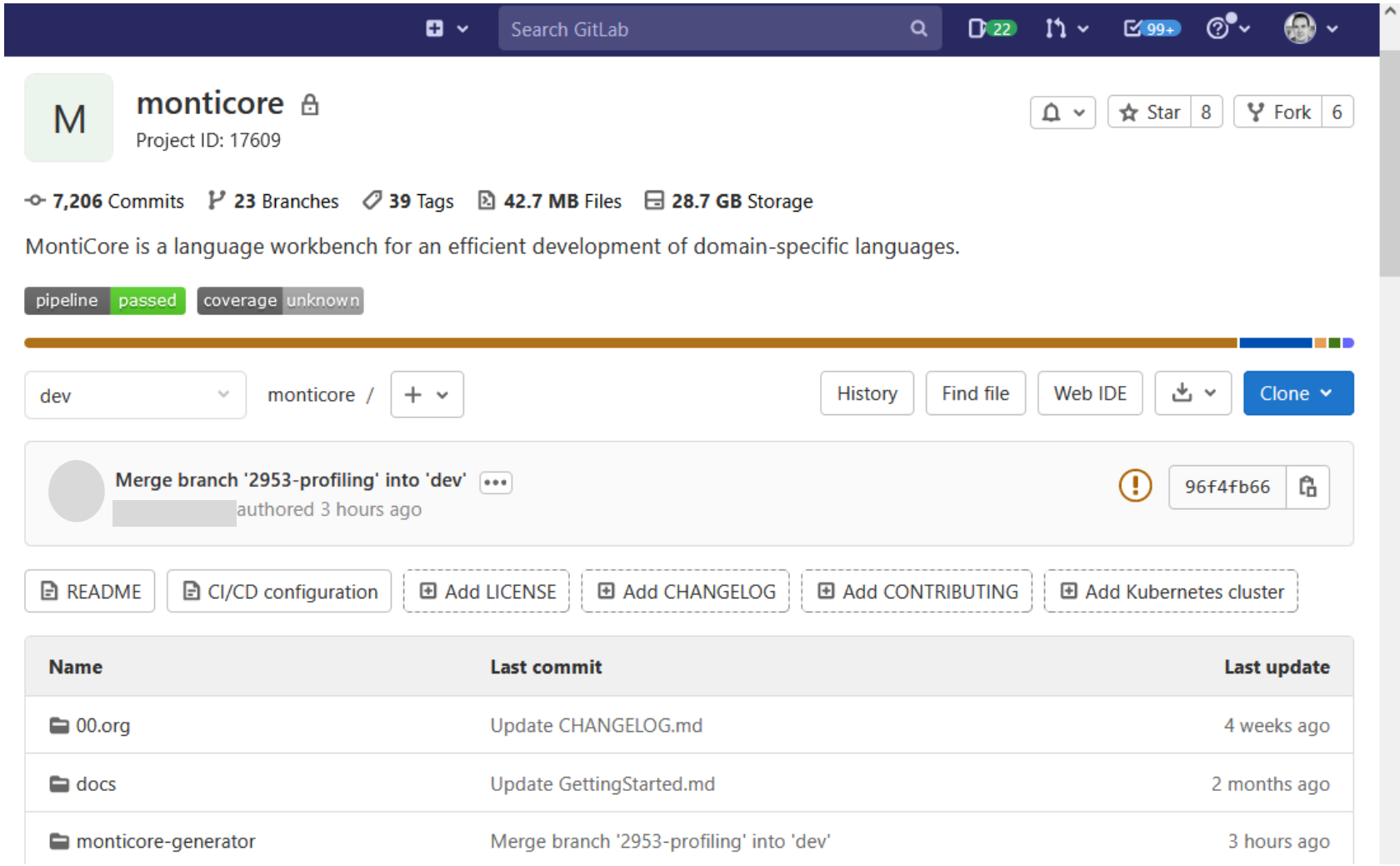
- Projekte sind gruppierbar

- Ansicht hier: MontiCore gruppe



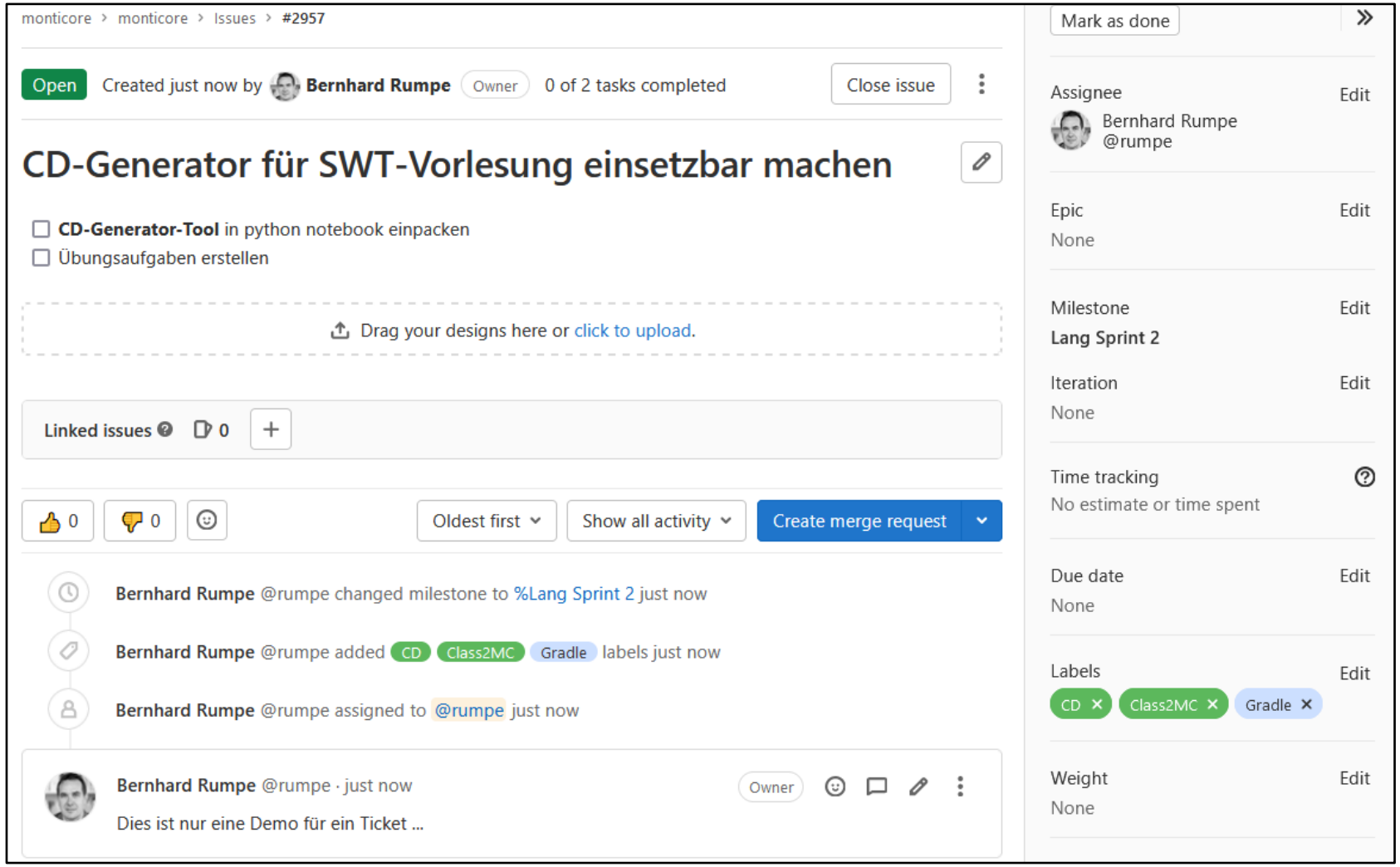
GitLab: Verwalten von Projekten

- Ansicht hier: MontiCore Projekt
- Infos über Status, #branches,
- CI pipeline passed
- Aktuell betrachteter Branch
- Letzte Aktion
- Etc.

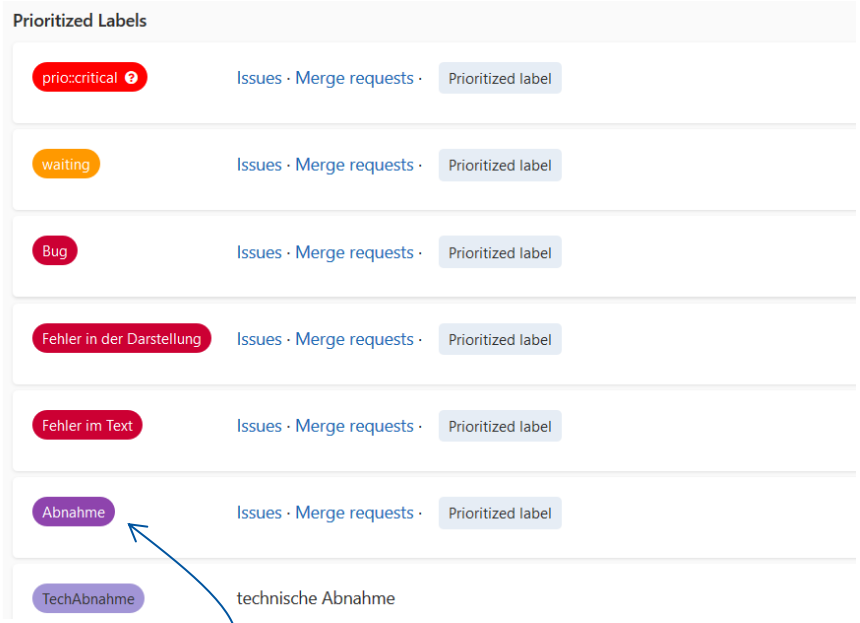


GitLab: Issues (Ticket Management)

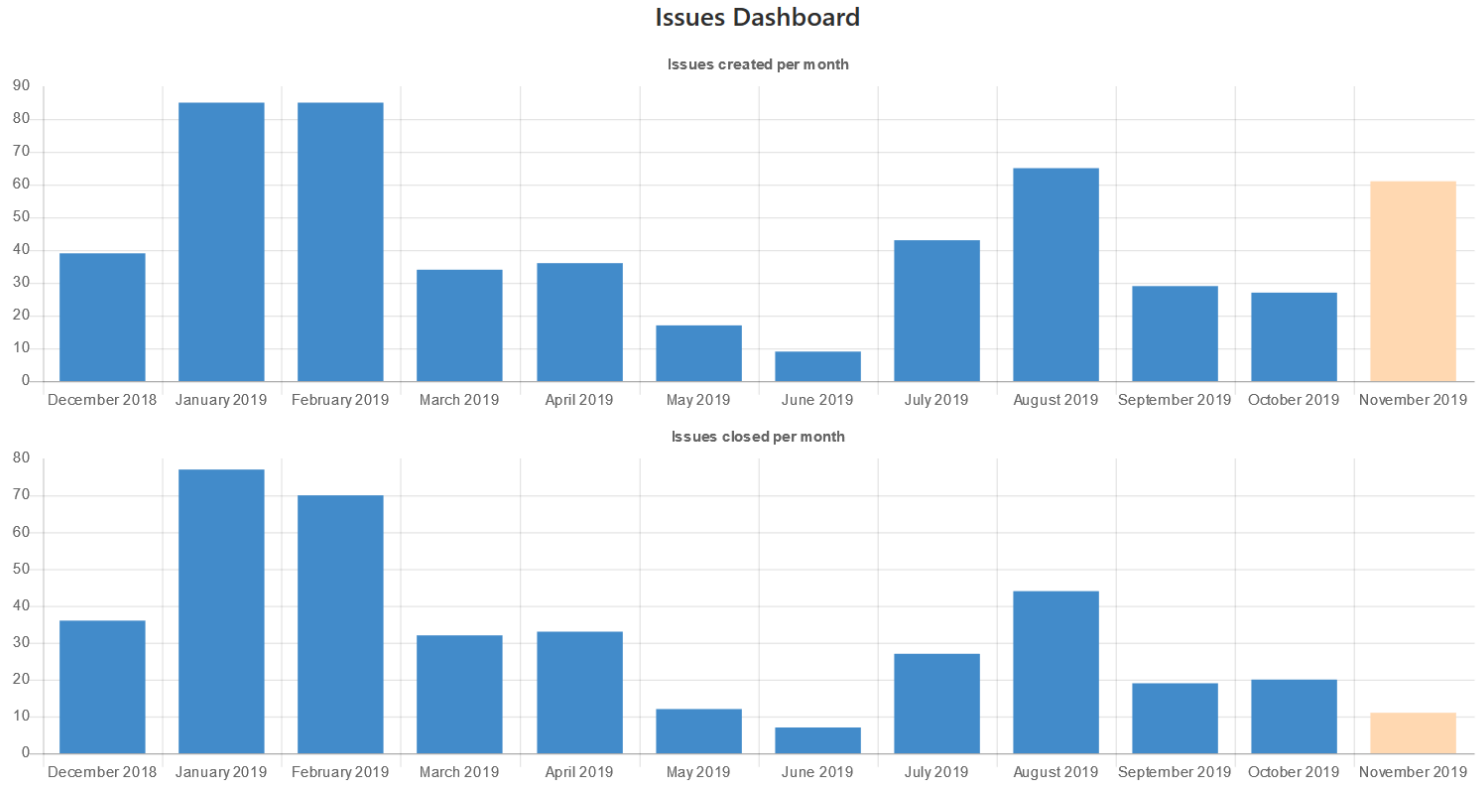
- Ansicht hier: ein Issue
- Titel + Erklärung
- Frei wählbare Eigenschaften:
  - Milestone
  - Time: due date
  - Labels, zB CD, Class2MS
  - Zugewiesen an …
- Kollaborative Einsicht & Editiermöglichkeiten (kein Ownership)
- Historie des Tickets
- Möglichkeit mit dem Code zu verlinken



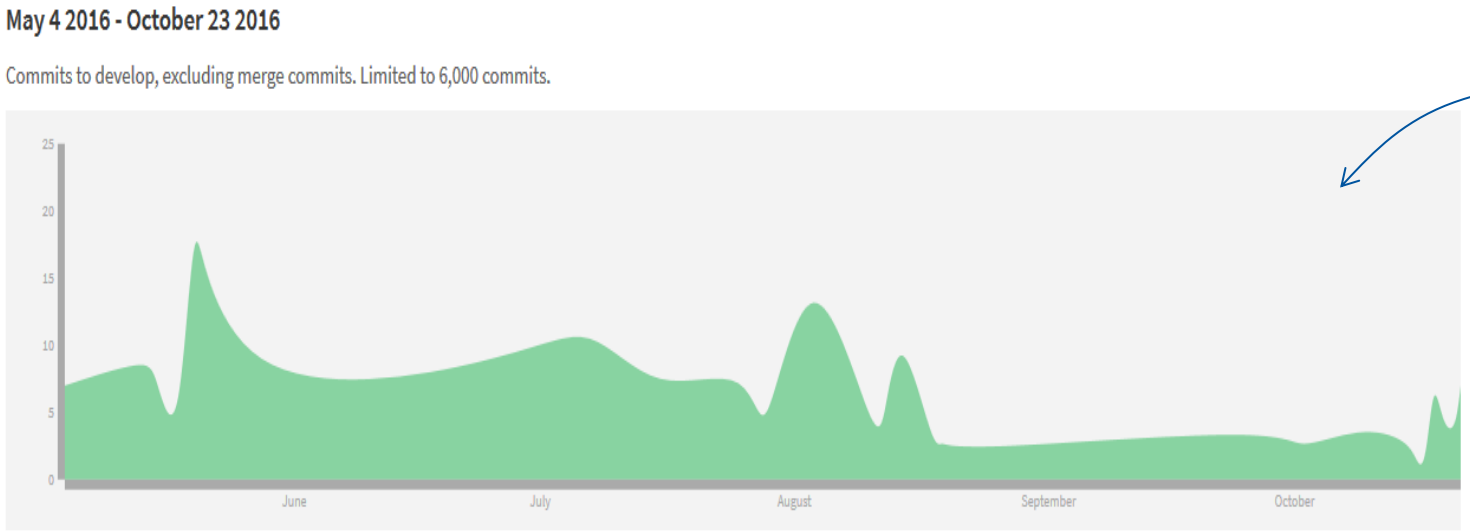
GitLab: Labels für Issues & Übersichten



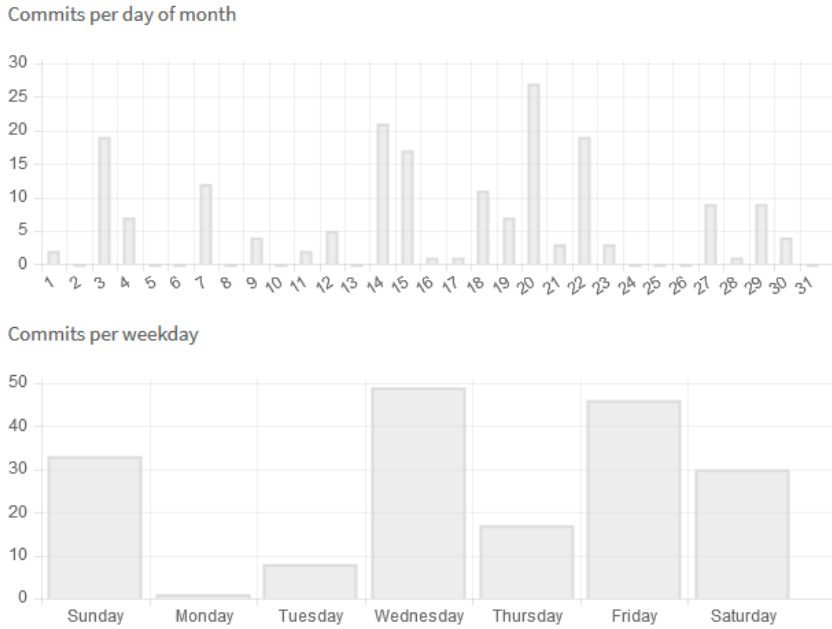
Labels selbst anlegen und priorisieren



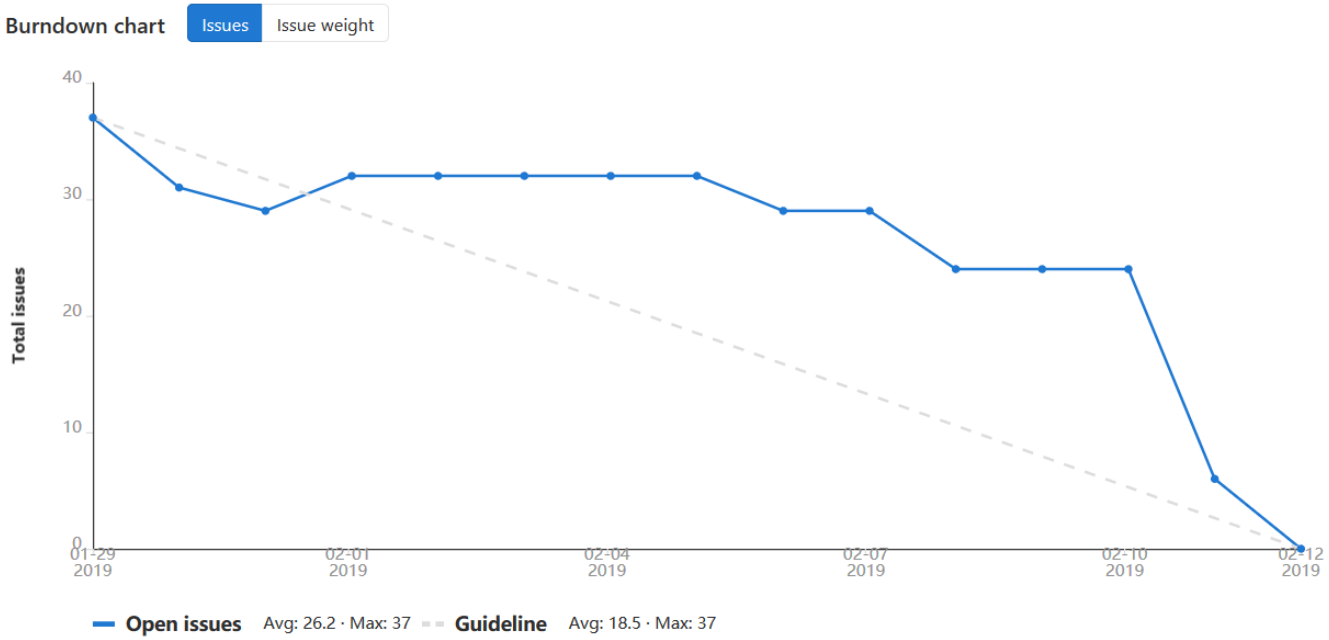
GitLab: Activity Reports



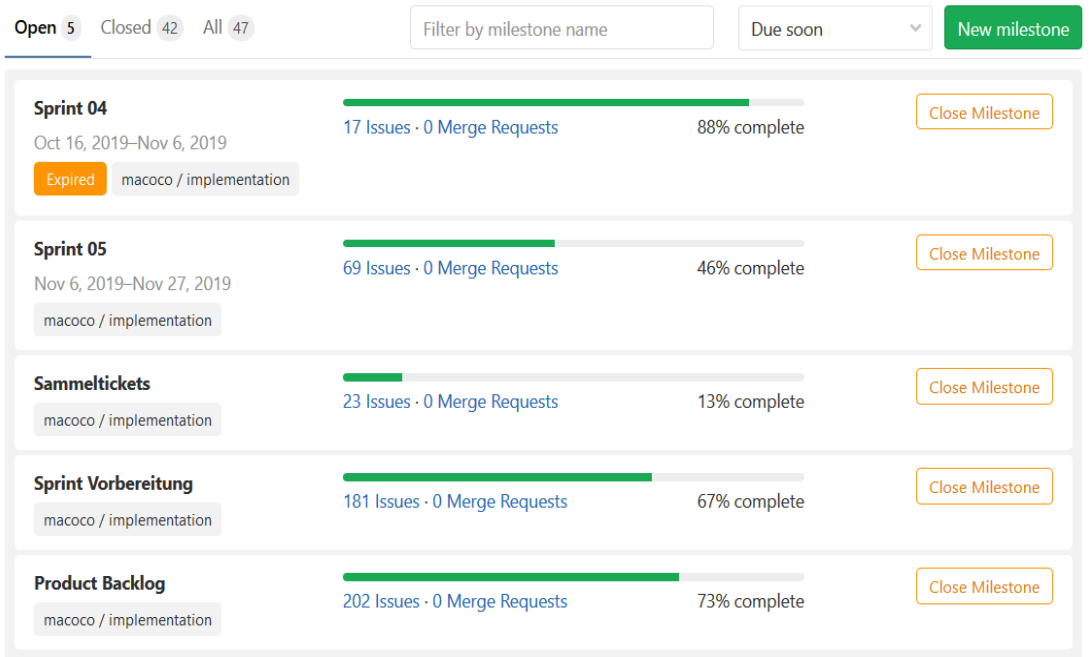
commit history
(auch per Developer)



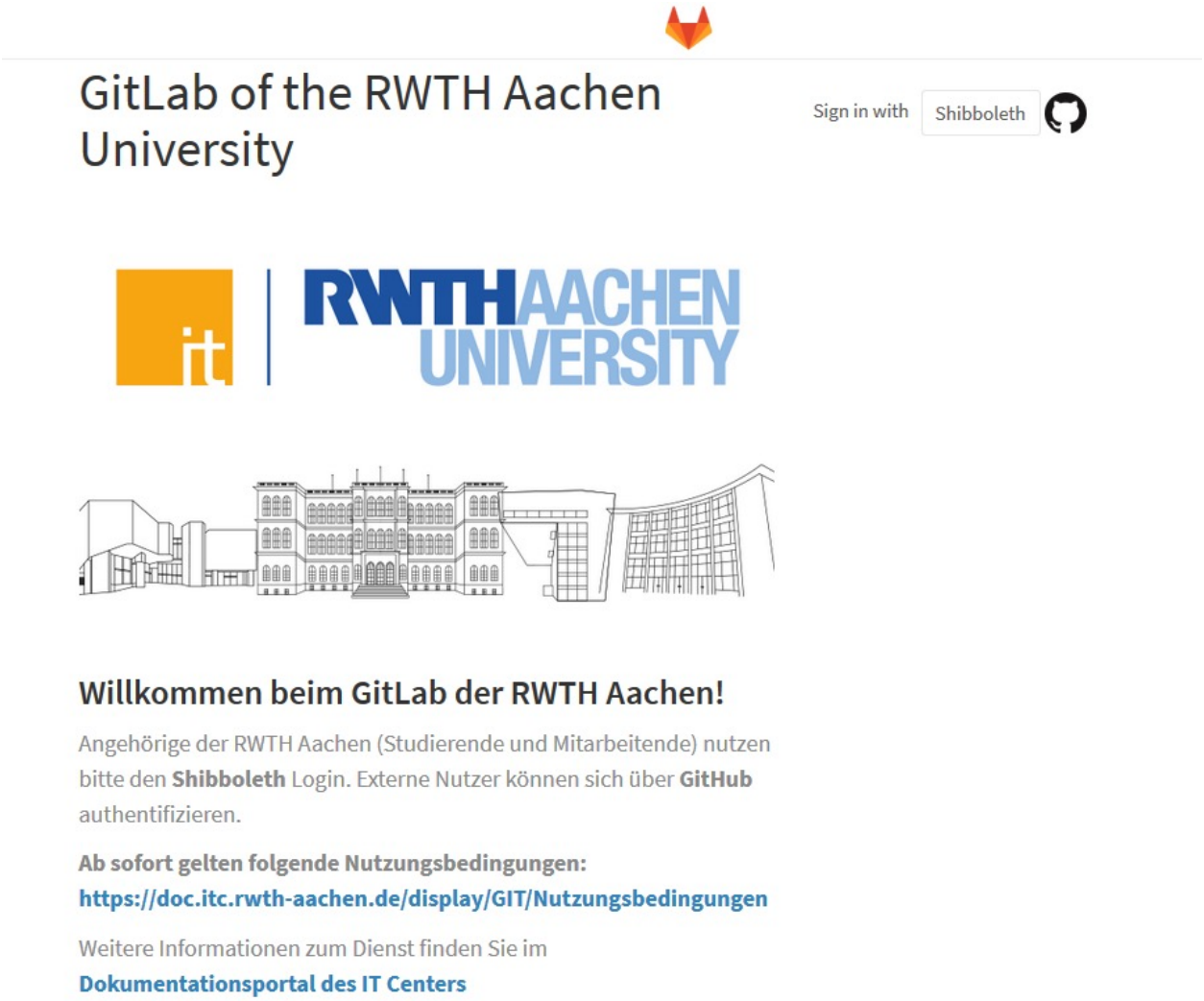
GitLab: Milestones



Wieviele Tickets sind bis zu dem Meilenstein zu erledigen bzw. abgeschlossen



GitLab @ RWTH



GitLab

https://git.rwth-aachen.de/

Kostenlos !

Login per TIM-Kennung
oder gitHub-Account

Git ist geeignet für alle Arten von Projekten inclusive Bachelor/Master/Doktorarbeit (java, python, C++, latex, etc.)

Was haben wir gelernt?

Werkzeuge

…vereinfachen die Verwaltung

…verbessern die Kommunikation

…steigern die Produktivität und Qualität

ABER:

Werkzeugauswahl schwierig

Anschaffung und Schulung teuer

Widerstände und Akzeptanz

Werkzeugintegration und –konfiguration nicht trivial

Versionierung

Parallele Bearbeitung von Software durch mehrere Personen

Protokollierung durchgeführter Änderungen

Rücknahme von Änderungen möglich

Datensicherung des Quelltextes

SVN – Subversion (zentral): Verschmelzen und Konfliktbehebung

Git (dezentral): Git Workflow

Automatisierung

Build Automatisierung

- Ant, Gradle, Maven & Make

Continous Integration

- Build-Stabilität
- Modul-Integration
- Autom. Testen
- Qualitäts-Analysen
- Reporting
- Automatisierte Verteilung auf Testsysteme

Qualitätsmanagement

- Metriken, Testabdeckung, Code Conventions

Projekt-management

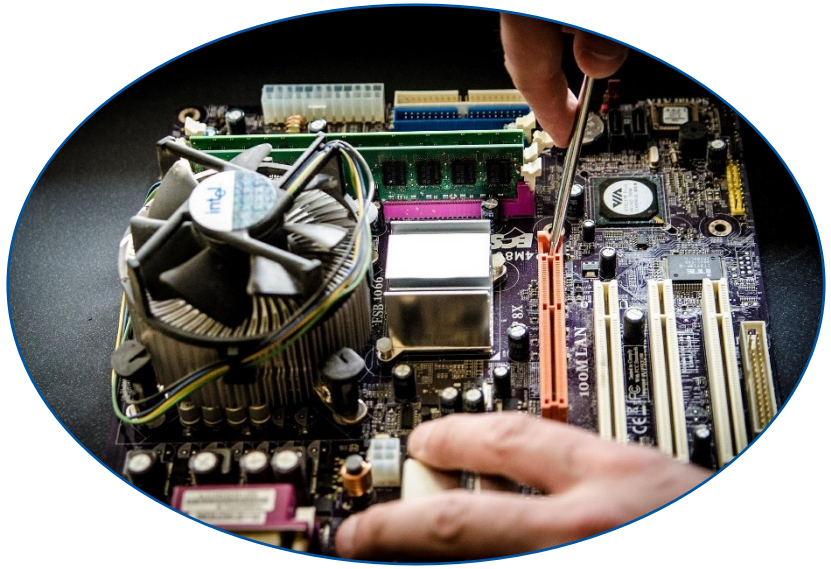
Integrierte Plattformen

Verbinden unterschiedliche Funktionalitäten

- z.B. Wiki, Bug-Reporting, Repository-Browser, Versionierung, Branches, Zeitverläufe, Meilensteinplanung, CI

Vorlesung Softwaretechnik

12. Komponenten



Prof. Bernhard Rumpe
Software Engineering
RWTH Aachen

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

Warum, was, wie und wozu? Komponenten

Warum?

- Wiederverwendbare Einheiten
- Flexibilität

Was?

- Spezifikation von Komponenten
- Verbindung von Komponenten

Wie?

- Formalisierung der Schnittstellen
- Architekturmuster für die Verteilung
- Architekturmuster für die Kommunikation
- Komponenten in Anwendungen

Wozu?

- Praxis: Viele Entwicklungen Komponenten-basiert bzw. nutzen Frameworks
- Entwicklungszeit reduzieren

Softwaretechnik

12. Komponenten
12.1. Motivation & Basis Konzepte

Prof. Bernhard Rumpe
Software Engineering
RWTH Aachen

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

SE Software Engineering | RWTH AACHEN UNIVERSITY

Probleme bei der Anpassung von (Standard-)Software

Beispiel: Australische Post und SAP R/3

- Geschäftssituation:
  Die lokalen Geschäftsführer haben eine Übersicht über alle Transaktionen.
  Die Gesamtgeschäftsleitung bekommt nur Berichte von den lokalen Geschäftsführern, hat aber keine Übersicht über einzelnen Transaktionen

- SAP R/3 bietet standardmäßig nur eine hierarchische Rechtevergabe, so dass die Geschäftsführung theoretisch jede Transaktion nachvollziehen kann.

- Die Software widerspricht somit der Firmenkultur, die Autonomie der Teilbereiche wird untergraben. Politische und soziale Widerstände sind zu bewältigen, da die Anpassung der Software in diesem Punkt *sehr kostspielig* wäre.

Was ist eine Komponente?

