Video

Objektorientierte Analyse und Design

Prof. Dr. Stephan Kleuker

Kernziele:

- Strukturen für erfolgreichen SW-Entwicklungsprozess kennen lernen
- Realisierung: Von der Anforderung zur Implementierung

Ich

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- Prof. Dr. Stephan Kleuker, geboren 1967, verheiratet, 2 Kinder
- seit 1.9.09 an der FH, Professur für Software-Entwicklung
- vorher 4 Jahre FH Wiesbaden
- davor 3 Jahre an der privaten FH Nordakademie in Elmshorn
- davor 4 ½ Jahre tätig als Systemanalytiker und Systemberater in Wilhelmshaven

- s.kleuker@hs-osnabrueck.de, Zoom-Termine kurzfristig per E-Mail vereinbar

Ablauf

- 2h Vorlesung + 2h Praktikum = 5 CP
- Praktikum (2er oder 3er oder 4er Gruppen):
  - Anwesenheit = (Übungsblatt vorliegen + Lösungsversuche zum vorherigen Aufgabenblatt + Fragen)
  - 12 Übungsblätter mit insgesamt ca. 100 Punkten
  - Praktikum mit 85 oder mehr Punkten bestanden
- Prüfung: Hausarbeit, 3er-Gruppen, Themen s. Webseite
- ~~Folienveranstaltungen sind schnell, bremsen Sie mit Fragen~~
- steuern Sie Ihr Lerntempo mit den Videos selbst (Pausetaste)
- von Studierenden wird hoher Anteil an Eigenarbeit erwartet
- Melden Sie sich in ILIAS zu VL und Praktikum an, Freischaltung sollte erfolgt sein
- Praktikum startet „sofort“ an nächsten geplanten Termin

Verhaltenscodex

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- Rechner sind zu Beginn der Veranstaltung aus
- Handys sind aus
- Wir sind pünktlich
- Es redet nur eine Person zur Zeit

- Sie haben die Folien zur Kommentierung in der Vorlesung vorliegen (Ihre Aufgabe), zwei Tage vor der VL liegen abends Unterlagen im Netz

  http://www.edvsz.hs-osnabrueck.de/kleuker/index.html

- Probleme sofort melden
- Wer aussteigt teilt mit warum

Praktikum genauer

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- Praktikumsaufgaben müssen jeweils als Ergebnisse im Praktikum der Folgewoche vorliegen; diese werden dort abgenommen
- Falls jemand nicht kommt, sind die Ergebnisse per E-Mail spätestens am Praktikumstag an den Praktikumsleiter zu schicken; werden in der Folgewoche abgenommen
- Aufgaben dürfen in Gruppen von maximal drei (minimal zwei) Studierenden bearbeitet werden; ***jeder*** muss in der Lage sein, jedes Gruppenergebnis vorzustellen (gerade auch bei evtl. späteren Abnahmen)
- Treten ähnliche Ergebnisse bei mehr als einer Arbeitsgruppe auf, werden diese bei allen Arbeitsgruppen gestrichen
- bei Lösungen aus dem Internet ist das Praktikum beendet

Praktikum - Aufgabenbearbeitung

- Bearbeitung in 3er/2er/4er-Gruppen
- sinnvoll: Pairprogramming, zwei Personen an einem Rechner
- Ansatz: eigene Tastatur und Maus mitbringen

USB-Stick (lokaler Speicher), neben Z:

private Tastatur und Maus von Studi

Veranstaltung im Studienkontext

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

+ Sie haben Kenntnisse in der OO-Programmierung (C++, Java)
+ [Sie können Datenbanken (Überschneidung bei Modellierung)]

= Sie können erfolgreich an dieser Veranstaltung teilnehmen

+ nächstes Semester: Veranstaltung Software-Engineering Projekt (Vorlesungsanteil zur Organisation von SW-Projekten in Unternehmen, großes Praktikumsprojekt, 10 CP)

Skript = Buch

Hinweis:

Aktuelle Bücher des Springer-Verlags Können über Web-Seite der Bibliothek als PDF legal heruntergeladen werden, Fachdatenbanken (DBIS)

weitere Literatur

Generell lesenswert:

- Jochen Ludewig, Horst Lichter, Software Engineering: Grundlagen, Menschen, Prozesse, Techniken, dpunkt.verlag, Heidelberg
- Bernd Oestereich, Axel Scheithauer, Analyse und Design mit UML 2.5, Oldenbourg, München
- C. Rupp, S. Queins, B. Zengler, UML 2 glasklar, Hanser, München Wien
- Ian Sommerville, Software Engineering, Addison Wesley, Boston

- (jeweils aktuellste Auflage)
- Spezialliteratur wird zum jeweiligen Kapitel genannt

Werkzeuge

- Programmierung mit Eclipse, Modellierung mit UMLet
  http://kleuker.iui.hs-osnabrueck.de/querschnittlich/SEU.pdf
- UMLet ist (fast) reines Malwerkzeug für verschiedene UML-Diagrammarten (etwas instabiler unter Linux)
- gibt SEU auf HS-Rechner identisch für zu Hause C:\kleukersSEU; ist verpflichtend zu nutzen

- gibt professionellere Werkzeuge, die aber nicht generell frei verfügbar sind (jedes Unternehmen kocht hier seinen eigenen „Werkzeugbrei“ zusammen)
- Bedeutung der Diagramme im Entwicklungsprozess unterschiedlich („fokussiert auf aktuelle UML-Diagramme“ oder nur „zentrales Hilfsmittel für Skizzen“)

Inhaltsverzeichnis

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- 2 Prozessmodellierung
- 1 Motivation von Software-Engineering
- ~~3 Vorgehensmodelle~~ — kurz, genauer nächstes Semester
- 4 Anforderungsanalyse
- 5 Grobdesign
- 6 Vom Klassendiagramm zum Programm
- 8 Optimierung des Designmodells
- 7 Konkretisierungen im Feindesign
- 9 Implementierungsaspekte
- ~~10 Oberflächengestaltung~~ — andere Veranstaltung
- ~~11 Qualitätssicherung~~ — Wahlfach
- ~~12 Umfeld der Software-Entwicklung~~ — nächstes Semester

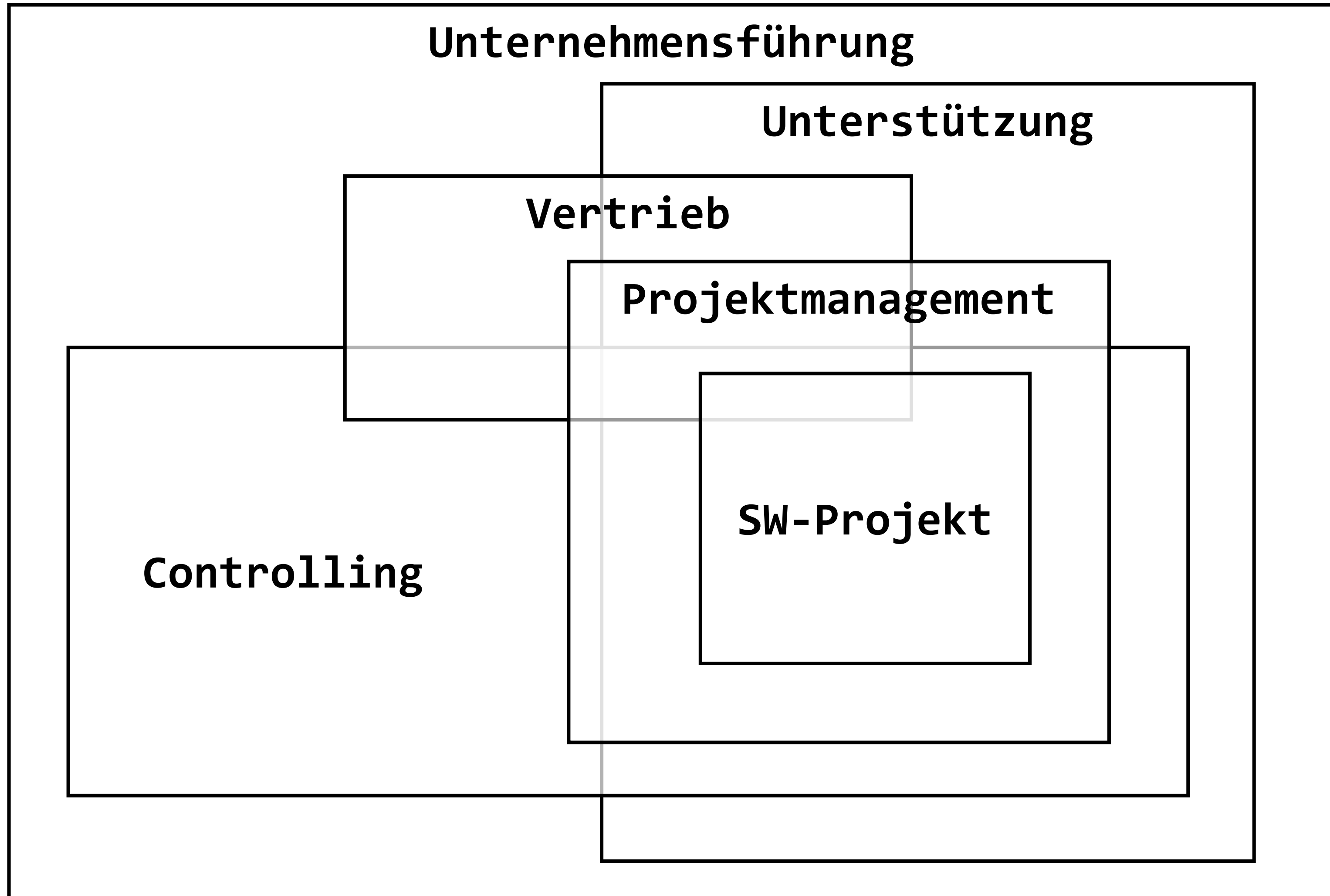
Video

2. Prozessmodellierung

- 2.1 Unternehmensprozesse
- 2.2 Prozessmodellierung mit Aktivitätsdiagrammen

Umfeld von SW-Projekten

2.1



Prozesse in Unternehmen aus SW-Projektsicht

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

(Annahme SW ist wichtiges Kernprodukt)

- Unternehmensführung gibt Geschäftsfelder und Strategien vor
- Vertriebsleute müssen potenzielle auftraggebende Firmen finden, überzeugen und Aufträge generieren
- Aufträge führen zu Verträgen, die geprüft werden müssen
- Das Personal für Aufträge muss ausgewählt werden und zur Verfügung stehen
- Der Projektablauf muss beobachtet werden, Abweichungen z. B. in Zeitplan müssen zu Steuerungsmaßnahmen führen

- Die SW muss realisiert werden

Rollenbegriff

- Unterschiedliche Menschen arbeiten in verschiedenen Rollen zusammen
- Rolle: genaue Aufgabenbeschreibung, mit Verantwortlichkeiten (was soll gemacht werden) und Kompetenzen (welche Entscheidungen können getroffen werden, z. B. „Arbeit anweisen“)
- Mensch kann in einem Unternehmen/Projekt mehrere Rollen haben
- Eine Rolle kann von mehreren Menschen besetzt werden
- Beispielrollen: Vertriebsleitung, Vertriebsmitarbeit, Projektleitung, mitarbeitende Personen in der Anforderungsanalyse, Entwicklung, Qualitätssicherung

Prozessbegriff

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

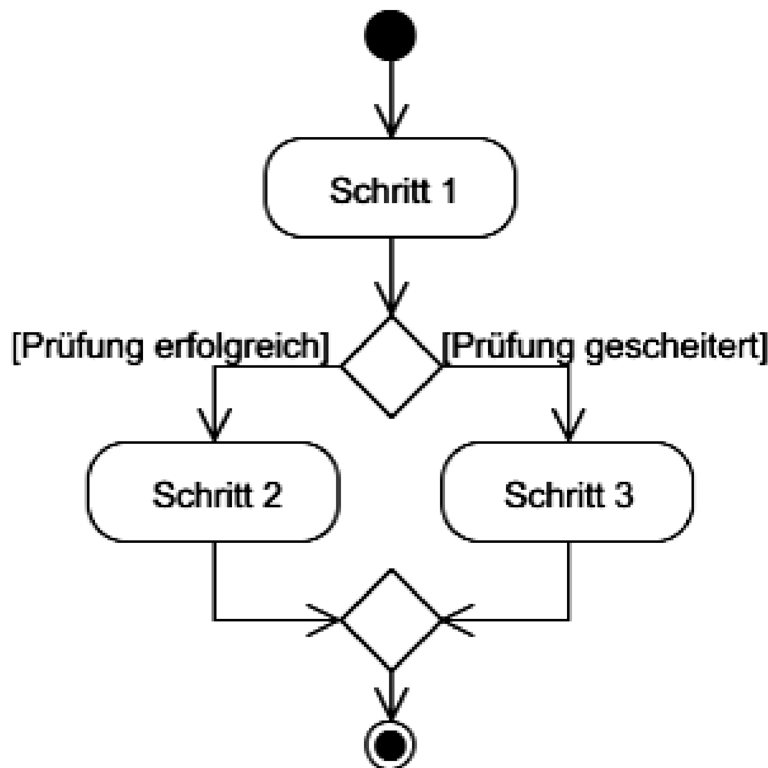
Prozessbeschreibungen regeln die Zusammenarbeit verschiedene Menschen (genauer Rollen),

- Was soll in diesem Schritt getan werden?
- Wer ist verantwortlich für die Durchführung des Schritts?
- Wer arbeitet in welcher Rolle in diesem Schritt mit?
- Welche Voraussetzungen müssen erfüllt sein, damit der Schritt ausgeführt werden kann?
- Welche Teilschritte werden unter welchen Randbedingungen durchgeführt?
- Welche Ergebnisse kann der Schritt abhängig von welchen Bedingungen produzieren?
- Welche Hilfsmittel werden in dem Prozessschritt benötigt?
- Welche Randbedingungen müssen berücksichtigt werden?
- Wo wird der Schritt ausgeführt?

Prozesse sind zu dokumentieren und zu pflegen

2.2

Zur Beschreibung werden folgende elementare Elemente genutzt:



genau ein Startpunkt

einzelner Prozessschritt (Aktion)

Kontrollknoten (Entscheidung)

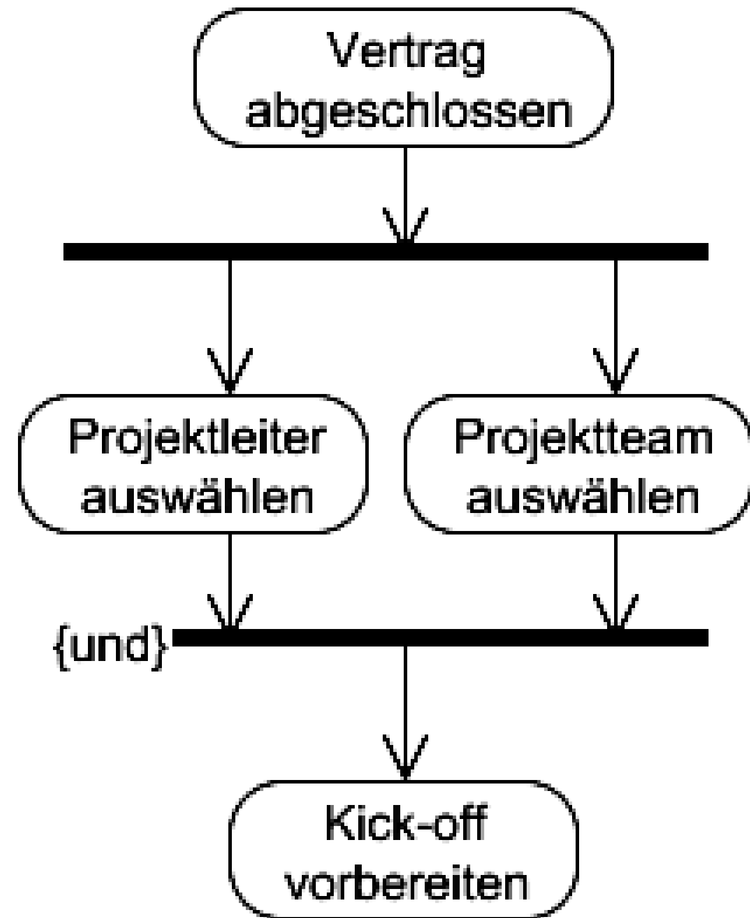
ausgehenden Kanten: Boolesche Bedingungen in eckigen Klammern

Kontrollknoten (Zusammenführung)

Endpunkt (Terminierung)

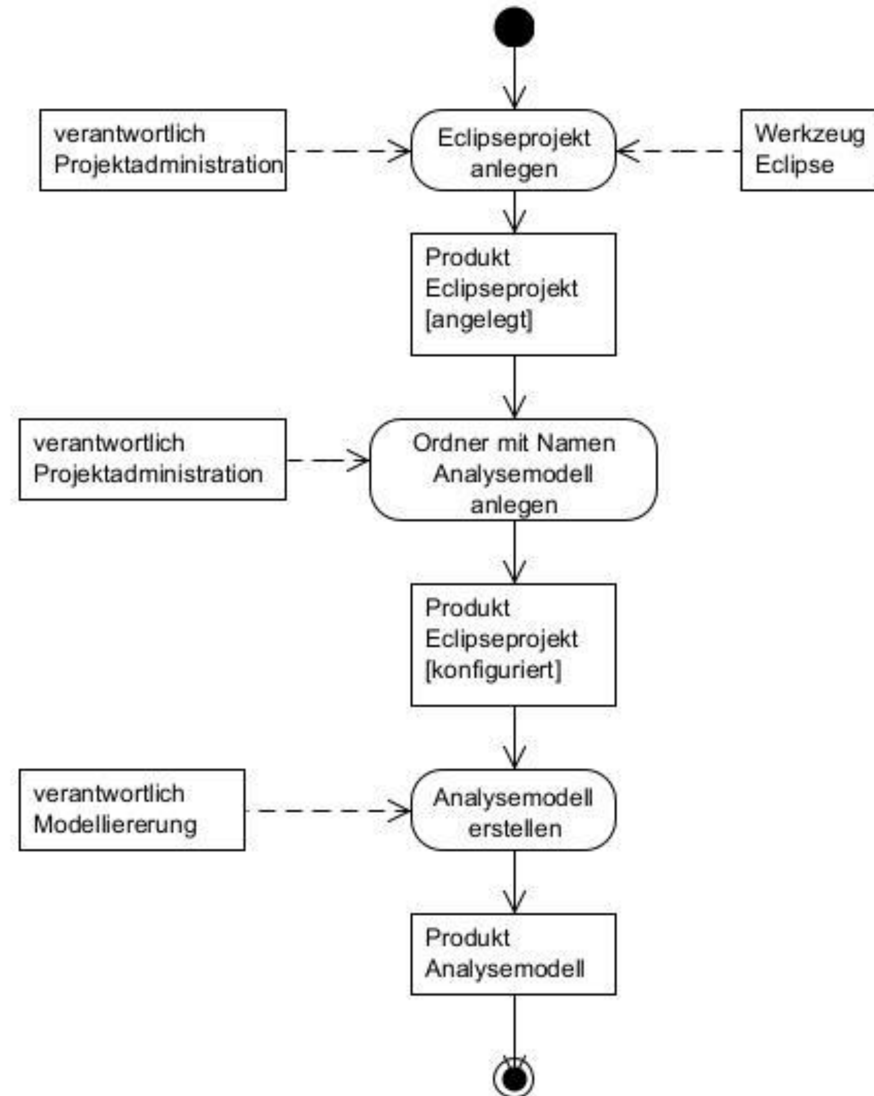
Parallelität in Prozessen

- Waagerechter oder senkrechter Strich steht für mögliche Prozessteilung (ein Pfeil rein, mehrere raus) oder Zusammenführung (mehrere Pfeile rein, ein Pfeil raus)
- Am zusammenführenden Strich steht Vereinigungsbedingung, z. B.
  - {und}: alle Aktionen abgeschlossen
  - {oder}: (mindestens) eine Aktion abgeschlossen
- UML 1.1 hatte andere Restriktionen



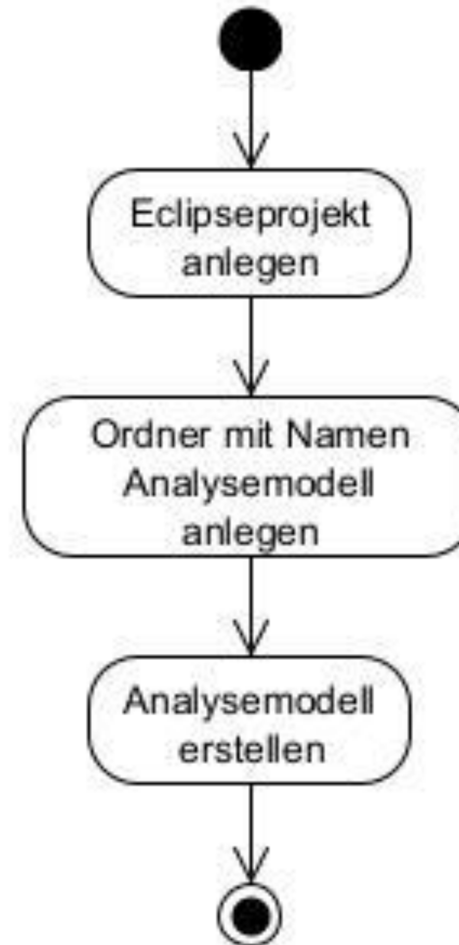
Beteiligte, Produkte, Werkzeuge (optional)

- Beteiligte Personen, Produkte, Werkzeuge werden hier als einfache Datenobjekte modelliert, dabei steht zunächst die Objektart und dann die genaue Bezeichnung
- In eckigen Klammern kann der Zustand eines Objekts beschrieben werden
- neben „verantwortlich“ noch „mitwirkend“ möglich
- auch Entscheidungen haben verantwortliche Personen



Anmerkungen

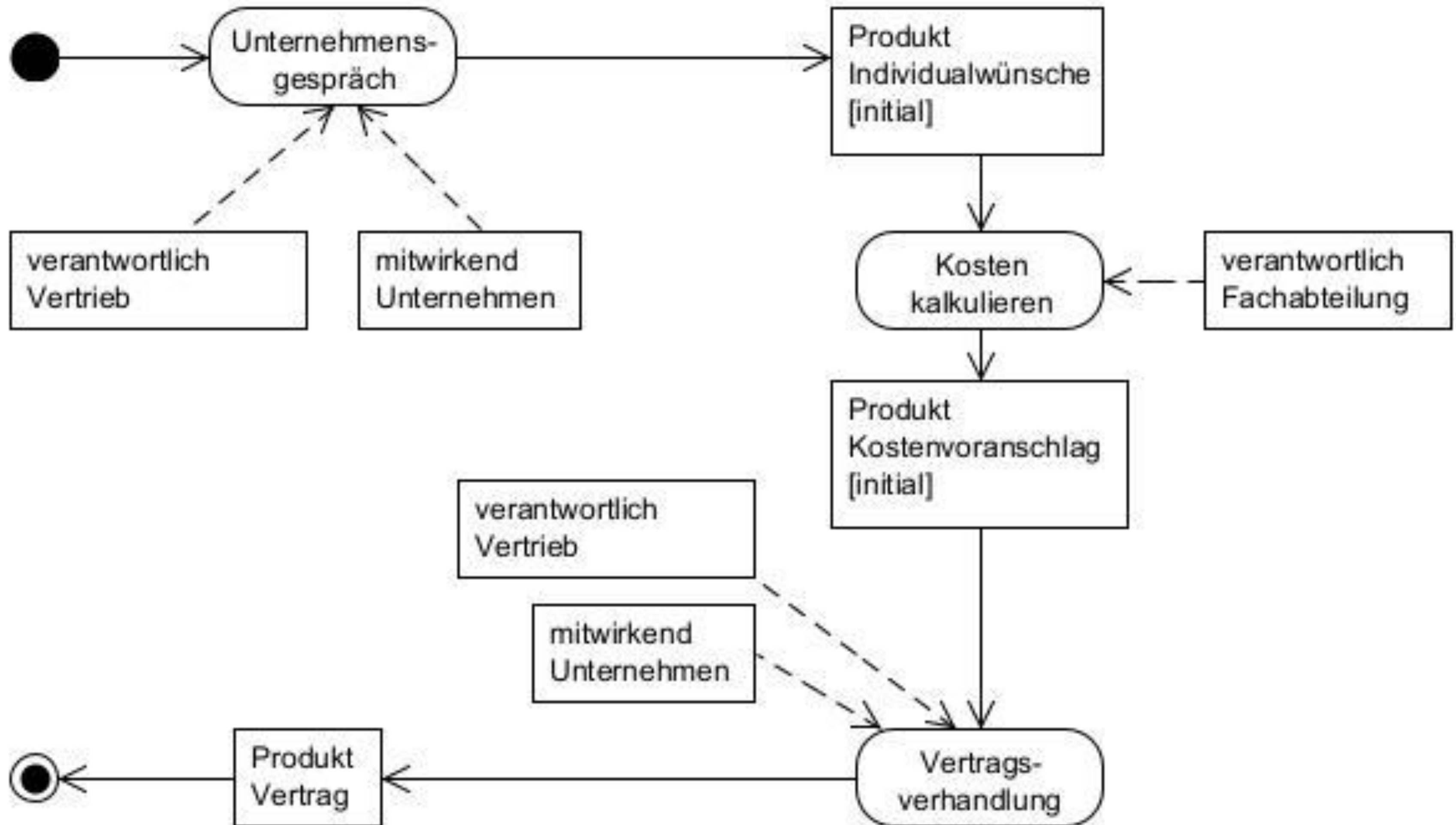
- immer erst ohne "Kästen" modellieren
- häufig alternative Darstellungen für Rollen und Werkzeuge
- Variante: nur Ablauf, Rest in Textdokumentation
- Buch alte Version: alle Linien durchgezogen



Beispiel: Vertrieb (1/4)

- Video Zu modellieren ist der Vertriebsprozess eines Unternehmens, das SW verkauft, die individuell für das beauftragende Unternehmen angepasst und erweitert werden kann
- Modelle werden wie SW inkrementell erstellt; zunächst der (bzw. ein) typische Ablauf, der dann ergänzt wird
- Typisches Szenario: Mitarbeitende Person des Vertriebs kontaktiert potenzielles beauftragendes Unternehmen und arbeitet individuelle Wünsche heraus; Fachabteilung erstellt Kostenvoranschlag; beauftragendes Unternehmen unterschreibt Vertrag; Projekt geht in Prozess Projektdurchführung (nicht modelliert)
- Beteiligt: Vertriebsmitarbeit, beauftragendes Unternehmen, Fachabteilung
- Produkt: Individualwünsche, Kostenvoranschlag, Vertrag
- Aktionen: Unternehmensgespräch, Kostenkalkulation, Vertragsverhandlung

Beispiel: Vertrieb (2/4)

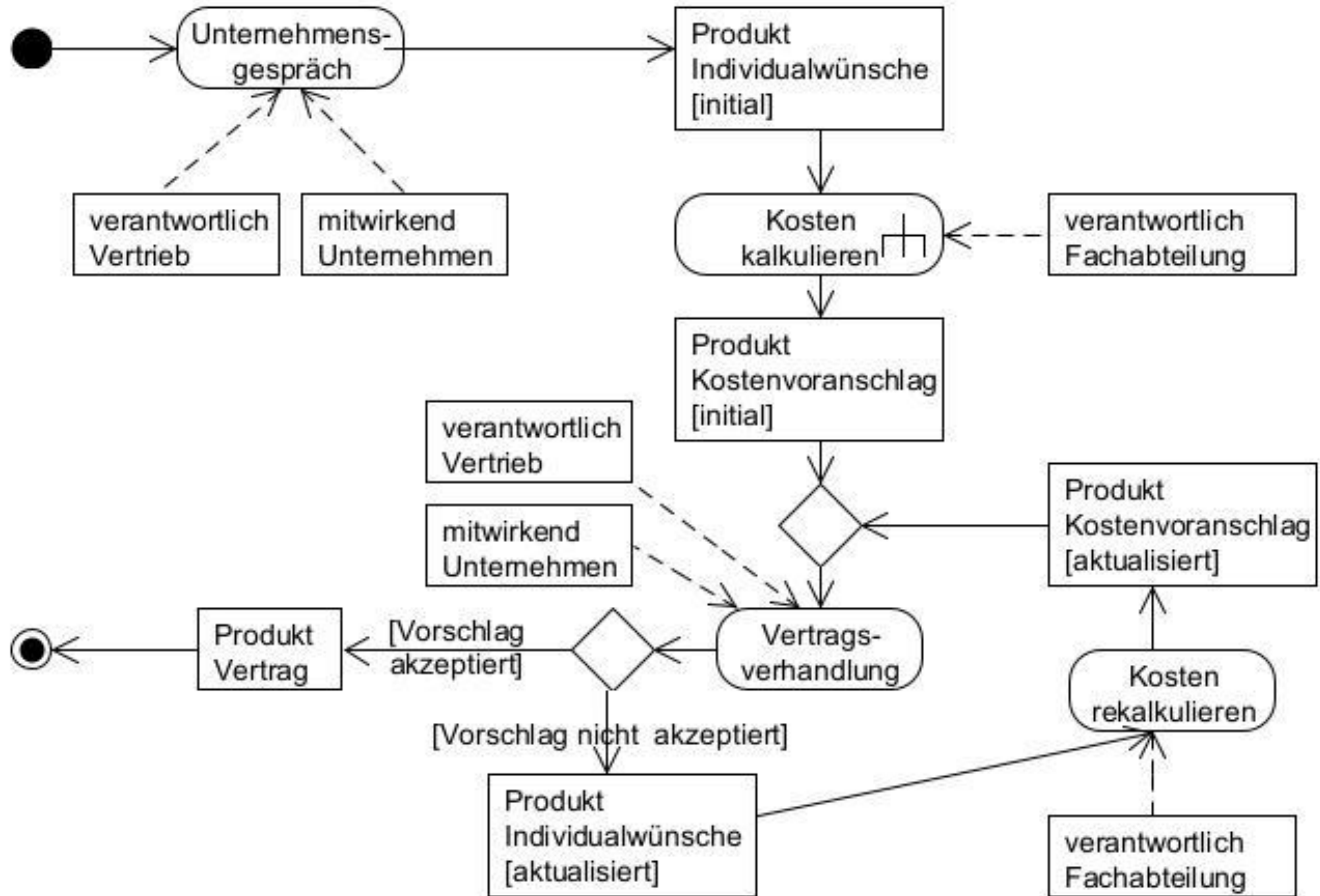


Beispiel: Vertrieb (3/4)

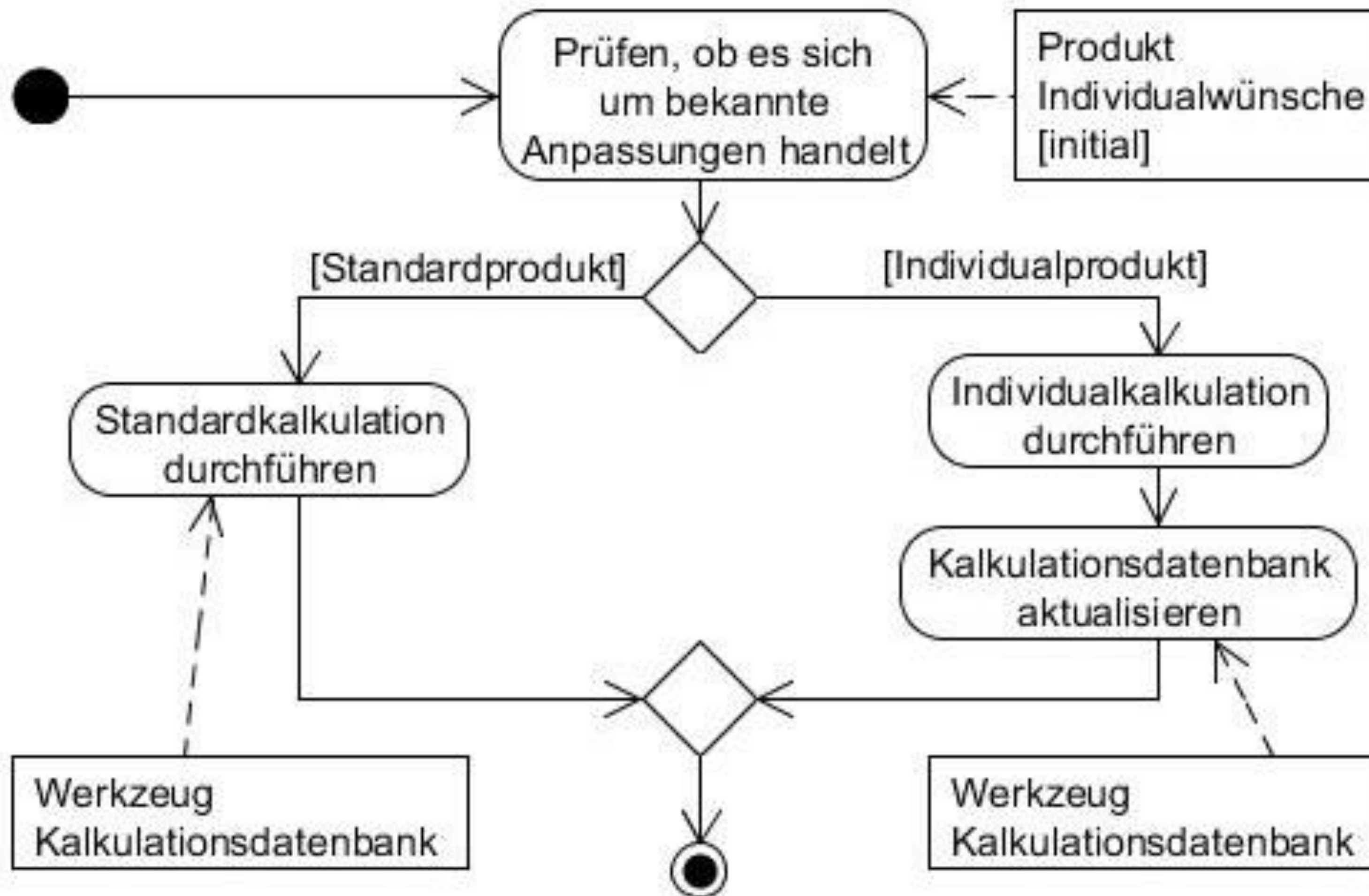
nächster Schritt: Einbau alternativer Abläufe

- Unternehmen ist am Angebot nicht interessiert
- In den Vertragsverhandlungen werden neue Rahmenbedingungen formuliert, so dass eine Nachkalkulation notwendig wird [nächste Folie]
- Bis zu einem Vertragsvolumen von 20 T€ entscheidet die Abteilungsleitung, darüber die Geschäftsleitung ob vorliegender Vertrag abgeschlossen werden soll oder Nachverhandlungen nötig sind
- Die Fachabteilung hat Nachfragen, die die mitarbeitende Person des Vertriebs mit dem potenziell beauftragenden Unternehmen klären muss

Beispiel: Vertrieb (4/4)



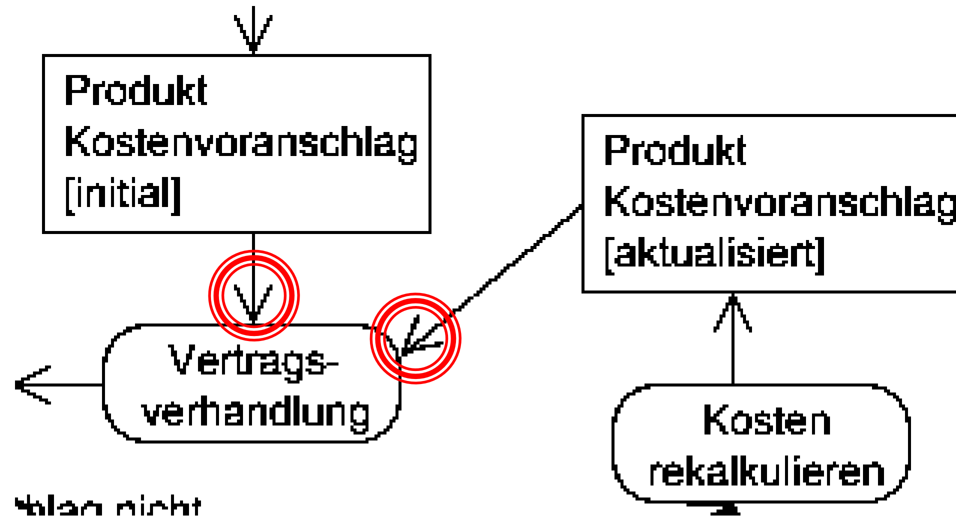
Prozessverfeinerung: Kosten kalkulieren



Anmerkung: Verantwortliche weggelassen, da immer „Projektbegleitung der Fachabteilung“

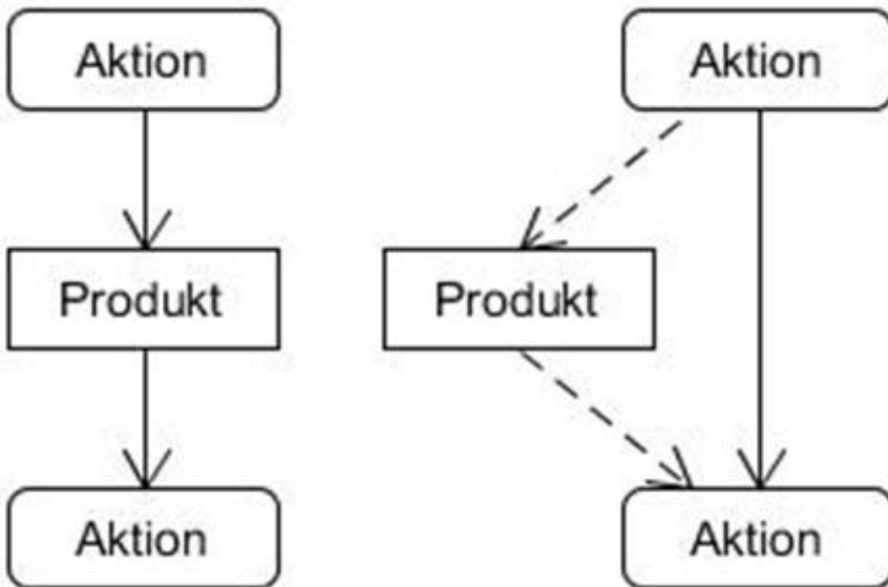
Modellierungsfalle

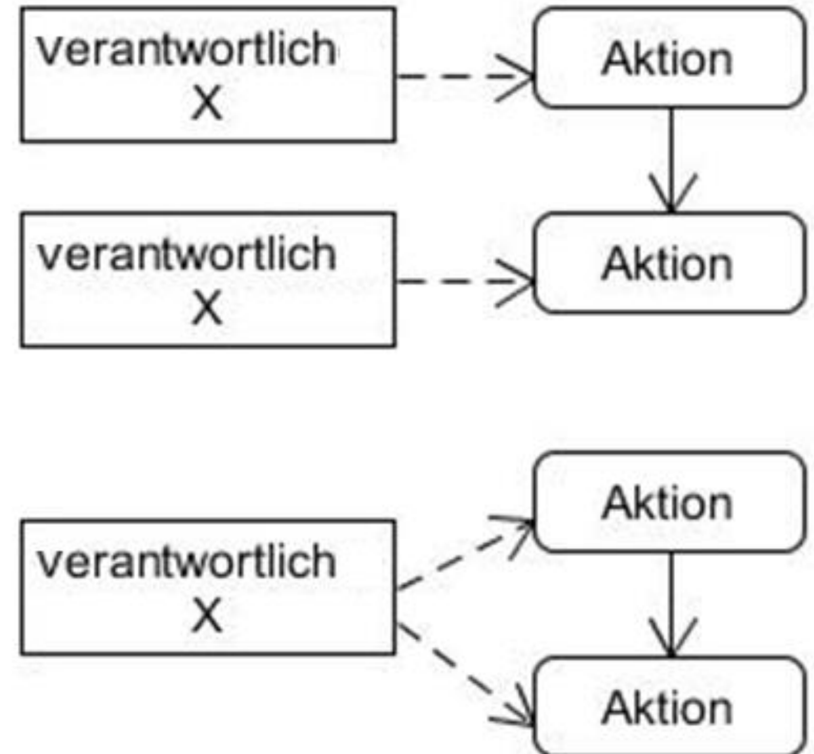
- Basierend auf Erfahrungen mit Flussdiagrammen könnte man zu folgender Modellierung kommen