- „Components are for composition“

  [Szy02, S. 3]

- „A modular part of a system that encapsulates its contents and whose manifestation is replaceable within its environment. A component defines its behavior in terms of provided and required interfaces. As such, a component serves as a type, whose conformance is defined by these provided and required interfaces (encompassing both their static as well dynamic semantics).“

  [UML 2.0 Superstructure Specification, 03-08-02, S. 7]

- „A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties“

  [Szy02, S. 41]

Komponenten (unsere Definition)

Eine Komponente ist eine Einheit

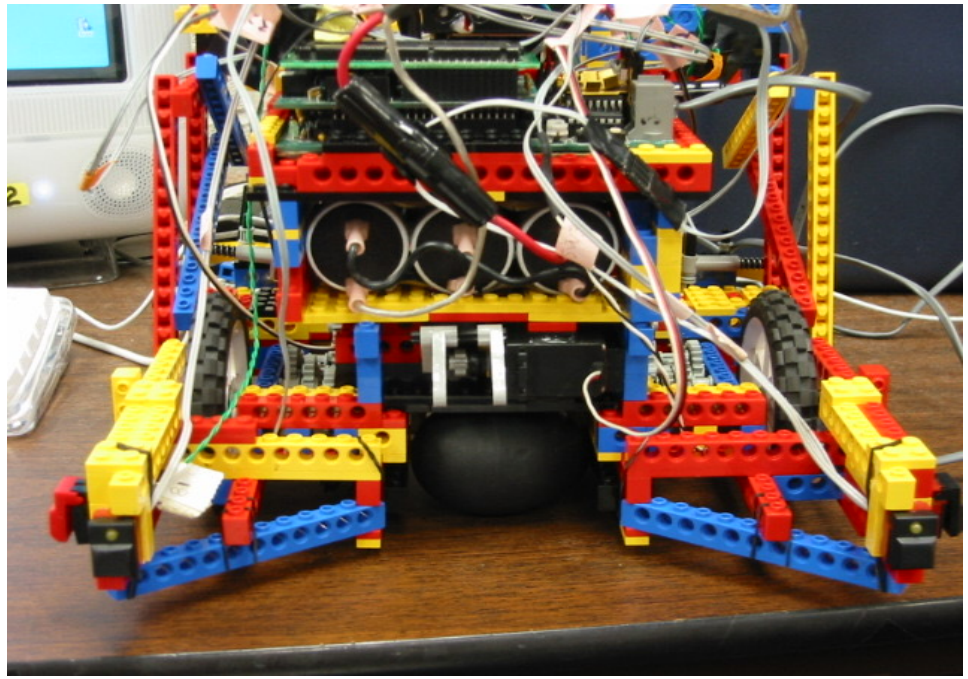
- die unabhängig auslieferbar und versioniert ist,
- die explizite Schnittstellen hat,
- von der Umgebung weitgehend unabhängig ist,
- mit anderen Komponenten kombinierbar ist,
- mehrfach verwendbar ist,
- und keinen nach außen beobachtbaren Zustand hat.

Glue Code

Die Komposition erfolgt durch so genannten „glue code“

- Verbindungen zwischen den Komponenten
- Manchmal Realisierung durch
  - Skriptsprache, Deployment Descriptor oder Wizard
- Komponenten sind typischerweise stabil, die Rekonfiguration der Verdrahtung kann leicht gewechselt werden

Besser nicht so



Spezifikation von Komponenten

- Menge von Schnittstellen
  - Erforderliche Schnittstellen
  - Angebotene Schnittstellen an die Umgebung

- Entwurfsmuster „Extension Interface“ wird zur Entkopplung der konkreten Schnittstelle von der Implementierung genutzt.

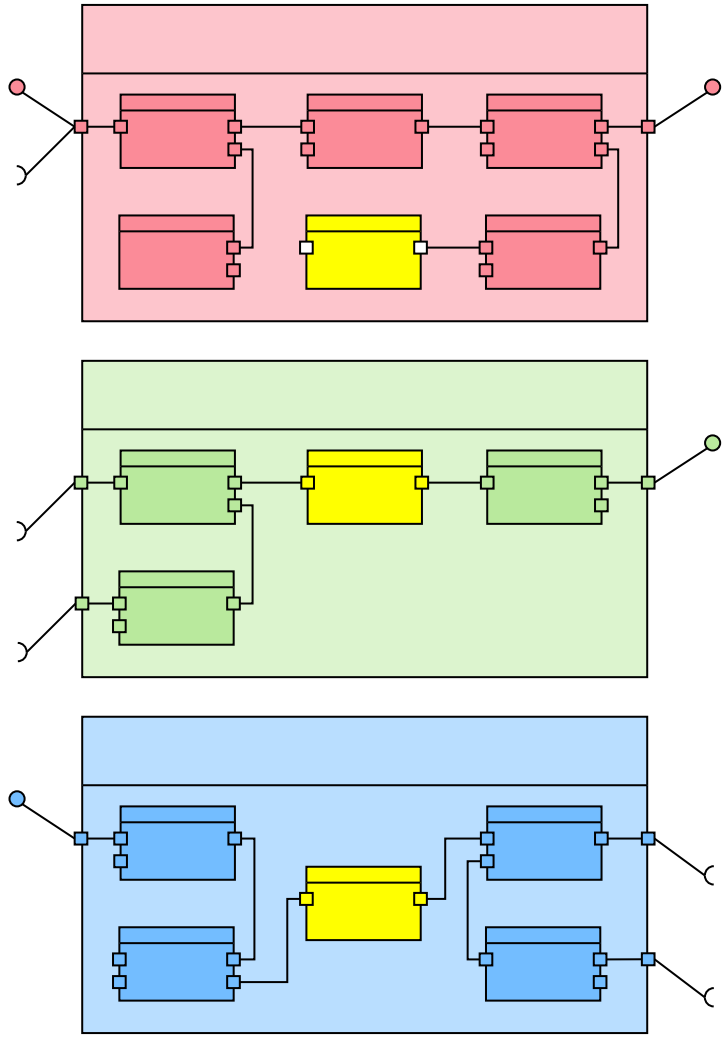
- Spezifikation kann nicht-funktionale Eigenschaften enthalten
  - Zeitverhalten
  - Protokolle
  - Speicherbedarf
  - Pre- und Postconditions von Operationen

- Spezifikation und Implementierung sind getrennte Artefakte

Wiederverwendung von Komponenten (1/2)

Wiederverwendung einzelner Komponenten in verschiedenen Systemen

- Produktivitätssteigerung
- „best in class“ Lösungen
- Qualitätssteigerung
- Kostenreduktion
- Risikominimierung
- Vereinfachte Wartung
- Geringerer Schulungsaufwand



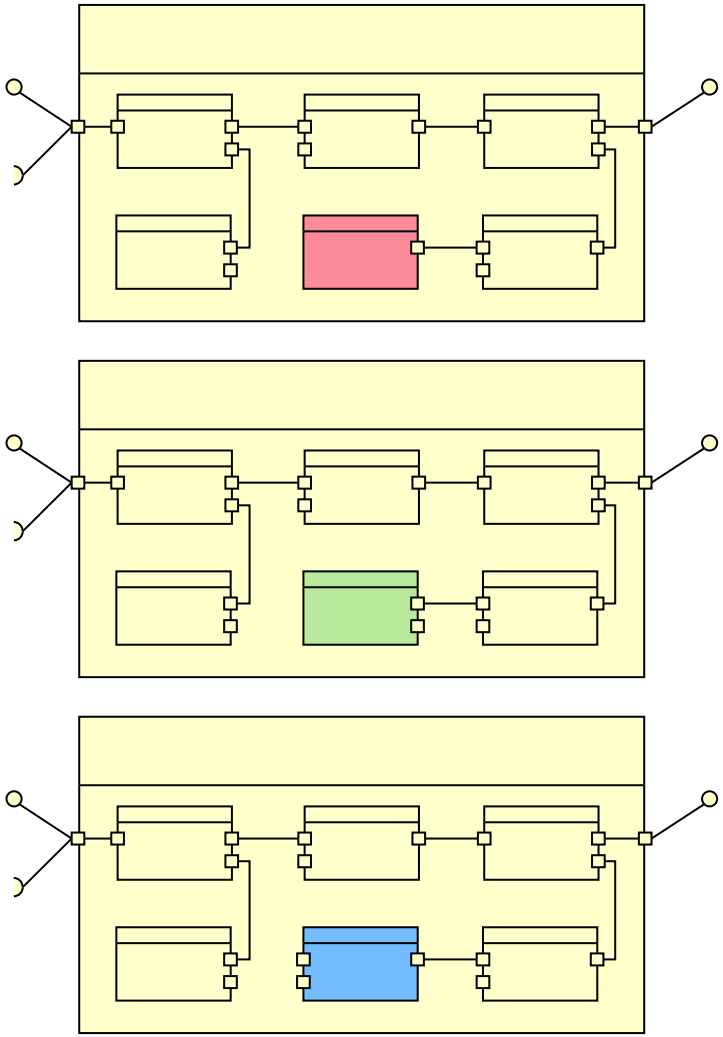
Wiederverwendung von Komponenten (2/2)

Oft viel wichtiger als die Wiederverwendung von Komponenten, ist die

Wiederverwendung ganzer Systeme
(Systemlandschaften)

und deren Anpassung durch Ersetzen einzelner Komponenten:

- Verschiedene Implementierung derselben Funktion
- Erweiterte Funktionalität
- Neue Version der Komponente



Präzise Definition von Komponentenspezifikationen

- Formalisierung durch Assumption/Guarantee
  - Assumption:
    Annahmen, die bei Aufruf durch die Umgebung erfüllt sein müssen oder allgemein gültig sind
  - Guarantee:
    Sofern die Annahmen erfüllt sind, gelten diese Aussagen über die von der Komponente gelieferten Ergebnisse

- Einfache Form:
  - Assumption = Aufrufsituation: Precondition
  - Guarantee = Ergebnissituation: Postcondition

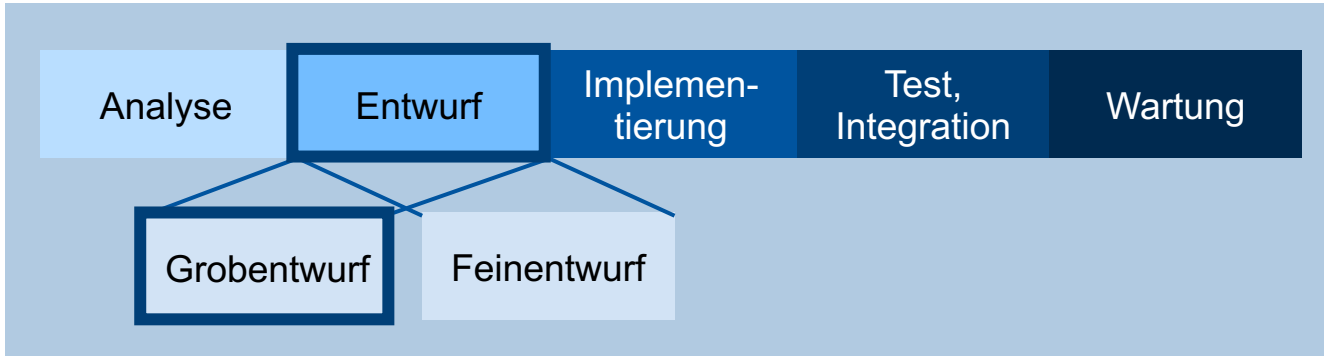
- Aufruffolgen und zeitliches Verhalten spielen oft ebenfalls eine Rolle

- Beispiel:
$$A_{Tempomat}(x) \equiv \forall t : x[user\_speed](t) \in \{40, ..., 220\}$$
$$G_{Tempomat}(x, y) \equiv \forall t : y[ACC\_speed\_opt](t + 1) = x[user\_speed](t)$$

Quelle: https://www.cqse.eu/publications/2008-umfassendes-architekturmodell-fr-das-engineering-eingebetteter-software-intensiver-systeme.pdf

Softwaretechnik

12. Komponenten
12.2. Architekturmuster für die Verteilung



Prof. Bernhard Rumpe
Software Engineering
RWTH Aachen

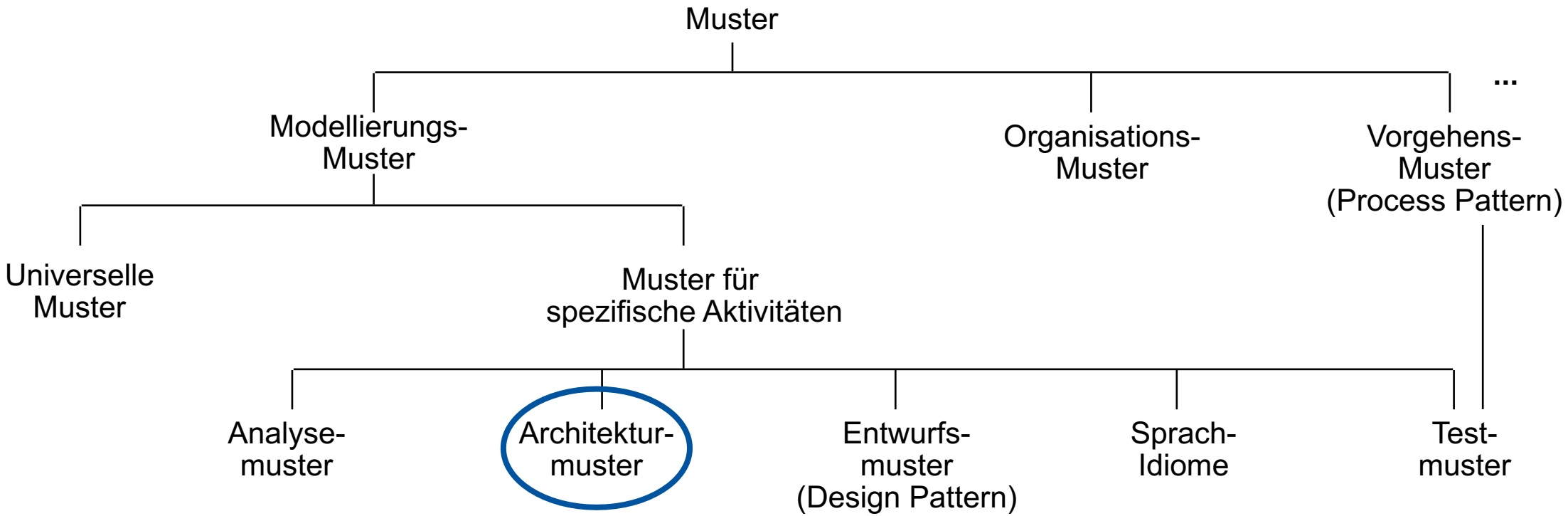
http://www.se-rwth.de/
@SE_RWTH

Literatur:

- Gamma/Helm/Johnson/Vlissides: Design Patterns, Addison-Wesley 1994 *(= „Gang of Four“, „GoF“)*
- Buschmann/Meunier/Rohnert/Sommerlad/Stal: A System of Patterns, Wiley 1996

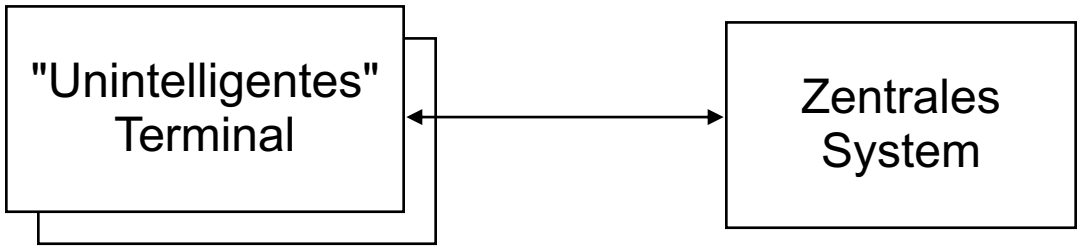
SE Software Engineering | RWTH AACHEN UNIVERSITY

Klassifikation von Mustern



- Weitere Unterteilung:
  - Allgemein anwendbare Muster (z.B. Komposition)
  - Domänenspezifische Muster (z.B. Kontostruktur im Finanzbereich)

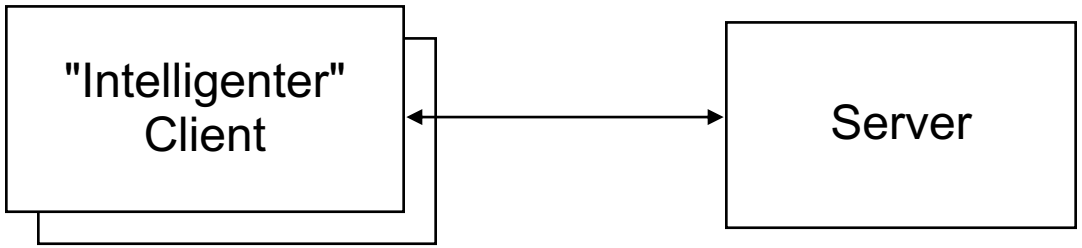
Verteilungsmuster "Zentrales System"



Beispiele:

- Klassische Großrechner-("Mainframe"-)Anwendungen
- Noch einfachere Variante:
  Lokale PC-Anwendungen (identifizieren Zentrale und Terminal)

Verteilungsmuster "Client/Server"



- Sogenannte "Two-Tier" Client/Server-Architektur
- Andere Namen:
  - "Front-end" für "Client", "Back-end" für "Server"

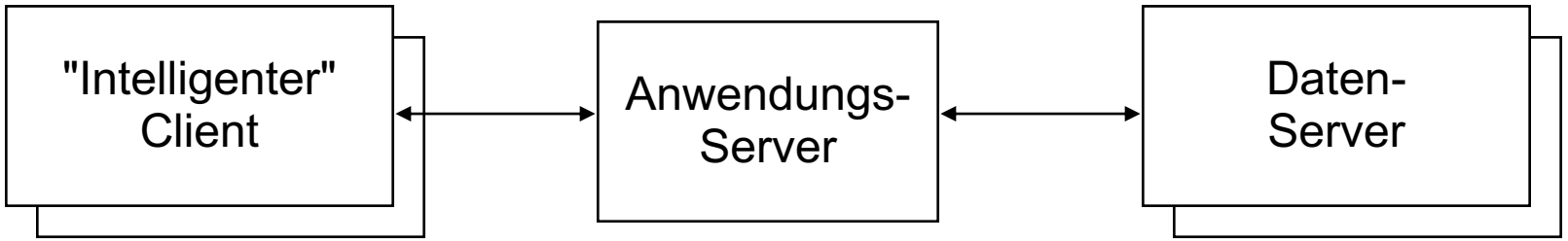
- Client:
  - Benutzungsschnittstelle
  - Einbindung in Geschäftsprozesse
  - Entkoppelt von Netztechnologie und Datenhaltung
- Server:
  - Datenhaltung, evtl. Fachlogik

"Thin-Client" und "Fat-Client"

- Thin-Client:
  - Nur die Benutzungsschnittstelle auf dem Client-System
  - Ähnlich zu zentralem System, aber oft Download-Mechanismen
  - Anwendungen:
    - "Screen-Scraping"
      (Umsetzung traditioneller Benutzungsschnittstellen in moderne Technologie)

- Fat-Client:
  - Teile der Fachlogik (oder gesamte Fachlogik) auf dem Client-System
  - Hauptfunktion des Servers: Datenhaltung
  - Entlastung des Servers
  - Zusätzliche Anforderungen an Clients (z.B. Installation von Software)

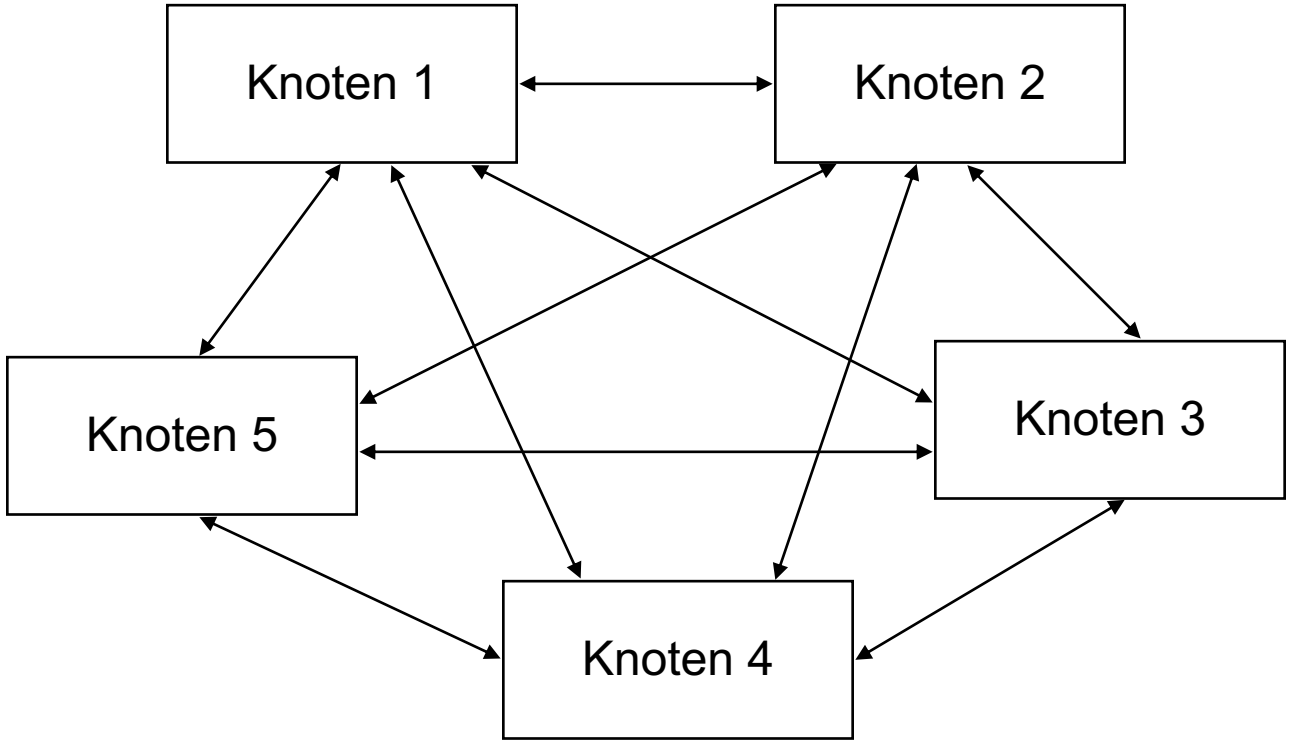
Verteilungsmuster "Three-Tier Client/Server"



- Client:
  - Benutzungsschnittstelle
  - evtl. Fachlogik
- Anwendungsserver:
  - evtl. Fachlogik
  - Verteilung von Anfragen auf verschiedene Server
- Server:
  - Datenhaltung, (Rechenleistung) etc.
- Kommunikation unter Servern meist breitbandig

- Heute üblich!

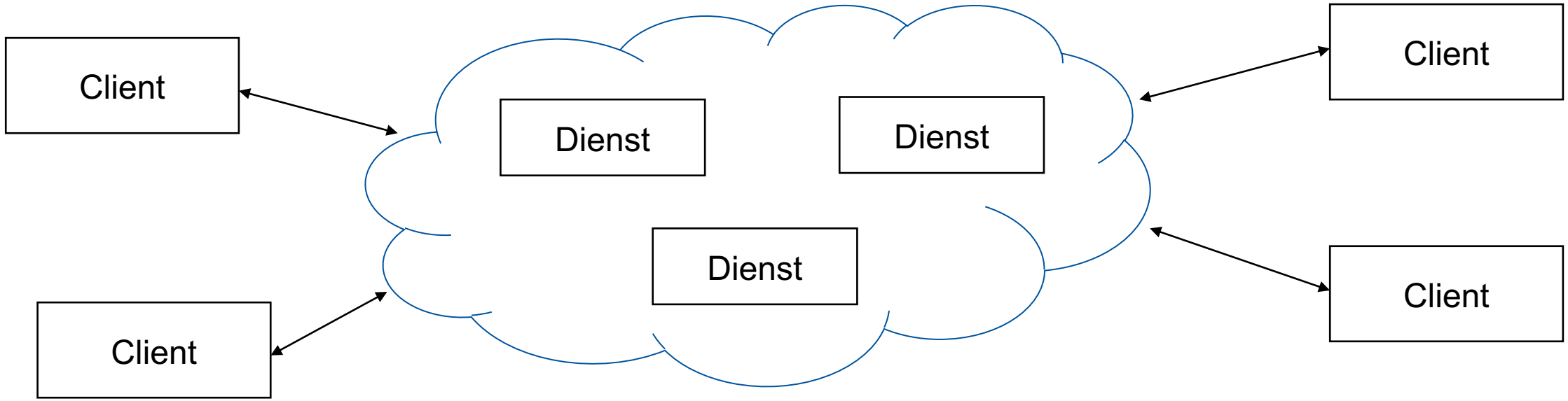
Verteilungsmuster "Föderation"



- Gleichberechtigte Partner (*peer-to-peer*)
- Unabhängigkeit von der Lokation und Plattform von Funktionen
- Verteilte kommunizierende Objekte

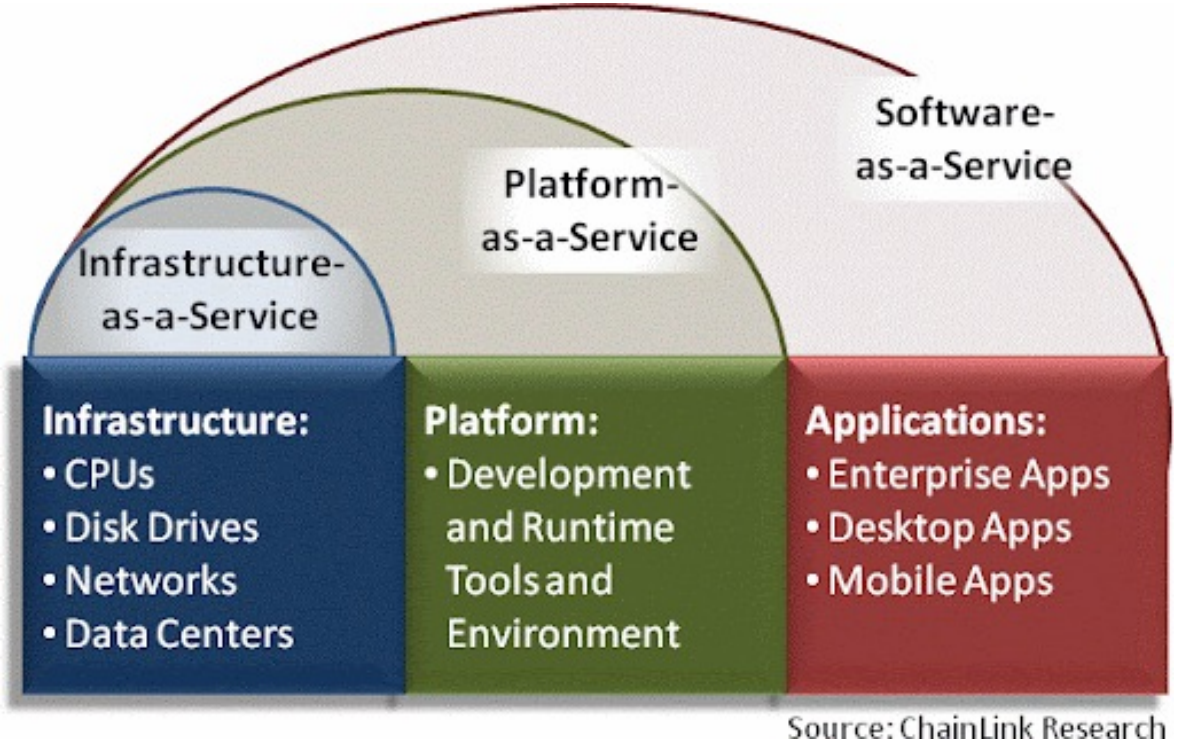
Cloud Computing

- Abstraktion von konkreten IT Ressourcen
  - z.B. Rechenkapazität, Datenspeicher, Netzwerkkapazitäten oder auch fertige Software
- Ressourcen werden dynamisch nach Bedarf zur Verfügung gestellt



3 Ebenen der Cloud-Dienste

- Infrastructure as a Service (IaaS):
  - Zugang zu virtualisierten Hardware-Ressourcen, wie Rechnern, Netzwerken und Speicher.
- Platform as a Service (PaaS):
  - Zugang zu Programmierungs- oder Laufzeitumgebungen mit flexiblen, dynamisch anpassbaren Rechen- und Datenkapazitäten.
  - Entwicklung eigener Software-Anwendungen innerhalb einer Softwareumgebung, die vom Dienstanbieter (Service Provider) bereitgestellt und unterhalten wird.
- Software as a Service (SaaS) oder Software on demand:
  - Zugang zu (laufenden) Software-Bibliotheken und kompletten Anwendungsprogrammen.



Source: ChainLink Research

Transparenz

- Transparenz bedeutet, dass ein Entwickler sich einer bestimmten Eigenschaft nicht bewusst sein muss, um diese zu nutzen.

- Transparenz erleichtert die Programmierung von verteilten Systemen.

  – Zugriffstransparenz
    - Auf lokale und verteilte Ressourcen kann auf dieselbe Weise zugegriffen werden

  – Ortstransparenz
    - Zugriff auf eine Ressource ist unabhängig von dessen Ort

  – Persistenztransparenz
    - Die Speicherung und das Laden von Objekten geschieht unbemerkt für den Entwickler

Entwurfsmuster: Client-Dispatcher-Server

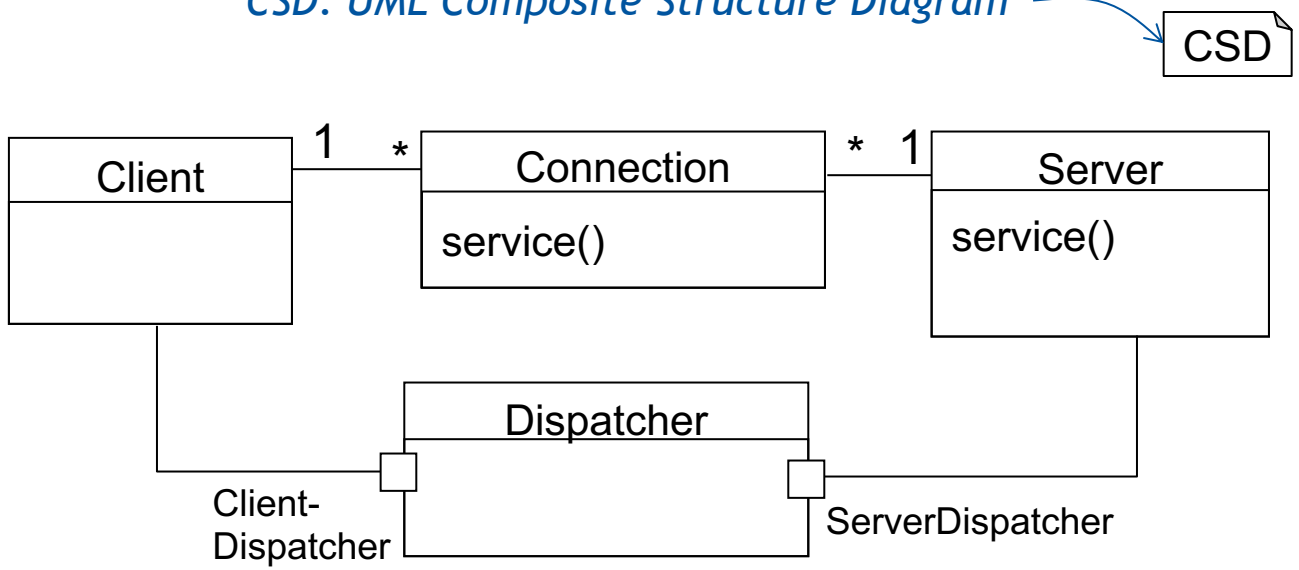
- *Problem:* Ein Client möchte Kontakt zum Server aufnehmen. Die genaue Adresse ist unbekannt.
  - Server nur über Namen bekannt
  - System will dynamisch einen Server zuordnen
    (z.B. für Load balancing in der Cloud)
  - Genauer Server irrelevant, nur Eigenschaft entscheidend
    (z.B. Auswahl des Mirror servers mit bester Verbindung)

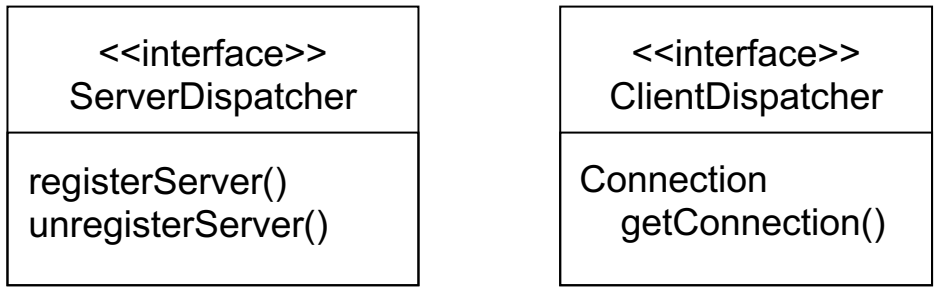
- *Lösung:*
  - Server durch Namen identifiziert und bei einem Dispatcher registriert
  - Client fordert von dem Dispatcher eine Verbindung zu einem Server.
  - Der Dispatcher wandelt Namen in physikalische Adresse um und stellt die Verbindung her.

Client-Dispatcher-Server

- Verbindungsaufbau über den Dispatcher
- Verbindung nach den Aufbau direkt zum Server über Connection-Objekt
  - kein Performanceverlust
  - Vermeidung eines Bottleneck beim Dispatcher

CSD: UML Composite Structure Diagram

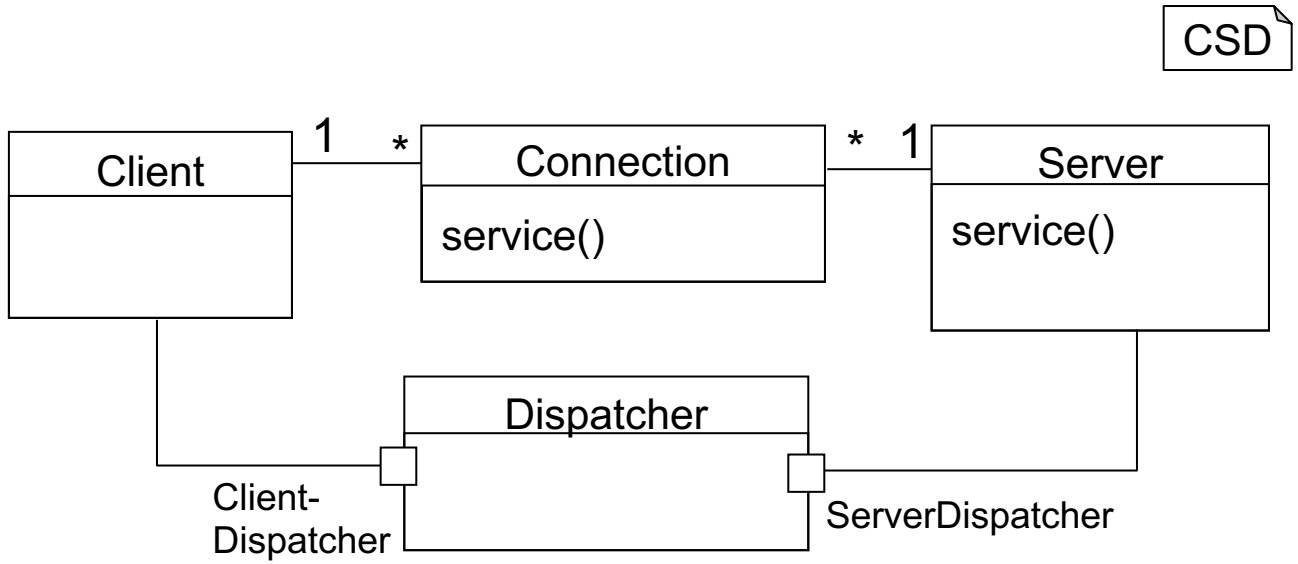




Parameter können Name oder Eigenschaften des Servers sein

Client-Dispatcher-Server: Vor- und Nachteile

- *Vorteile*:
  - Server können dynamisch ausgetauscht werden.
  - Verbindungen zum Server werden durch einen Namen oder eine Eigenschaft aufgebaut. (Ortstransparenz)
  - Re-Konfigurierbarkeit
  - Fehlertoleranz:
    - Redundante Server können sich bei einem Ausfall beim Dispatcher registrieren.



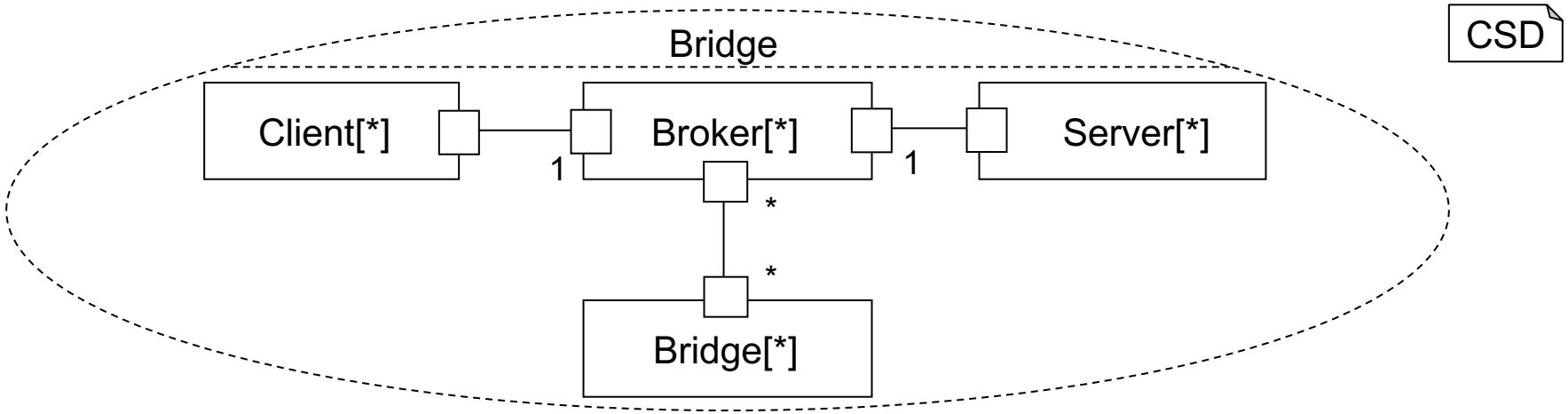
- *Nachteile*:
  - Overhead durch Kommunikation mit Dispatcher
  - Änderungen Dispatcher-Interface haben weitreichende Folgen
  - Dispatcher: Single-Point-of-Failure

Architekturmuster: Broker

- *Probleme*:
  - Kommunikation von Client- und Server-Komponenten soll unabhängig sein
    - von konkret beteiligten Systemen
    - vom Standort der Client- und Server-Komponenten
  - Dynamik
    - Komponenten sollen zur Laufzeit hinzugefügt, ausgetauscht und entfernt werden können
- *Lösung*:
  - Das Broker-Architekturmuster dient zur Strukturierung von verteilten Systemen mit entkoppelten Komponenten
  - Einsatz von Remote Procedure Call (RPC)
    - Aufruf einer Prozedur (Funktion, Methode) auf einem Server, als wenn diese lokal wäre
    - realisiert Zugriffstransparenz

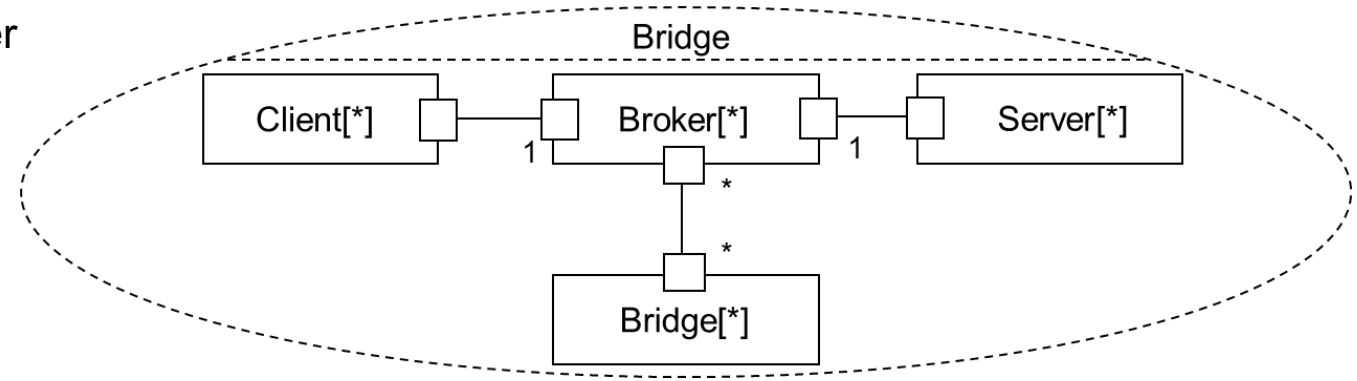
Architekturmuster: Broker

- Eine Broker-Komponente übernimmt die Kommunikation zwischen Client und Servern und stellt eine API bereit.
- Server registrieren sich beim Broker.
- Kommunikation zwischen Komponenten über Nachrichten.
- Methodenaufruf auf entfernte Komponenten wird über Proxies in Nachrichten umgewandelt (Verwendung der Broker-API).
- Ein Broker leitet Nachrichten an die entsprechenden Server weiter und übermittelt Ergebnisse und Ausnahmen an den Client.
- Verschiedene Broker können über Bridges miteinander kommunizieren.



Broker Vorteile & Nachteile

- *Vorteile*:
  - Ortstransparenz: Standorte von Server und Client nicht gegenseitig bekannt.
  - Austauschbarkeit der Komponenten ohne Änderungen an der Architektur
  - Broker-Systeme sind portabel
    - Reimplementierung des Brokers möglich, aber
    - Proxies nutzen unveränderte Schnittstelle



- *Nachteile*:
  - Geringe Effizienz
    - Die Kommunikation über einen Broker ist langsamer als direkte Kommunikation zwischen Komponenten
  - Fehlertoleranz-Maßnahmen notwendig
    - Sonst: Ausfall eines Brokers betrifft alle Komponenten.

Variante: Direct Communication Broker

- Broker wird stark belastet, falls alle Kommunikation über ihn läuft
- Deshalb: Nur Verbindungsaufbau über Broker
  - Großteil der übrigen Kommunikation über Proxies

- Beispiel: CORBA
  - Common Object Request BROKER Architecture
  - OMG-Standard

Skeleton = Server-Side Proxy

Stub = Client-Side Proxy

ORB = Object Request Broker

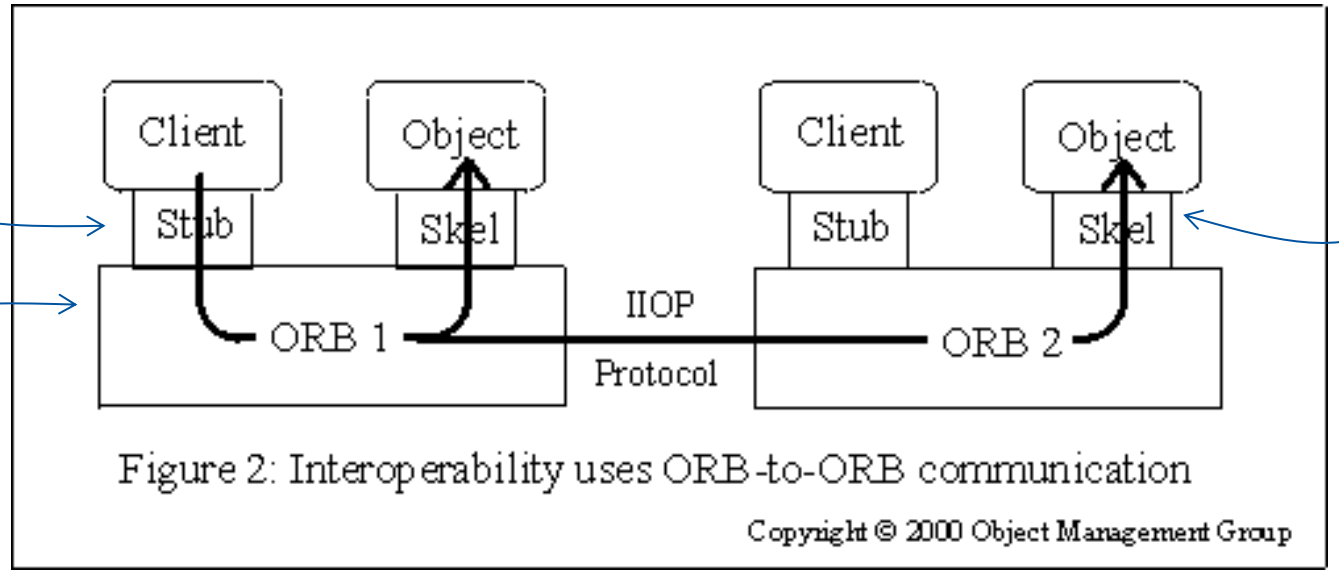
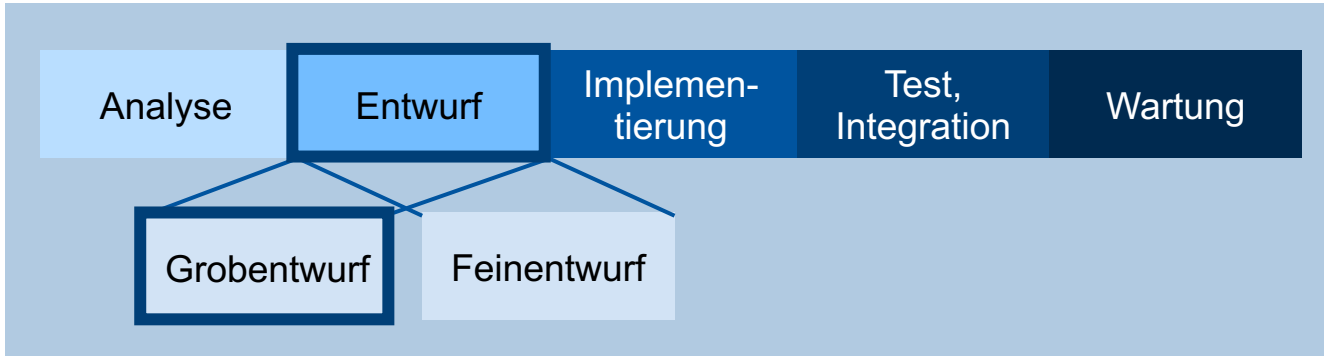


Figure 2: Interoperability uses ORB-to-ORB communication

Softwaretechnik

12. Komponenten
12.3. Architekturmuster für die Kommunikation



Prof. Bernhard Rumpe
Software Engineering
RWTH Aachen

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

Literatur:

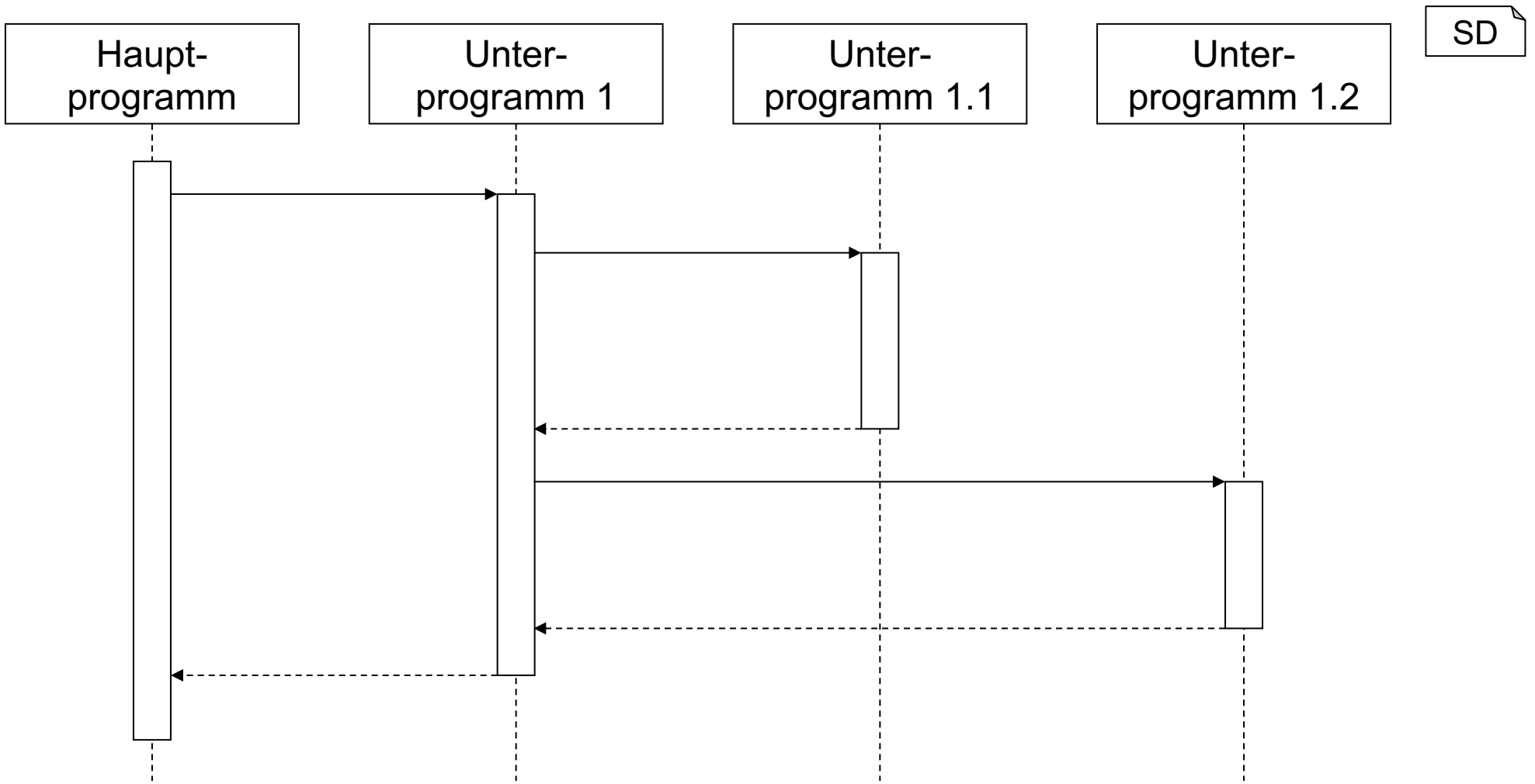
- Gamma/Helm/Johnson/Vlissides: Design Patterns, Addison-Wesley 1994 *(= „Gang of Four“, „GoF“)*
- Buschmann/Meunier/Rohnert/Sommerlad/Stal: A System of Patterns, Wiley 1996

SE Software Engineering | RWTH AACHEN UNIVERSITY

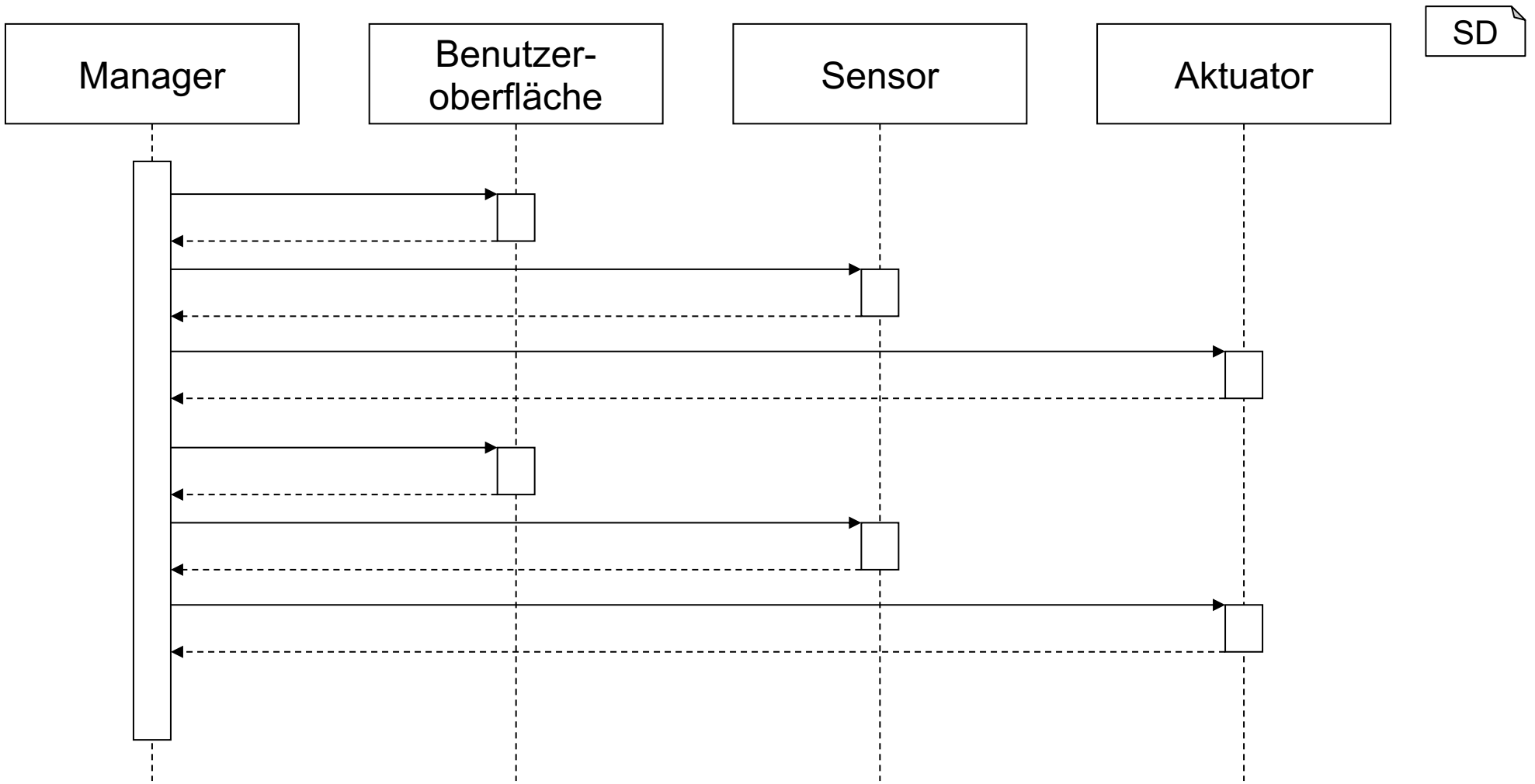
Architekturmuster der Kommunikation

- Kommunikation und Ablaufsicht der Architektur:
  - Definition nebenläufiger Systemeinheiten (z.B. Prozesse)
  - Steuerung der Abfolge von Einzelfunktionen
  - Synchronisation und Koordination
  - Reaktion auf externe Ereignisse
  - Datenflüsse (Wie fließen Daten?)
  - Kommunikationsflüsse (Wer ruft wen auf?)
- Darstellung
  - z.B. durch Sequenzdiagramme

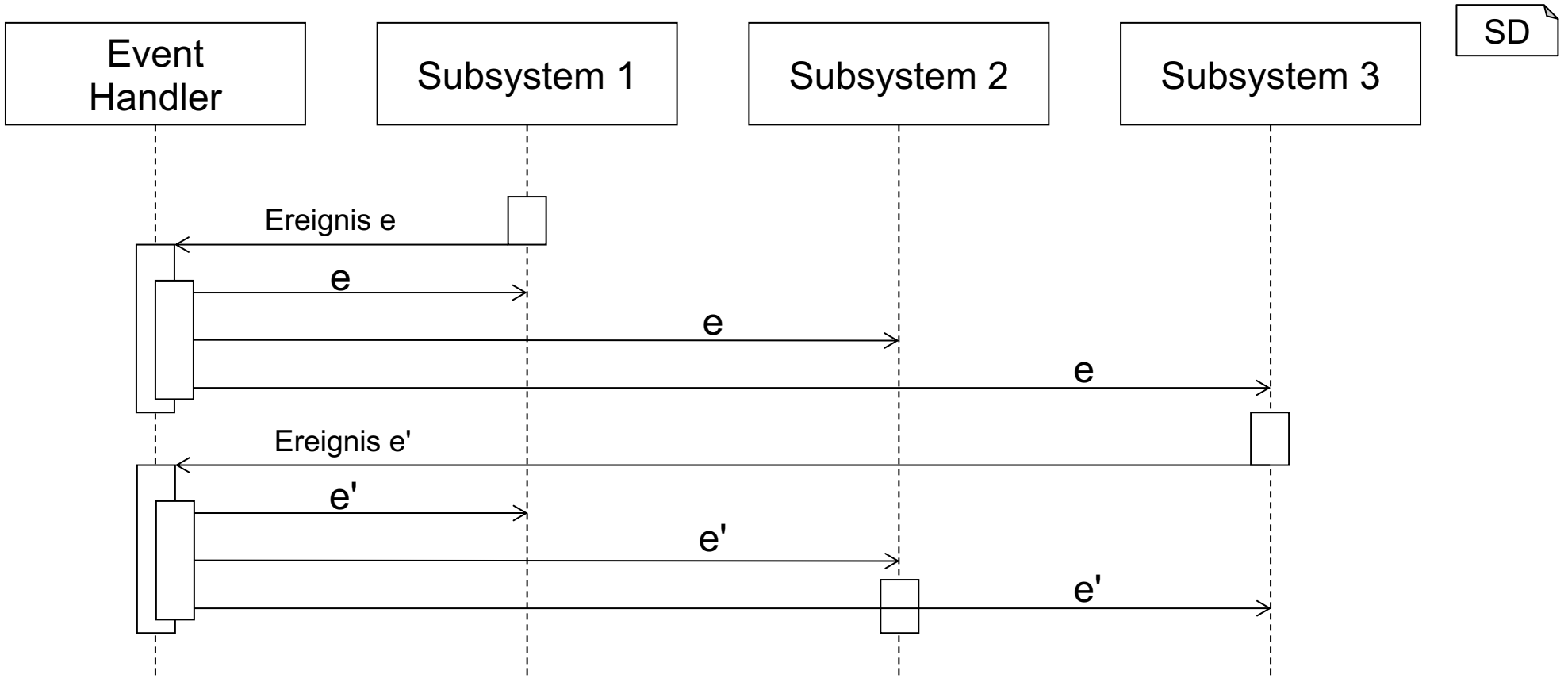
Steuerungsmuster "Call-Return"



Steuerungsmuster "Master-Slave"

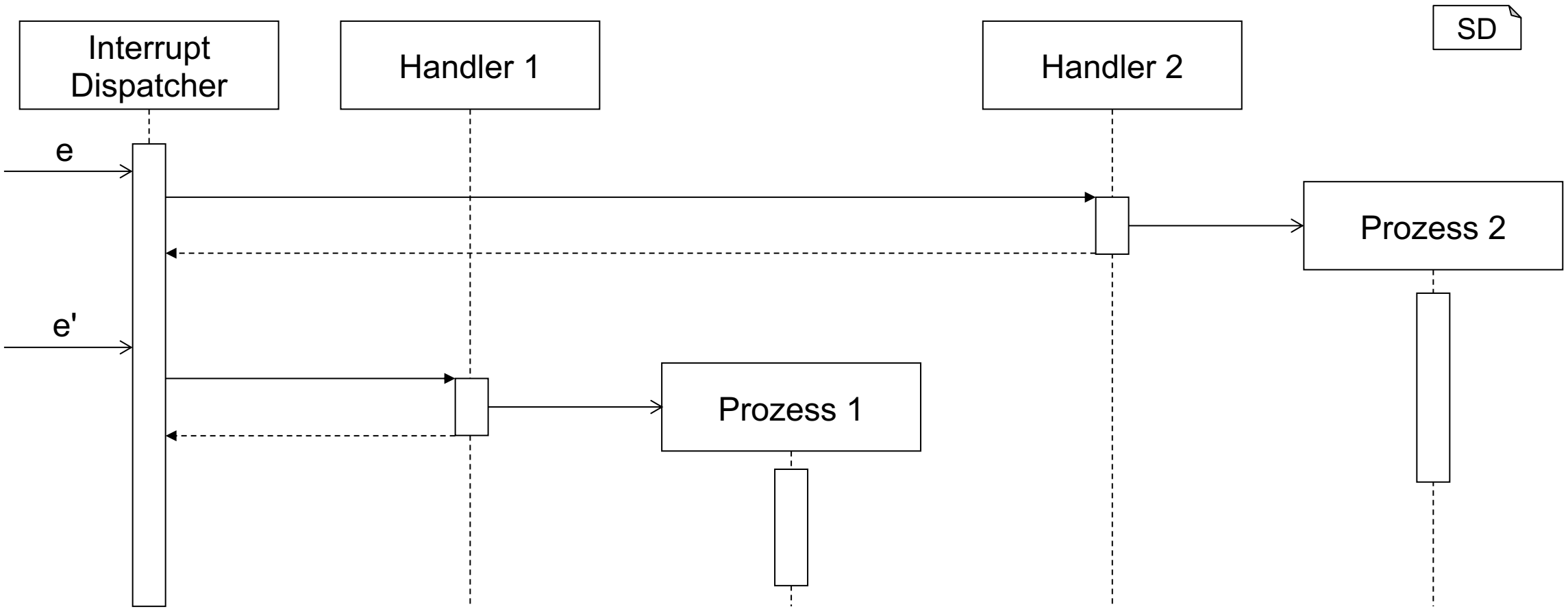


Steuerungsmuster "Selective Broadcast"



typisch bei asynchroner Kommunikation

Steuerungsmuster "Interrupt"



Speicherbare Aufrufe

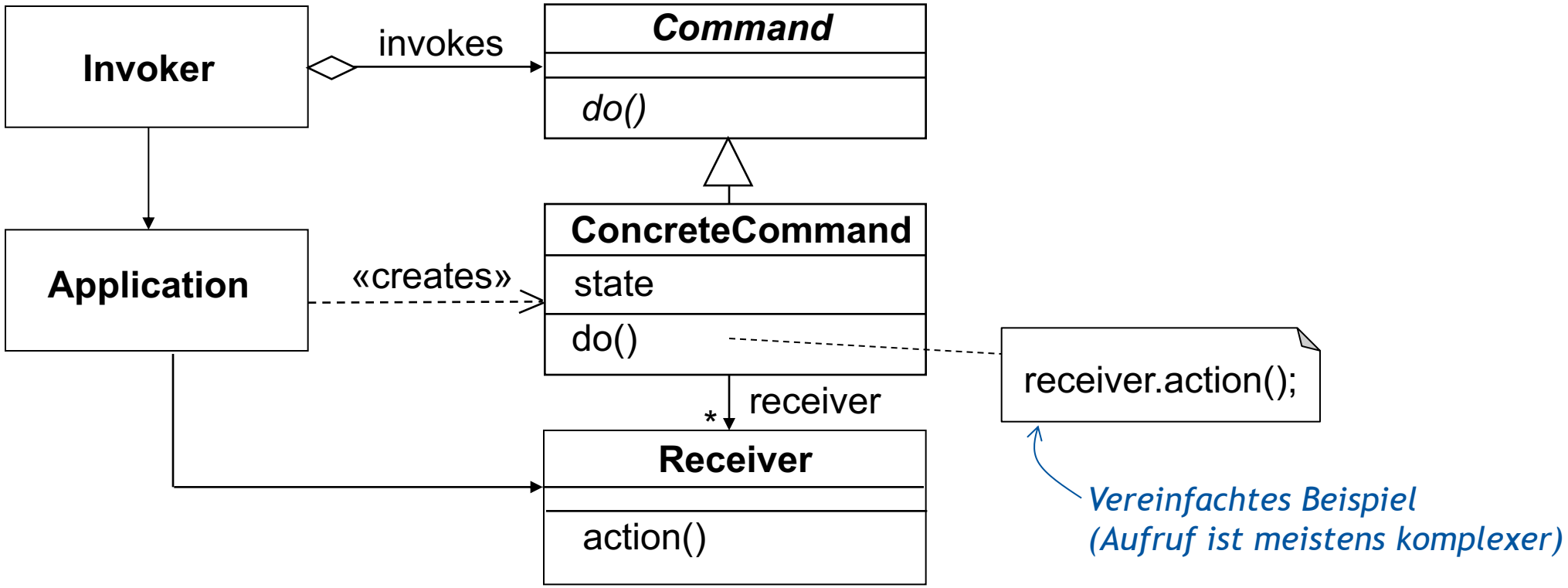
- Es kann notwendig sein, Aufrufe zu speichern und übertragen:
  - Methodenaufrufe zur späteren Ausführung abspeichern
  - Verteilte Systeme: Aufrufe übermitteln und in Warteschlangen verwalten
  - Aufzeichnen von Aufrufsequenzen (Recovery-Vorsorge)

- Command-Muster
  - Aufrufe als Objekte
  - Command ist (zusammen mit Memento) das wichtigste Muster für die Realisierung von "undo"-Operationen.