- Dies würde nach UML-Semantik bedeuten, dass für die Aktion Vertragsverhandlung zwei Kostenvorschläge (initial und aktualisiert, zwei eingehende Kanten) vorliegen müssten
- Wenn verschiedenen Wege zu einer Aktion führen sollen, muss vor der Aktion ein Zusammenführungs-Kontrollknoten stehen

Modellierungsvarianten





Produkt existiert und wird in Aktion bearbeitet

Problem Lesbarkeit

- Diagramme können leicht komplex werden

Lösungsmöglichkeiten:

- Verteilung von Diagrammen auf mehrere Seiten mit Ankerpunkten
- Verzicht, alle Elemente in einem Diagramm darzustellen (z. B. Produkte weglassen; dies nur in der immer zu ergänzenden Dokumentation erwähnen)
- Diagramme hierarchisch gestalten; eine Aktion kann durch ein gesamtes Aktivitätsdiagramm verfeinert werden, z. B. ist „Kosten kalkulieren“ eigener Prozess; dies sollte im Modell sichtbar werden

Problem Abstraktionsgrad

- Frage: Wann nur eine Aktion, wann mehrere Aktionen
- Indikator: Mehrere Aktionen zusammenfassen, wenn
  - nur ein Produkt entsteht, das ausschließlich in diesen Aktionen benötigt wird („lokale Variable“)
  - oder diese von nur einer Person/Rolle bearbeitet werden
- Typischerweise Prozesshierarchie:
  - Unternehmensebene; d.h. ein Diagramm für jeden Prozess der Kern-, Management- und Supportprozesse
  - Prozessebene: Verfeinerung des Prozesses, so dass alle nur intern sichtbaren Rollen und Produkte sichtbar werden
  - Arbeitsprozess: Individuelle Beschreibung der Arbeitsschritte einer Rolle für eine/ mehrere Aktionen
- Probleme: Flexibilität und Akzeptanz

Video

1. Motivation von Software-Engineering

Historie des SW-Engineering (1/4)

- Ende 60er
  - Bedarf für Softwaretechnik neben der reinen Programmierung erstmals voll erkannt (u. a. NATO Software Engineering Conference, Garmisch, 1968)
  - Vorher sind zahlreiche große Programmentwicklungen (möglich durch verbesserte Hardware) gescheitert
  - Arbeiten von Dijkstra 1968 (u.a. gegen Verwendung von GOTO) und Royce 1970 (Software-Lebenszyklus),
    - Top-Down-Entwurf, graphische Veranschaulichungen (Nassi-Shneiderman Diagramme)
- Mitte 70er
  - Top-Down-Entwurf für große Programme nicht ausreichend, zusätzlich Modularisierung erforderlich
  - Entwicklung der Begriffe *Abstrakter Datentyp, Datenkapselung* und *Information Hiding*

Historie des SW-Engineering (2/4)

- Ende 70er
  - Bedarf für präzise Definition der Anforderungen an ein Softwaresystem, Entstehen von Vorgehensmodellen, z. B. *Structured Analysis Design Technique* (SADT)
- 80er Jahre
  - Vom Compiler zur Entwicklungsumgebung (Editor, Compiler, Linker, symbolischer Debugger, Source Code Control Systems)
  - Weiterentwicklung der Modularisierung und der Datenkapselung zur objektorientierten Programmierung
- 90er Jahre
  - Objektorientierte Programmierung nimmt zu (wieder ausgehend von der Implementierung)
  - Neue Programmiersprache Java (ab Mitte 80er C++)
  - Anwendungs-Rahmenwerke (*Application Frameworks*) zur Vereinfachung von Design und – vor allem – Programmierung

Historie des SW-Engineering (3/4)

- 90er Jahre
  - Geeignete Analyse- und Entwurfsmethoden entstehen (*Coad/Yourdon*, *Rumbaugh, Booch, Jacobson* und andere)
- 1995
  - Vereinigung mehrerer Ansätze zunächst als *Unified Method* (UM) von Booch und Rumbaugh, dann kommt Jacobson hinzu (Use Cases).
  - 3 Amigos definieren die *Unified Modeling Language* (UML) als Quasi-Standard.
- 1997
  - UML in der Version 1.1 bei der OMG (Object Management Group) zur Standardisierung eingereicht und angenommen
  - UML ist jedoch keine Entwicklungsmethode (Phasenmodell), *nur* eine Beschreibungssprache
- 1999
  - Entwicklungsmethode: *Unified Process* (UP) und *Rational Unified Process* (RUP) (erste Version)

Historie des SW-Engineering (4/4)

- Heute
  - Vorgehensweisen auf individuelle Projektanforderungen abgestimmt
  - CASE-Methoden und –Tools orientieren sich an der UML
  - Stand 07/2011: UML 2.4.1
  - Stand 09/2015: UML 2.5
  - Stand 12/2017: UML 2.5.1 (http://www.uml.org/)
  - Aufbauend auf Analyse und Design erzeugen Codegeneratoren Programmgerüste
  - Haupttätigkeiten bei Softwareentwicklung sind Analyse und Design, vieles andere versucht man zu automatisieren (!?)

Warum scheitern SW-Projekte (kleine Auswahl)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- Die Software wurde wesentlich zu spät geliefert
- Die Software erfüllt nicht die Wünsche der nutzenden Personen
- Die Software läuft nicht auf den vereinbarten Rechnersystemen, sie ist zu langsam oder kommt mit dem Speicher nicht aus
- Die Software kann nicht erweitert werden oder mit anderer Software zusammenarbeiten
- …

Antworten des Software-Engineering

- 1967: Prägung des Begriffs Software-Krise
- Lösungsansätze:
  - Programmiersprachen: kontinuierliche Einführung von Abstraktion (Datentypen, Funktionen, Modulen, Klassen, Bibliotheken, Frameworks)
  - Dokumentation: Einheitliche Notationen für Entwicklungsergebnisse (UML)
  - Entwicklungsprozesse: Aufgabenbeschreibungen, wann was wie gemacht wird
  - Vorgehensmodelle: Entwicklung passt sich an Bedürfnisse der nutzenden/bezahlenden Personen an

Definitionsversuch Software-Engineering

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

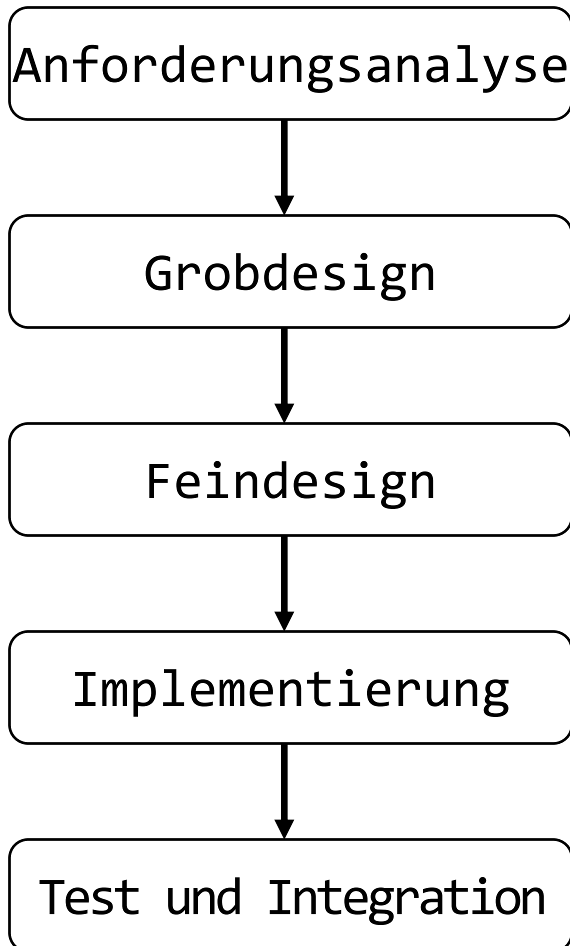
Zusammenfassend kann man Software-Engineering als die Wissenschaft der systematischen Entwicklung von Software, beginnend bei den Anforderungen bis zur Abnahme des fertigen Produkts und der anschließenden Wartungsphase definieren. Es werden etablierte Lösungsansätze für Teilaufgaben vorgeschlagen, die häufig kombiniert mit neuen Technologien, vor Ihrer Umsetzung auf ihre Anwendbarkeit geprüft werden. Das zentrale Mittel zur Dokumentation von Software-Engineering-Ergebnissen sind UML-Diagramme.

Video

3. Vorgehensmodelle

nur kurzer Einblick (★ nur als Vorausschau, nicht Teil der VL)

Die Phasen der SW- Entwicklung



- Erhebung und Festlegung des WAS mit Rahmenbedingungen
- Klärung der Funktionalität und der Systemarchitektur durch erste Modelle
- Detaillierte Ausarbeitung der Komponenten, der Schnittstellen, Datenstrukturen, des WIE
- Ausprogrammierung der Programmiervorgaben in der Zielsprache
- Zusammenbau der Komponenten, Nachweis, dass Anforderungen erfüllt werden, Auslieferung

Wasserfallmodell

Anforderungsanalyse

Grobdesign

Feindesign

Implementierung

Test und Integration

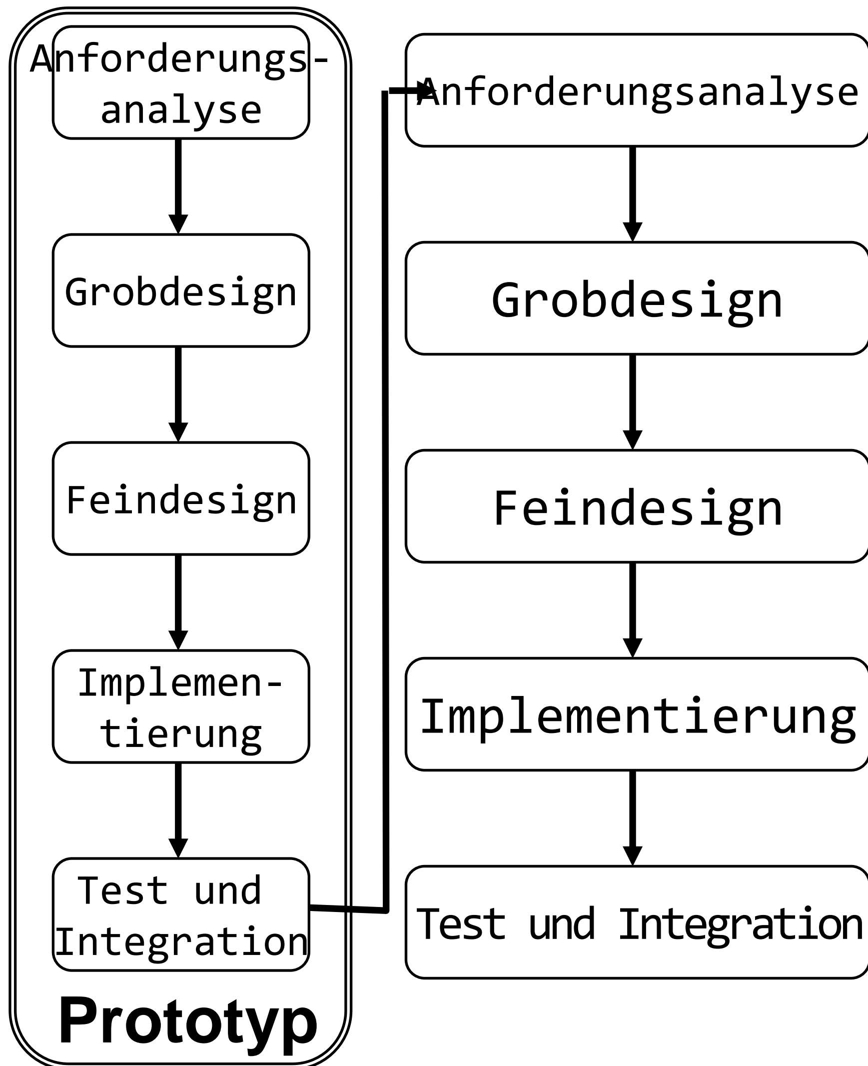
Merkmale:
Phasen werden von oben nach unten durchlaufen

Vorteile:
- Plan auch für IT-unerfahrene verständlich
- einfache Meilensteinplanung
- lange Zeit häufigste Prozessgrundlage

Nachteile:
- Anforderungen müssen 100%-ig sein
- späte Entwicklungsrisiken werden spät erkannt
- Qualität des Design passt sich Zeitplan an

Optimierung:
es ist möglich, in die vorherige Phase zu springen

Prototypische Entwicklung



Merkmale:

- potenzielle Probleme frühzeitig identifiziert,
- Lösungsmöglichkeiten im Prototypen gefunden, daraus Vorgaben abgeleitet

Vorteile:

- **frühzeitige Risikominimierung**
- schnelles erstes Projektergebnis

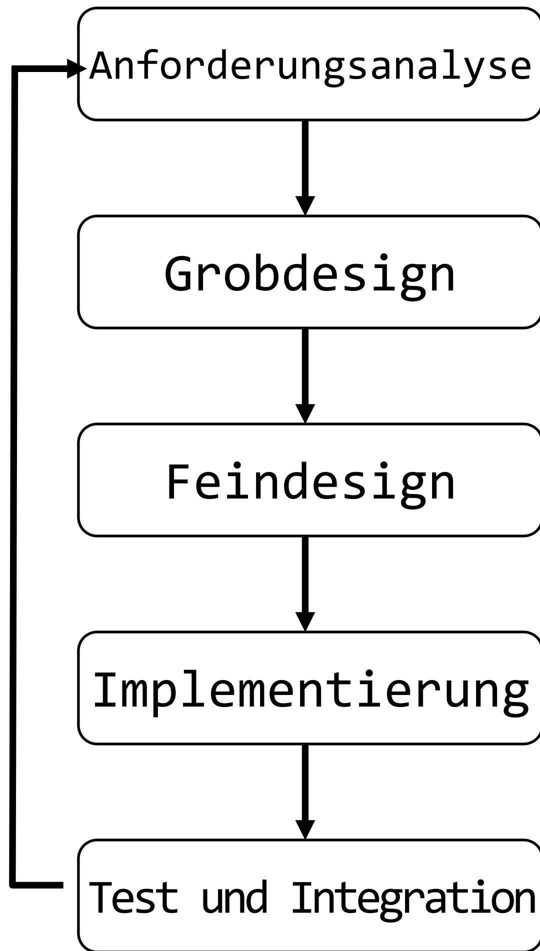
Nachteile:

- Anforderungen müssen fast 100%-tig sein
- Prototyp (illegal) in die Entwicklung übernommen
- Endergebnis zu schnell erwartet

Optimierung:

es ist möglich, in die vorherige Phase zu springen (auch vorheriges Modell)

Iterative Entwicklung



Merkmale:

- Erweiterung der Prototypidee; SW wird in Iterationen entwickelt
- In jeder Iteration wird System weiter verfeinert
- In ersten Iterationen Schwerpunkt auf Analyse und Machbarkeit; später auf Realisierung

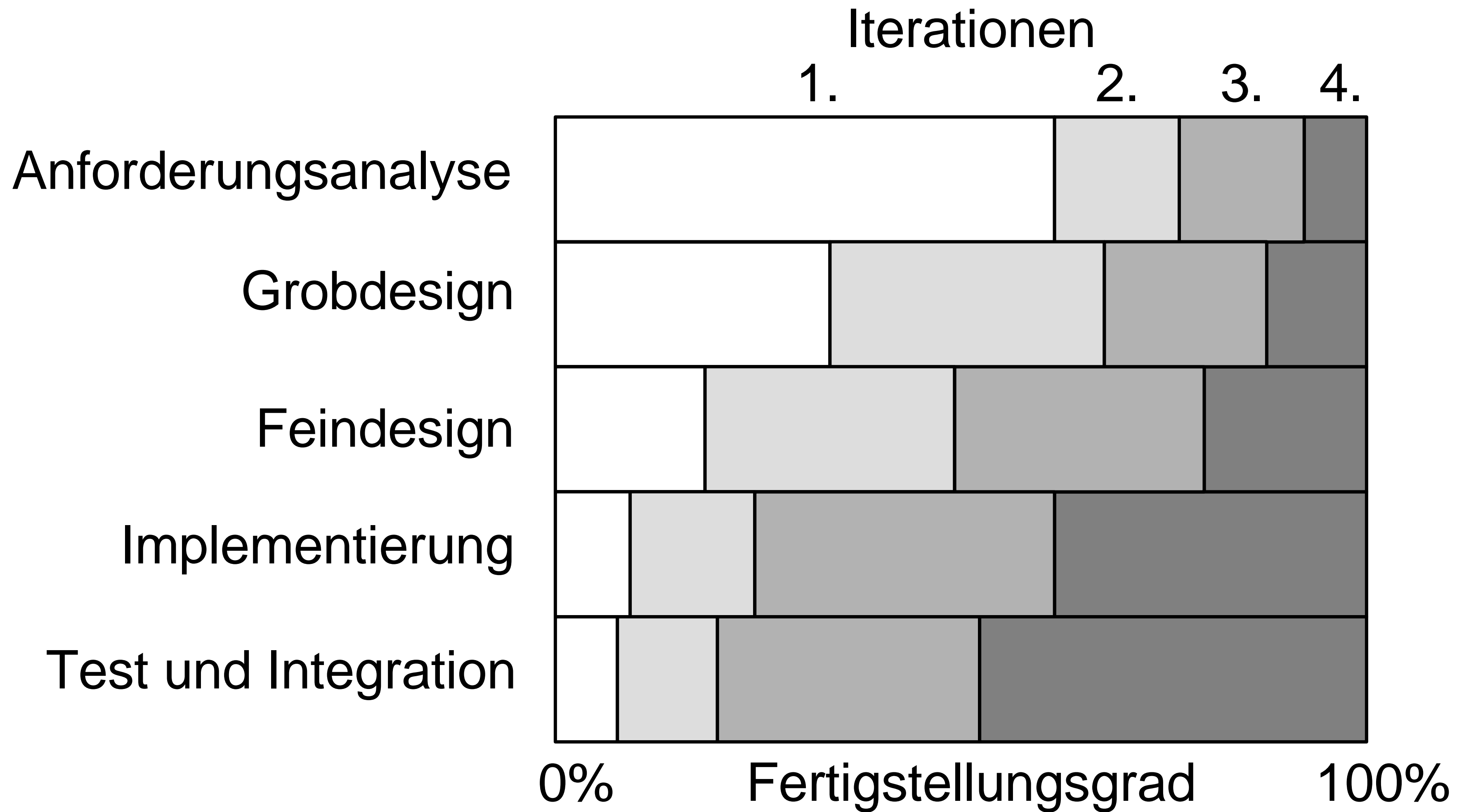
große Vorteile:

- dynamische Reaktion auf Risiken
- Teilergebnisse mit auftraggebenden Personen diskutierbar

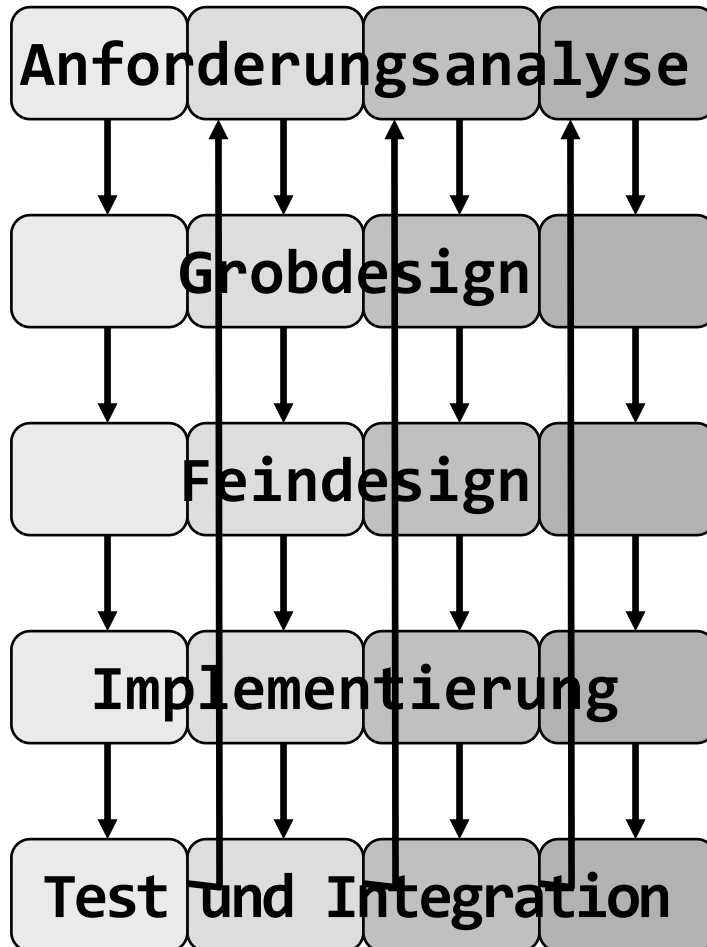
Nachteile im Detail:

- schwierige Projektplanung
- schwierige Vertragssituation
- zu schnelles Endergebnis erwartet (GUI = fertig)
- Anforderungen als beliebig änderbar angesehen

Fertigstellung mit Iterationen



Iterativ Inkrementelle Entwicklung (State of the Art)



Bsp.: vier Inkremente

Merkmal:

- Projekt in kleine Teilschritte zerlegt
- pro Schritt neue Funktionalität (Inkrement) + Überarbeitung existierender Ergebnisse (Iteration)
- n+1-ter Schritt kann Probleme des n-ten Schritts lösen

Vorteile:

- siehe „iterativ“
- flexible Reaktion auf neue funktionale Anforderungen

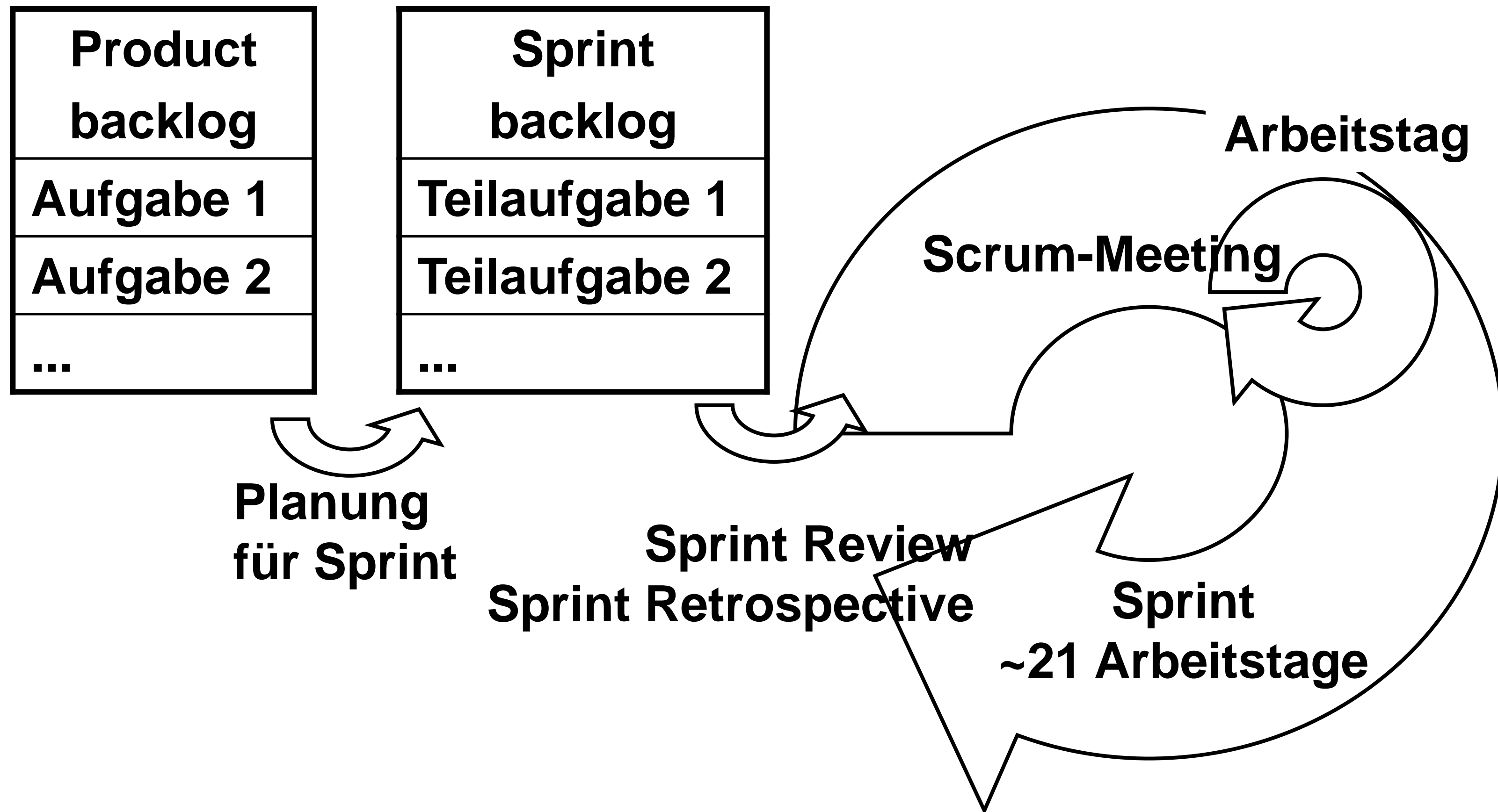
Nachteile:

- siehe „iterativ“ (etwas verstärkt)

Optimierung/Anpassung:

Anforderungsanalyse am Anfang intensiver durchführen

Agile Methoden – Beispiel Scrum



Video

4. Anforderungsanalyse

- 4.1 Stakeholder und Ziele
- 4.2 Klärung der Hauptfunktionalität (Use Cases)
- 4.3 Beschreibung typischer und alternativer Abläufe
- 4.4 Ableitung funktionaler Anforderungen
- 4.5 Nicht-funktionale Anforderungen
- 4.6 Lasten- und Pflichtenheft

Literatur:

- [RS] C. Rupp, SOPHIST GROUP, Requirements- Engineering und – Management, Hanser Fachbuchverlag
- [OW] B. Oestereich, C. Weiss, C. Schröder, T. Weilkiens, A. Lenhard, Objektorientierte Geschäftsprozessmodellierung mit der UML, dpunkt.Verlag

so nicht (1/4): Beispiel-Szenario

Zur Stundenerfassung und Abrechnung werden von den in Projekten mitarbeitenden Personen spezielle Excel-Tabellen jeden Freitag ausgefüllt und am Montag von der Projektleitung bei der Verwaltung abgegeben.

Die zuständige Sachbearbeitung überträgt dann die für den Projektüberblick relevanten Daten manuell in ein SAP-System. Dieses System generiert automatisch eine Übersicht, aus der die Geschäftsführung ablesen kann, ob die Projekte wie gewünscht laufen.

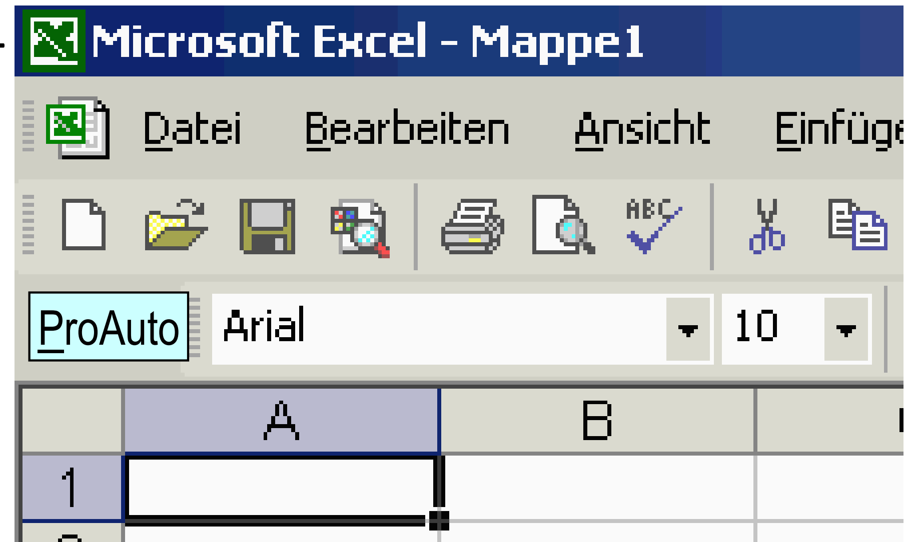
Dieser Bericht liegt meist am Freitag der Woche vor. Die Bearbeitungszeit ist der Geschäftsführung zu lang, deshalb soll der Arbeitsschritt automatisiert werden.

so nicht (2/4): Die Projektplanung

- Projekt „Projektberichtsautomatisierung“ (ProAuto) beschlossen
- Leiter der hausinternen IT-Abteilung über anstehende Aufgabe informiert, er erhält Beschreibung der Excel-Daten und gewünschter SAP-Daten
- Leiter stellt fest, dass seine Abteilung Know-how und die Kapazität hat Projekt durchzuführen, legt Geschäftsführung Projektplan mit Aufwandsschätzung vor
- Geschäftsführung beschließt, Projekt intern durchzuführen, kein externes Angebot einzuholen

so nicht (3/4): Die Schritte zum Projektmisserfolg

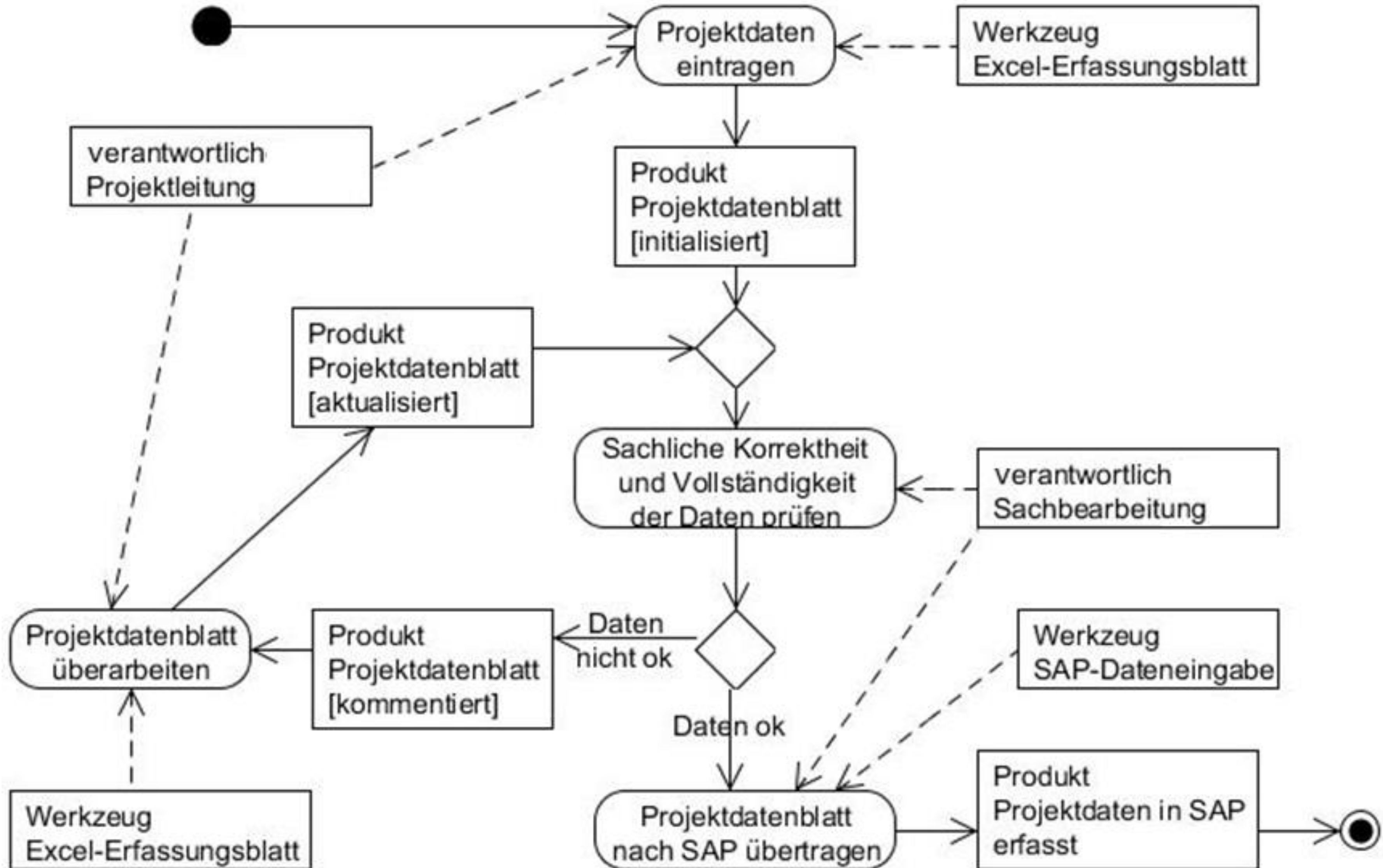
HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES



- IT-Abteilung analysiert Excel-Daten und Daten die in das SAP-System eingefügt werden können
- Kurz nach dem geschätzten Projektende liegt technisch saubere Lösung vor, Excel wurde um Knopf erweitert; Projektleitung kann per Knopfdruck die Daten nach SAP überspielen
- Vier Wochen nach Einführung wird die Leitung der IT-Abteilung entlassen, da Daten zwar jeden Montag vorliegen, sie aber nicht nutzbar sind; erzürnte Geschäftsleitung hat deshalb falsche Entscheidungen getroffen
- Projekt wird an Beratungsfirma neu vergeben

so nicht (4/4): so doch, Geschäftsprozessanalyse

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES



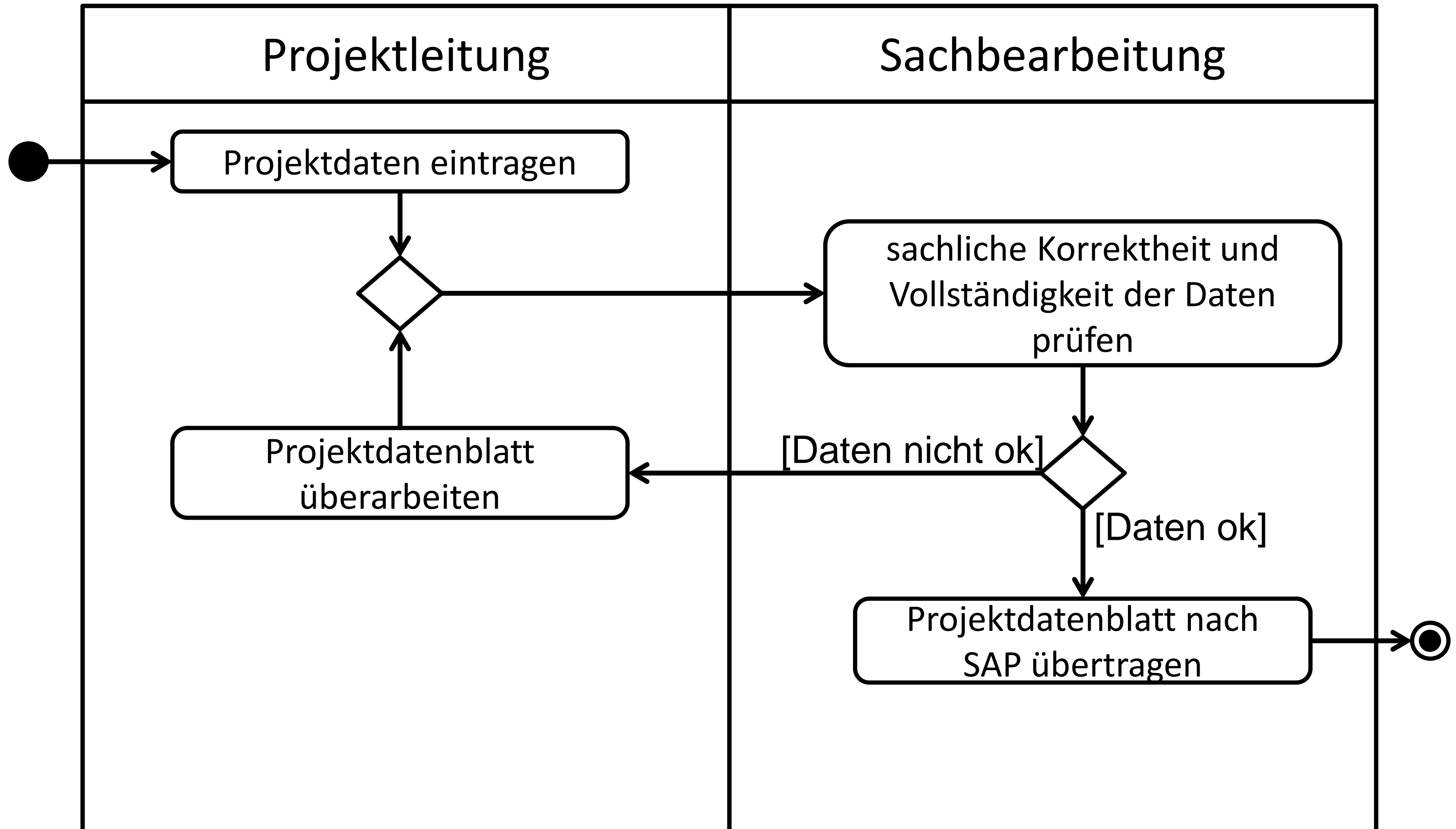
Einschub: Swimlanes (1/2)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- Idee: jede verantwortliche Rolle für mindestens eine Aktion bekommt eine Swimlane
- Aktionen werden jeweils in die Swimlane der verantwortlichen Rolle eingeordnet
- Swimlanes können horizontal oder vertikal angeordnet werden

- Vorteil: schnelle Übersicht über Verantwortlichkeiten
- Nachteil: recht viel Platz für wenige Aktionen benötigt

Einschub: Swimlanes (2/2)



Aufgabe der Anforderungsanalyse

4.1

Bestimmung aller Anforderungen an die zu erstellende Software bzw. an das zu erstellende DV-System, Anforderungen müssen

- vollständig,
- notwendig ("WAS statt WIE"),
- eindeutig und
- richtig ("abgestimmt als Teil einer Zielhierarchie") sein.

Bemerkung zur Ablauforganisation: Anforderungen müssen nicht notwendig in einer Phase vor Beginn des Entwurfs vollständig bestimmt werden

Probleme mit Anforderungen an große Systeme

- Auftraggebende, nutzende, betreibende Personen etc. sind häufig verschiedene Personen, unterschiedliche Personen haben teilweise widersprüchliche Anforderungen
- die Effekte des angestrebten Systems sind schwer vorhersehbar
- Anforderungen ändern sich im Laufe der Entwicklungszeit
- großer Umfang der Anforderungen
- komplexe Interaktion mit anderen Systemen

- Erste Aufgabe: Ermittlung der Stakeholder
- Definition: Eine Person, die Einfluss auf die Anforderungen hat, da sie vom System betroffen ist (systembetroffene Person)
- Zweite Aufgabe: Ermittlung der Ziele des Systems

Checkliste zum Finden von Stakeholdern (1/3) [RS]

Video

- nutzende Personen des Systems
  - Die größte und wichtigste Gruppe, liefert Großteil der fachlichen Ziele
  - Durchdachtes Auswahlverfahren für die Nutzungsrepräsentanten nötig (Vertrauensbasis der gesamten Nutzungsgruppe berücksichtigen!)
- Management des auftragnehmenden Unternehmens (wir)
  - Gewährleisten die Konformität mit Unternehmenszielen und Strategien, sowie der Unternehmensphilosophie
  - Sind die Sponsoren!
- Personen mit Entscheidungsgewalt des auftraggebenden Unternehmens
  - Wer ist für die Kaufentscheidung verantwortlich?
  - Liefer-Vertrags-Zahlungskonditionen?
- prüfendende, auditierende Personen
  - sind für Prüfung, Freigabe und Abnahme notwendig
- entwickelnde Personen
  - nennen die technologiespezifischen Ziele

Checkliste zum Finden von Stakeholdern (2/3)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- Wartungs- und Servicepersonal
  - Wartung und Service muss unkompliziert und zügig durchzuführen sein
  - Wichtig bei hohen Stückzahlen
- Produktbeseitigung
  - Wichtig, wenn ausgeliefertes Produkt nicht nur Software umfasst, Frage der Beseitigung (z.B. Umweltschutz), kann enormen Einfluss auf die Zielsetzung einer Produktentwicklung haben
- Schulungs- und Trainingspersonal
  - Liefern konkrete Anforderungen zur Bedienbarkeit, Vermittelbarkeit, Hilfesystem, Dokumentation, Erlernbarkeit,
- Marketing und Vertriebsabteilung
  - Marketing und Vertrieb als interne Repräsentanten der externen Wünsche des Auftraggebers und der Marktentwicklung

Checkliste zum Finden von Stakeholdern (3/3)

- Systemschutz
  - stellt Anforderungen zum Schutz vor Fehlverhalten von Stakeholdern
- Standards und Gesetze
  - vorhandene und zukünftige Standards/Gesetze berücksichtigen
- Person die Projekt oder Produkt ablehnen
  - Die Klasse der kritisch eingestellten Personen - vor allem zu Beginn des Projekts wenn möglich mit einbeziehen, sonst drohen Konflikte
- Kulturkreis
  - setzt Rahmenbedingungen, z.B. verwendete Symbolik, Begriffe, ...
- Meinungsführung und die öffentliche Meinung
  - beeinflussen oder schreiben Ziele vor, Zielmärkte berücksichtigen

Regeln für die Definition von Zielen

Video

Ziele müssen
- vollständig,
- korrekt,
- konsistent gegenüber anderen Zielen und in sich konsistent,
- testbar,
- verstehbar für alle Stakeholder,
- umsetzbar — realisierbar,
- notwendig,
- eindeutig und positiv formuliert sein.

Zwei weitere Merkmale:
- Lösungsneutralität
- einschränkende Rahmenbedingungen

Hinweis: Ziele sind abstrakte Top-Level-Anforderungen

Schablone zur Zielbeschreibung

Ziel	Was soll erreicht werden?
Stakeholder	Welche Stakeholder sind in das Ziel involviert? Ein Ziel ohne Stakeholder macht keinen Sinn.
Auswirkungen auf Stakeholder	Welche Veränderungen werden für die Stakeholder erwartet?
Rand-bedingungen	Welche unveränderlichen Randbedingungen müssen bei Zielerreichung beachtet werden?
Abhängigkeiten	Ist dieses Ziel mit anderen Zielen unmittelbar verknüpft? Dies kann einen positiven Effekt haben, indem die Erfüllung von Anforderungen zur Erreichung mehrerer Ziele beiträgt. Es ist aber auch möglich, dass ein Kompromiss gefunden werden muss, da Ziele unterschiedliche Schwerpunkte haben.
Sonstiges	Was muss organisatorisch beachtet werden?

Projektbeschreibung

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

Zu entwickeln ist ein individuell auf die Unternehmenswünsche angepasstes Werkzeug zur Projektverwaltung. Dabei sind die Arbeitspakete (wer macht wann was) und das Projektcontrolling (wie steht das Projekt bzgl. seiner Termine und des Budgets) zu berücksichtigen. Projekte werden zur Zeit ausgehend von Projektstrukturplänen geplant und verwaltet.

Projekte können in Teilprojekte zerlegt werden.

Die eigentlichen Arbeiten finden in Arbeitspaketen, auch Aufgaben genannt, statt.

Projekte werden von zusammenzustellenden Projektteams bearbeitet, die zugehörigen Daten der mitarbeitenden Personen sind zu verwalten. Zur Ermittlung des Projektstands tragen mitarbeitende Personen ihre Arbeitszeit und den erreichten Fertigstellungsgrad in das System ein.

Ziele für eine Projektmanagementsoftware (1/3)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

Ziel	1. Die Software muss die Planung und Analyse aller laufenden Projekte ermöglichen
Stakeholder	Projektplanung, Projektleitung, mitarbeitende Personen, Controlling (alle als Nutzende des Systems)
Auswirkungen auf Stakeholder	Projektplanung: Alle Planungsdaten fließen in das neue Werkzeug, es gibt sofort eine Übersicht, wer an was, von wann bis wann arbeitet. Projektleitung: Die Projektleitung ist immer über den Stand informiert, er weiß, wer an was arbeitet. mitarbeitende Person: Teammitglieder sind verpflichtet, ihre Arbeitsstunden und erreichten Ergebnisse in das Werkzeug einzutragen. Sie sehen, für welche Folgearbeiten sie wann verplant sind. Controlling: Hat Überblick über Projektstand.
Rand-bedingungen	Existierende Datenbestände sollen übernommen werden. Die Randbedingungen zur Verarbeitung personalbezogener Daten sind zu beachten.
Abhängigkeiten	-
Sonstiges	Es liegt eine Studie des auftraggebenden Unternehmens vor, warum kein Produkt vom Markt zur Realisierung genommen wird.

Ziele für eine Projektmanagementsoftware (2/3)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

Ziel	2. Das auftraggebende Unternehmen soll von der fachlichen Kompetenz unseres Unternehmens überzeugt werden.
Stakeholder	Management, Entwicklung
Auswirkungen auf Stakeholder	Management: Der Projekterfolg hat große Auswirkungen auf die nächsten beiden Jahresbilanzen. Entwicklung: Es werden hohe Anforderungen an die Software-Qualität gestellt.
Rand-bedingungen	Es muss noch geprüft werden, ob langfristig eine für beide Seiten lukrative Zusammenarbeit überhaupt möglich ist.
Abhängigkeiten	Überschneidung mit dem Ziel 3, da eine Konzentration auf die Wünsche des auftraggebenden Unternehmens eventuell einer Verwendbarkeit für den allgemeinen Markt widersprechen kann.
Sonstiges	Das Verhalten des neuen auftraggebenden Unternehmens bei Änderungswünschen ist unbekannt.

Ziele für eine Projektmanagementsoftware (3/3)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

Ziel	3. Das neue Produkt soll für einen größeren Markt einsetzbar sein.
Stakeholder	Management, Vertrieb, Entwicklung, Rechtsabteilung
Auswirkungen auf Stakeholder	Management: Es soll eine Marktposition auf dem Marktsegment Projektmanagement-Software aufgebaut werden. Vertrieb: In Gesprächen mit potenziell auftraggebenden Unternehmen wird das neue Produkt und seine Integrationsmöglichkeit mit anderen Produkten ab Projektstart beworben. Entwicklung: Die Software muss modular aufgebaut aus Software-Komponenten mit klaren Schnittstellen bestehen. Rechtsabteilung: Klärung der Lizensierung
Randbedingungen	-
Abhängigkeiten	zu Ziel 2 (Beschreibung dort)
Sonstiges	Eine Analyse der Konkurrenz auf dem Markt liegt vor. Es sind Möglichkeiten für neue, den Markt interessierende Funktionalitäten aufgezeigt worden.

Rahmenbedingungen und weiteres Vorgehen

Traceability:

- alle Anforderungen müssen sich auf ein Ziel zurückführen lassen
- alle Ziele benötigen einen Stakeholder (Ökonomie-Check)

Kommunikation:

- die ausgewählten Stakeholder müssen nun detaillierter befragt und dauerhaft in das Projekt integriert werden

Warum der ganze Aufwand:

- Vergessene Ziele und Stakeholder führen zu massiven Change Requests

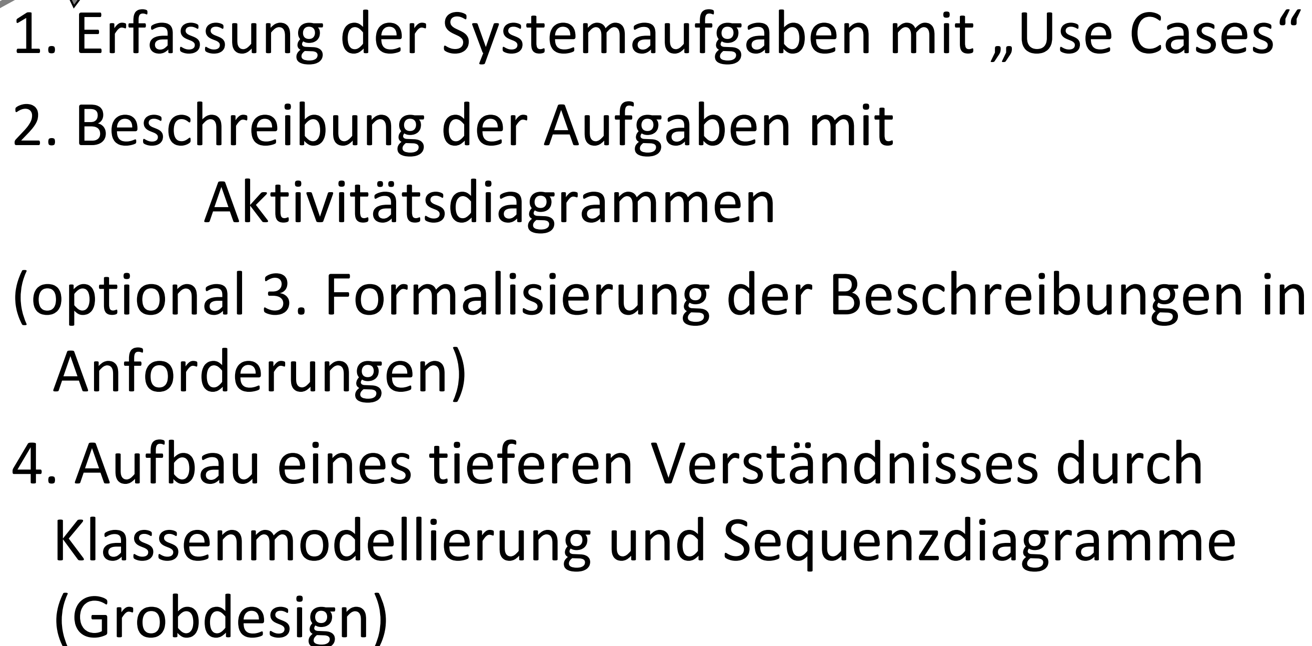
Das eigentliche SW-Projekt kann beginnen.

Überblick über den Analyseprozess

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

4.2

Video



Erfragung des WAS?

- Zentrale Frage:

  Was sind die Hauptaufgaben des Systems?

- Wer ist an den Aufgaben beteiligt?

- Welche Schritte gehören zur Aufgabenerfüllung?

  => Aufgaben werden als Use Cases (Anwendungsfälle) beschrieben

  => Beteiligte werden als Aktoren festgehalten (können meist aus der Menge der Stakeholder bzw. deren Rollen entnommen werden)

Use Case (Anwendungsfall)

- Use Case beschreibt in der Sprache der Stakeholder, d.h. in natürlicher Sprache, eine konsistente und zielgerichtete Interaktion der nutzenden Person mit einem System, an deren Anfang ein fachlicher Auslöser steht und an deren Ende ein definiertes Ergebnis von fachlichem Wert entstanden ist
- Ein Use Case beschreibt das gewünschte externe Systemverhalten aus Sicht einer nutzenden Person und somit Anforderungen, die das System erfüllen soll
- eine Beschreibung was es leisten muss, aber nicht wie es dies leisten soll
- Unterscheidung in Geschäftsanwendungsfall (business use case) formuliert aus Geschäftssicht (z. B. Vertriebsprozess vom Anfang) und Systemanwendungsfall (system use case) formuliert aus Sicht der durch die neue SW zu lösenden Aufgabe

Business Use Case [OW]

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

2.1.8 Geschäftsanwendungsfall

- Verwandte Begriffe: engl. *business use Case, Geschäftsfall.*

Definition

- Ein Geschäftsanwendungsfall beschreibt einen geschäftlichen Ablauf, wird von einem geschäftlichen Ereignis ausgelöst und endet mit einem Ergebnis, das für den Unternehmenszweck und die Gewinnerzielungsabsicht direkt oder indirekt einen geschäftlichen Wert darstellt.

Beschreibung

- Bei einem Geschäftsanwendungsfall wird die Frage nach der möglichen systemtechnischen Umsetzung noch nicht gestellt, sondern völlig unabhängig davon ganz allgemein aus geschäftlicher Sicht beschrieben.

- Beispiel: Business Use Case „Angebotserstellung“

System Use Case [OW]

2.1.9 Systemanwendungsfall

- Verwandte Begriffe: engl. System use case

Definition

- Ein Systemanwendungsfall ist ein Anwendungsfall, der speziell das für außen stehende Akteure (nutzende Person oder Nachbarsysteme) wahrnehmbare Verhalten eines (Hard-/Software-) Systems beschreibt.

Beschreibung

- Aus UML- und Softwareentwicklungssicht ist der Systemanwendungsfall die normale Form eines Anwendungsfalles. In Abgrenzung zu den verschiedenen Arten von Geschäftsanwendungsfällen beschreibt ein Systemanwendungsfall konkret das Verhalten bzw. den Arbeitsablauf, wie er durch ein System (z. B. Software) unterstützt wird. Dabei wird das äußerlich wahrnehmbare Verhalten beschrieben, also was das System macht, aber nicht wie es dies tut.

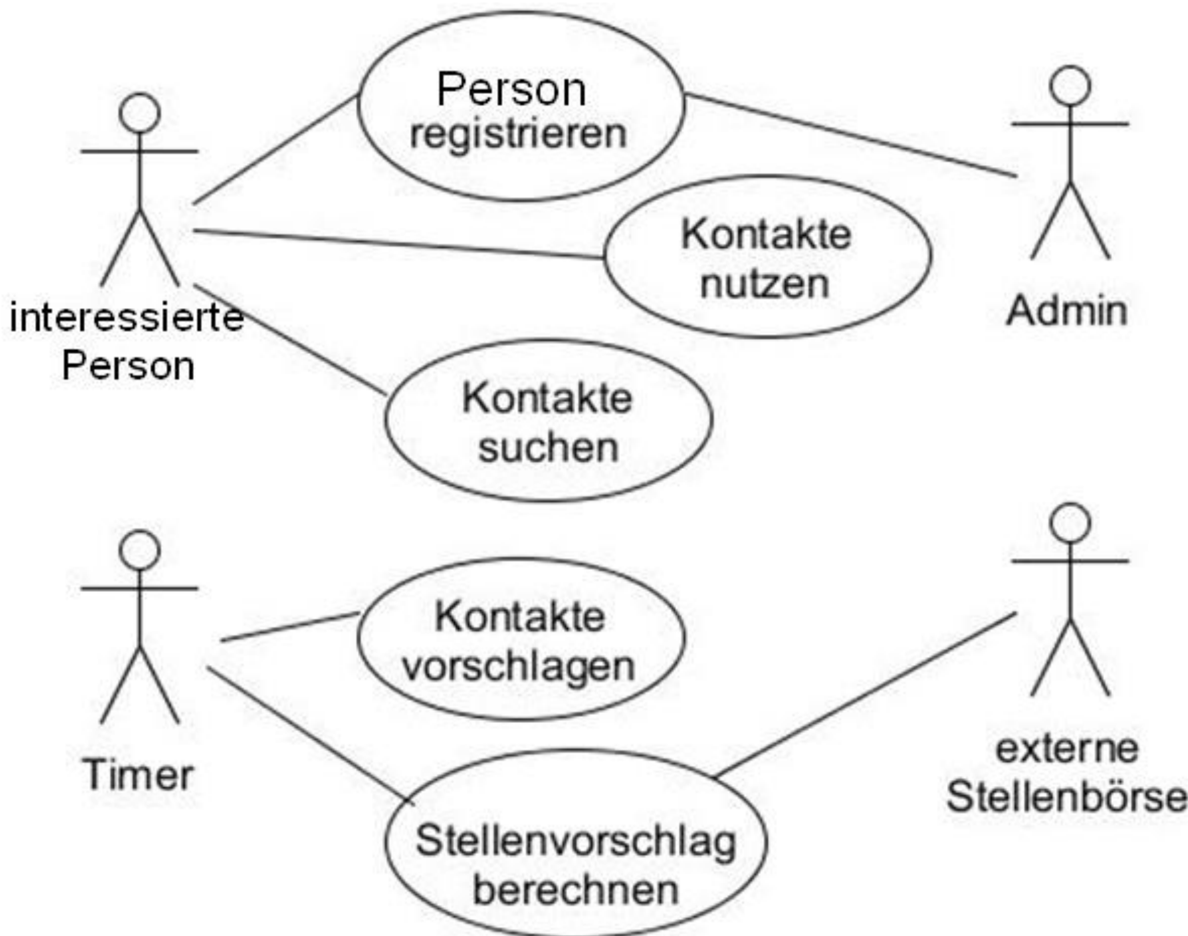
Zusammenhang der Use Case Arten

- Für ein neu geplantes SW-System wird zunächst analysiert, welche Prozesse mit der SW unterstützt werden sollen (Geschäftsprozessmodellierung)
- Oft geht mit Modellierung auch eine Optimierung einher
- Man erhält zentrale Aufgaben, die das SW-System übernehmen soll (Business Use Case)
- Ausgehend davon werden die Aufgaben geplant, die das SW-System unterstützen/ausführen soll, dies sind die System Use Cases
- Häufig gehört zu einem Business Use Case ein System Use Case, d. h. es gibt die gleiche Überschrift, aber eine unterschiedliche Beschreibung (im System Use Case steht die Nutzung des neues SW-Systems im Mittelpunkt)
- Es kann weitere System Use Cases geben, die z. B. die Systemwartung oder neue Analysemöglichkeiten betreffen

Wege zur Use Case-Ermittlung

- moderierter Workshop zentraler Stakeholder
- Beobachtung der Personen, die das bisherige oder ein vergleichbares System nutzen
- Fragebögen
- Interviews
- auftraggebende Person vor Ort im Projekt
- Analyse von Altsystemen und Dokumenten der auftraggebenden Personen
- Simulationsmodelle

Darstellungsbeispiel: Business-Netzwerk



externe Sicht der nutzenden Person auf die Aufgaben des Systems

Aktoren können Personen oder andere Systeme oder interne Auslöser sein

Use Cases können in Teilpaketen strukturiert werden

das zu entwickelnde System tritt nie Aktor auf, kann als Kasten um UC stehen

Systematische Use-Case Ermittlung (1/4)

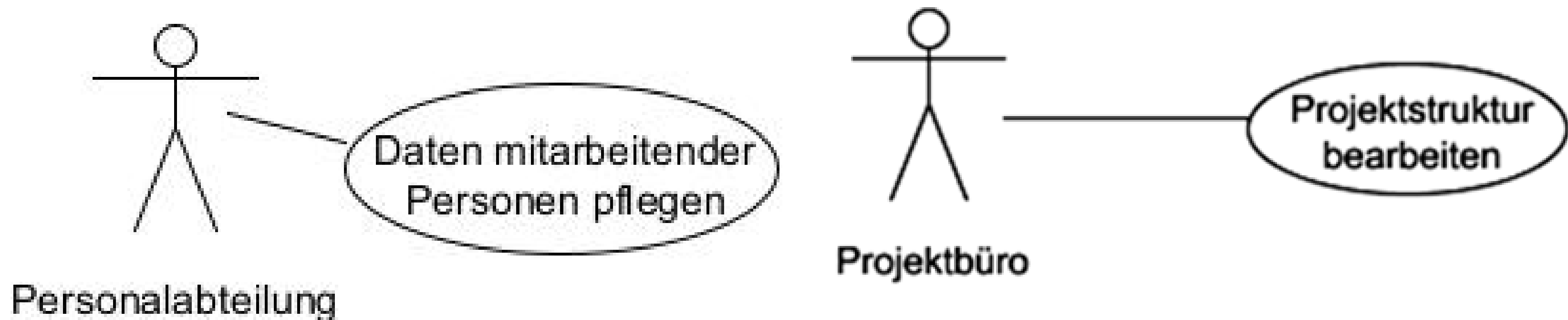
1. Welche Basisinformationen / Objekte sind zu bearbeiten (keine Detailmodellierung, keine Informationen, die aus anderen berechenbar sind)?