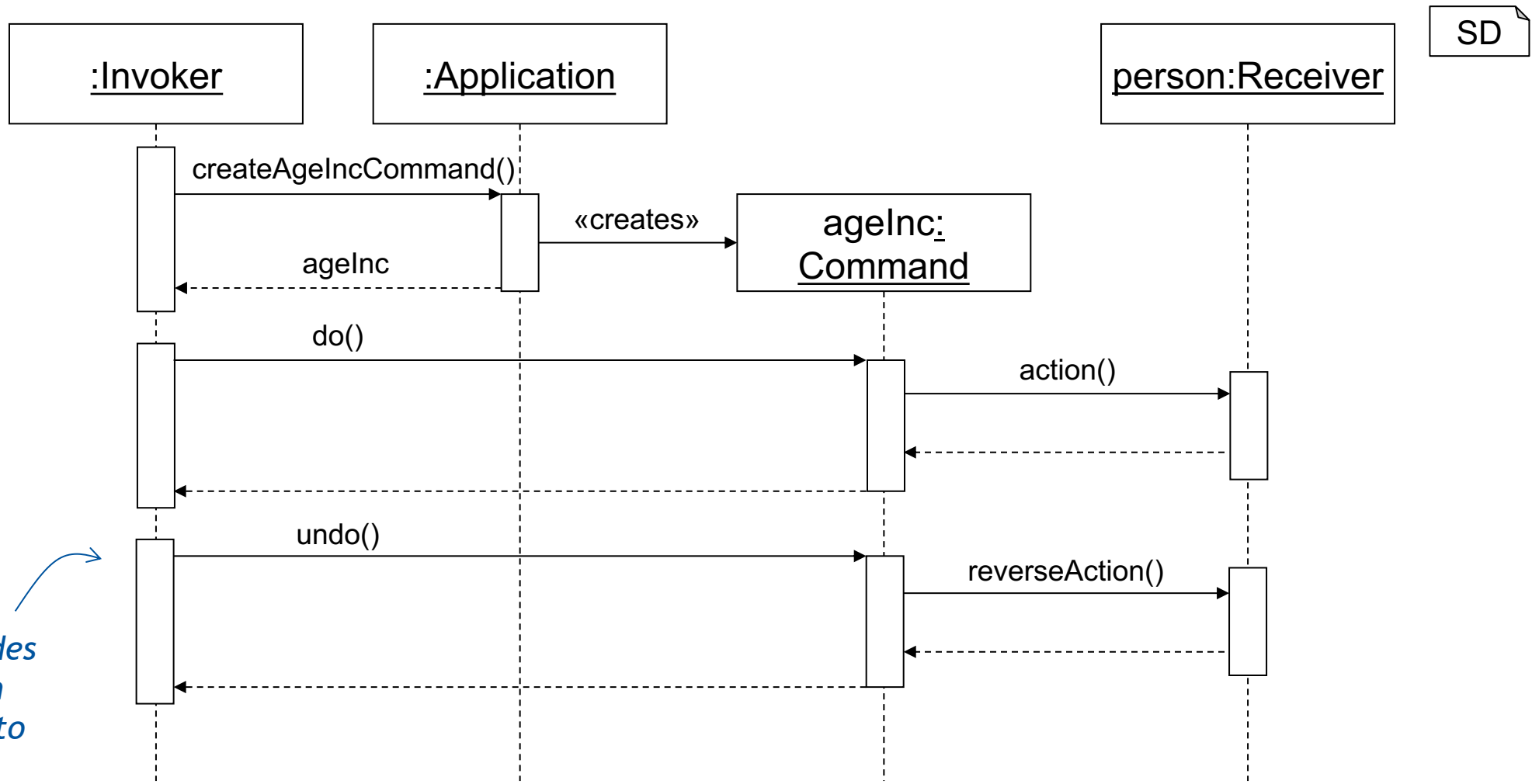
- Zentrale Verwaltung von Commands und Memento durch den CommandProcessor

Entwurfsmuster Command

- *Problem:* Aufrufende Objekte (z.B. aus einer Benutzungsoberfläche) sollen nichts über die aufzurufende Operation bzw. deren Empfänger wissen.
- *Lösung:* Befehle (commands) als Objekte

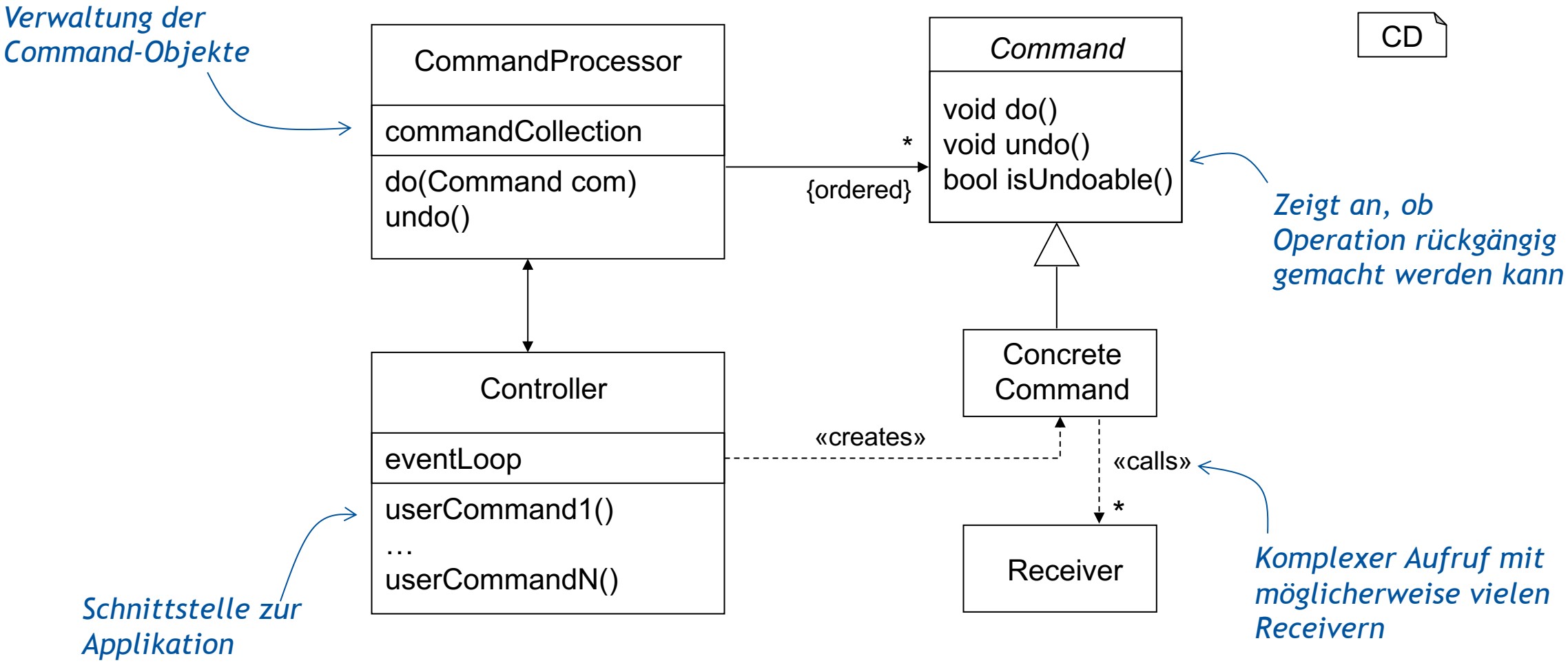


Ablauf eines Aufrufs



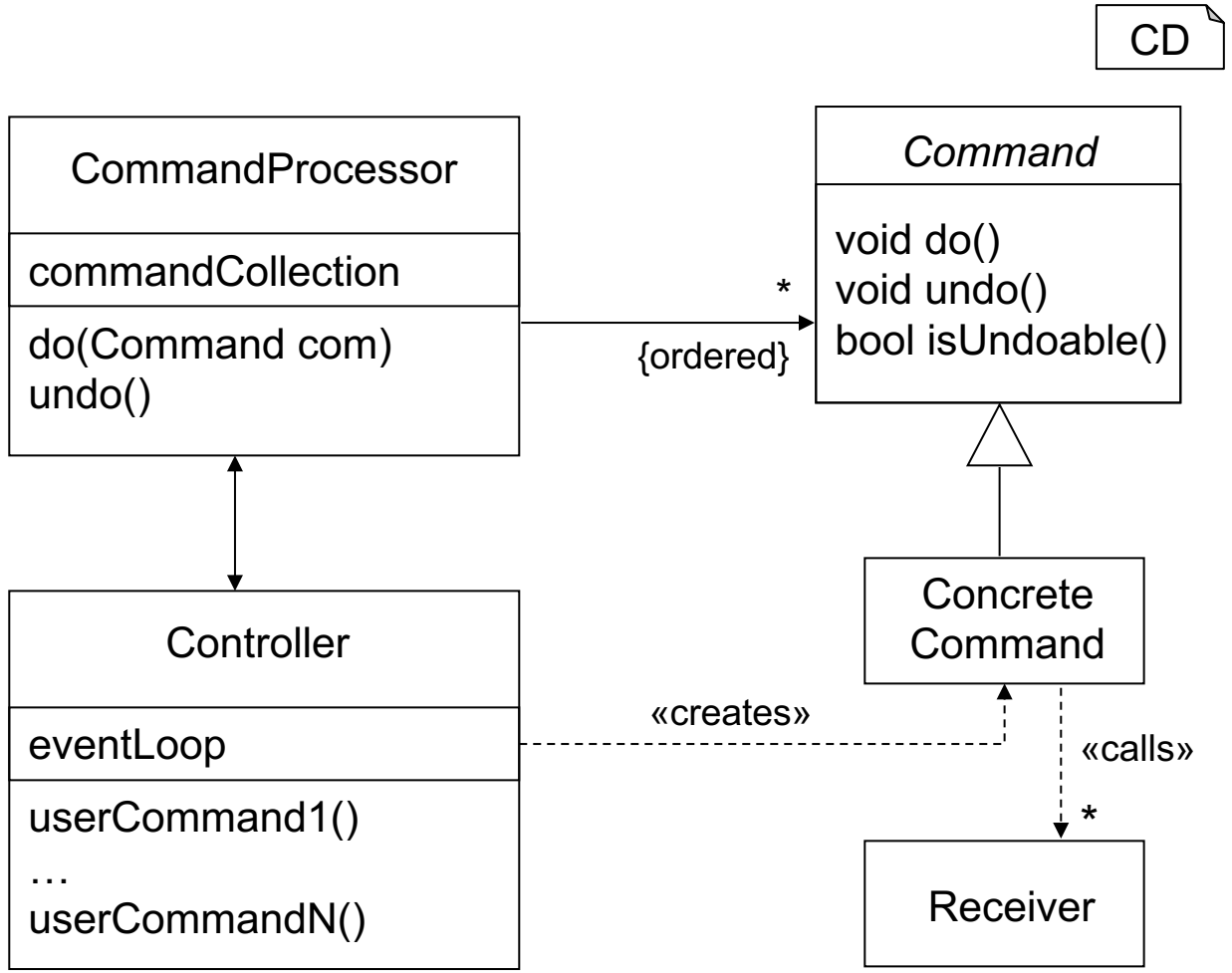
Erweiterung des Musters durch Undo/Memento

Entwurfsmuster: CommandProcessor



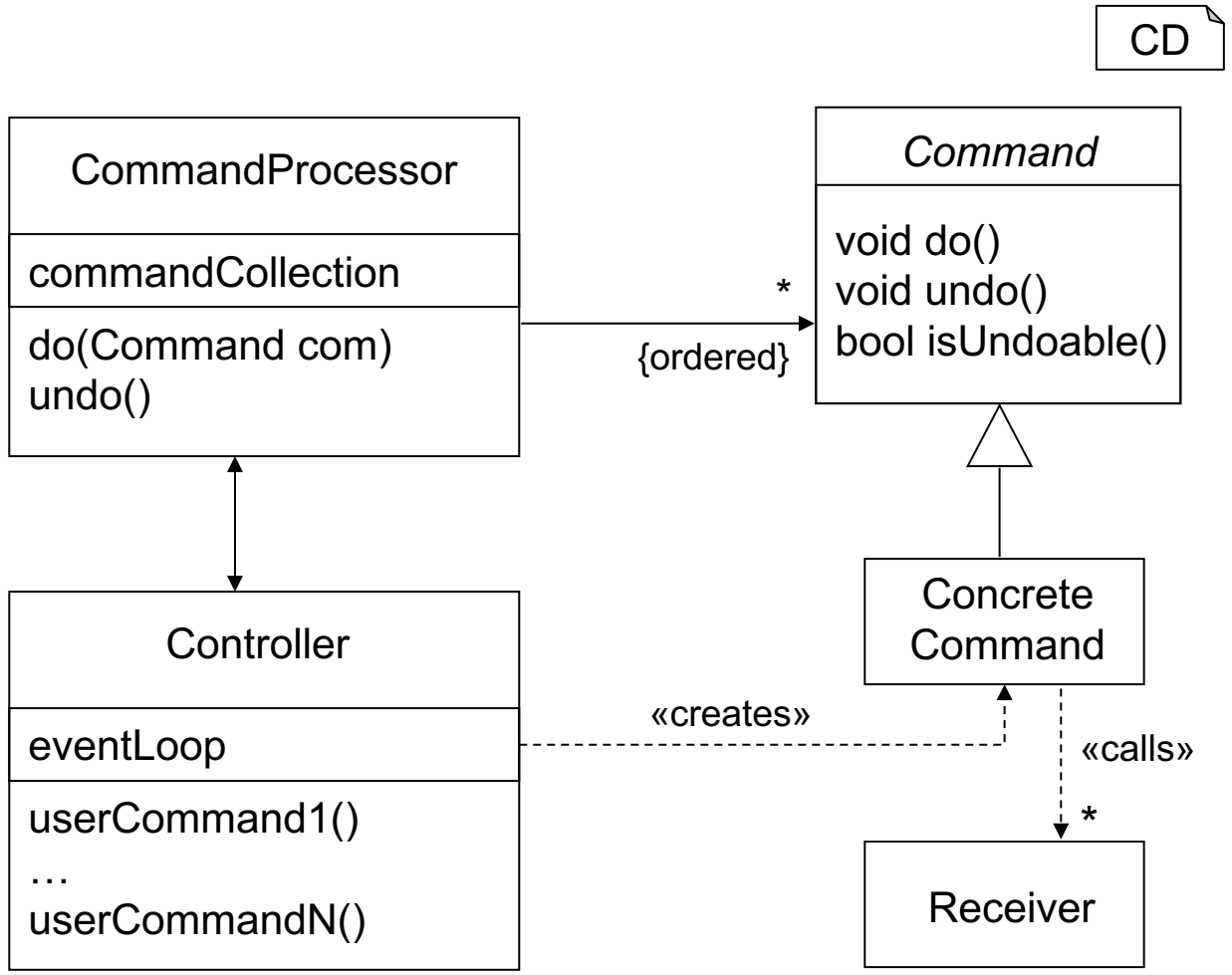
Rollen im Muster CommandProcessor

- CommandProcessor
  - Verwaltet Command-Objekte und führt diese aus
  - Ermöglicht Undo
- Controller
  - Stellt der Anwendung die Funktionen zur Verfügung
- Command
  - abstraktes Kommando mit do/undo-Signaturen
- ConcreteCommand
  - Aufrufimplementierung
- Receiver
  - kompliziertere Anwendungsfunktionalität
  - Genutzt von ConcreteCommands



CommandProcessor Vor- und Nachteile

- *Vorteile:*
  - Trennung der Programmlogik von der Benutzeroberfläche durch Kommandos
    - Einfache Modifizierbarkeit der Benutzerschnittstelle
    - Performantes Nutzerverhalten
  - Beeinflussung der Anwendung durch CommandProcessor möglich
    - automatisierte Tests
    - Makros
- *Nachteile:*
  - Geringere Effizienz als direkte Methodenaufrufe
  - Gefahr zu vieler Command-Klassen
    - Strukturierung der Klassen vorteilhaft
  - Aktionen sollten keine interaktiven weiteren Benutzereingaben verlangen



Softwaretechnik

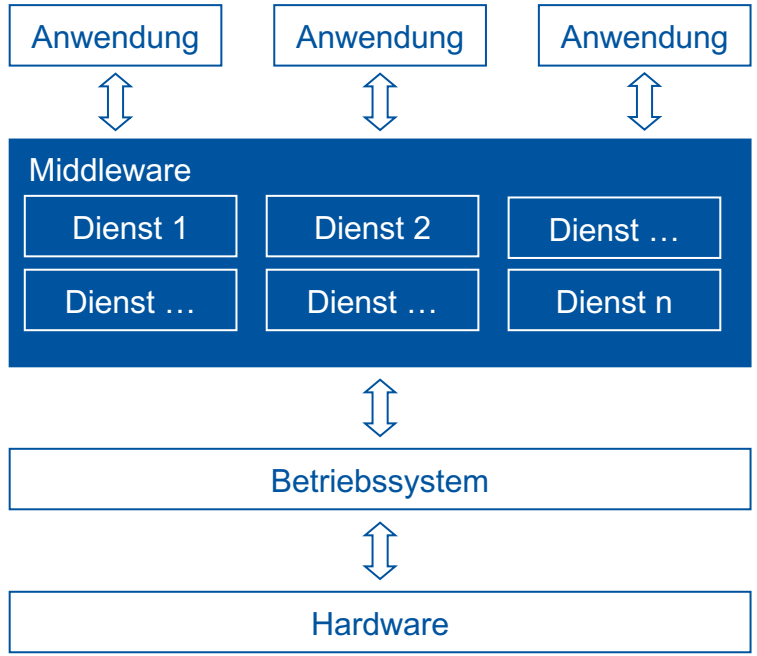
12. Komponenten
12.4. Middleware: Java EE & .NET

Prof. Bernhard Rumpe
Software Engineering
RWTH Aachen

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

Definition: Middleware

- Middleware ist die Zwischenschicht zwischen
  - Anwendung(en) und
  - Betriebssystem
- Bietet Funktionalität und Schnittstellen in Form von Diensten an
- Erleichtert die Entwicklung von verteilten Software-Systemen
  - Kommunikation zwischen Prozessen
- Typische Dienste
  - Namensdienste
  - Persistenz
  - Transaktionen
  - Security
  - Load-Balancing
  - Event Service
  - Diagnosis



Probleme der 3-Schichten-Architektur

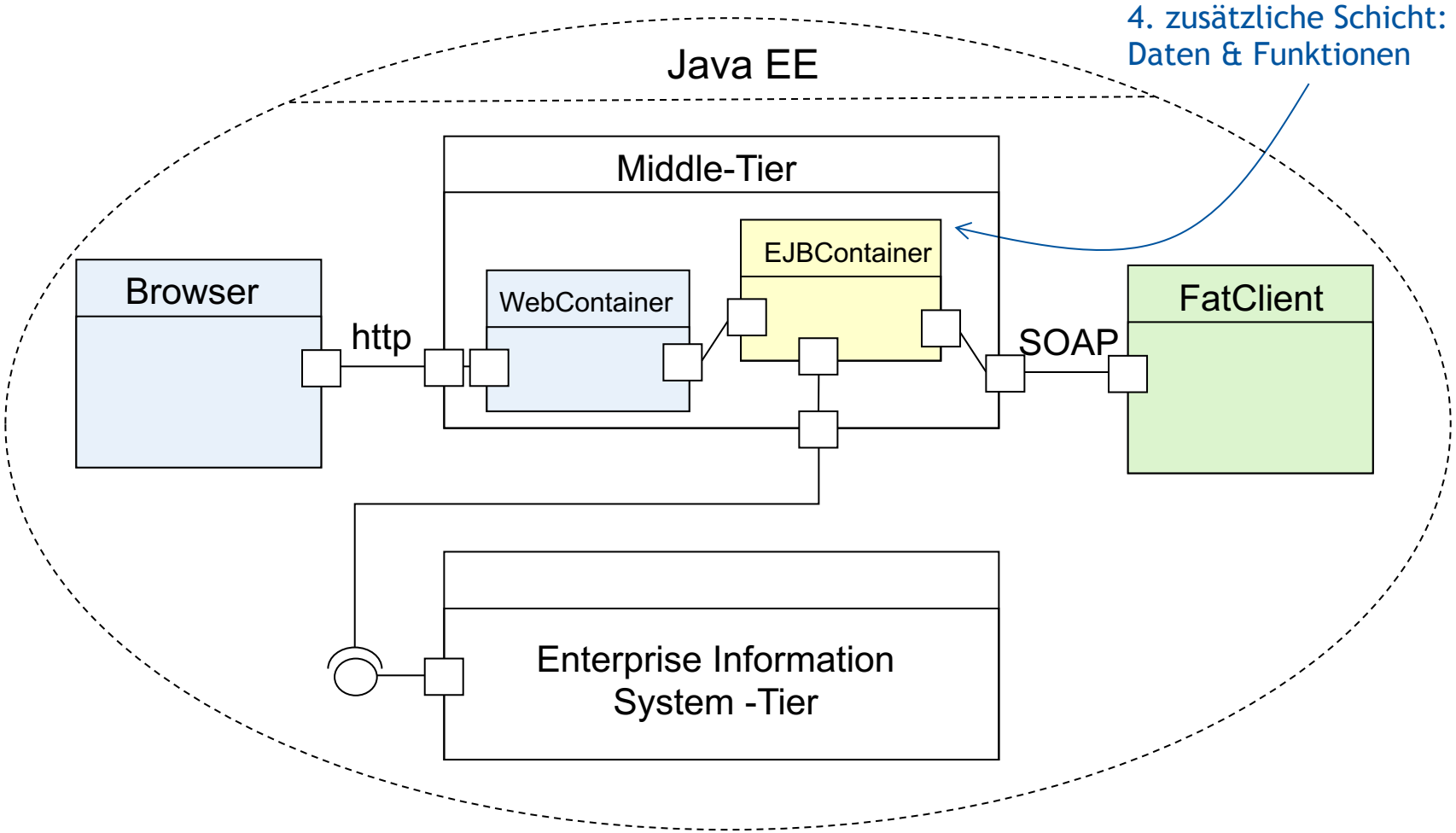
- 3-Schichten-Architektur
  - Client
    - Webbrowser
  - Serveranwendung
    - PHP, Java Servlets
  - Datenhaltung
    - RDBMS

- *Problem:*
  - Der Client ist ein so genannter „Thin-Client“, der wenig über die reine Darstellung hinaus leisten kann
  - Darstellungslogik und Anwendungslogik werden von der Serveranwendung realisiert: Server stark belastet
  - Vermischung von Thin-Clients und Think-Clients erschwert Erstellung und Wartung

- *Lösung:*
  - Einführen einer 4.Schicht

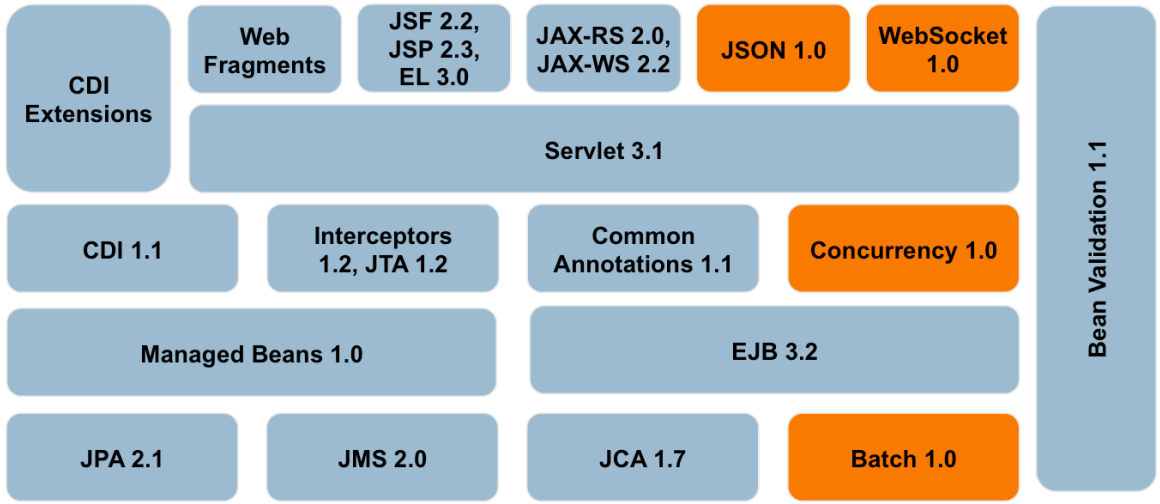
Java EE 4-Schicht-Architektur

- Java Enterprise Edition
- zwei wesentliche Bestandteile eines Java EE Servers
  - ein EJB-Container als Laufzeitumgebung für Enterprise Java Beans
  - ein Web-Container als Laufzeitumgebung für Servlets und Java Server Pages (JSP)



Java EE

- Java Plattform, Enterprise Edition (Java EE) ist die Spezifikation einer Standardarchitektur für die Ausführung von Java-EE-Applikationen.
  - Architektur nutzt Softwarekomponenten und
  - stellt Dienste zur Verfügung.
- Ziele
  - modulare Komponenten
  - verteilte, mehrschichtige Anwendungen
  - klar definierte Schnittstellen zwischen Komponenten bzw. Schichten
  - Softwarekomponenten unterschiedlicher Hersteller interoperabel
  - skalierbare verteilte Anwendung

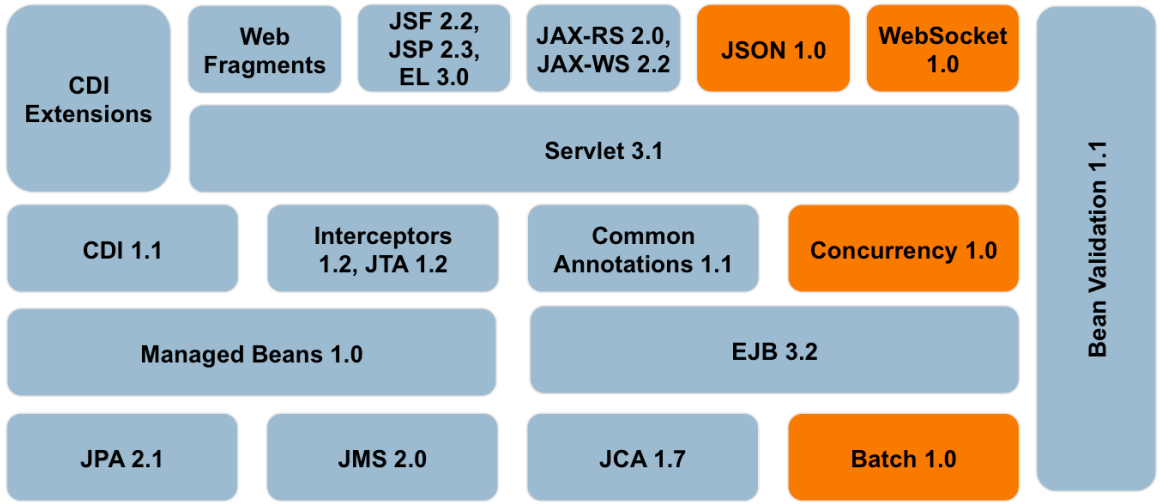


Java EE Infrastruktur

- Java EE Komponenten erfordern als Laufzeitumgebung eine spezielle Infrastruktur: den Java EE Application Server.

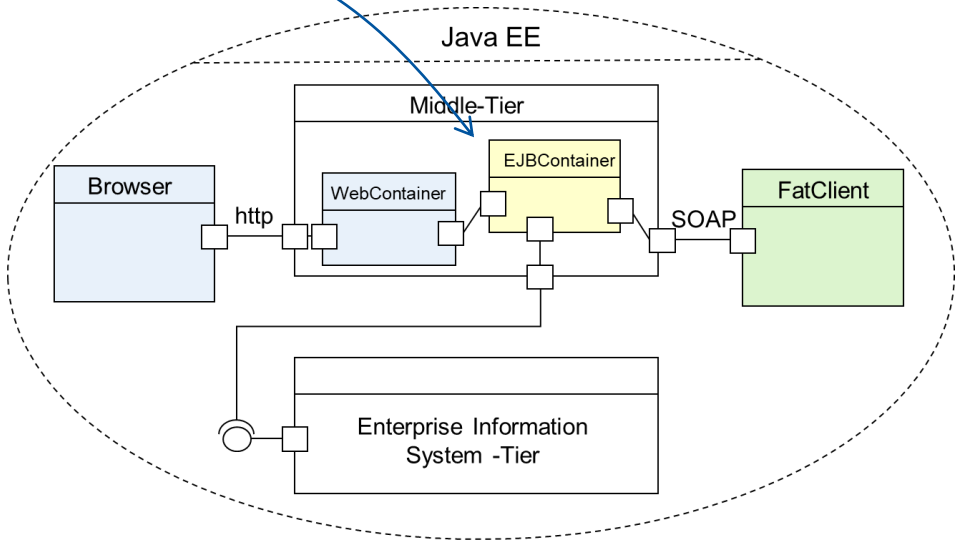
- Server bietet technische Funktionalitäten wie
  - Sicherheit (Security)
  - Transaktionsmanagement
  - Namens- und Verzeichnisdienste
  - Kommunikation zwischen Java-EE-Komponenten
  - Management der Komponenten (inklusive Instanziierung)
  - Unterstützung für die Installation (Deployment)

- Server kapselt Zugriff auf die Ressourcen des Betriebssystems (Dateisystem, Netzwerk ...)



Aufgaben eines EJB-Containers

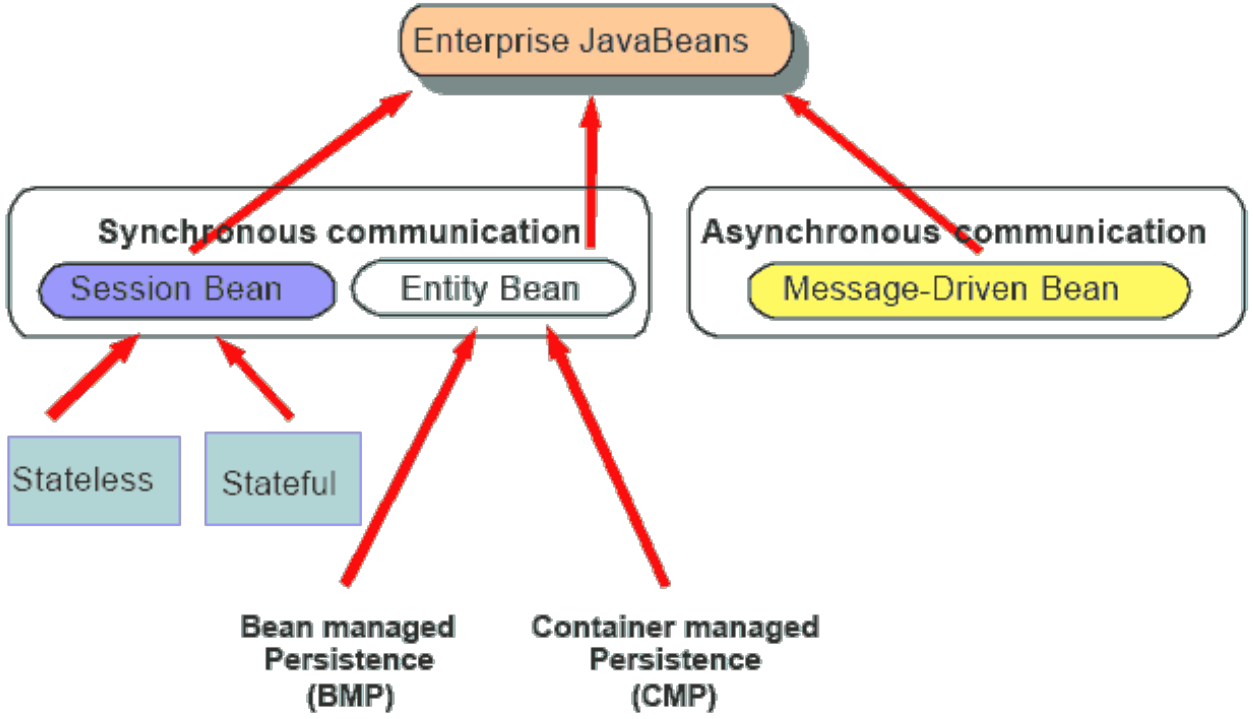
- Enterprise Java Beans (EJB)
  - sind die Komponenten des Java EE Frameworks
- EJB-Container „hegt und pflegt“ EJB‘s
- Auswahl der spezifizierten Aufgaben (Java EE/EJB 2.1 / 3.0)
  - Lebenszyklus Verwaltung von EJBs
  - Persistenzverwaltung
  - Transaktionen
  - Sicherheit
  - Zustandsverwaltung von Session Beans
  - Ortstransparenz beim Zugriff auf EJBs
  - Webservice-Anbindung
- eventuell zusätzlich (Differenzierung der EJB-Container-Anbieter)
  - Load balancing
  - …



Typ 1: Entity Beans

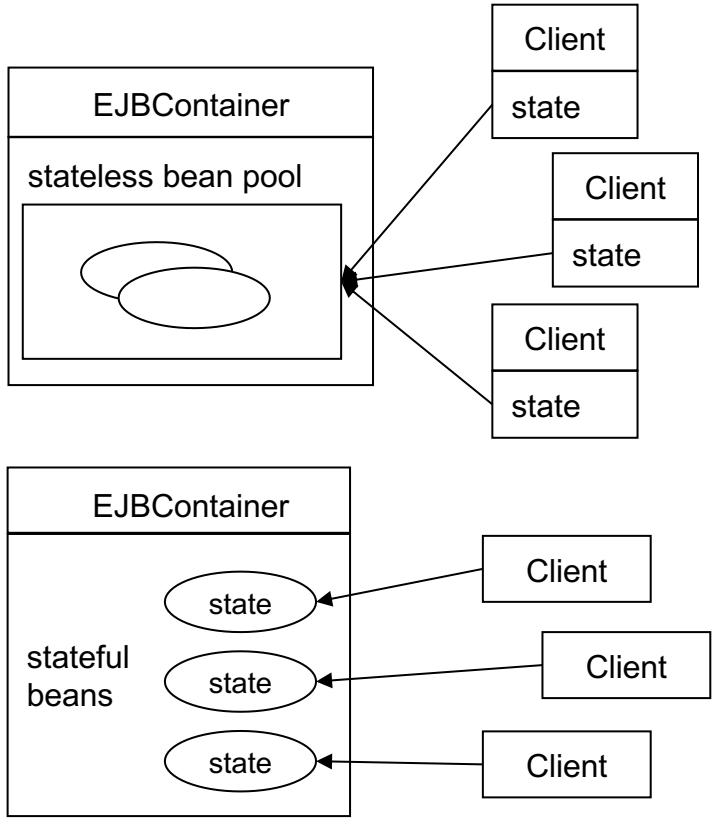
- Entity Beans
  - sind POJOs (Plain Old Java Objects)
  - Objektorientierte Repräsentation der Daten-Schicht
  - Nebenläufiger Zugriff durch mehrere Clients möglich
  - haben meist ein langes Leben: daher Persistenz