   Beispiel (Projektmanagementsystem): Projekte, mitarbeitende Personen

   Prüfe ob neues System Basisinformationen verwaltet oder Sie aus existierenden Systemen stammen

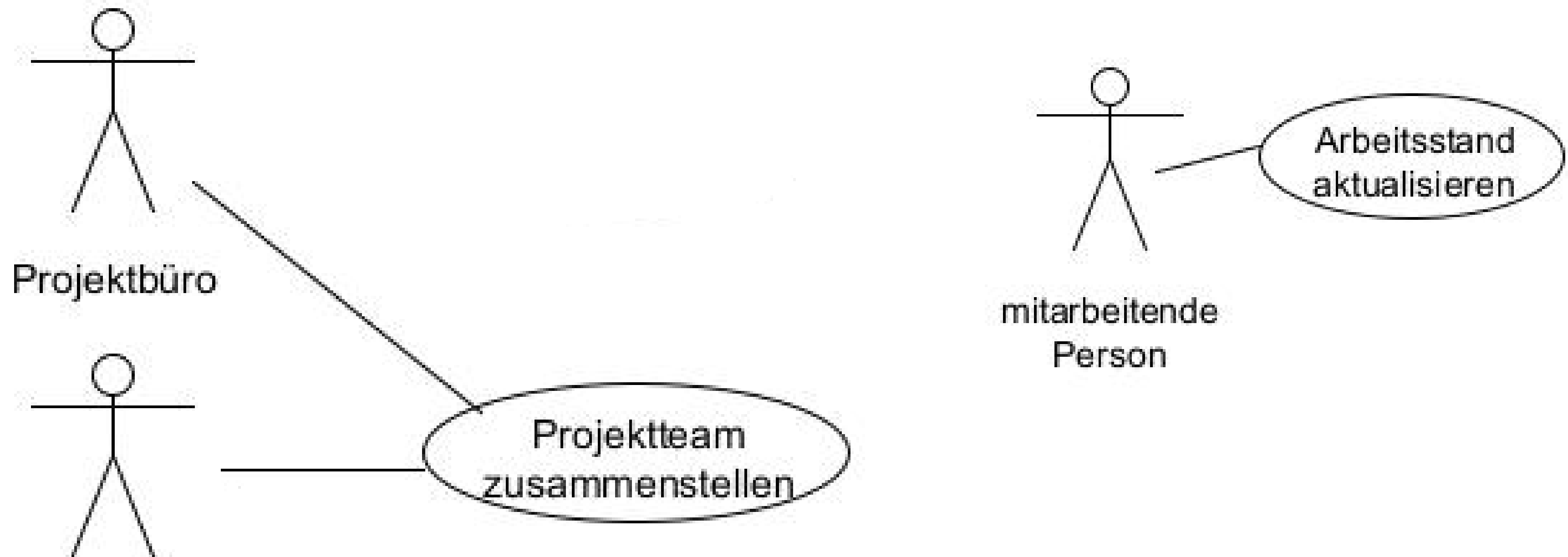
   neues System: Use Case „Basisinformation XY verwalten" gefunden (evtl. in „anlegen", „bearbeiten", „löschen" trennen)

   existierendes System: tritt als Aktor auf, wenn Daten benötigt



Systematische Use-Case Ermittlung (2/4)

2. Welche Prozessinformationen sind zu verwalten, also dynamisch entstehende Daten, Daten zur Verknüpfung von Basisinformationen
   Beispiel: Projektteams, Arbeitsstunden der mitarbeitenden Personen
   Ergänze Use Cases, die die Verknüpfung der Daten herstellen

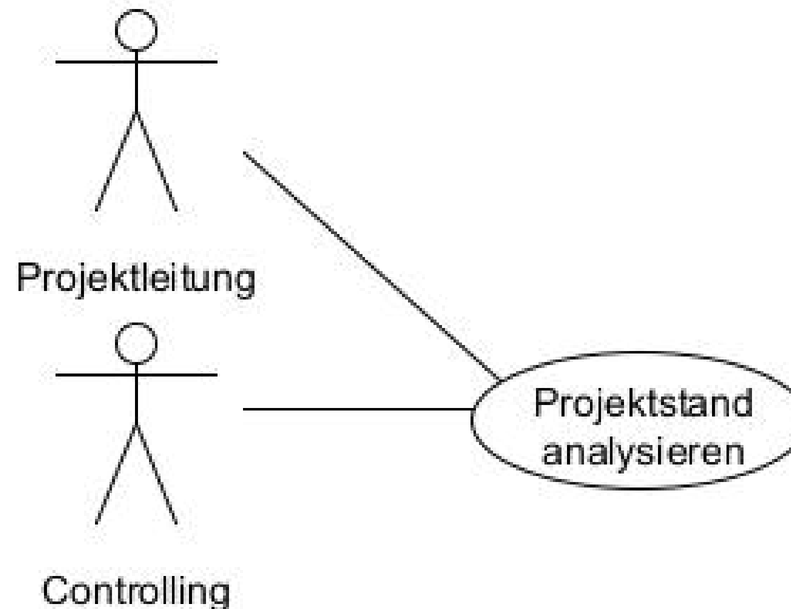


Systematische Use-Case Ermittlung (3/4)

3. Ermittle Funktionalität, die auf Basis der Verarbeitung von Basis- und Prozessinformationen benötigt wird

   abstrakte Beispiele: Entscheidungsprozesse/ Analyseprozesse zur Auswertung (Statistiken, Übersichten)

   Ergänze Use Case für jede der Prozessarten (Art bedeutet, Zusammenfassung eng verwandter Funktionalität)



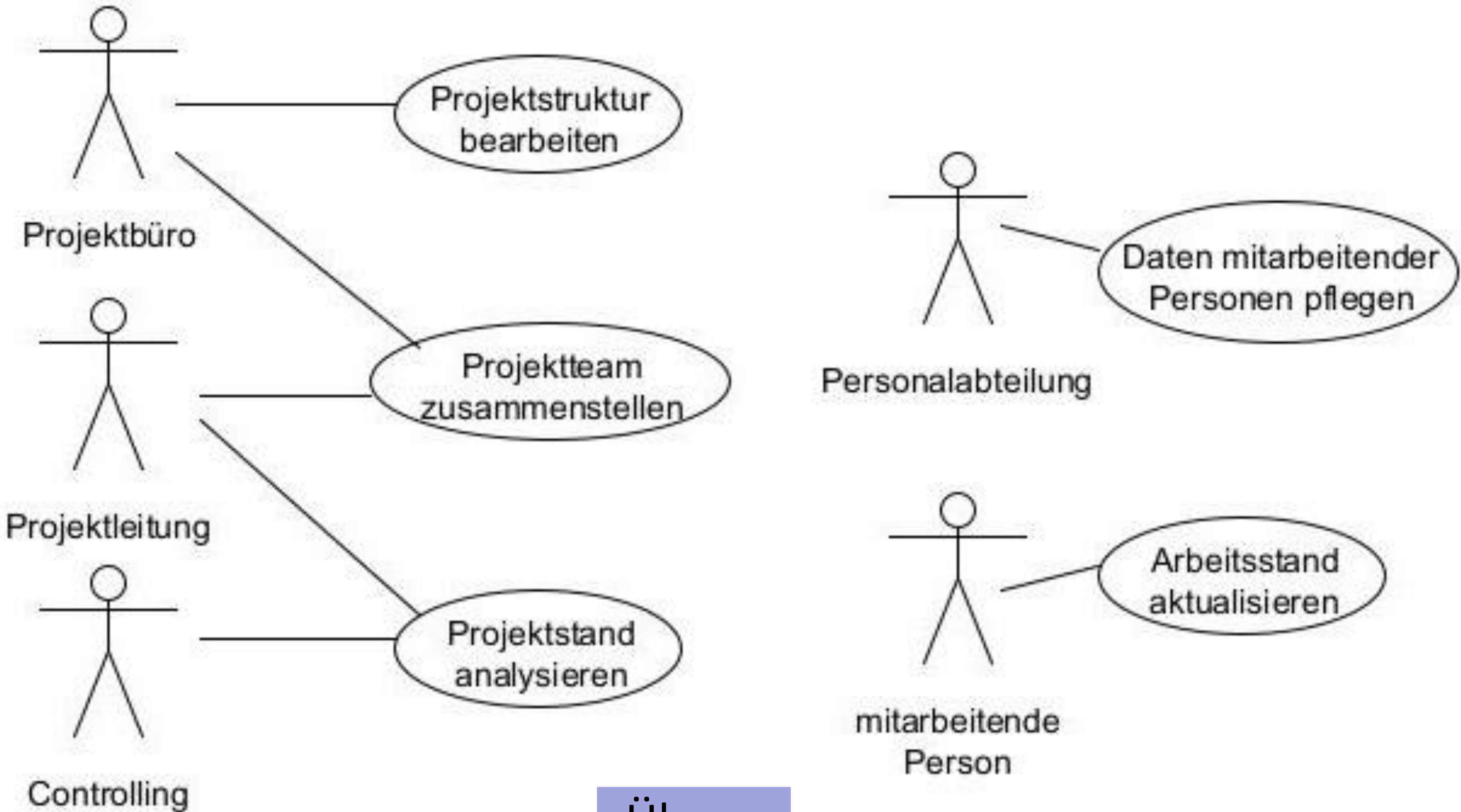
Systematische Use-Case Ermittlung (4/4)

4. Ermittle Use Cases zur reinen Systempflege insofern es besondere Herausforderungen gibt

   abstrakte Beispiele: langfristige Datenhaltung, Systemstart, Systemterminierung

   Zeichne Use Case-Diagramm und ergänze Aktoren (z. B. Stakeholder, genutzte Systeme, Timer) und Dokumentation

Abgeleitetes Use Case-Diagramm



Übung

Use Case-Erstellung genauer

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- Beschreibung eines Use Cases
  - zunächst verbal
  - relativ abstrakt, wird später verfeinert
- Leitfragen für die Ermittlung von Aktoren und Prozessen
  - Welcher Aktor löst Use Case aus?
  - Welche Aktoren sind am Use Case beteiligt?
  - Welche Aufgaben sind im Use Case zu erfüllen?
  - Wer ist verantwortlich für Planung, Durchführung, Kontrolle der Aufgaben?
  - Welche Ereignisse starten Use Case, treten im Use Case auf?
  - Welche Bedingungen sind zu beachten?
  - Was sind die Ergebnisse des Use Cases?
  - Welche Beziehungen gibt es zu welchen anderen Use Cases?

Verfeinerung der Use Case-Dokumentation

Video

4.3

- Im ersten Schritt werden in Use Cases nur Hauptaufgaben des Systems beschrieben
- Zur Dokumentation der Use Cases gehört zunächst nur eine grobe kurze Beschreibung (maximal 5 Sätze) des Inhalts
- Im nächsten Schritt wird dieser Inhalt konkretisiert. Dabei ist es sinnvoll, auf eine Dokumentationsschablone zurück zu greifen (oder eine für das Projekt zu entwickeln)
- Im ersten Schritt der Beschreibungsentwicklung wird nur der typische Ablauf des Use Cases ohne Alternativen, dann mit Alternativen beschrieben

Dokumentationsschablone für Use Cases (1/3)

Name des Use Case	1	kurze prägnante Beschreibung, meist aus Verb und Nomen
Nummer	1	eindeutige Nummer zur Verwaltung, sollte von der eingesetzten Entwicklungsumgebung vergeben werden
Paket	2	bei sehr komplexen Systemen können Use Cases in Teilaufgabenbereiche zusammengefasst werden, diese Bereiche können in der UML als Pakete dargestellt werden
Erstellung	1	wer hat den Use Case erstellt und wer mitgearbeitet
Version	1	aktuelle Versionsnummer, möglichst mit Änderungshistorie, wer hat wann was geändert
Kurzbeschrei-bung	1	kurze Beschreibung, was mit dem Use Case auf welchem Weg erreicht werden soll,
beteiligte Aktoren (Stakeholder)	1	welche Aktoren sind beteiligt, wer stößt den Use Case an

Dokumentationsschablone für Use Cases (2/3)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

Fachverant-wortlich	1	wer steht auf fachlicher Seite für Fragen zum Use Case zur Verfügung und entscheidet auf Auftraggebenderseite für die Software über den Inhalt
Referenzen	2	Nennung aller Informationen, die bei der späteren Ausimplementierung zu beachten beziehungsweise hilfreich sind, können Verweise auf Gesetze, Normen oder Dokumentationen existierender Systeme sein
Vorbedingungen	2	was muss erfüllt sein, damit der Use Case starten kann
Nachbedin-gungen	2	wie sieht das mögliche Ergebnis aus, im nächsten Schritt sind auch die Ergebnisse alternativer Abläufe zu berücksichtigen
typischer Ablauf	2	welche einzelnen Schritte werden im Use Case durchlaufen, dabei wird nur der gewünschte typische Ablauf dokumentiert
alternative Abläufe	3	welche Alternativen existieren zum typischen Ablauf

Dokumentationsschablone für Use Cases (3/3)

Kritikalität	3	wie wichtig ist diese Funktionalität für das Gesamtsystem
Verknüpfungen	3	welche Zusammenhänge bestehen zu anderen Use Cases
funktionale Anforderungen	4	welche konkreten funktionalen Anforderungen werden aus diesem Use Case abgeleitet
nicht-funktionale Anforderungen	4	welche konkreten nicht-funktionalen Anforderungen werden aus diesem Use Case abgeleitet

- Nummer gibt Iteration an, in der das Feld gefüllt wird
- typischer und alternative Abläufe werden jetzt genauer betrachtet
- funktionale und nicht-funktionale Anforderungen weiter hinten in diesem Abschnitt

Beispielbeschreibung (1/2)

Name des Use Case	Projektstruktur bearbeiten
Nummer	U1
Paket	-
Erstellung	Achmed Analytiker
Version	1.0, 30.01.2019, Erstellung
Kurzbeschrei-bung	Im Projektbüro tätige Personen haben die Möglichkeit, Projekte mit Teilprojekten anzulegen und zu bearbeiten.
beteiligte Aktoren (Stakeholder)	Projektbüro (startet Use Case durch Auswahl der Funktionalität im zu erstellenden System)
Fachverantwortlich	Lisa Leitung (zentrale Ansprechpartnerin des auftraggebenden Unternehmens)
Referenzen	Handbuch zur Führung von Projekten des auftraggebenden Unternehmens

Beispielbeschreibung (2/2)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

Vorbedingungen	Die Software ist vollständig installiert und wurde gestartet.
Nachbedingungen	Neue Projekte und Teilprojekte sowie Änderungen von Projekten und Teilprojekten wurden vom System übernommen.
typischer Ablauf	1. Nutzende Person wählt Funktionalität zur Bearbeitung von Projektstrukturen 2. Nutzende Person legt Projekt mit Projektstandarddaten an 3. Nutzende Person ergänzt neue Teilprojekte 4. Nutzende Person verlässt Funktionalität
alternative Abläufe	Die nutzenden Person kann existierendes Projekt auswählen, Die nutzenden Person kann Daten eines Teilprojekts ändern
Kritikalität	sehr hoch, System macht ohne Funktionalität keinen Sinn

Hinweise zu Use Cases (1/2)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- Verwende für den Use Case eine sinnvolle Bezeichnung, die mindestens aus einem echten Substantiv und einem aktiven Verb ("Antrag erfassen") oder dem zugehörigen Gerundium ("Antragserfassung") besteht!
- Definiere zuerst den fachlichen Auslöser und das fachliche Ergebnis, um Anfang und Ende des Use Cases festzulegen!
- Formuliere den Use Case so abstrakt wie möglich und so konkret wie nötig!
- Betreibe *zunächst* keine Zerlegung in abgeleitete, sekundäre Use Cases!
- Standardisiere die Sprache in den Use Cases!