- Persistenz wird (normalerweise) von Entity Managern gesteuert
  - Werden vom Container bereit gestellt
  - Unterstützen CRUD-Operationen
    - Create
    - Read (Find)
    - Update
    - Delete



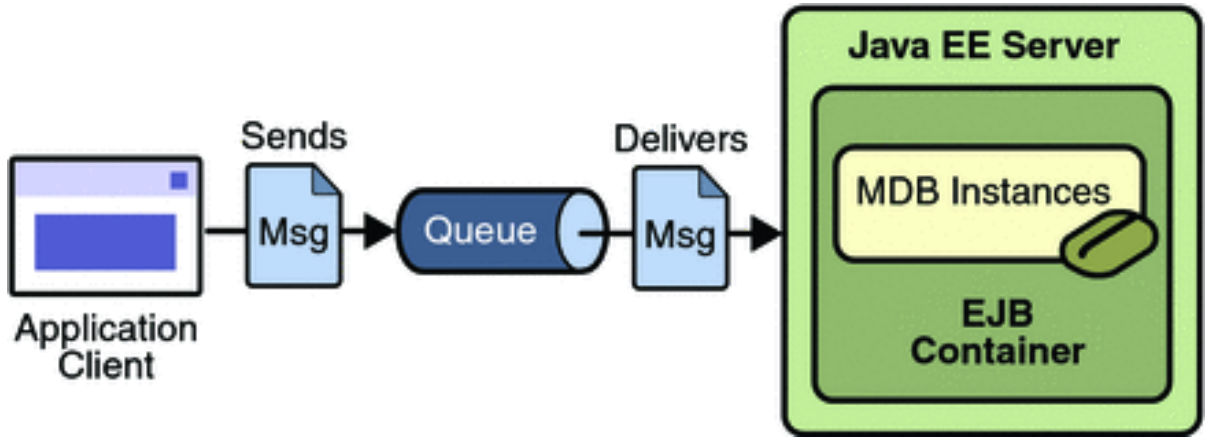
Typ 2: Session Beans

- Session Beans
  - Beinhalten Anwendungslogik
  - bearbeiten Anfragen
  - sind kurzlebig ("Session"), werden selbst nicht in der DB gespeichert
- Varianten:
  - stateless Sessions beans
    - Reine Funktionalität ohne Gedächtnis, nicht an einen Client gebunden
    - notwendige Daten liegen beim Client und/oder dann doch in der Datenbank
    - Können als Web Service aufgerufen werden
    - `lightweight` - wenig Ressourcen am Server notwendig
    - Skaliert sehr gut auf viele Rechner
  - stateful Session beans
    - Sessionverwaltung, keine persistenten Daten
    - EJB-Container bindet Session-Object an einen Client und leitet seine Aufrufe immer an dieselbe Instanz weiter

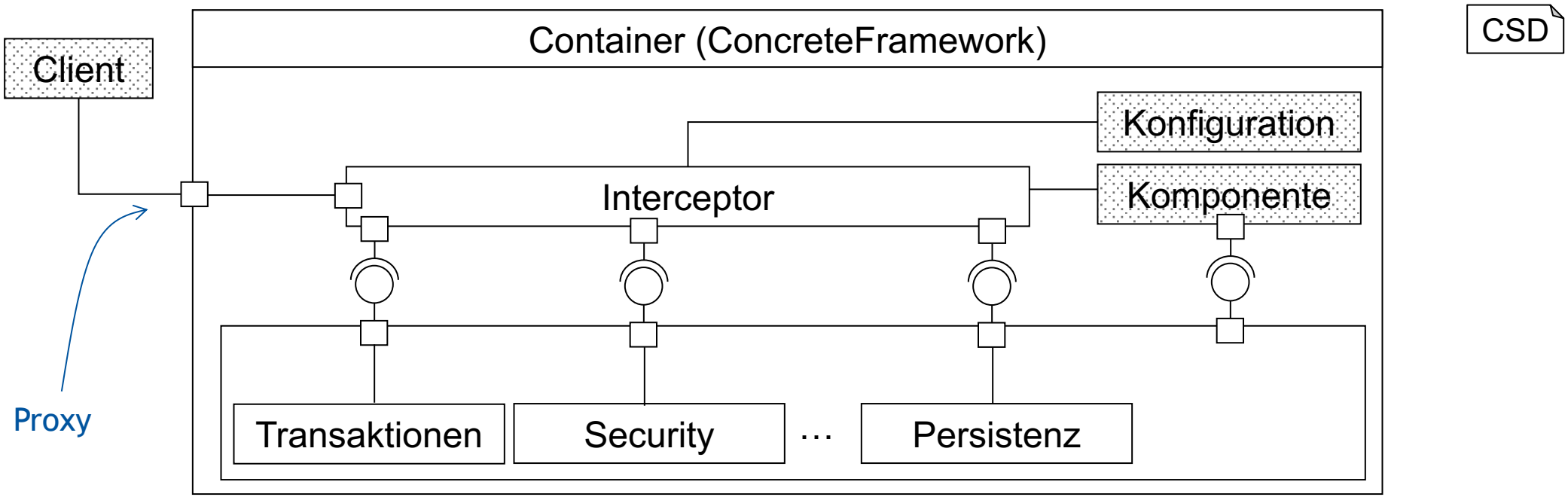


Typ 3: Message Driven Beans

- Message-driven Bean
  - Kommunikation von EJB über Servergrenzen
    - Aufgaben-Zuteilung
    - Datenaustausch
  - Asynchrone Kommunikation
  - Verwenden JMS (Java Message Service)



Grundsätzliches Designmuster für EJB‘s: Interceptor-Proxy



- Verwendung des Interceptor-Proxy-Musters
- Verhalten der Komponente wird durch die Konfiguration beeinflusst
- Der Container (z.B. ein Java EE Applikationsserver) stellt Dienste zur Verfügung

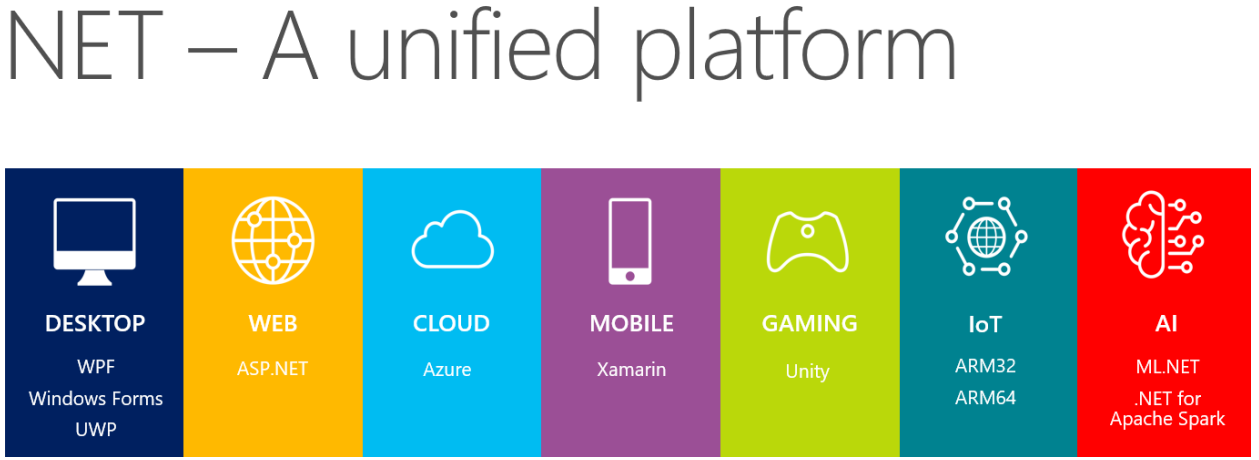
Alternativer Ansatz: Microsoft .NET

- Windows-Plattform gilt/galt nicht als sicher und stabil
  - Sicherheitslücken
  - Viren/Würmer/Trojaner
  - Probleme durch DLL-Hell
  - viele nicht interoperable Programmiersprachen

- .NET als Lösung
  - verschiedene Sicherheits-Features
  - keine Versionskonflikte
  - objektorientierte Plattform
    - „normale“ Anwendungen
    - verteilte Systeme
    - Web-Anwendungen
  - Einen Zoo von binärkompatiblen Programmiersprachen (z: C#, F#, Visual Basic, ~TypeScript)

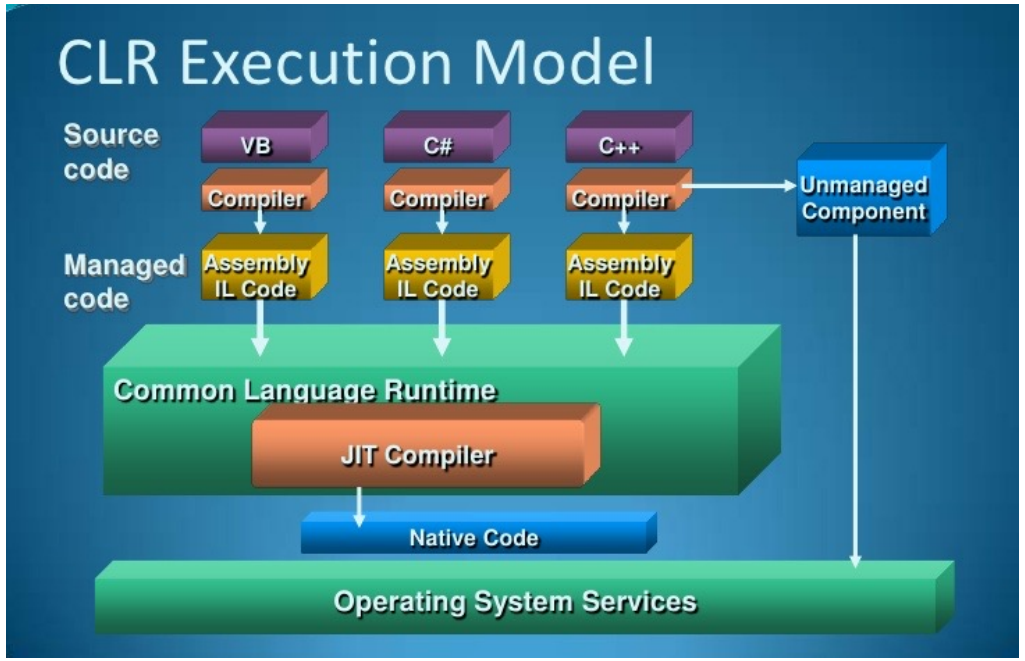
- Konkurrenz zu Sun/Oracles Java Enterprise Edition

Microsoft® .NET



Aufbau des .NET Frameworks (Auszug)

1. Zwischensprache (Intermediate Language)
   - „objektorientierte Assemblersprache“
   - vergleichbar mit Java ByteCode
2. Laufzeitumgebung (Common Language Runtime, CLR)
   - Just-In-Time Compilierung der IL in nativen Code
   - Kompilierter Code wird für weitere Ausführung gespeichert
   - Vergleichbar mit Java Runtime Environment
3. gemeinsames Typsystem (Common Type System, CTS)
   - Verwendung in allen .NET-Sprachen
4. umfangreiche Klassenbibliothek (Framework Class Lib, FCL)
   - ASP.net (dynamische Webseiten)
   - ADO.net (Datenbankanbindung)
   - Vergleichbare Funktionalität mit Java Klassenbibliothek



Bildquelle: https://www.itmagazine.ch/artikel/69503/_Net_Framework_4_8_mit_verbessertem_Malware-Schutz.html

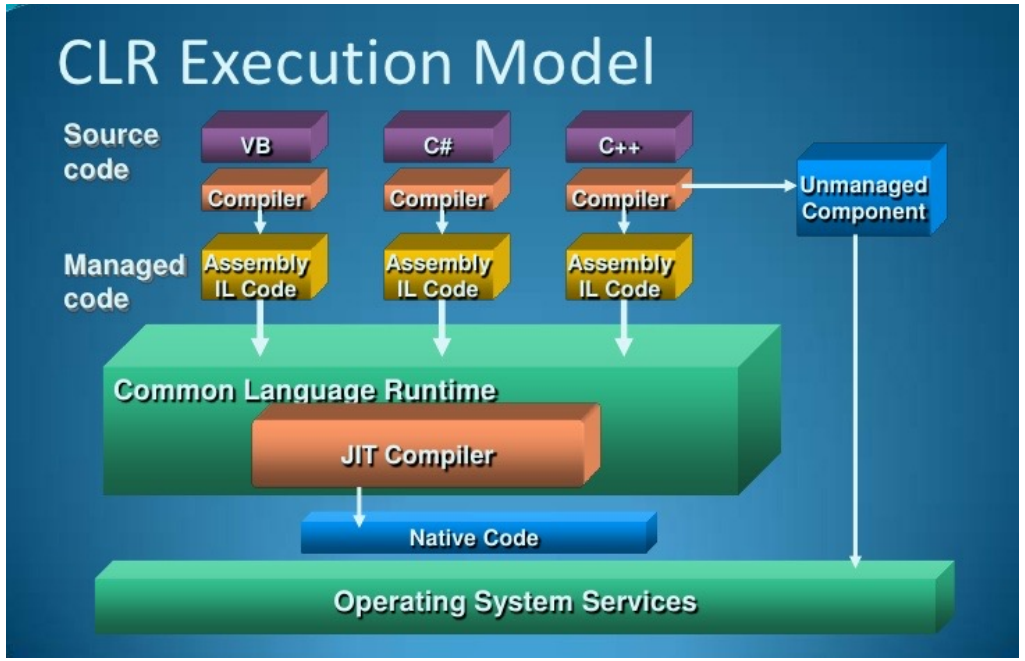
Common Language Infrastructure (CLI)

CLR | CTS | FCL | …

Assembly (= Komponente)

- Grundbaustein von .NET Framework-Anwendungen.
- Kleinste Einheit für
  - Weitergabe
  - Wiederverwendung
- Beinhalten Versions- und Sicherheitsinformationen
- Beschreiben sich selbst
  - bereitgestelltes Interface
  - implementierte Interfaces
- Inhalt kann über mehrere Dateien verteilt sein

- Assemblies können als Komponentenmodell verstanden werden



Bildquelle: https://www.itmagazine.ch/artikel/69503/_Net_Framework_4_8_mit_verbessertem_Malware-Schutz.html

Common Language Infrastructure (CLI)

Was haben wir gelernt?

Komponenten …

… unterstützen Modularität

… beschleunigen die Entwicklung durch Wiederverwendung

Verbindung durch Schnittstellen

Wiederverwendung einzelner Komponenten oder ganzer Frameworks

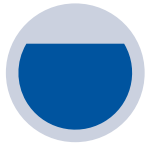
Muster zur Verteilung

Architekturmuster für verteilt benutzte Komponenten (HW)

Zentrales System, Client/Server, Three-Tier Client/Server, Föderation, Cloud Computing

Transparenz-Arten: Zugriff, Ort, Persistenz

Client-Dispatcher-Server, Broker, Direct Communication Broker

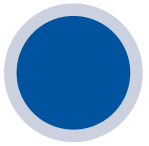


Muster zur Kommunikation

Architekturmuster: Kommunikation und Ablauf zwischen Komponenten

Call-Return, Master-Slave, Selective Broadcast, Interrupt

Speicherbare Aufrufe: Command



Middleware

… zwischen Anwendung(en) und Betriebssystem

4-Schicht-Architektur: Java Enterprise Edition

Komponenten-Art, z.B. Enterprise Java Beans (EJBs)

Vorlesung Softwaretechnik

13. Software-Produktlinien

Prof. Bernhard Rumpe
Software Engineering
RWTH Aachen

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

SE Software Engineering | RWTH AACHEN UNIVERSITY

Warum, was, wie und wozu? Software-Produktlinien

Warum?

- Aus einer Plattform individuelle Produkte ableiten
- "Halb"-Indivuduelle Kundenlösungen von der Stange

Was?

- Domänen Engineering
- Application Engineering

Wie?

- Prozessschritte
- Artefakte
- Feature Modelle
- Vorteile und Risiken

Wozu?

- Praxis: insbesondere in größeren Unternehmen mit ähnlichen Produkten
- Variabilitäts- & Konfigurations-management

Softwaretechnik

13. Software-Produktlinien

Prof. Bernhard Rumpe
Software Engineering
RWTH Aachen

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

Literatur:

- Pohl/Böckle/van der Linden: Software Product Line Engineering, Springer, 2007

SE Software Engineering | RWTH AACHEN UNIVERSITY

Produktlinien

- Produktlinien bezeichnen die Idee aus einer Plattform individuelle Produkte abzuleiten
- Die Nutzung gemeinsamer Komponenten spart Produktions- und Entwicklungskosten
- Erlaubt die rentable Entwicklung von Spartenmodellen mit kleineren Stückzahlen

- Software-Produktlinien
  - Übertragung der Idee auf die Software-Entwicklung

Volkswagen PQ35 Plattform

Ableitung von Produkten

Audi A3

VW Touran

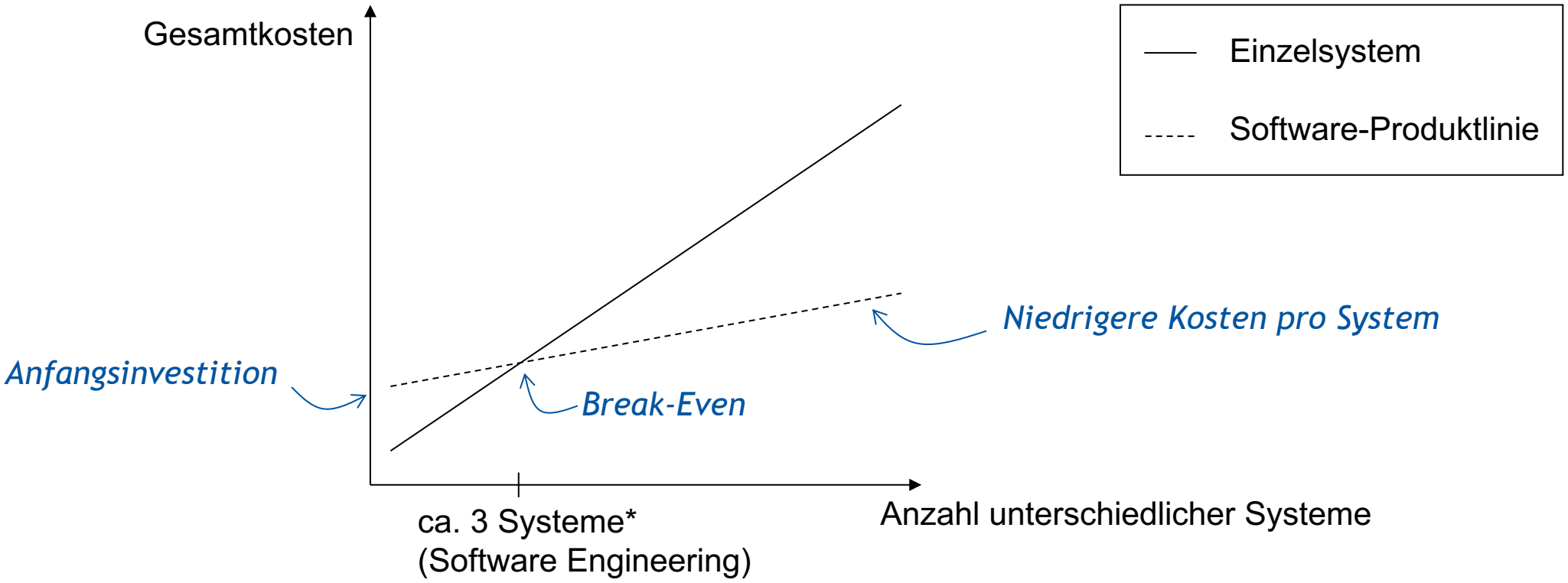
Seat Altea

Skoda Octavia II
SEAT Toledo III
SEAT León II
VW Eos
...

Beispiel: HP inkjet product family (Owen)

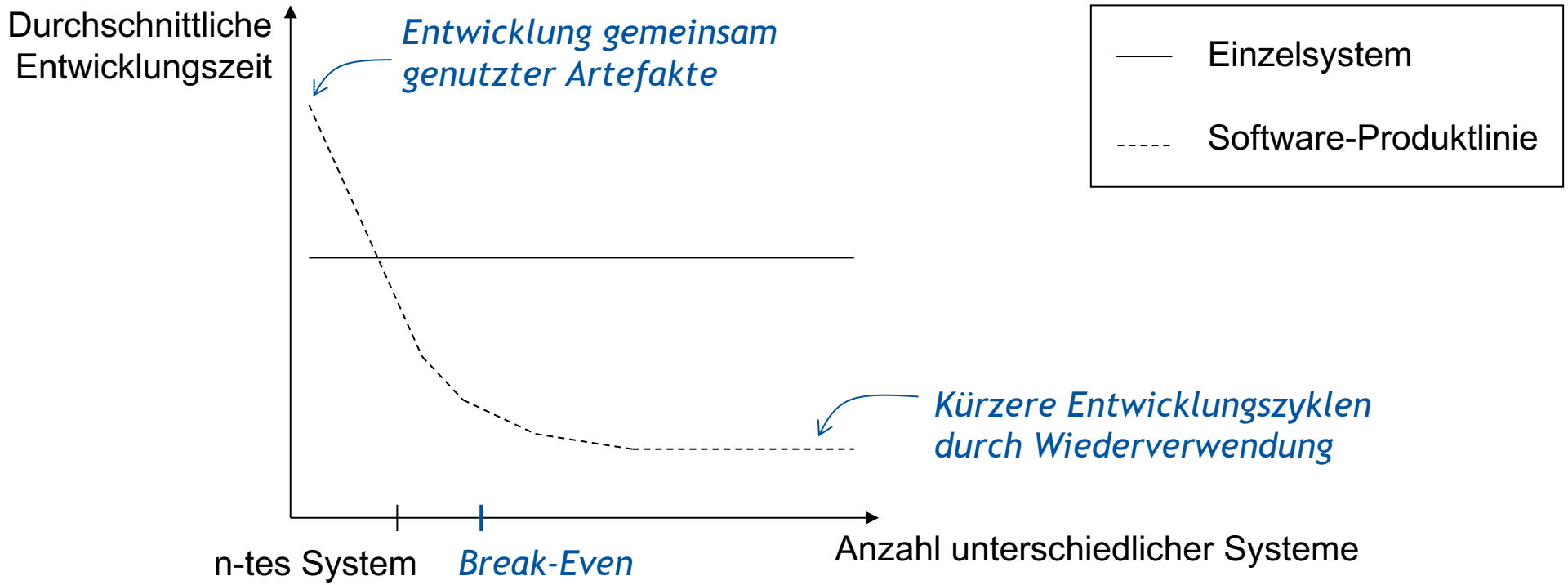
- Inkjet Drucker von HP
  - Software für die Firmware von Druckern
  - Realisierung durch einen Produktlinienansatz (Owen)
  - Nutzen eine gemeinsame Architektur mit austauschbaren Komponenten
- Owen (Stand 2007)
  - 30-35 Produkte pro Jahr
  - Codebasis > 5M LOC
  - 97 % eines typischen Produkts sind Produkt-unabhängig
- Vergleich mit Einzelentwicklung (Stand 2000)
  - 3x kürzere Entwicklungszeit
  - 4x kleinere Teamgröße
  - 25x kleinere Fehlerdichte

Kosten der Einzelsystementwicklung gegenüber einer Software-Produktlinie



* Zusammenstellung verschiedener Untersuchungen aus [CN02, S. 226]

Entwicklungszeit der Einzelsystementwicklung gegenüber einer Software-Produktlinie



- Vorteil eines Produktlinenansatzes
  - Erzeugung von einer Vielzahl von Varianten bei vernünftigen Kosten und Entwicklungszeit

Vorteile einer Software-Produktlinie

- Wiederverwendung einzelner Module erlaubt
  - Reduzierung der Entwicklungskosten
  - Erhöhung der Qualität
  - Reduzierung der Time-To-Market

- Für alle Produkte der Familie
  - Reduktion des Wartungsaufwands
    - Bessere Möglichkeiten zur Auswertung der Folgen einer Änderung
  - Reduktion der Gesamtkomplexität
    - Mehrfache Verwendung derselben Module
  - Verbesserung der Kostenschätzung für ein neues Produkt

- Vorteile für den Kunden
  - Ähnlichkeiten zwischen den Produkten
  - "Höhere Qualität für einen niedrigeren Preis"

Grundbegriffe

Definition: *Mass customisation*

"Mass customisation is the large-scale production of goods tailored to individual customers' needs."

[Dav87]

Definition: *Software Platform*

"A software platform is a set of software subsystems and interfaces that form a common structure from which a set of derivative products can be efficiently developed and produced."

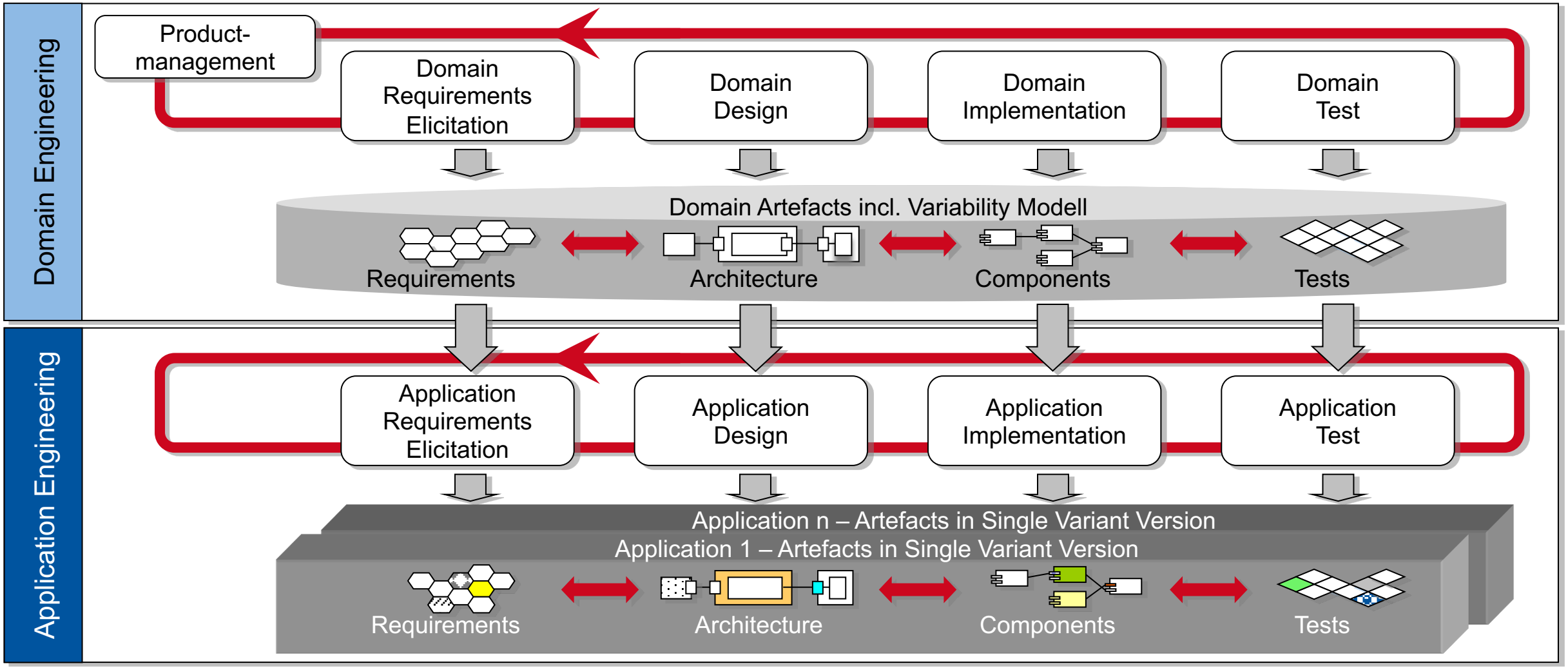
[ML97]

Definition: *Software product line engineering*

"Software product line engineering is a paradigm to develop software applications (software-intensive systems and software products) using platforms and mass customizations."

[PBL05]

Domain Engineering und Application Engineering [PBL05]



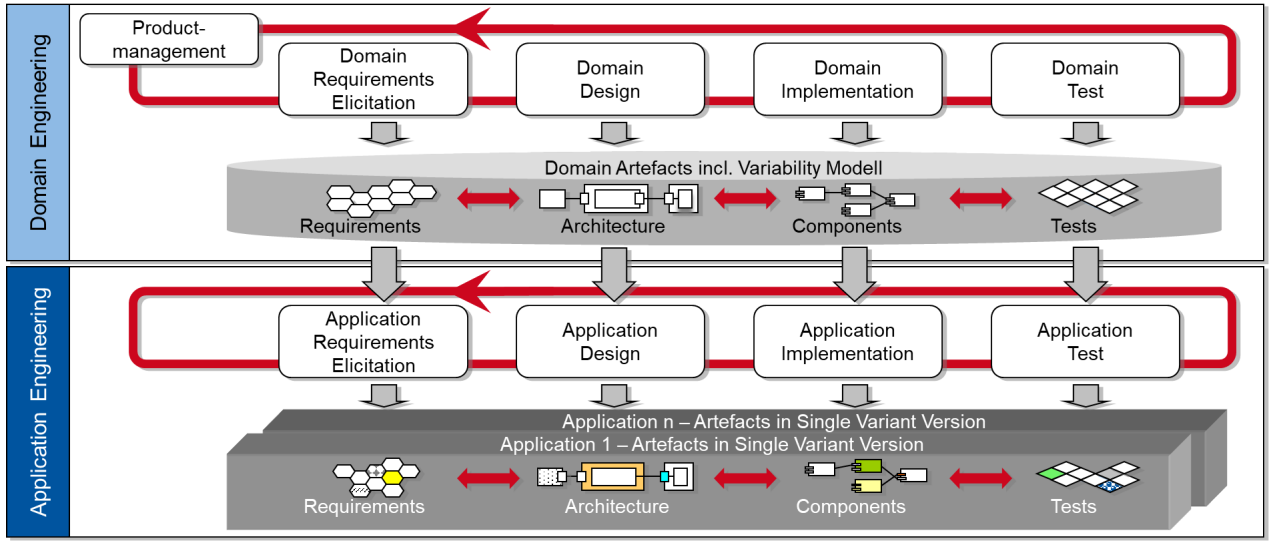
Trennung in Domänen und Applikation Engineering

Domänen Engineering

- Erfassung aller relevanten Konzepte der Domäne
- Realisierung einer wieder verwendbaren Plattform
  - Gemeinsamkeiten und
  - Unterschiede der einzelnen Produkte
- Wieder verwendbare Komponenten
  - Kostenersparnis
  - Qualitätssteigerung

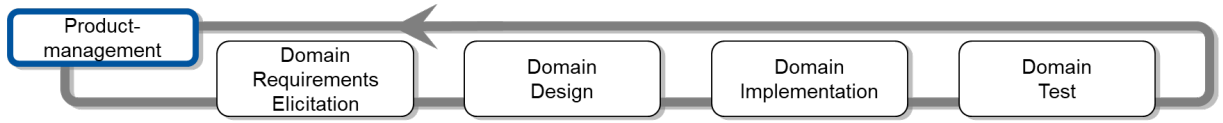
Applikation Engineering

- Ableitung konkreter Produkte aus einer Plattform
  - Auswahl der wieder verwendbaren Komponenten
  - Integration individueller Komponenten



Domänen Engineering: Produktmanagement

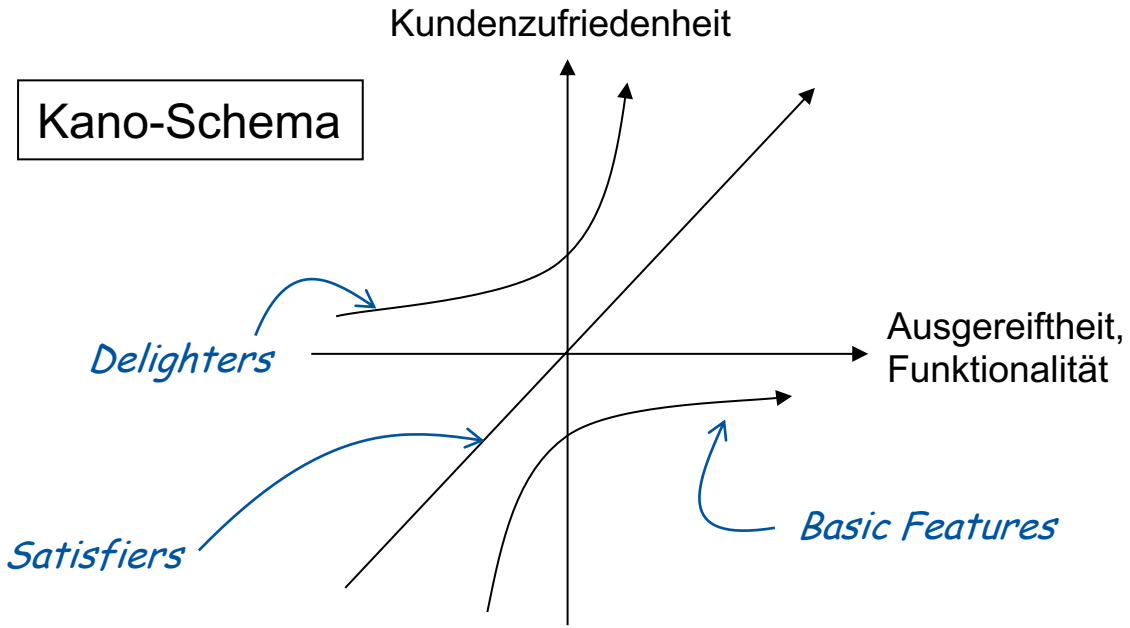
Aufgaben Produktmanagement



- Anwendungsbereich (Scope) festlegen
  - Zu schmal = Zu kleine Anzahl an relevanten Produkten
  - Zu breit = Keine Synergieeffekte

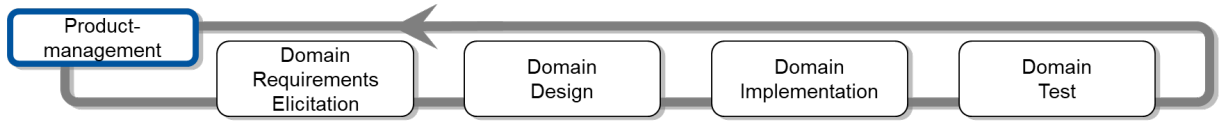
- Kundenbefragungen
  - Identifikation von Kundenwünschen
  - 20 bis 30 Interviews reichen für 90 - 95% der relevantesten Kundenwünsche
    [Griffin and Hauser 1993]

- Z.B. Kano-Schema beachten

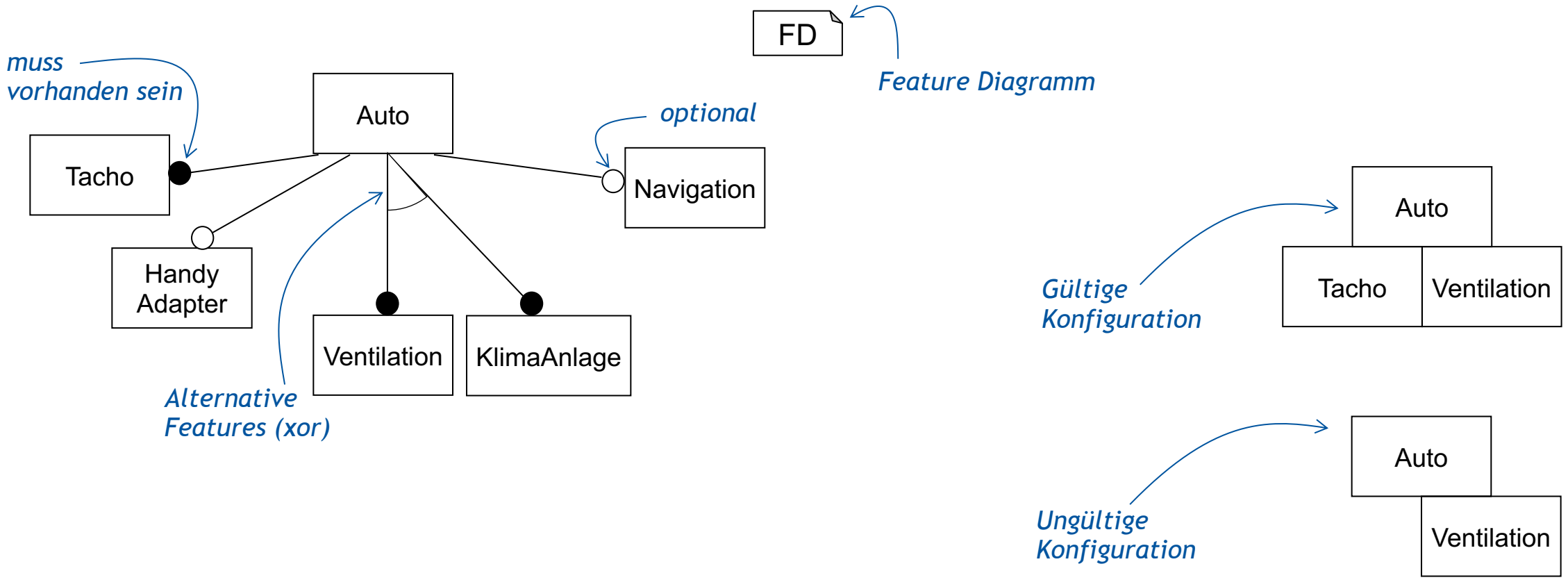


Domänen Engineering: Anforderungsanalyse

- Anforderungsanalyse wie für ein normales System
  - Funktionale Anforderungen
  - Qualitätsattribute
- Textbasierte oder modellbasierte Methoden möglich
- Zusätzlich zur normalen Anforderungsanalyse:
  - Analyse der Gemeinsamkeiten und Unterschiede aller Produkte
  - Erstellen eines Variabilitätsmodells zur Beschreibung der möglichen Konfigurationen
    - Darstellung als Feature-Diagramm
  - Antizipieren von voraussichtlichen Änderungen (z.B. future market needs)

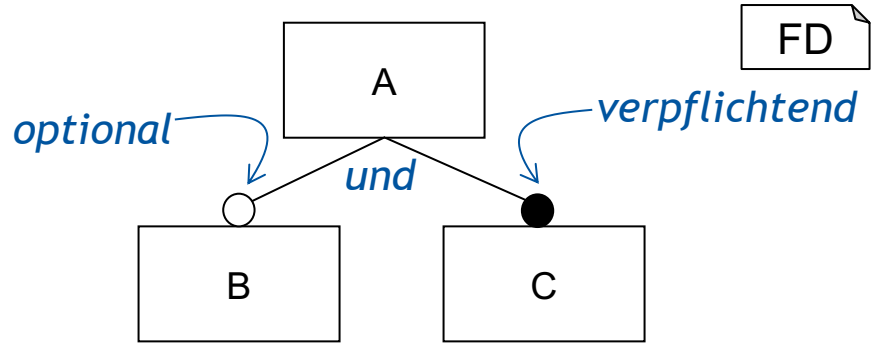


Beschreibung des Variabilitätsmodells durch Feature-Diagramme



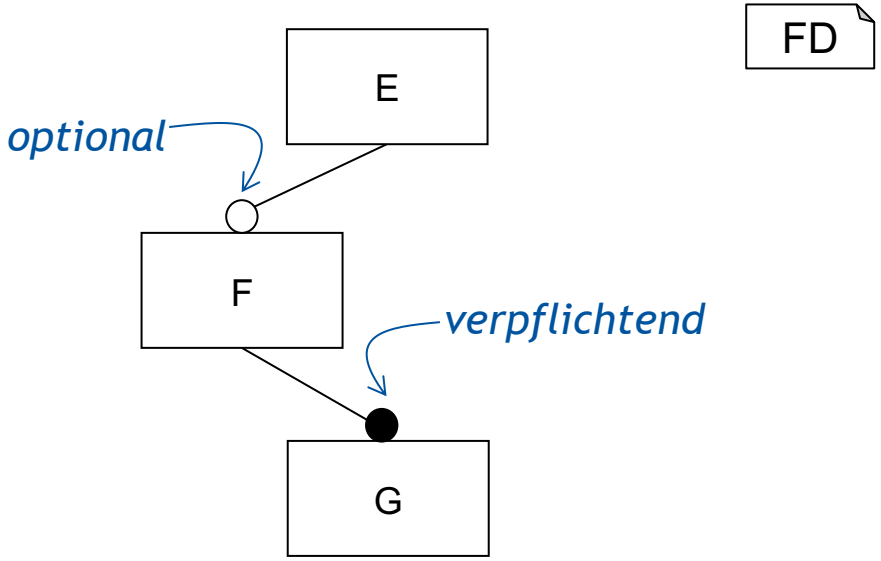
- Feature-Diagramme beschreiben *gültige Konfigurationen*
- Wesentliches Ergebnis einer Domänen Engineering Anforderungsanalyse

Feature Diagramme: Optional, Verpflichtend



- Genau dann wenn Feature A gewählt wurde, kann auch Feature B gewählt werden
- Genau dann wenn Feature A gewählt wurde, muss auch Feature C gewählt werden

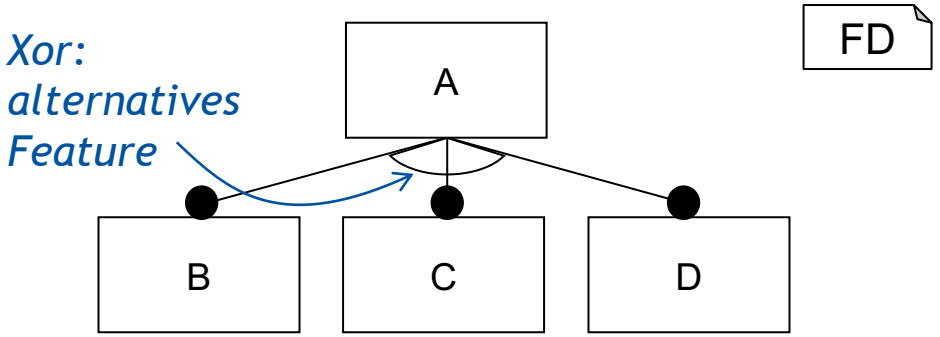
- Valide Kombinationen: {A,C} und {A,B,C}



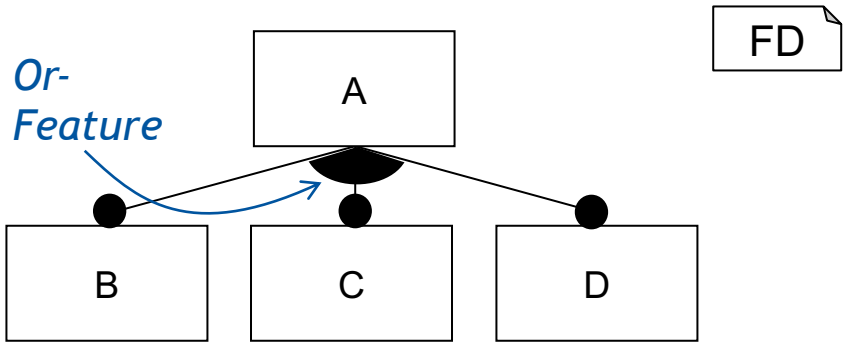
- Genau dann wenn Feature E gewählt wurde, kann Feature F gewählt werden
- Genau dann wenn Feature F gewählt wurde, muss auch Feature G gewählt werden

- Valide Kombinationen: {E} und {E,F,G}

Feature Diagramme: Xor, Or



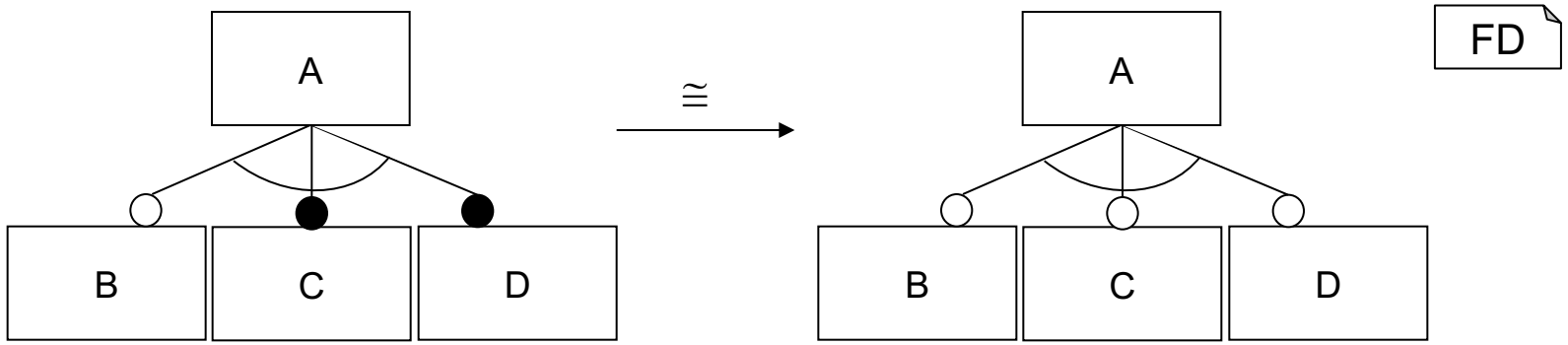
- Wenn Feature A gewählt wurde, muss genau ein Feature B oder C oder D gewählt werden („xor“), sonst keines
- Valide Kombinationen: {A,B} und {A, C} und {A,D}



- Wenn Feature A gewählt wurde, muss mindestens ein Subfeature gewählt werden, sonst keines
- Valide Kombinationen: {A,B} und … {A,B,C,D} (7)

Semantische Äquivalenz / Normalisierung von Feature-Diagrammen - 1

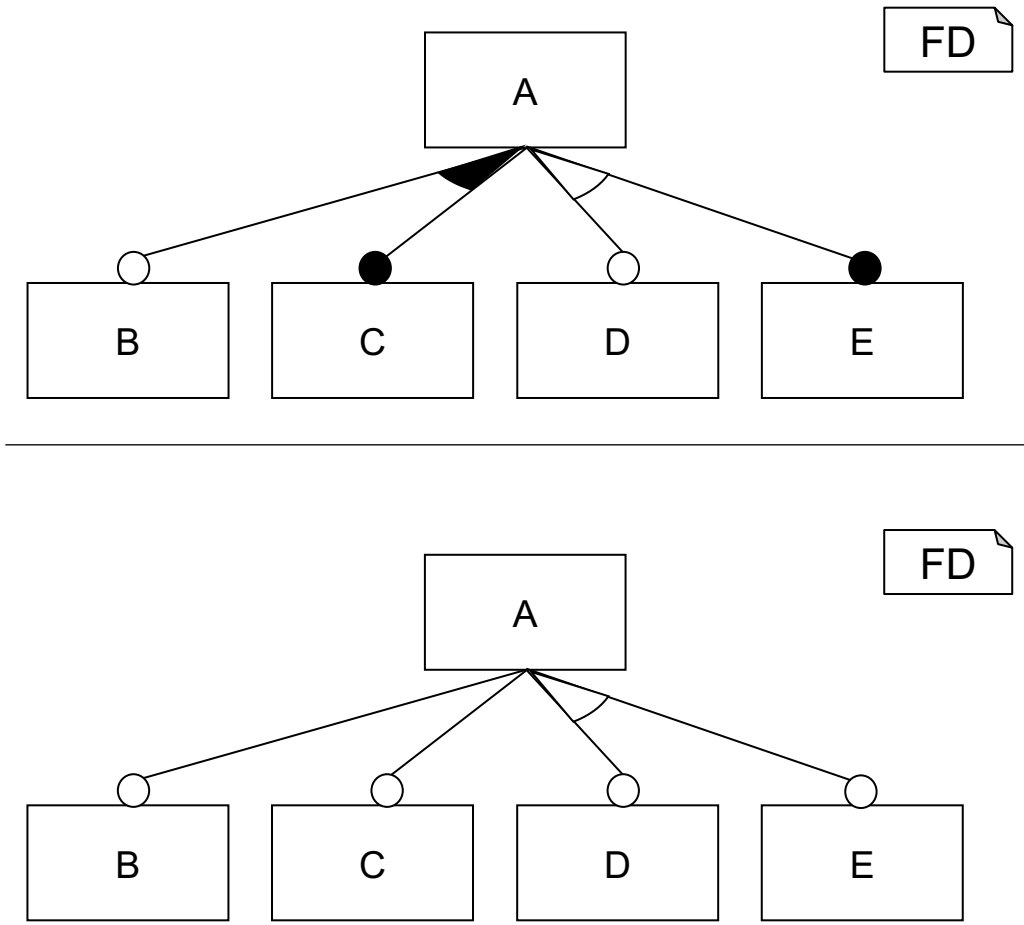
- Vereinfacht Lesbarkeit und Verständnis
- Sehr ähnlich zur Logik:



- Xor: Wenn nicht alle Zweige verpflichtend sind:
  - dann sind eigentlich auch alle optional

- Denn Auswahl von „Zweig zu B“ ohne Auswahl von B = keine Auswahl

Feature Diagramme: Kombinationen

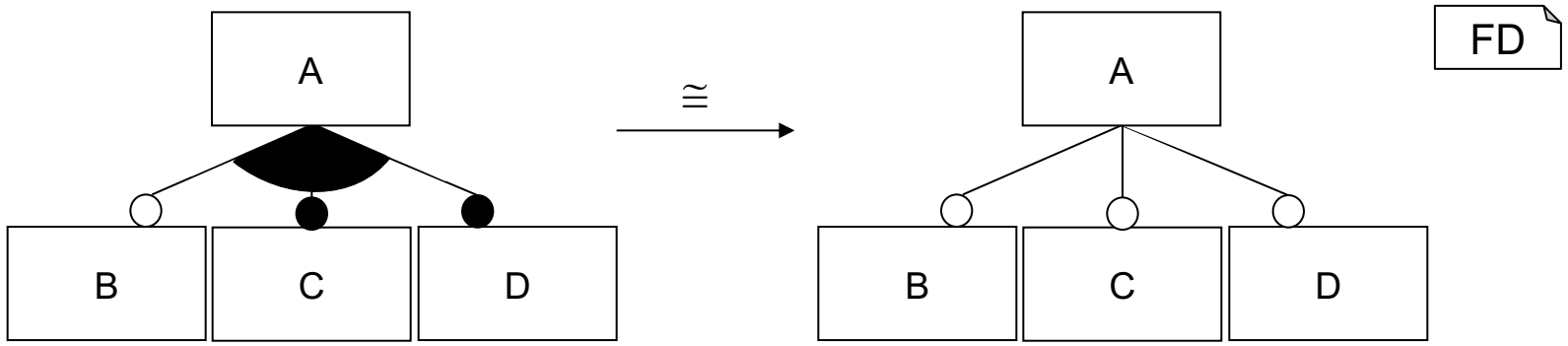


- Weitere Kombinationen möglich, aber redundant, da semantisch äquivalent

- Valide Kombinationen in beiden Fällen:
- {A},
- {A,B}, {A,C},
- {A,B,C},
- {A,D}, {A,B,D}, {A,C,D},
- {A,B,C,D},
- {A,E},
- {A,B,E}, {A,C,E} und {A,B,C,E}

Semantische Äquivalenz / Normalisierung von Feature-Diagrammen - 2

- Vereinfacht Lesbarkeit und Verständnis



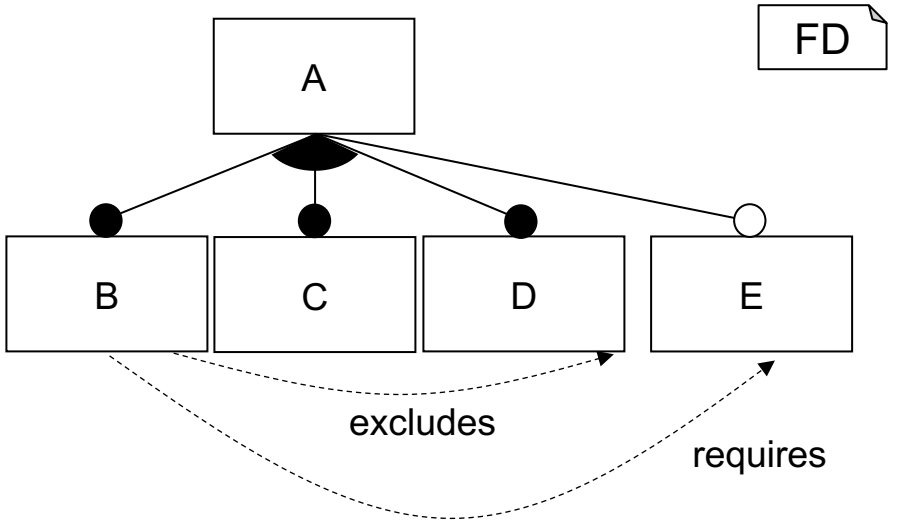
- Or: Wenn eines optional ist, können auch die anderen optional sein
  - Das „or“ kann durch „und“ mit „optional“ ersetzt werden

- Denn Auswahl von „Zweig zu B“ ohne Auswahl von B = keine Auswahl

Erweiterungen zu Featurediagrammen

Es existieren Erweiterungen zu Feature-Diagrammen

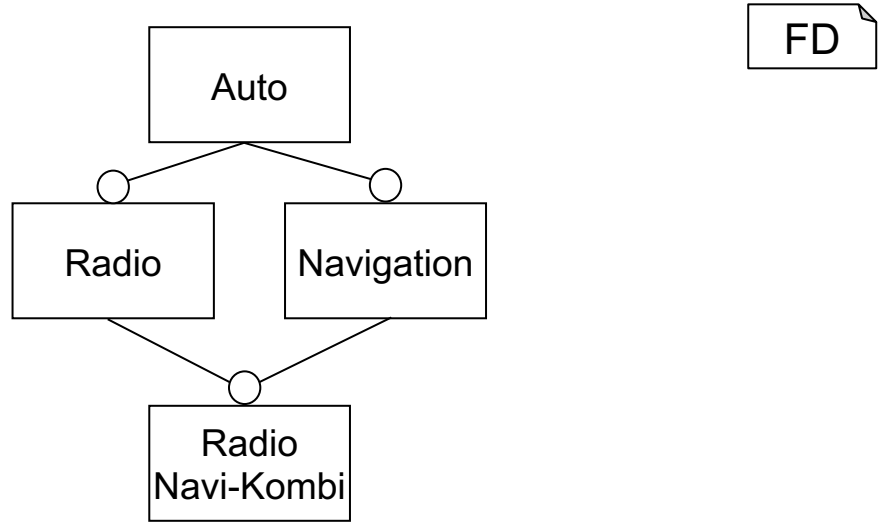
- Constraints, z.B.
  - B requires D
  - B excludes E



Erweiterungen

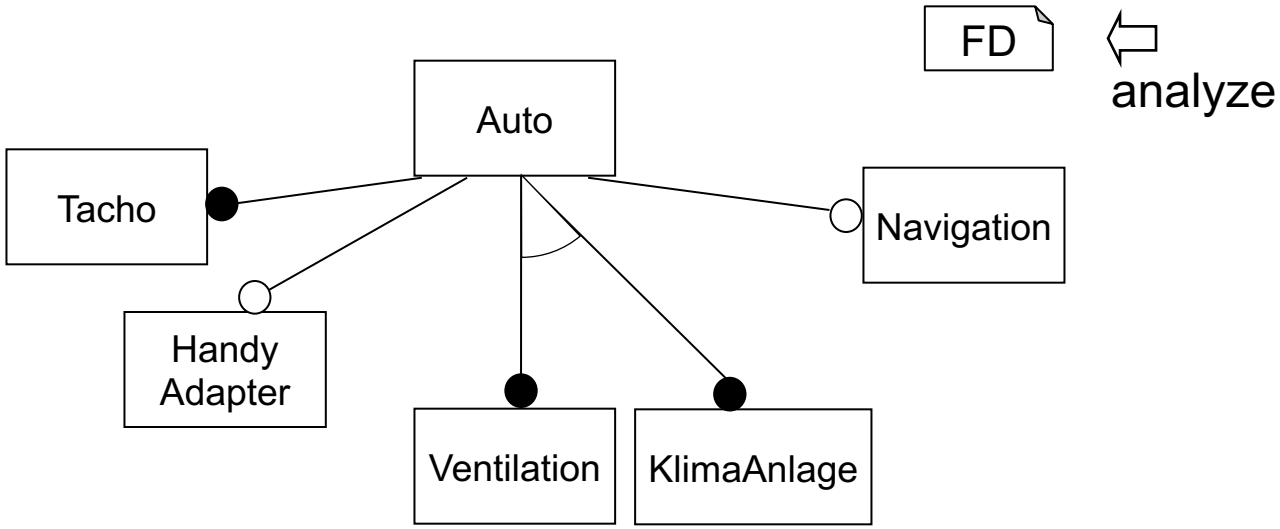
Es existieren Erweiterungen zu Feature-Diagrammen

- Feature-DAGs (Directed Acyclic Graph)



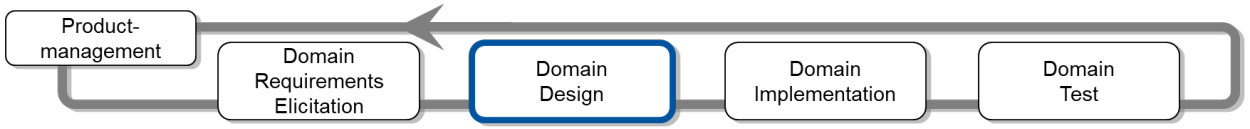
Beispiel: Konfigurationen

- Wieviele gültige Konfigurationen hat das Featuremodell?
- Zählen sie alle gültigen Konfigurationen auf
- Ist {Auto, Tacho, Navigation} eine gültige Konfiguration?



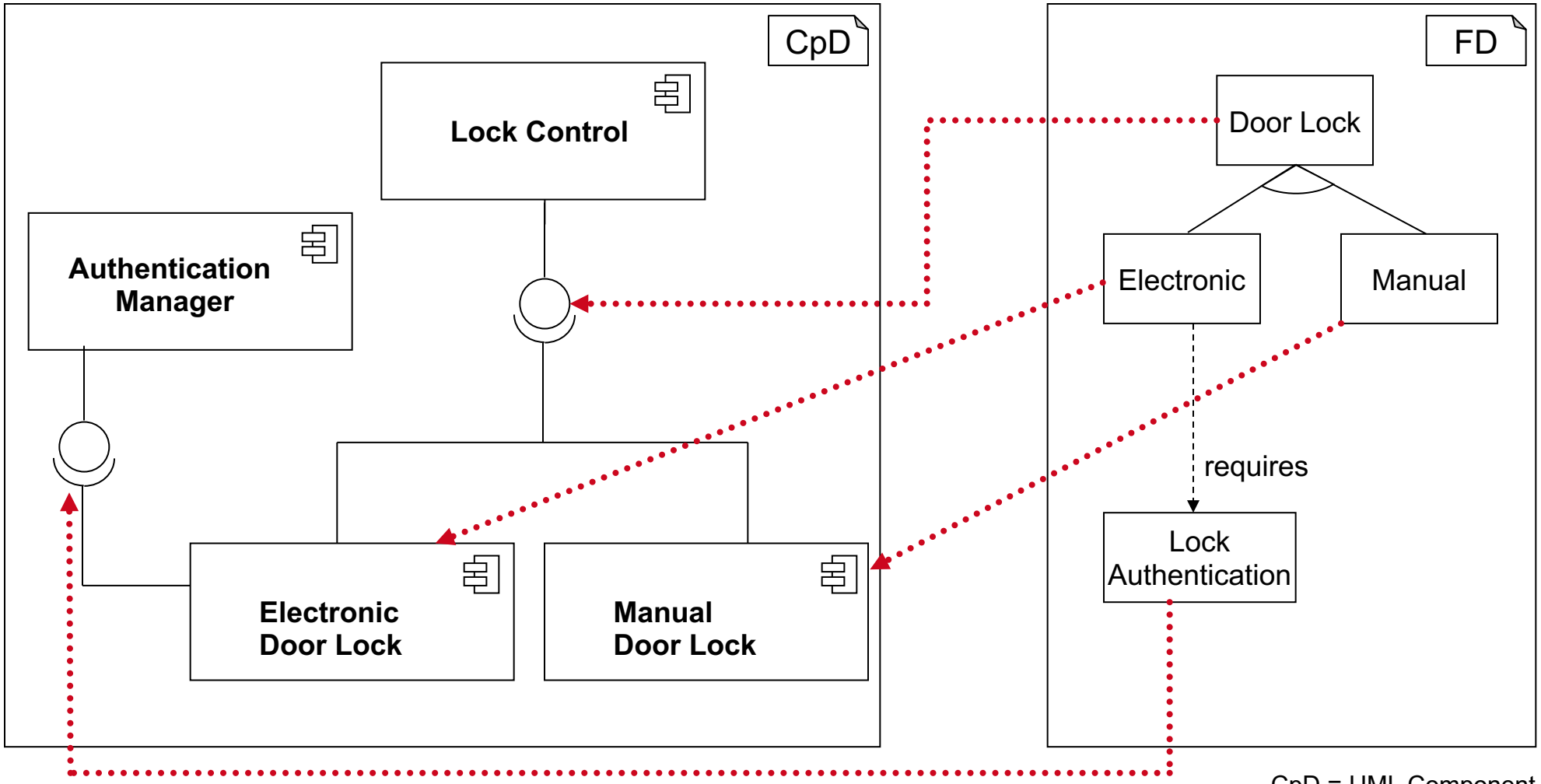
Domänen Engineering: Entwurf

- Entwurf einer Referenz-Architektur der Produktlinie
- Unterschiede zu einem Entwurf eines Einzelsystems
  - Variationspunkte werden modelliert
  - Flexibilität besonders wichtig
    (Komponenten-basierte Technologien vorteilhaft)
  - Entwurfsregeln für das Ableiten konkreter Produkte aus der Referenzarchitektur
  - Identifikation von
    - wiederverwendbaren Elementen
      (Entwicklung im Domänen Engineering)
    - vs. Applikationsspezifische Elemente
      (Entwicklung im Applikation Engineering)



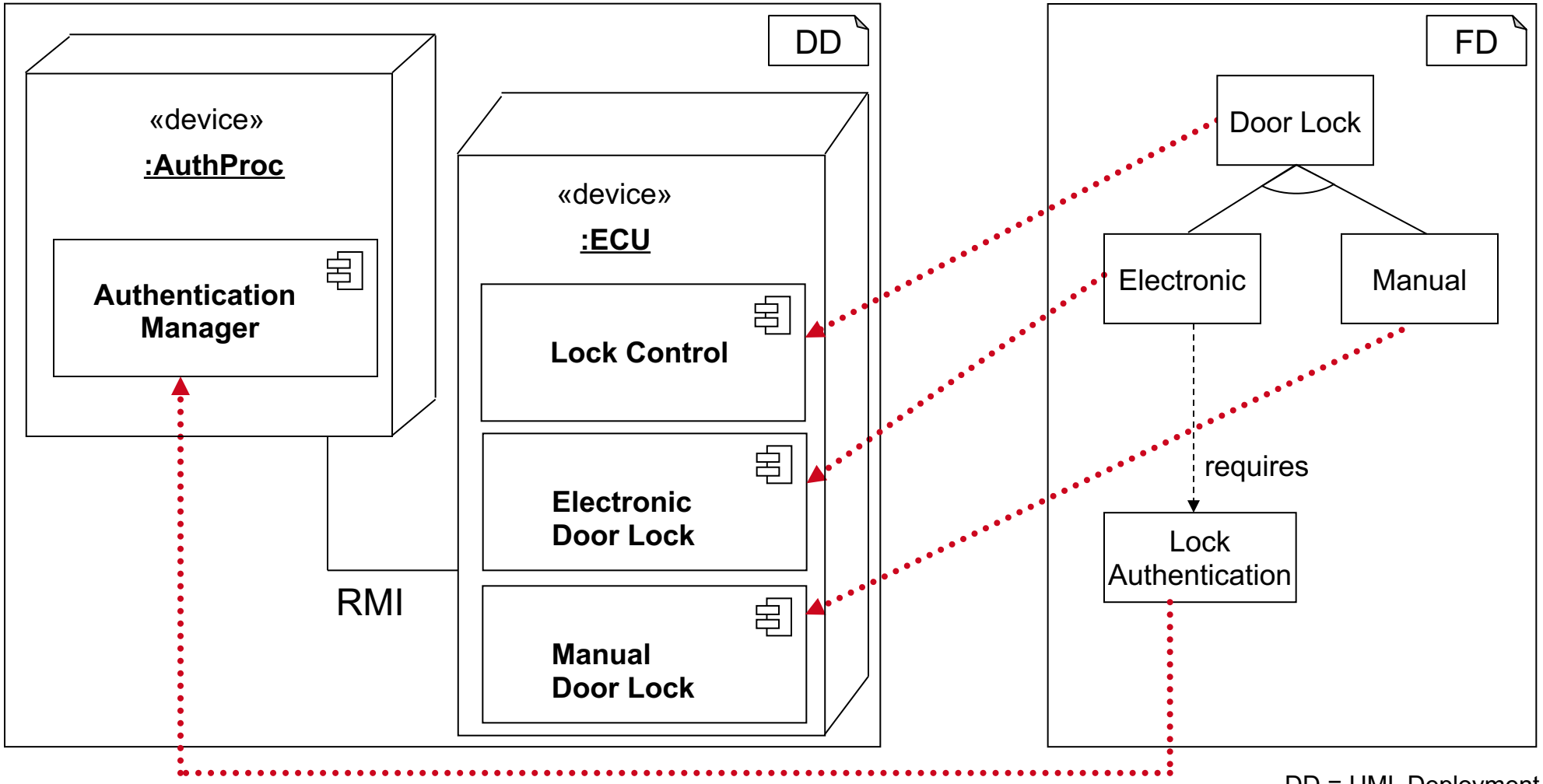
- Entwurfsdokumente
  - Dokumentation wie für ein Einzelsystem
  - Struktur- und Verhaltensmodelle des Design enthalten Bezüge zum Variabilitätsmodell

Dokumentation von Variabilität in Entwurfsdokumenten: Komponenten



CpD = UML Component Diagram

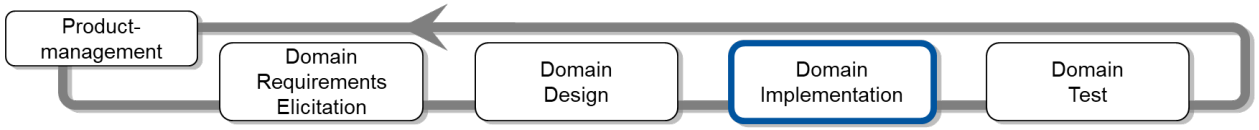
Dokumentation von Variabilität in Entwurfsdokumenten: Verteilung



DD = UML Deployment Diagram

Domänen Engineering: Realisierung

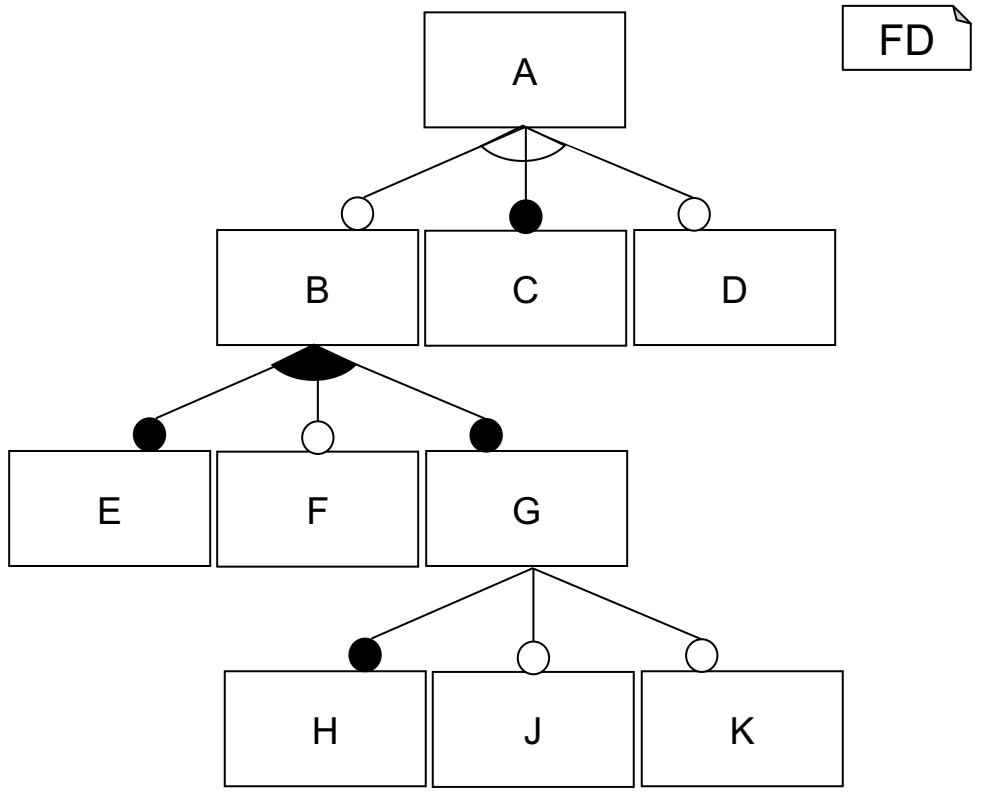
- Detaillierter Entwurf und Implementierung
- Schnittstellenentwurf und -dokumentation
  - Abstraktionsgrad wichtig
    - Zu detailliert = Zu wenig Anwendungen
    - Zu abstrakt = Wenig nützlich
  - Gute Dokumentation notwendig für Wiederverwendung
- Konfigurationsmanagement
  - Welche Komponente wird in welcher Version in welchen Produkten verwendet?