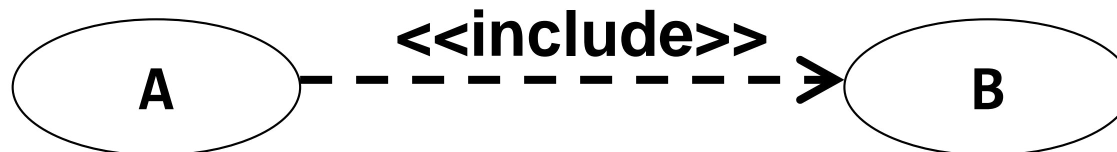
Hinweise zu Use Cases (2/2)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- Use Cases eignen sich nicht zur funktionalen Zerlegung, d.h. ein Use Case beschreibt keine einzelnen Schritte, Operationen oder Transaktionen (bspw. "Vertrag drucken", „Auftrags-Nr. erzeugen" etc.), sondern relativ große Abläufe (bspw. "Neuen Auftrag aufnehmen")
- Es wird keine Ablaufreihenfolge definiert, hierzu gibt es andere Ausdrucksmittel, z.B. Aktivitätsdiagramme
- Use Cases belassen das Sprachmonopol beim Stakeholder, wodurch die Use Cases angreifbarer und besser kritisierbar werden
- Bereits hier sinnvoll: Glossar anlegen (Begriffe und Prozesses definieren)

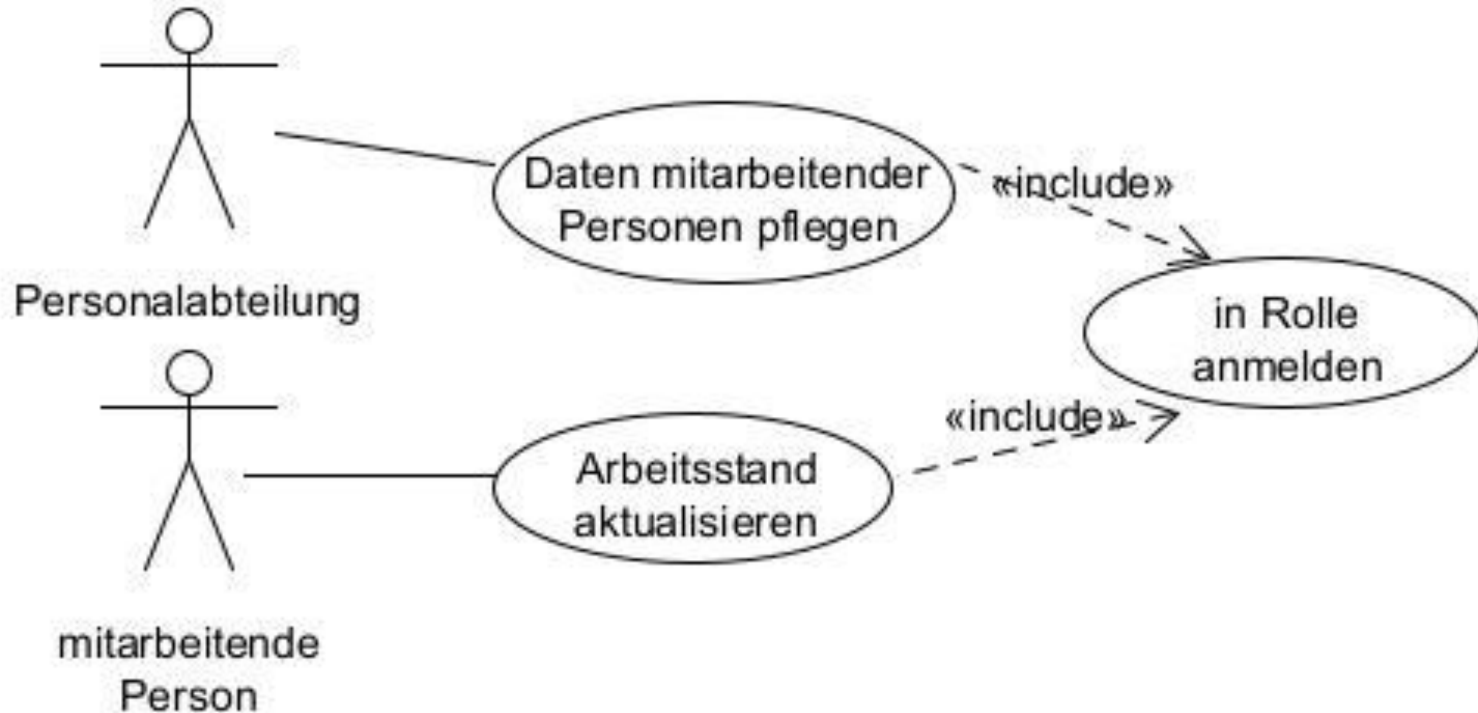
Analyse von Use-Case-Dokumentationen

- es kann passieren, dass identische Abläufe mehrfach beschrieben werden
- diese (nicht trivialen) Abläufe können als eigene Use Cases ausgegliedert werden; man sagt dann „ein Use Case nutzt einen anderen Use Case“
- UML-Darstellung:



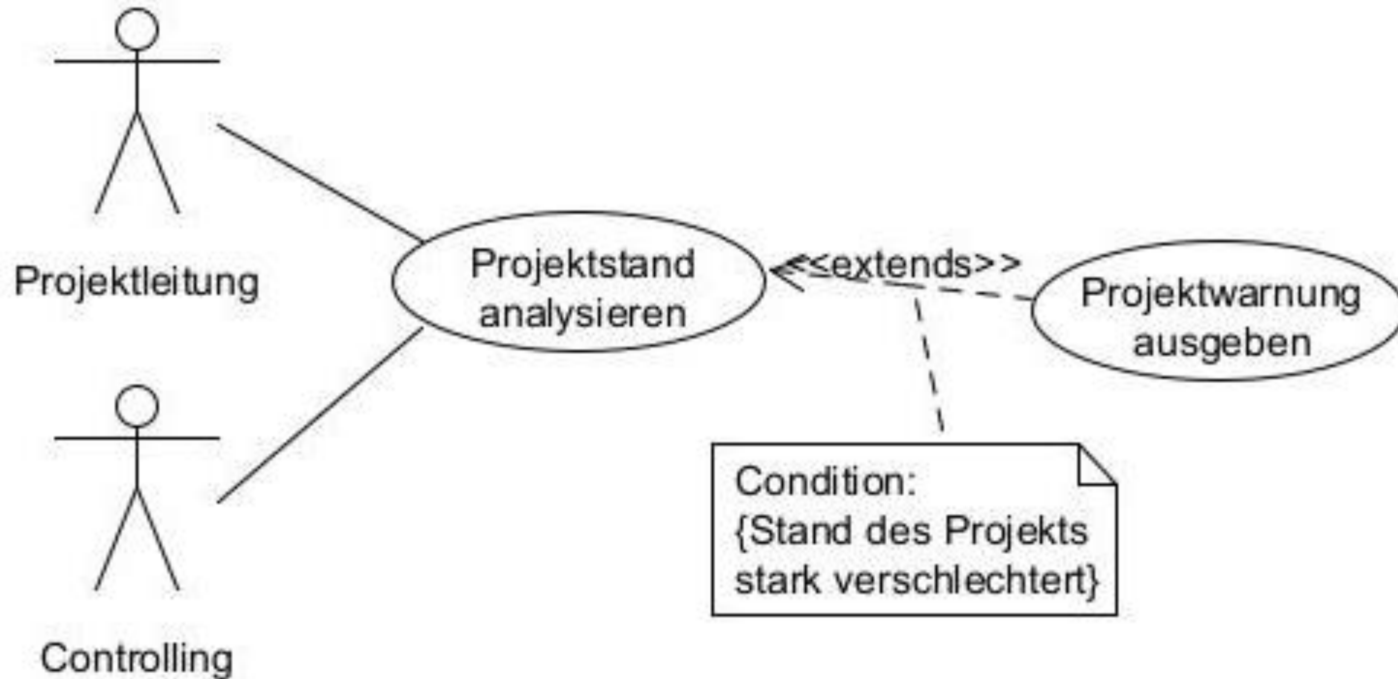
- In spitzen <<Klammern>> stehen sogenannte Stereotypen, mit denen man UML-Elementen zusätzliche Eigenschaften zuordnen kann

Beispiel zu <<include>>



<<extend>>

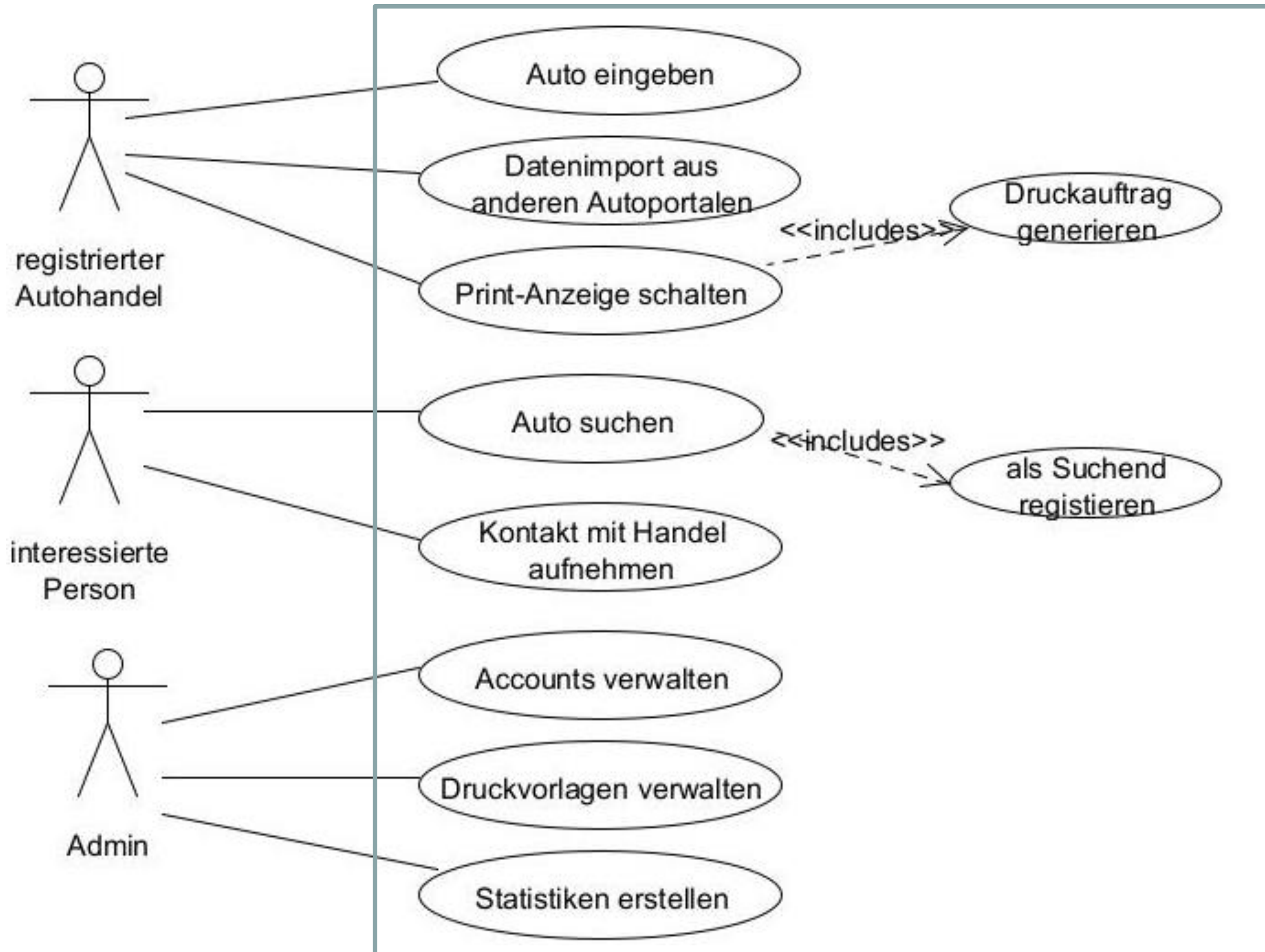
- Seltene Variation des erweiterten Use Cases
- Wird nur unter bestimmter Bedingung ausgeführt, z. B. Sonderfall oder Fehlerbehandlung
- eigentlicher Use Case nicht durch Spezialfälle überfrachtet



Hinweis zu <<include>>, <<extend>> (persönlich)

- <<include>> ist ein sehr nützlicher Stereotyp, der die Dokumentation verkürzen kann
- Gerade bei in der Modellierung unerfahrenen auftraggebenden Unternehmen sollte <<include>> zunächst verheimlicht werden, da sonst funktionale Zerlegungen in Bäumen das Ergebnis sind
- <<include>> wird dann bei der Dokumentation und späteren Verfeinerung bei der Umstrukturierung der Use Cases als Optimierung eingesetzt
- Hinweis: <<extend>> und weitere nicht erwähnte Möglichkeiten werden hier ignoriert, da es auftraggebende Unternehmen, genauer Personen ohne IT-Background, eher verwirrt

weiteres Use Case – Diagramm: Online-Autobörse



Beschreibung verschiedener Abläufe

Video

- Bei Projekten mit enger Bindung (z.B. bei engen Beziehungen zwischen AG und IT-Abteilung bei Inhouse-Projekten) können Use Cases (oder User Stories) als Anforderungsdokumentation ausreichen, wenn das Projekt in kleinen Iterationen und der Möglichkeit eines großen Einflusses der auftraggebenden Partei entwickelt wird
- Oftmals ist die Beschreibung der Use Cases aber zu ungenau, gerade bei der Darstellung typischer und Alternativer Abläufe stellt sich die rein sprachliche Beschreibung als recht aufwändig heraus
- Da die UML eine graphische Sprache ist, stellt sie auch für Ablaufbeschreibungen eine grafische Darstellungsmöglichkeit, nämlich *Aktivitätsdiagramme,* zur Verfügung

Modellierungsrichtlinie für Aktivitätsdiagramme

Modelliere zu jedem Use Case genau ein Aktivitätsdiagramm

- Mache aus den Use Case-Schritten Aktionen
- Zerlege die Aktionen ggfls. mit einem Aktivitätsdiagramm, so dass sie stets genau einen fachlichen Arbeitsschritt repräsentieren
- Ergänze den Ablauf um alle bekannten fachlichen Ausnahmen, fachlichen Fehler und fachlichen Ablaufvarianten, so dass das Diagramm eine vollständige Beschreibung aller zulässigen Ablaufmöglichkeiten darstellt

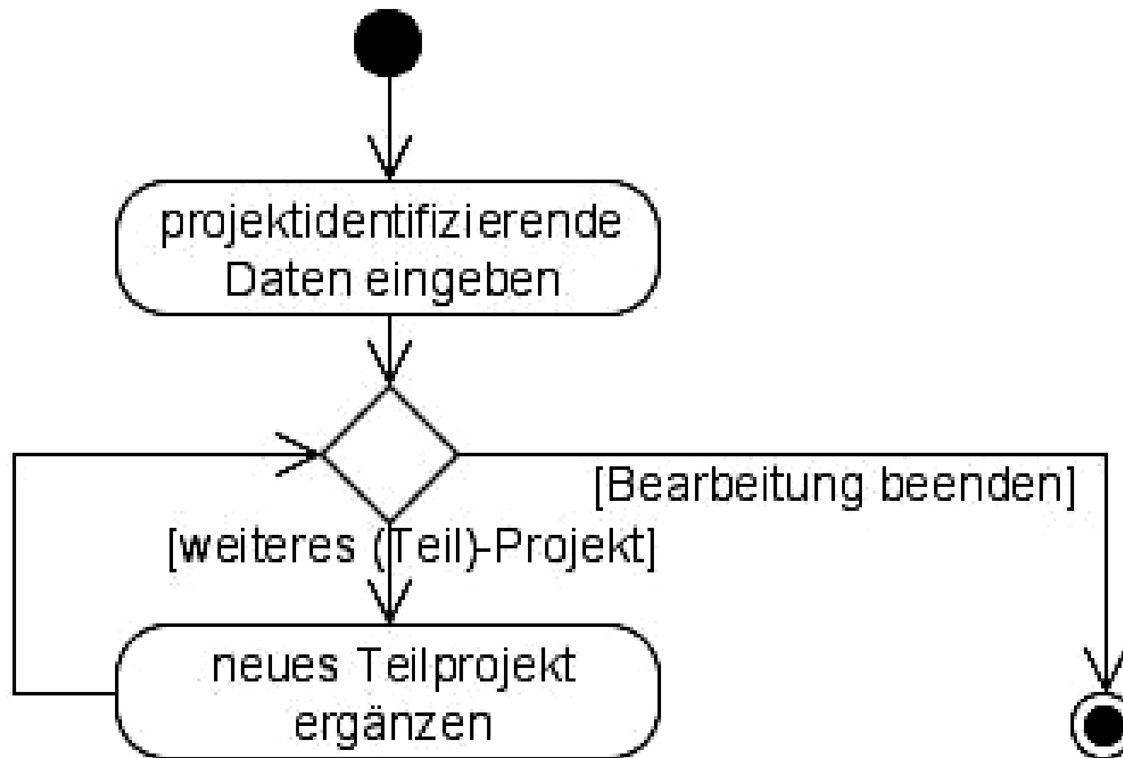
(sinnvoll jetzt oder später) Modelliere den Objektfluss:

- Beschreibe zu jeder Aktion die vorausgesetzten (zu verarbeitenden) und resultierenden (erzeugten oder veränderten) Geschäftsobjekte (Produkte).
- Unterscheide, bei welchen ausgehenden Transitionen bzw. Bedingungen welche Objekte bzw. Objektzustände resultieren

Aktivitätsdiagramm mit typischen Ablauf

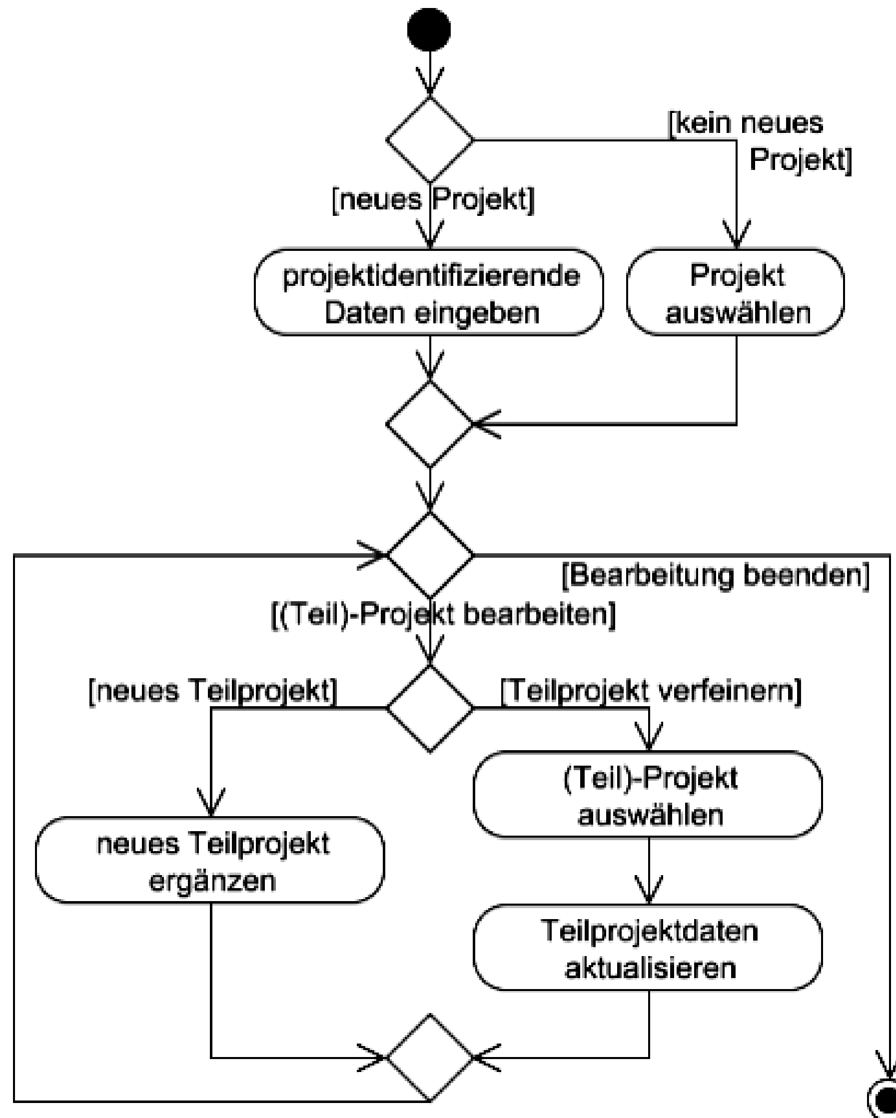
HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