- Unterschiede zur Einzelsystemrealisierung
  - Ergebnis ist Einzelkomponente bzw. Framework, keine lauffähige Applikation
  - Wiederverwendung in verschiedenen Kontexten vorgesehen
    - Parametrisierung der Schnittstelle
    - Konfiguration ist integraler Bestandteil der Komponenten
    - Robustes Design

Domänen Engineering: Realisierung von Variabilität

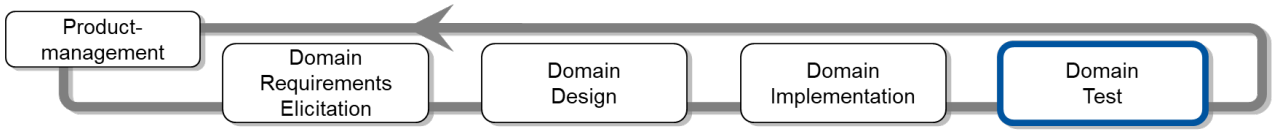
- Techniken zur Realisierung von Variabilität
  - Vor der Kompilierung
    - Codegenerierung (Parametrisierung z.B. durch domänenspezifische Sprachen)
    - Modellgetriebene Ansätze (Parametrisierung durch PIM to PSM Transformationen)
  - Zur Kompilierzeit
    - Pre-Compiler macros
  - Zur Linkzeit
    - Gradle/Makefiles
  - Während des Ladens
    - Konfigurationsdateien
  - Zur Laufzeit
    - Zentrale Registry
  - Im Code
    - Zusätzliche Attribute zur Konfiguration



- Feature-Diagramme werden hier für ggf. alle diese Zwecke eingesetzt!

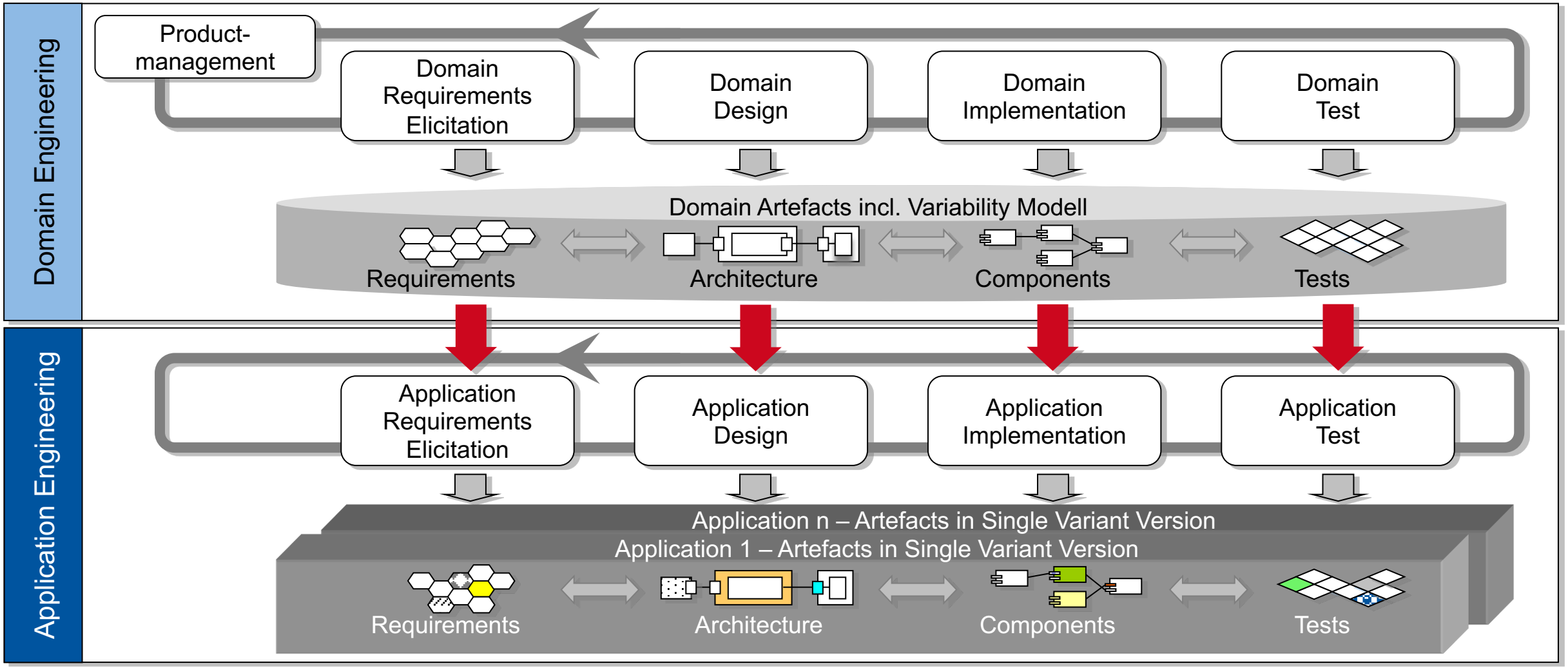
Domänen Engineering: Testen

- Unterschiede zum Einzelsystemtest
  - Testen von einzelnen parametrisierten Komponenten bzw. des Frameworks, nicht ein Gesamtsystem
- Teststrategien
  - Nachbilden der Umgebung
    - Umgebung im Test (teilweise) aufbauen durch Mock-Objekte
      (vgl. Kapitel über das Testen)
  - Sample Application Strategy
    - Erstellen von Beispielanwendungen
    - Testen auf Applikationsebene mit Fokus auf betrachteter Komponente



- Teststrategien (2)
  - Commonality and Reuse Strategy
    - Erstellung von generischen Testfällen
    - Auflösung der Variabilität der Testfälle im Applikations-Testen
    - Testen auf Applikationsebene
  - Parameterüberdeckung
- Eine geeignete Teststrategie ist eine Kombination aller Ansätze

Ableiten von Applikationen aus der Plattform [PBL05]



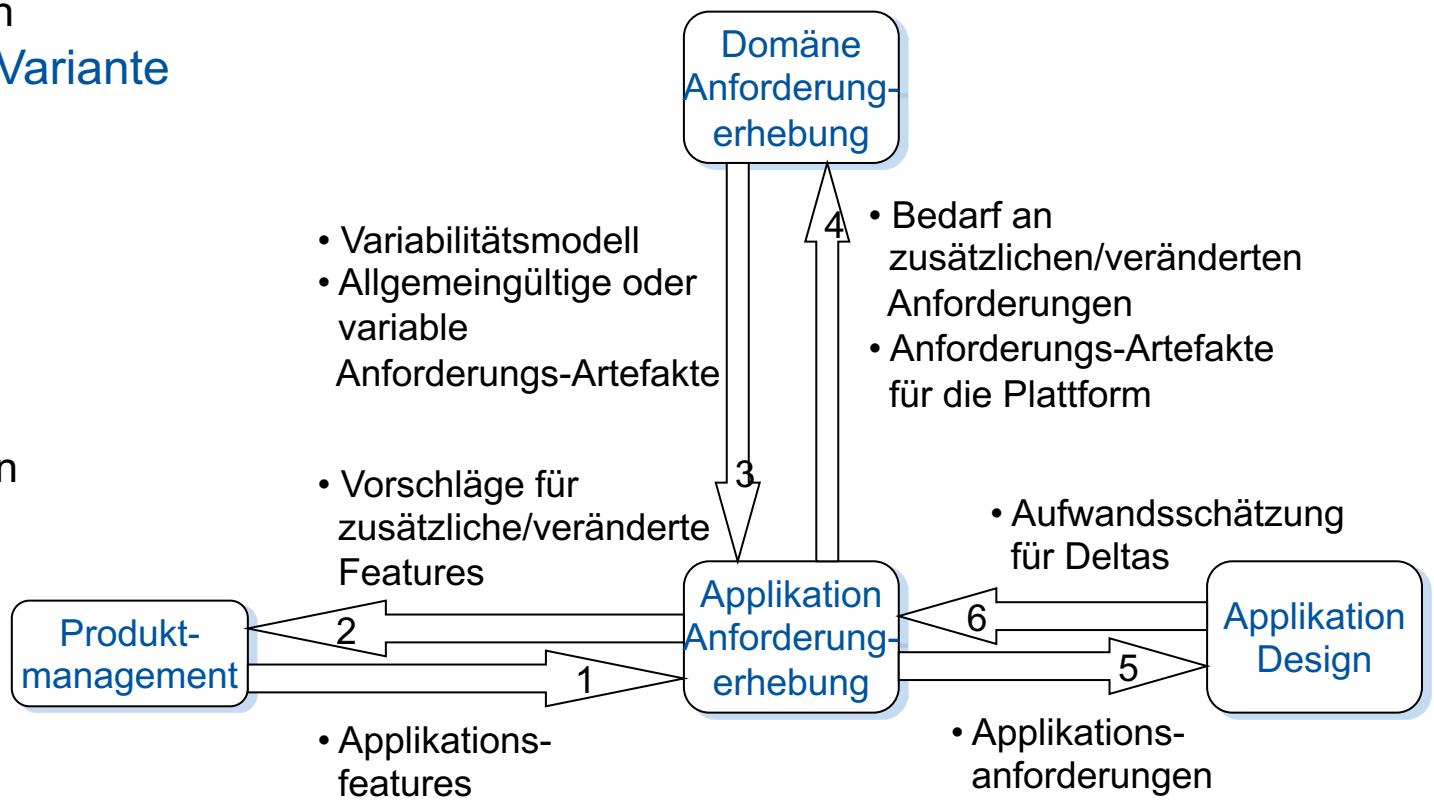
Applikations-Engineering: Anforderungen

- Anforderungsanalyse
  - Bestimmung der Unterschiede (Deltas) zwischen Anforderungen der Plattform und der Applikation
- Auswahl und Dokumentation der gewählten Variante aus dem Variabilitätsmodell

- Tracing zwischen Applikations- und Domain-Anforderungen

- Effekte auf die Plattform
  - Zusätzliche Freiheitsgrade/Parametrisierung von Anforderungen
  - Ziele
    - Weiterentwicklung der Plattform
    - Minimierung der Deltas

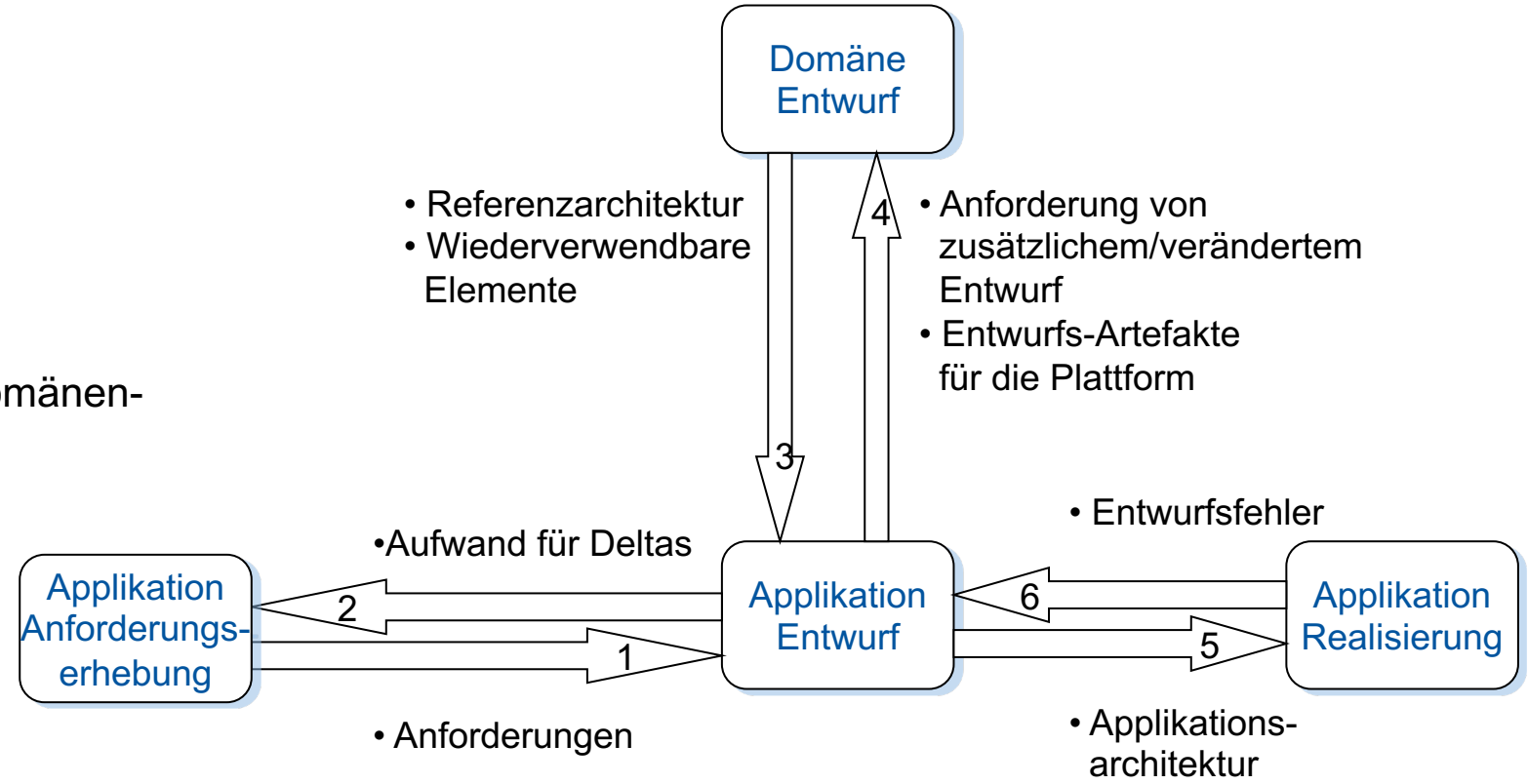
- Konsequenz: Domänen Engineering ist eine Querschnitts- keine Linienfunktion



Application Engineering: Entwurf

Aufgaben:

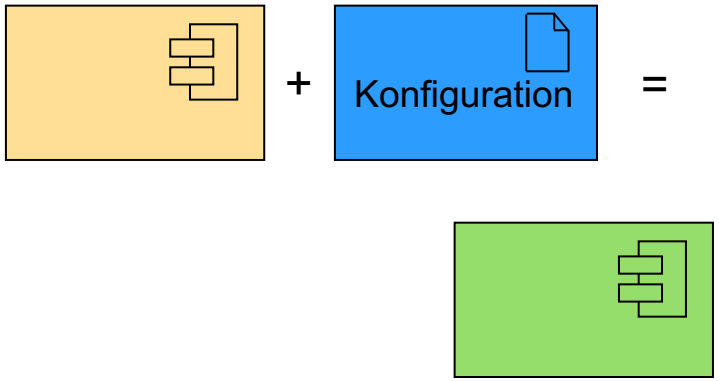
- Bestimmung von Aufwand und Kosten der Deltas
  - Neue Komponenten
  - Veränderungen der Architektur
  - Testaufwand
- Konkretisierung der Referenzarchitektur
  - Auflösung der Variabilität
    - Festlegung der verwendeten Varianten
  - Wiederverwendung von Elementen des Domänen-Entwurfs
    - Übernahme der Diagramme
    - Sichten auf die Referenzarchitektur
  - Realisierung der Deltas aus der Anforderungsanalyse

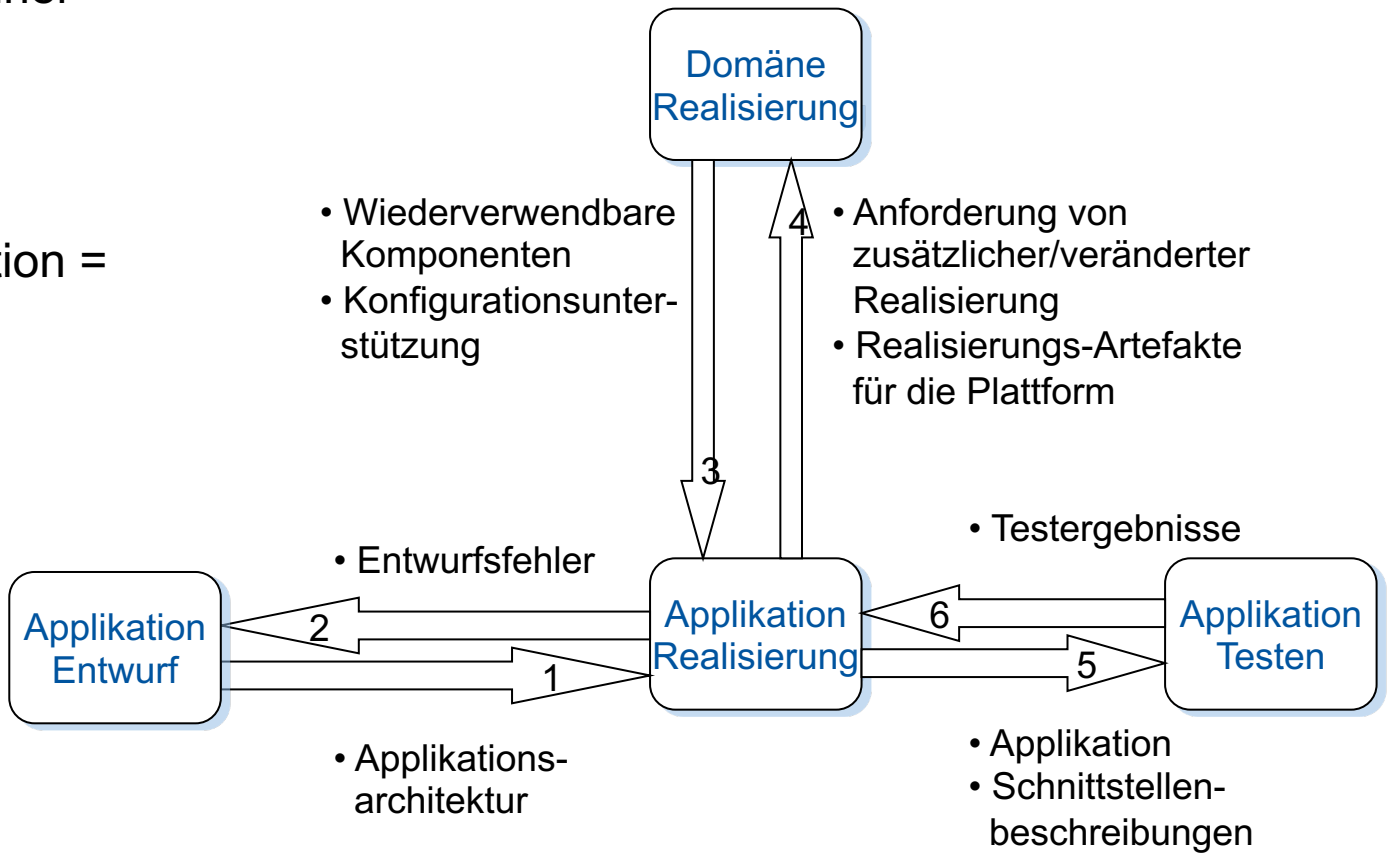


Application Engineering: Realisierung

- Auflösung der Variabilität =
  Überführung einer Domänen-Komponente in einer Applikationskomponente
  - Auswahl der passenden Komponenten
  - Erstellen der Komponentenkonfiguration

- Passende Domänen-Komponente + Konfiguration = Applikations-Komponente





Application Engineering: Test

- Tests vor allem für die Deltas zur Plattform
  - Testet ob Anpassungen erfolgreich
  - Relativ wenig Testaufwand, wenn die Plattform gut getestet ist
- Optionen: Statt Wiederverwendung konkreter Tests
  - Nutzung der Testinfrastruktur zur effizienten Testdefinition
  - Nutzung eines Testgenerators für die Produktlinie
- Fokus auf Integrationstests
  - Testen der integrierten Komponenten
  - Testet richtige Verschaltung und Parametrisierung
- denn: Unit-Tests bereits im Domänen-Engineering
  - Zentrale Sicherung der Qualität für wieder verwendbare Komponenten

Einführung von Software-Produktlinien: Prozessmodelle zur Wiederverwendung

- Prozessmodelle: wann die Investition zur Etablierung einer Wiederverwendungsstrategie erfolgen kann:
- Proaktives Modell
  - Wiederverwendbare Komponenten werden im Voraus entwickelt
  - Produkte basieren auf diesen Komponenten
- Reaktives Modell
  - Produkte werden zunächst unabhängig entwickelt
  - Bei Gelegenheit werden wieder verwendbare Komponenten erstellt und ausgetauscht
- Extraktives Modell
  - Verwendung einer oder mehrerer Produkte als Ausgangspunkt der Produktlinie
- Mögliche Organisationsformen:
- Zentralisierte Organisationseinheit
  - beschäftigt sich explizit mit wiederverwendbaren Komponenten
    - Entwicklung
    - Aufbereitung
    - Verwaltung
- Verteiltes Organisation
  - Wiederverwendbare Komponenten werden jeweils in den Projekten der Produktlinie entwickelt und ausgetauscht

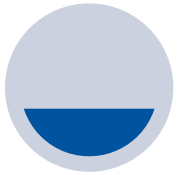
Einführung von Software-Produktlinien: Organisationsmodelle (2/2)

- Mischform, Beispiel: HP/Owen [MO07]
  - 1 Plattformteam
    - Pflegt und entwickelt die Plattform
  - 4 Produktteams
    - mit jeweils verschiedenen Produkten
    - unterschiedlichen Kernkompetenzen, die zur Weiterentwicklung der Plattform beitragen
  - Business Steuerungsteam
    - Kommunikationsschnittstelle Management-Technisches Team
    - keine abschließende Entscheidungsgewalt
  - Technische Steuerungsteam (inkl. Chefarchitekt)
    - Erhält Konsistenz der Architektur
    - Koordiniert Weiterentwicklung

Risikofaktoren einer SPL

- Anwendungsbereich
  - Zu klein: Zu wenig Produkte um Investitionen zu rechtfertigen
  - Zu groß: Aufwand individuelle Produkte aus den Gemeinsamkeiten zu entwickeln ist zu groß
- Fehlendes Führungsteam/Architekt zur Einhaltung des Ansatzes
  - Kompetenz und Einfluss erforderlich
- Fehlende Unterstützung des Managements
  - Klare Geschäftsziele zur Adaption der SPL notwendig
  - Unterstützung des Ansatzes bei ersten Schwierigkeiten
- Unzureichende Ausbildung des Entwicklungsteams
  - SPL schwieriger als Einzelsystementwurf
  - Gute Planung Architektur und wieder verwendbarer Einheiten
  - Unterstützung des Ansatzes notwendig

Was haben wir gelernt?

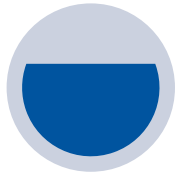


SPL

… ermöglichen es aus einer Plattform individuelle Produkte abzuleiten

… sparen Zeit und Kosten ca. ab 3. System

… reduzieren Wartungsaufwand, time-to-market und Komplexität, erhöht Qualität, erleichtert Kostenschätzung

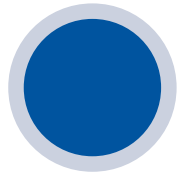


Prozesse

Domain Engineering & Application Engineering

Schritte & Artefakte:

- Produktmanagement
- Anforderungen: Feature Modelle
- Design: Architektur
- Entwicklung: Komponenten
- Tests



Einführung

Prozessmodelle zur Wiederverwendung:

- Proaktiv, reaktiv, extraktiv

Organisationsmodelle

- Verteilt, zentralisiert, Mischformen

Vorlesung Softwaretechnik

14. Ausblick

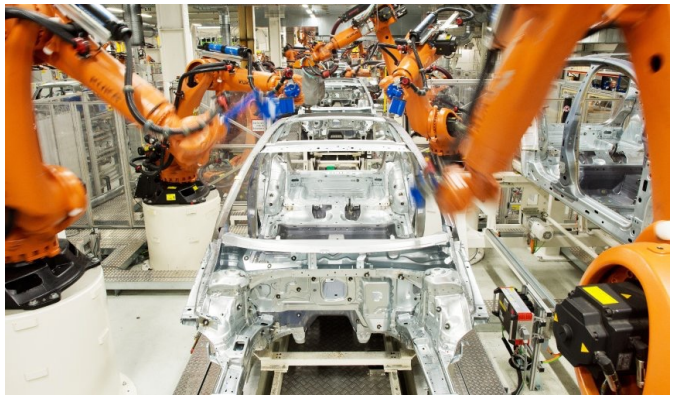
Prof. Bernhard Rumpe
Software Engineering
RWTH Aachen

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

SE Software Engineering | RWTH AACHEN UNIVERSITY

Zukunft?

- Automatisierung und Digitalisierung beschleunigen die Veränderung der Arbeitswelt
- Neue technische Optionen verändern unsere private und gesellschaftliche Lebenswelt
- Herausforderungen wie Klima und Gesundheit erfordern neue Innovationen



- Bedarf in Generalist:innen & Spezialist:innen
  - Daten A.
  - S. in KI und Maschinellem Lernen
  - Big Data S.
  - S. in digitaler Strategieentwicklung
  - S. in Prozessautomatisierung
  - Geschäftsentwicklungs-Experten
  - S. in digitaler Transformation
  - A. für Informationssicherheit
  - Software und Anwendungsentwickler
  - S. für das Internet der Dinge
- Erwartete persönliche Fähigkeiten
  - Kritisches Denken
  - Analysefähigkeiten
  - Problemlösungskompetenz
  - Selbst-Management
  - Aktives Lernen
  - Resilienz
  - Stresstoleranz
  - Flexibilität

Quelle (zum Teil): WEF Future of Jobs (Weltwirtschaftsforum), 2020

Softwaretechnik

14. Ausblick
14.1. Zusammenfassung: Warum, was, wie und wozu Softwaretechnik?

Prof. Bernhard Rumpe
Software Engineering
RWTH Aachen

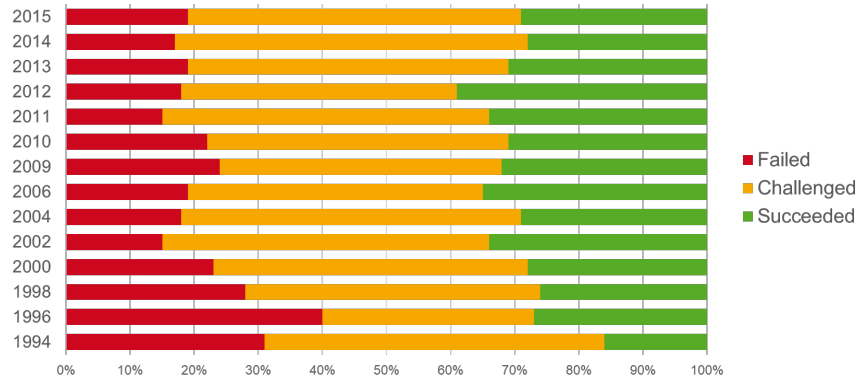
http://www.se-rwth.de/
@SE_RWTH

SE Software Engineering | RWTH AACHEN UNIVERSITY

Warum?

Wiederholung

- Größe der Systeme wächst
- Komplexität der Systeme steigt
- Heterogenität nimmt zu
- Software ist lange in Betrieb



Quelle: CHAOS Report, 1994-2015, Standish Group International, Inc.

- Softwareprojekte scheitern, kosten mehr als geplant, werden später fertig
- Software ist nicht wiederverwendbar, nur schwer erweiterbar
- Fehler in Sicherheitskritischer Software verursachen große Schäden

Was? Software Engineering

Wiederholung

> „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 drei Aspekte:
  - Kosten
  - Termine
  - Qualität
    (Korrektheit, Zuverlässigkeit, Performanz, Sicherheit, Nutzbarkeit, Verständlichkeit, Weiterentwickelbarkeit, Anpassbarkeit, Wartbarkeit)

Was? Aufgabenstellungen der Softwaretechnik

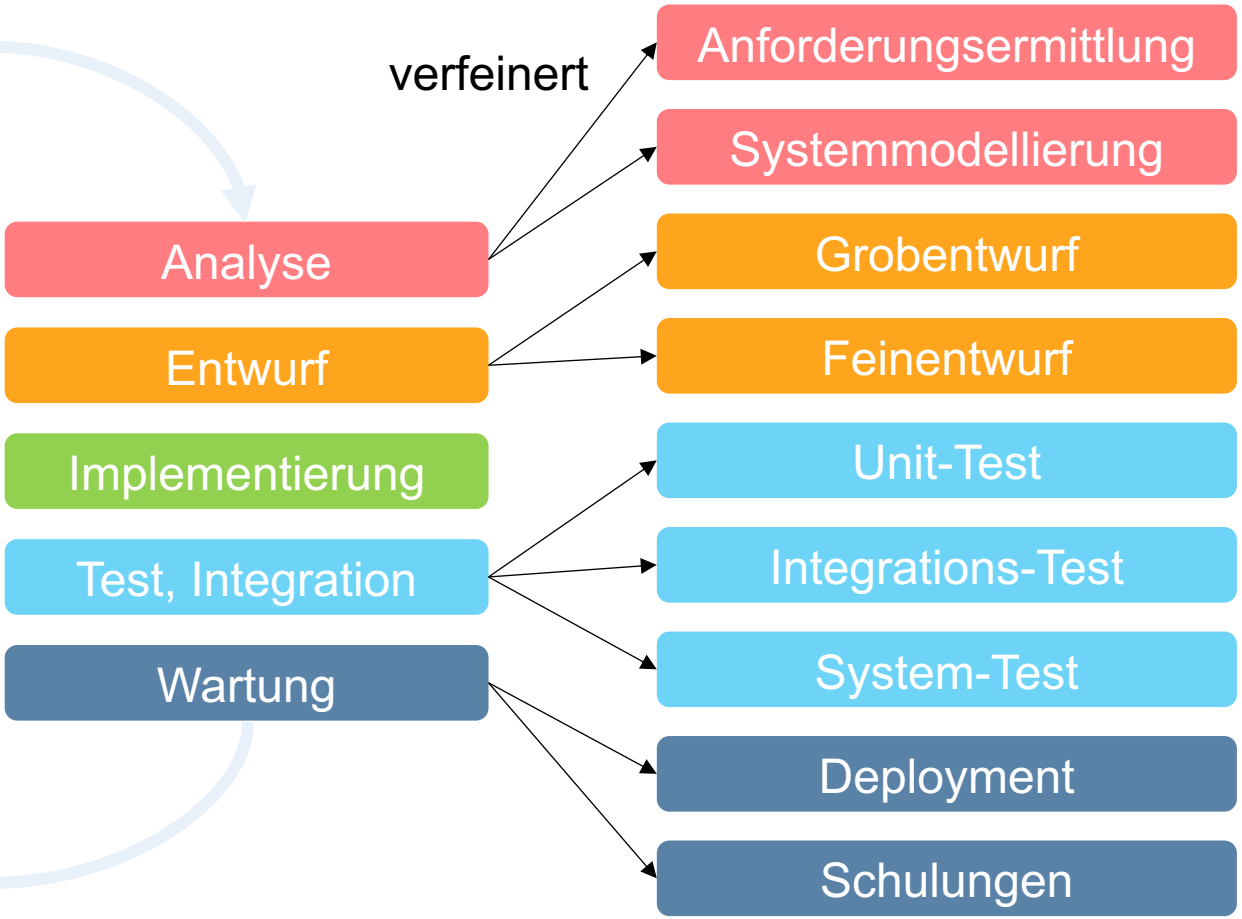
- Softwareentwicklung ist mehr als nur Programmieren!

- Dazu gehören:
  - Management großer und komplexer Projekte → 2. Vorgehensmodelle
  - Schätzung von Terminen und Kosten → (nicht behandelt)
  - Erfassung von Kunden- und Marktanforderungen → 3. Anforderungsanalyse, 4. Systemanalyse
  - Änderungsmanagement → 2. Vorgehensmodelle, 11. Werkzeuge
  - Sicherstellung eines hohen Qualitätsniveaus → 2-13 (durchgängig)
  - Wartung und Weiterentwicklung von Altsystemen → 4. Systemanalyse, 6. Softwareentwurf, 12. Komponenten
  - Guter Programmierstil → 8. Implementierung
  - Entwicklungswerkzeuge → 11. Werkzeuge
  - Prinzipien wie Abstraktion, Strukturierung, Hierarchisierung und Modularisierung → 3-13

Softwaretechnik beinhaltet viele weitere Themen und bietet sehr viel mehr vertiefendes Wissen!

Zeitlicher Verlauf; Aktivitäten; Phasen; Iterationen

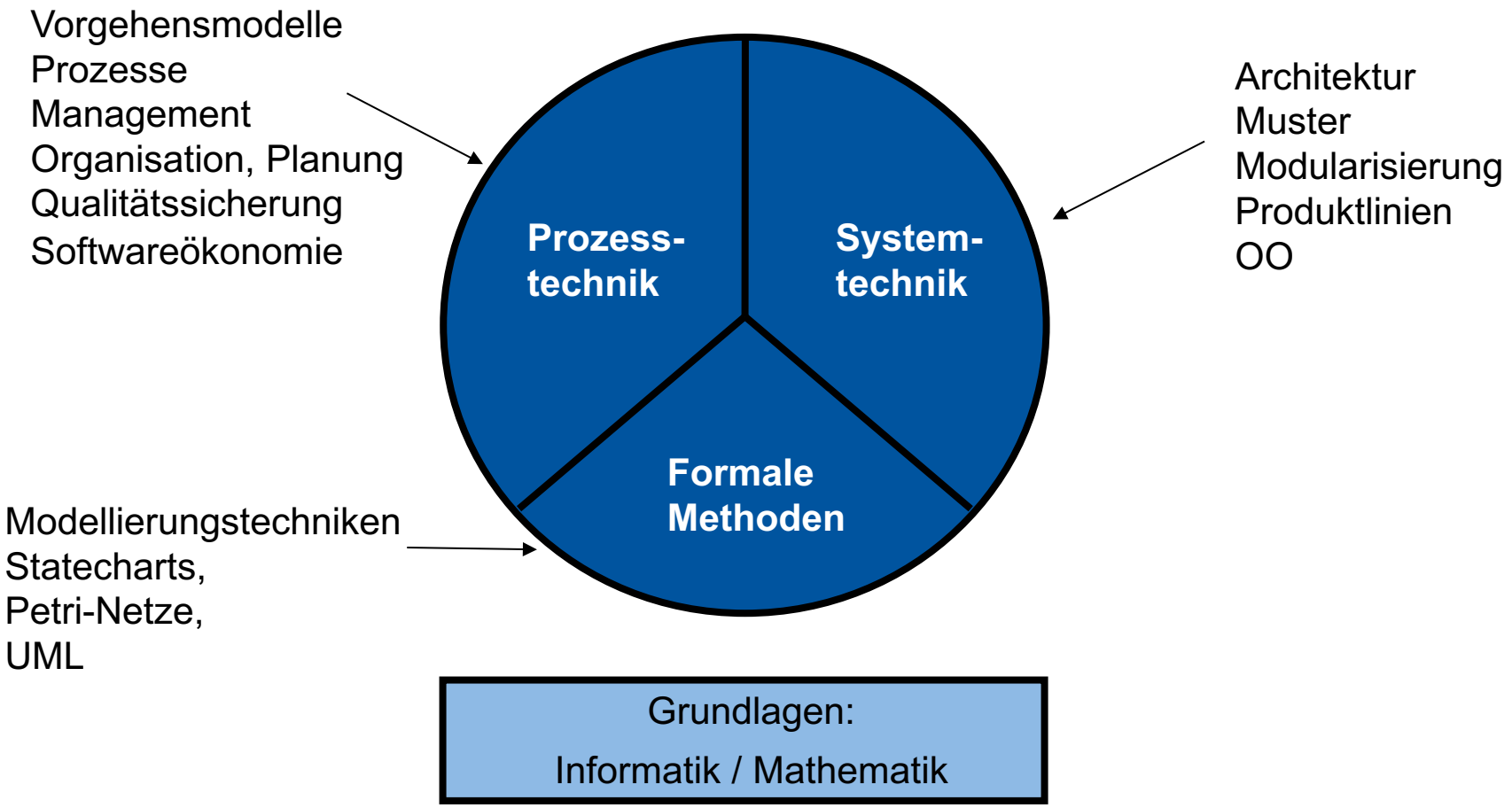
Wiederholung



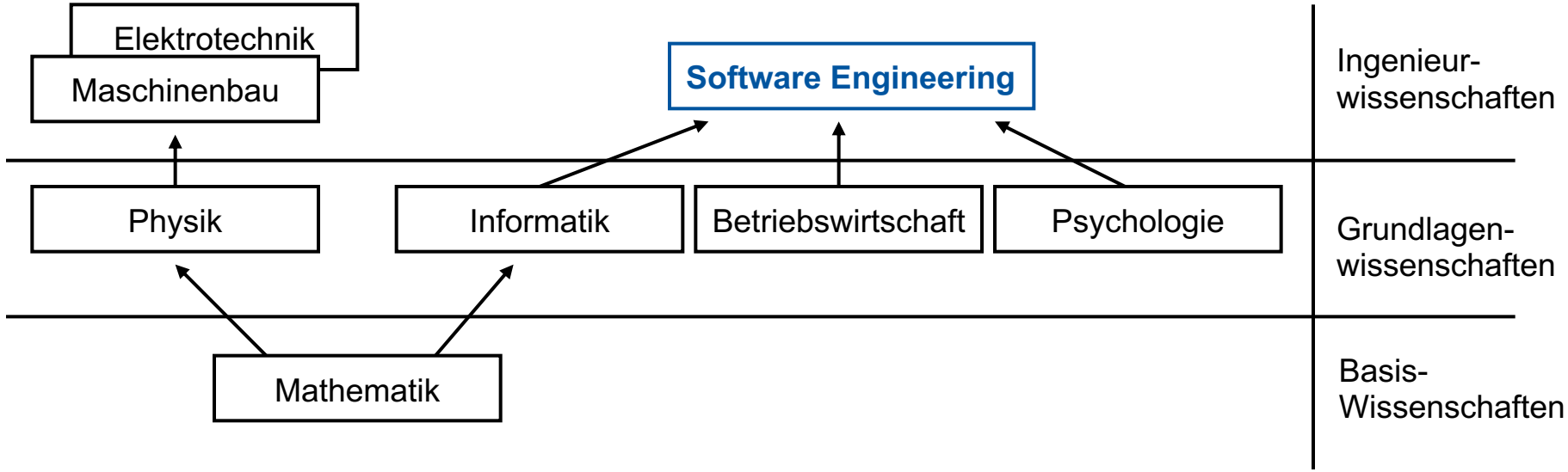
Parallel dazu

- Projekt-Management
- Konfigurations-Management
- Varianten/Versions-Management
- Änderungs-Management
- Tool-Management

Was? Gliederung des Software Engineering



Software Engineering vs. Informatik

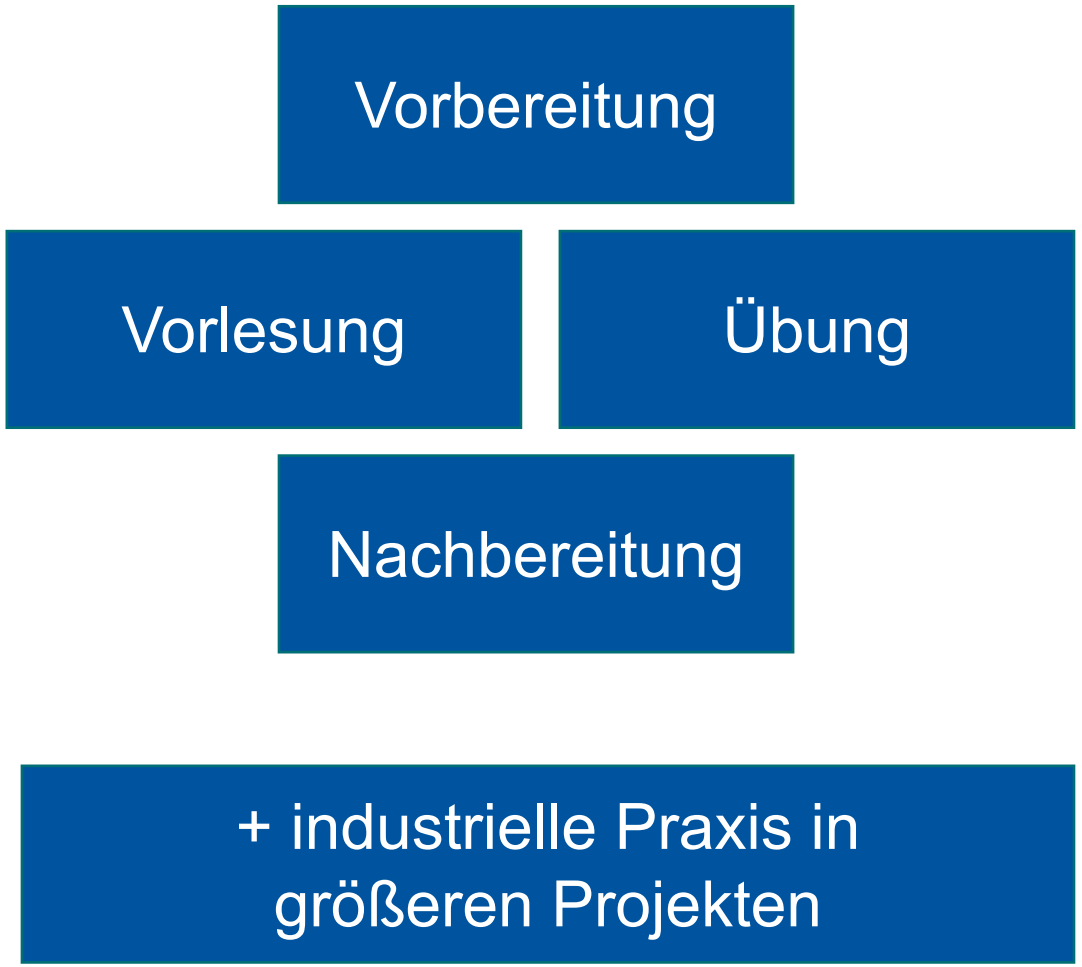


- Software Engineering ist der ingenieurwissenschaftliche Teil der Informatik (analog zur Beziehung Maschinenbau und Physik)

Wie? Überblick

Wiederholung

- Motivation & Organisatorisches
- **Vorgehensmodelle**
- **Analyse**
  - Anforderungen | Modellierung
  - System | Modellierung | Muster
- **Softwareentwurf (Design)**
  - Systementwurf | Oberflächen | Muster
- **Implementierung & Generative Entwicklung**
  - Implementierung | Generative SWT
  - Werkzeuge
- **Test**
  - Qualität in der SWT und Testen im Speziellen
- **Wiederverwendung & Variabilität**
  - Komponenten und Wiederverwendung
  - Softwareproduktlinien und Variabilität
- Ausblick



Softwaretechnik

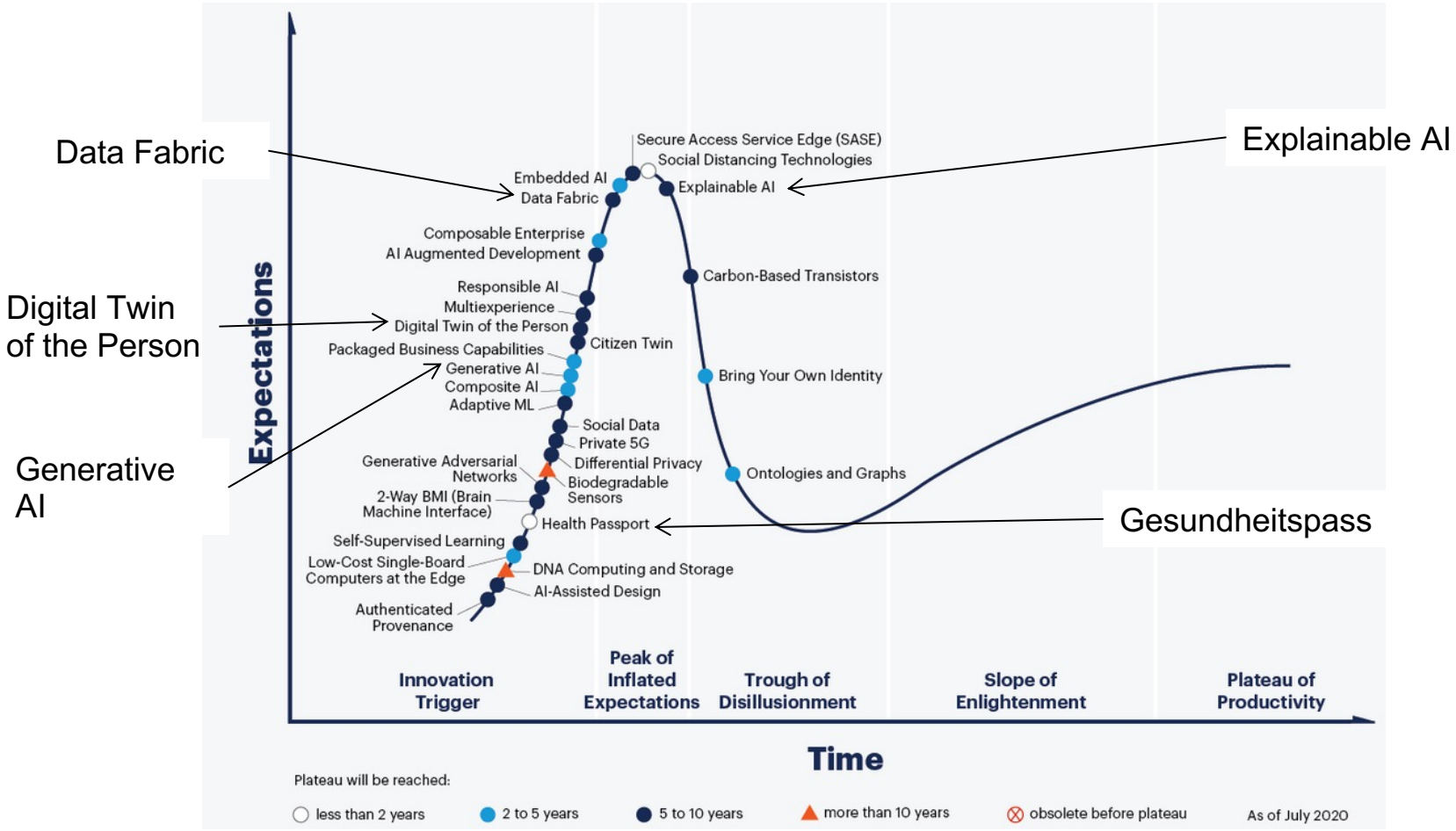
14. Ausblick
14.2. Herausforderungen der Softwaretechnik

Prof. Bernhard Rumpe
Software Engineering
RWTH Aachen

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


BERLIN POLICY JOURNAL
The AI Race
Künstliche Intelligenz in der Medizin
Harvard Business Review
When AI Becomes an Everyday T
Künstliche Intelligenz
Die Revolution der Roboter
Science
AAAS
Artificial Intelligence in Agriculture
Volume 2 | June 2019
Wissenschaftsjahr 2019
KÜNSTLICHE INTELLIGENZ
SPECIAL ISSUE
ARTIFICIAL INTELLIGENCE
TRADERFOX
aktien
02/2018
Künstliche INTELLIGENZ
TREND REPORT
Wachstum durch KI
JOURNAL OF ACCOUNTANCY
GEO
Bald sind Maschinen intelligenter als wir. Werden sie ...
Freund oder Feind?
Blitz-Rekord
Medizin
Abheben
Leben im Dorf
NATURE.COM/NATURE
2 February 2017
Vol. 542, No. 7639
nature
ALL SYSTEMS GO
NEUROLOGIE
TRONOMIE
gefährlich ist
Hypernova?
KÜNSTLI
INTELLIGEN
Was uns schon bald erwartet
Artificial Intelligence and Law
KÜNSTLICHE INTELLIGENZ IN DER LOGISTIK
Begriffe, Anwendungen und Perspektiven
ssi-schaefer.com
Whitepaper
OKT. 2018
AI

Gartner Hype Cycle: Emerging Technologies, 2020



Quelle: gartner.com/smarterWithGartner, 2020

Herausforderungen an die Softwaretechnik: Anwendungsgebiete

- Innovative Anwendungsgebiete mit hohem Software-Bedarf:
  - Integriertes Verkehrsmanagement
  - Sicherheit der Menschen (Unfälle)
  - Soziale Netzwerke
  - Energie-Reduktion
  - Logistik
  - Produktion: Industrie 5.0
  - Health, Wellness-Monitoring
  - Vernetzung der Welt
    - Home, Car, Shopping, Work, Freizeit
    - Telematik, Galileo, Mobile Devices
    - Smart & Intelligent, IoT
  - Simulation der Welt
    - Verkehrssysteme, Klimasysteme, Biologische Systeme
    - Wirtschaftliche und Soziale Systeme
  - Digitalisierung und Verstehen der Welt:
    - Daten, deren Modelle und Meta-Modelle
    - Machine Learning, Intelligenz

Herausforderungen an die Softwaretechnik: Beispiel Autonomes Fahren

Automobile Zukunft Die fünf Stufen des autonomen Fahrens

1

assistiert
Fahrassistenz-systeme wie Tempomat oder Kollisionswarner sorgen für mehr Sicherheit und Komfort. **Der Fahrer wird unterstützt, muss das System jedoch ständig überwachen;** bereits weit verbreitet.

2

teilauto-matisiert
Das System hält in bestimmten Situationen die Spur, beschleunigt und bremst. **Der Fahrer muss ständig aufmerksam sein.** Er darf die Hand nur kurz vom Lenkrad nehmen; z.T. schon verfügbar.

3

hoch auto-matisiert
Das Auto kann in bestimmten Situationen selbstständig fahren, z.B. in einem Stau. **Der Fahrer darf sich für längere Zeit vom Verkehrsgeschehen abwenden.** Er muss trotzdem jederzeit bereit sein, wieder die Kontrolle zu übernehmen. Erste Fahrzeuge sind angekündigt.

4

voll auto-matisiert
Kein Fahrer bei vorher definierten Bedingungen erforderlich. Das System kann alle Situationen allein bewältigen.

5

DER SPIEGEL

fahrerlos
Ein Fahrer ist überflüssig. Das System fährt autonom unter allen Bedingungen. Wird derzeit nur für Roboshuttles entwickelt, noch nicht für Privat-Pkw.

Herausforderungen an die Softwaretechnik: Beispiel Umsetzung von Recht in IT

Die ideale Welt aus Sicht...

...der Juristen*

- Recht führt, Technik folgt. Der Phantasie des Gesetzgebers sind keine Grenzen gesetzt.
- Recht ist abstrakt und generell, Anwendung auf den konkreten Fall erfolgt mittels Auslegung.
- Recht regelt nur das Wesentliche. Der Rest ist Sache der vollziehenden Verwaltung.
- Ermessensspielräume ermöglichen Einzelfallgerechtigkeit.
- Gesetzgebung folgt politischen Prioritäten und ist zeitlich flexibel, Umsetzung erfolgt möglichst schnell.

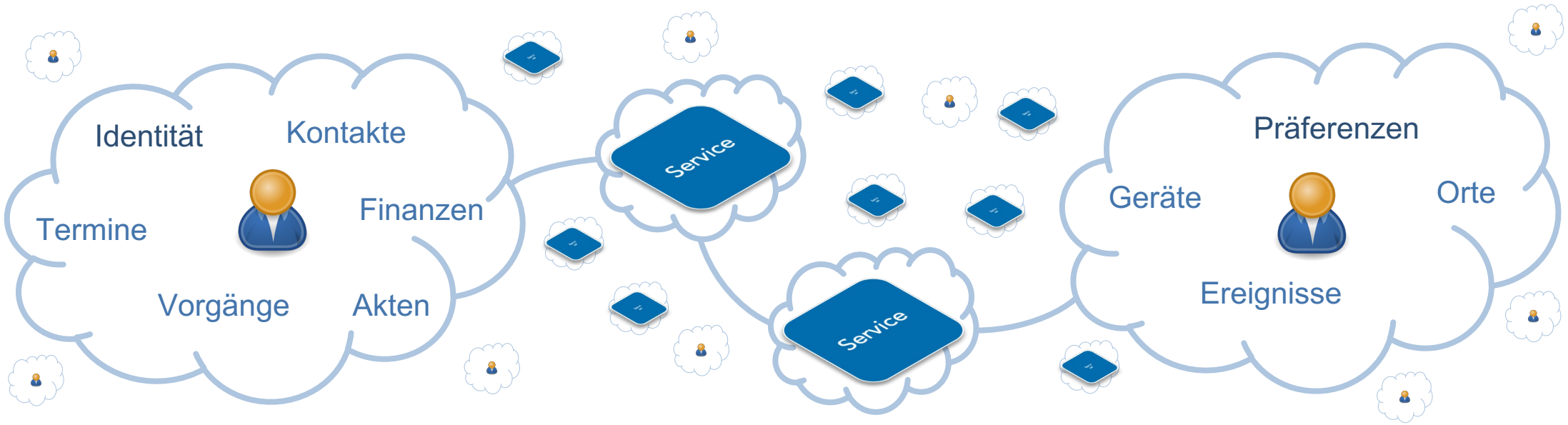
...der Techniker*

- Vor der Regelung wurde die technische Umsetzbarkeit geklärt.
- Recht ist so konkret formuliert, dass es unmittelbar in Technik übersetzt werden kann.
- Alles, was umgesetzt werden muss, ist auch rechtlich geregelt.
- Nur gebundene „wenn-dann-Entscheidungen".
- Umsetzungszeitpunkt folgt Releaseplänen. Recht ist zu einem von der Umsetzung bestimmten Zeitpunkt zuvor fertig, und darauf kann man sich auch verlassen.

(aus der Welt von Recht und Politik)

Vernetzung der Welt | Internet of Things

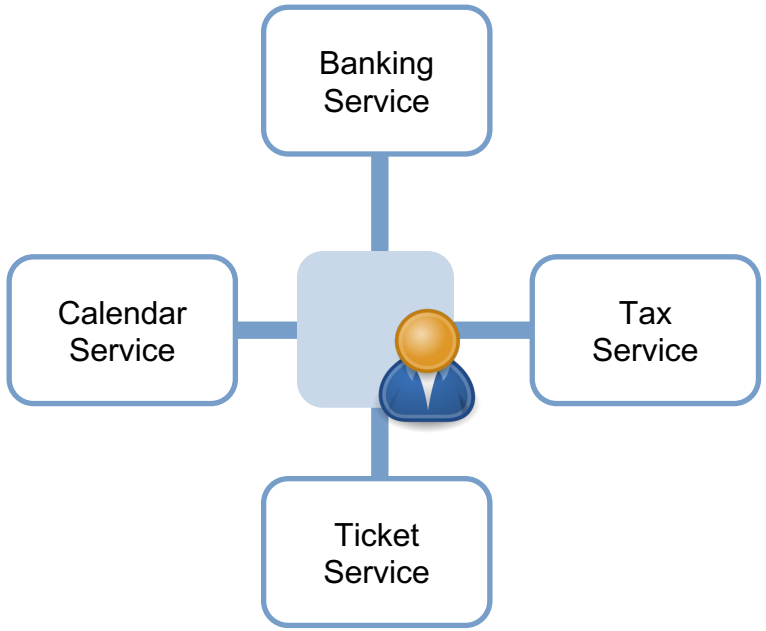
- Vernetzung von Informationen, Diensten, Geräten und Menschen
- Anwendungen laufen zunehmend in globalem, heterogenen Kontext

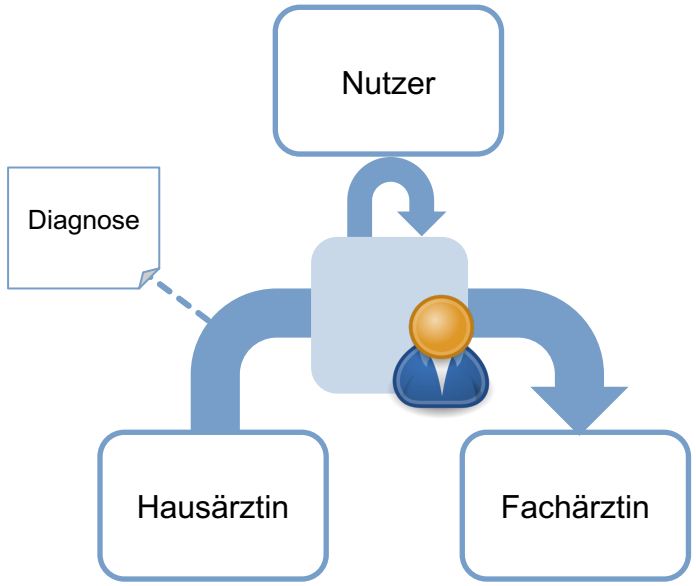


- Wie können solche Netzwerke autonom unser Leben vereinfachen?
- Wie können wir sie vertrauenswürdig machen?

Vernetzung der Welt | Cloud

- Persönlicher Verwaltungsdienst in der Cloud





- Verwaltet Vorgänge und Daten einer Person autonom
- Kann modell-basiert agil erweitert und angepasst werden

Vernetzung der Welt | Notwendigkeiten

- Komfortable, für jedermann einfach bedienbare Human-Machine-Interfaces
- Sichere Systeme
  - Datensicherheit
  - Authentizität der Handelnden, ...
- Effiziente Entwicklung
- Agile Anpassung des Systems
  - Vorschriften, Gesetze, Kundenwünsche, ...
- Zuverlässige und verfügbare Systeme
- Integrierbare Systeme
  - Offene Schnittstellen, Standards
- Komplexität vor Nutzern verbergen
- Intelligente Unterstützung
  - Gegenseitige Unterstützung der Nutzer ?! -> soziales Netz

Herausforderungen an die Softwareentwicklung

- Effizienz der Entwicklung
  - Agilität der Upgrades / Weiterentwicklung
  - Komplexität managen
- Make or Buy
  - Eigenentwicklung, Offshoring, Kauf
- Global verteilte Entwicklung
  - Werkzeuge/Infrastruktur zur Entwicklung
- Qualität
  - Korrektheit & Zuverlässigkeit
  - Sicherheit der Daten, Privacy
- Cloud
- Intelligenz (KI, Neuronale Netze, Maschinelles Lernen)
- Interoperabilität der Applikationen verteilt handelnder Institutionen
- Technologien
- Diversifikation in Anwendungsdomänen
  - vs. allgemeine Grundlagen, Methoden, Werkzeuge

Konsequenzen für die SWT-Forschung

- Agile, qualitativ hochwertige SE-Prozesse
  - definieren und etablieren
  - Verzahnung mit traditionellen Ingenieurs-Prozessen
- Technologien weiterentwickeln
  - Plattformen, Frameworks, Komponenten
- Entwicklungswerkzeuge verbessern
  - Intelligente Code-Generatoren
  - Analysewerkzeuge
  - Testgeneratoren
  - Simulation
- Modellierung
  - Domänenspezifische, Problem-angepasste Sprachen
  - Werkzeuge, Infrastrukturen

Zusammenfassung zur Vorlesung

- Softwaretechnik ist eine Schlüsselqualifikation von Informatikern/-innen und Informatik-nahen Berufen
- SWT-Kenntnisse erlauben die Durchführung großer IT-Projekte
  von der Planung bis zum Qualitätsmanagement
- Der Inhalt dieser Vorlesung beschränkt sich auf Kernelemente der Softwaretechnik
- Weiterführende Themen sind z.B.:
  - Projektmanagement
  - Muster
  - Model-Engineering
  - Softwarearchitektur
  - Testen
  - Software Language Engineering
  - Domänenspezifisches SE
  - Model-Driven Systems Engineering

Wir wünschen viel Vergnügen bei der Nutzung von Software Engineering bei innovativen Produkten und spannenden Projekten!

Interesse bei uns dabei zu sein?

Automotive / Cyber-Physical Systems

- Autonomous Driving
- Simulation
- Deep Learning
- Data Science
- Modeling Embedded Systems
- Generative Test-Driven Development

Model-Driven Systems Engineering

- Systems Modeling Languages
- Domain-Specific Application
- Software Architectures
- Semantic Tool Integration
- Variability & Product Lines
- Industry 4.0, CPPS, Robotics

Model-Based Assistance and Information Services

- Models in Information and Workflow Systems
- Human-Centered Assistance
- Behavior/ Process Modeling
- DSLs for rights, roles, privacy
- Digitalization and digital transformation

Modeling Language Engineering

- Development Tools
- Language Workbench MontiCore
- UML, SysML, Architecture DL
- Domain-specific languages (DSL)
- Generation, synthesis
- Testing, analysis, verification
- Software architecture, evolution
- Agile methods