Use Case: Projektstruktur bearbeiten



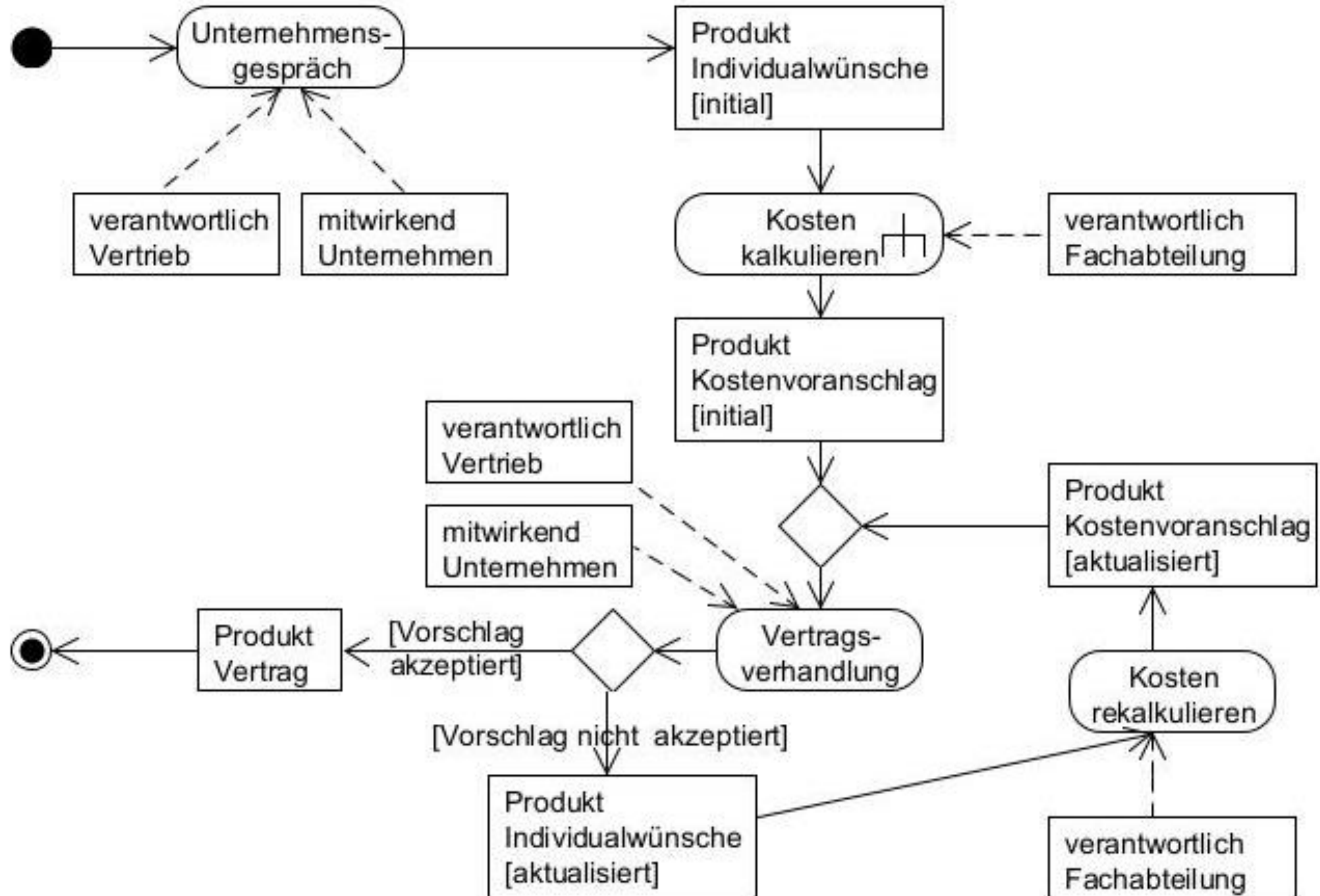
Anmerkung: typischer Ablauf ist immer einfache Sequenz von Aktionen, Ausnahme wie hier: einfache Schleifen

Aktivitätsdiagramm um Alternativen ergänzt

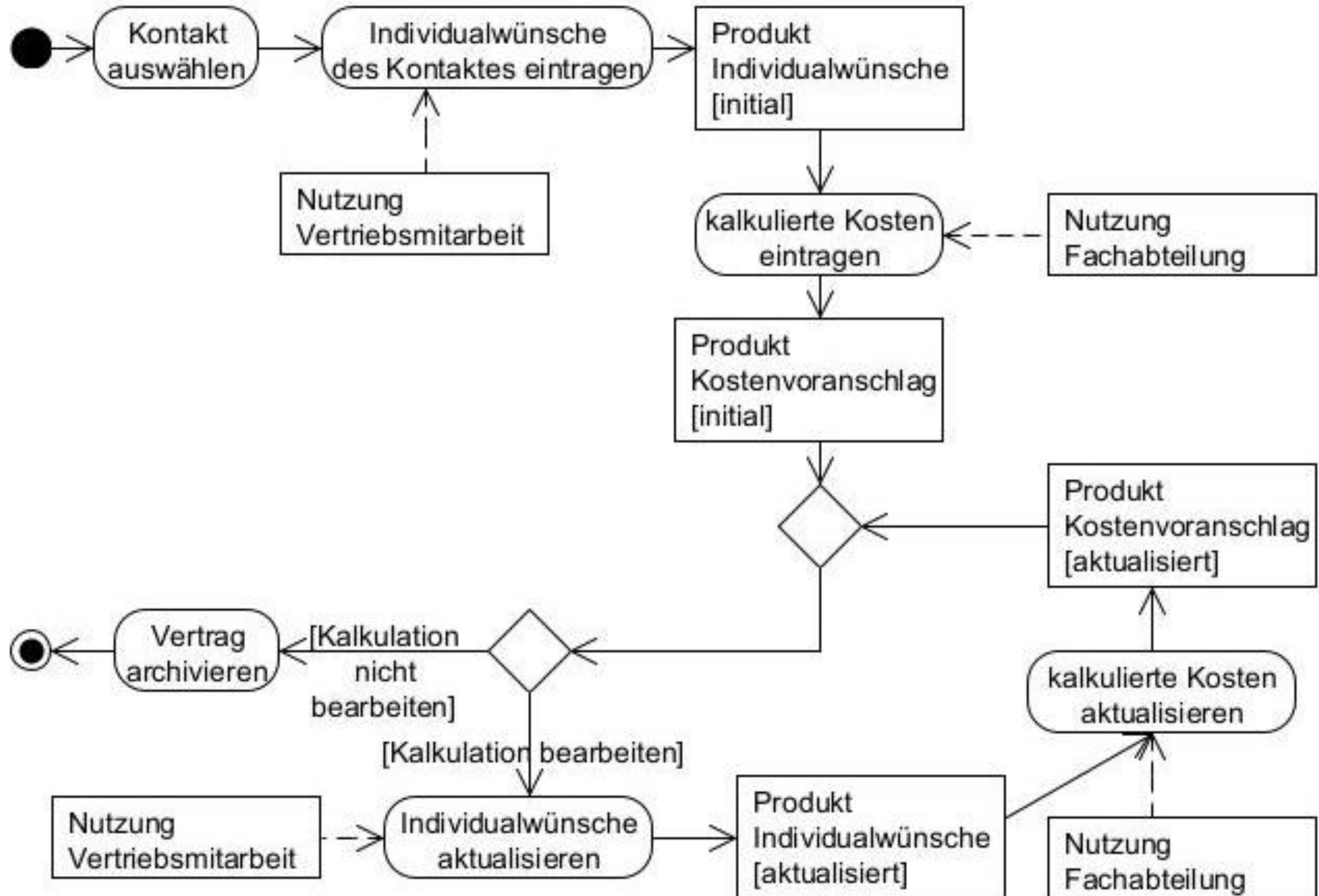


Erinnerung: Modellierung aus Business-Sicht

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES



Modellierung aus System-Sicht



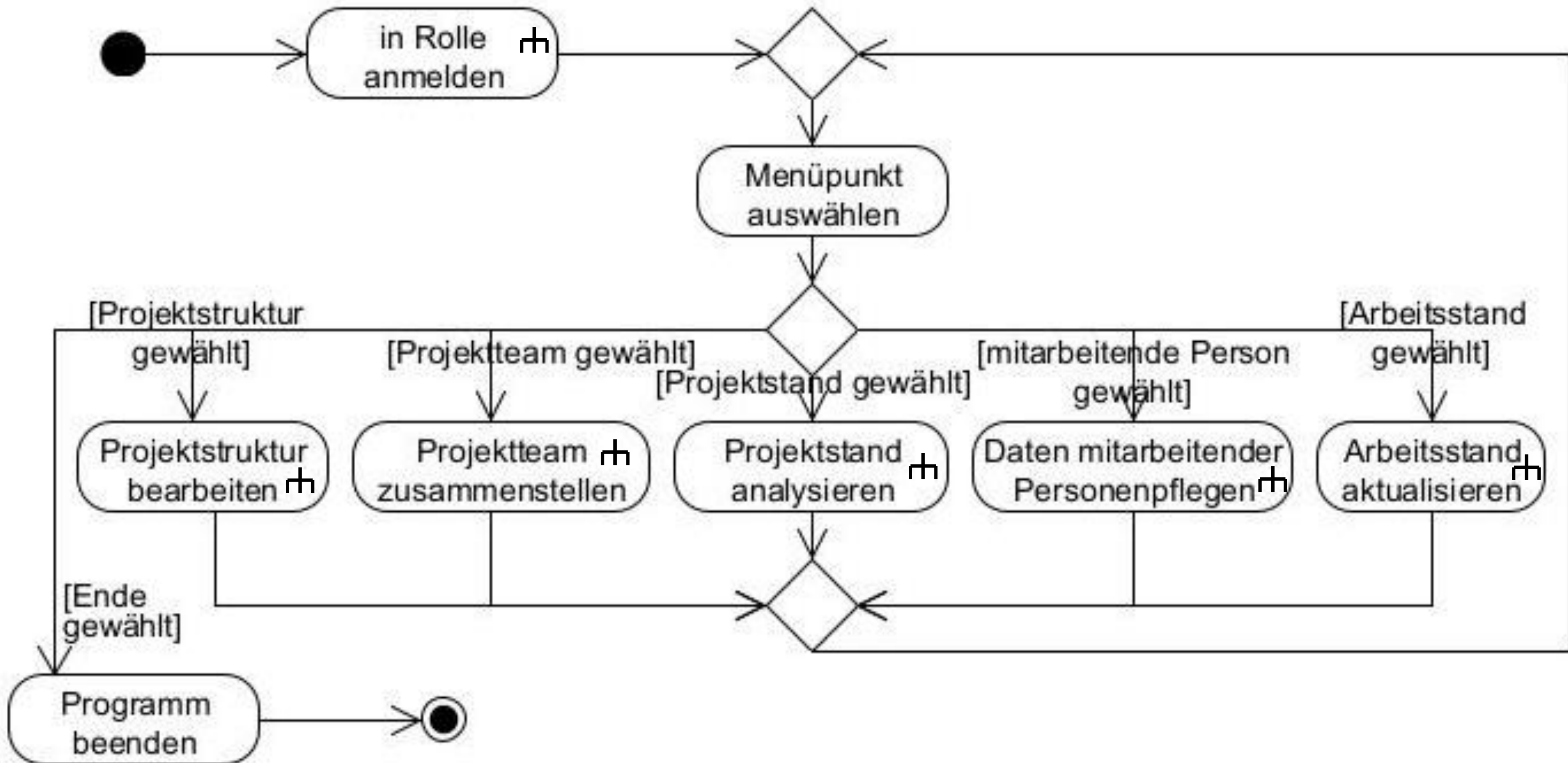
n+1 Aktivitätsdiagramme (1/2)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- typisch: zu jedem Use Case ein Aktivitätsdiagramm (ggfls. mit Verfeinerung)
- Ansatz ausreichend, wenn keine zentrale Steuerung (z. B. WebServices)
- Für zentrale Steuerung wird ein zusätzliches Aktivitätsdiagramm benötigt, dass diesen Ablauf zeigt (z. B. GUI mit Nutzungsauswahl)

n+1 Aktivitätsdiagramme (2/2)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES



Formulierung von Anforderungen

4.4

Video

- Analog zu Use Cases sind Aktivitätsdiagramme zu dokumentieren: was unter Nutzung welcher Hilfsmittel unter Berücksichtigung welcher Nebenbedingungen gilt
- Beschreibungen können oft unvollständig oder unklar formuliert sein, sind zu prüfen
- Statt Fließtextdokumentation von Aktivitätsdiagrammen, kann eine Darstellung von systematisch abgeleiteten textuellen Anforderungen sinnvoll sein
- Man benötigt Ansatz, Texte möglichst präzise zu formulieren

Sprache als Darstellungsmittel

Formulierte Anforderungen

- sind in natürlicher Sprache verfasst
- gewissen Prozessen bei der Entstehung unterworfen

Entstehungsprozesse

- verändern/verfälschen die beabsichtigte Bedeutung einer Anforderung
- hat jeder Mensch, sind regelgeleitet

Glossar

- Zentrales Hilfsmittel der Anforderungsanalyse
- Aufbau: Fachbegriff – Erklärung
- Wichtig: Fachbegriff kann auch Halbsatz sein
- Kann detaillierte Erklärungen oder Referenzen auf Fachliteratur enthalten
- muss von auftraggebenden und entwickelnden Personen verstanden werden

Arbeitspaket	Synonym für Projektaufgabe
Projektaufgabe	Nicht weiter zerlegte Aufgabe mit zugewiesenen Rollen zur Bearbeitung; gleiche Ausgangsdaten wie Projekt
Projektausgangs-daten	automatisch vergebene eindeutige Projektnummer, Projektname, geplanter Start- und Endtermin, geplanter Aufwand

Probleme mit natürlich-sprachlichen Formulierungen

- Hauptprozesse der menschlichen Modellbildung
  - Tilgung
  - Generalisierung
  - Verzerrung (z. B. durch Nominalisierung)
- Problem: Anforderungen werden für Menschen mit anderer Modellbildung (da andere Erfahrungen) unsauber formuliert

- In Prosatexten sind Wiederholungen unerwünscht; bei Anforderungen müssen immer die gleichen Worte für den gleichen Sachverhalt genutzt werden

Definition: Tilgung

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- Tilgung ist ein Prozess, durch den wir unsere Aufmerksamkeit selektiv bestimmten Dimensionen unserer Erfahrungen zuwenden und andere ausschließen. (Bandler/Grinder)

- Beispiel: Die Fähigkeit des Menschen, In einem Raum voller sprechender Menschen alle anderen Geräusche auszuschließen oder auszufiltern, um der Stimme einer bestimmten Person zuzuhören.

- problematisch für Anforderungen: implizite Annahmen, unvollständige Vergleiche

Beispiele für Tilgungen (1/2)

- Grundstruktur: Manche Prozessworte (Verben und Prädikate) implizieren zwei oder mehr Substantivargumente
- Sprachliche Vertreter
  - Eingeben: Wer? Was? Wie? Wo? Wann?
  - Anzeigen: Was? Wo? In welcher Weise? Wann?
  - Übertragen: Wer? Was? Von wo? Wohin? Wann?
  - „Die Auszahlungsmöglichkeit soll überprüft und die Auszahlung verbucht werden"
  - Überprüfen: Wer überprüft? Was wird überprüft? Nach welchen Regeln wird überprüft? Wann wird überprüft? Wie?
  - Verbuchen: Wer verbucht? Was wird verbucht? Wann wird es verbucht? Wie?

Beispiele für Tilgungen (2/2)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- Grundstruktur: Der Bezugspunkt. die Messbarkeit und die Messgenauigkeit für einen Komparativ oder Superlativ fehlt.
- Sprachliche Vertreter: Adjektiv + Endung "-er/en", "-ste" oder "more", "less", "least", oder "weniger", "mehr"
- In beiden Sprachen: Adjektive wie leicht, easy, schwer, complicated, ...
- Für durchschnittlich große Menschen soll das Display im normalen Bedienabstand gut lesbar sein.
- Die Eingabe des angeforderten Geldbetrages soll vom System durch eine intuitive Nutzungsführung so unterstützt werden, dass Fehleingaben minimiert werden.
  - Kann man den Sachverhalt überhaupt messen?
  - Ist der Bezugspunkt des Vergleiches angegeben?
  - Mit welcher Messgenauigkeit wird gemessen?

Definition: Generalisierung

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- Generalisierung ist der Prozess, durch den Elemente oder Teile eines persönlichen Modells von der ursprünglichen Erfahrung abgelöst werden, um dann die gesamte Kategorie, von der diese Erfahrung ein Beispiel darstellt, zu verkörpern. (Bendler/Grindler)

- Beispiel: Ein Kind verbrennt sich an einer heißen Herdplatte die Hand. Es sollte für sich die richtige Generalisierung aufstellen, dass es schmerzhaft ist auf heiße Herdplatten zu fassen.

- problematisch für Anforderungen: Universalquantoren, unvollständige Bedingungen

Generalisierung durch Universalquantoren

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

Universalquantoren

- Grundstruktur: Menge an Objekten wird zusammengefasst
- Sprachliche Vertreter:
  - Im Deutschen: nie, immer, kein, jeder, alle, ...
  - Im Englischen: never, ever, not, each, always, ...
- Frage:
  - Wirklich alle/jede, immer/nie? Gibt es keine Ausnahme?
  - Achtung! Auch Sätze ohne Universalquantoren überprüfen, die keine Angaben über die Häufigkeit enthalten!

Beispiele für Generalisierungen

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- Jede Auszahlung soll für die Rückverfolgbarkeit zusätzlich mit einem Zeitstempel etikettiert werden.
  - Wirklich jede Auszahlung?

- Das System soll eine Sicherung von aufgezeichneten Auszahlungsdaten auf ein externes Speichermedium ermöglichen.
  - Durch jede Person? Immer? Aller Auszahlungsdaten?

Definition: Verzerrung

- Verzerrung ist der Prozess, etwas mittels Überlegungen, Fantasie oder Wünschen, so umzugestalten, dass ein neuer Inhalt oder eine neue Bedeutung entsteht. (Dörrenbacher)

- Beispiel: Behauptung, dass auf A dann B folgt oder Gedankenlesen
  - Da jemand zu spät ist, ist das Projekt gefährdet
  - Ich denke, der mag mich nicht
  - Er sollte wissen, wie ich mich jetzt fühle

Verzerrung: Beispiele und Analyse

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- Die nutzende Person muss zunächst sein Login und dann sein Passwort eingeben.
- Der nutzenden Person muss am Anfang immer die Übersichtsseite gezeigt werden.
- Die nutzende Person muss eingeloggt sein, um die Übersicht zu sehen.
- „Das muss genau wie Word aufgebaut sein“

- Was führt zur Annahme, dass diese Reihenfolgen notwendig sind?
- Was würde sich bei einer anderen Reihenfolge oder Verlassen einer Einschränkung ändern?
- Welche Eigenschaften von Word sind wichtig; warum muss es so sein

Verzerrung durch Nominalisierung

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- Grundstruktur: Ein Prozesswort (Verb oder Prädikat) wird zu einem Ereigniswort (Substantiv oder Argument) umgeformt.
- Dadurch wird ein Vorgang zu einem Ereignis und viele vorgangsrelevante Informationen gehen verloren.
- Es ist möglich, dass sich die Bedeutung der Aussage dadurch ändert
  - Die Berechtigung für die Administration des Geldautomaten
  - Die Auszahlung wird nach der Buchung durchgeführt
  - Wer? zahlt wann? Wem? Was? Unter Einhaltung welcher Regeln? Mit welcher Zuverlässigkeit? Mit welcher Verfügbarkeit?
  - Wer? bucht wann? Was? Wohin? Unter Einhaltung welcher Regeln? Mit welcher Zuverlässigkeit? Mit welcher Verfügbarkeit?

Erkennen von Nominalisierungen

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

Fragen/Vorgehen:

- Intuition, Sprachgefühl
- Suche nach ähnlichem Prozesswort
- Sprachtest durch Einsetzen in "ein(e) andauernde(r) …". Wahre Substantive passen nicht in diese Aussage

Beispiele:

- Bei der Auswahl der Auszahlungsfunktion soll die …
- der Anzeige, Nutzungsführung, Bestätigung, ...
- die Eingabe, Erfassung, ...
- das Ereignis, die Meldung, ...
- die Buchung, Ausgabe, Prüfung, ...

Anmerkung: Nominalisierung wird oft auch als Tilgung angesehen
http://nlpportal.org/nlpedia/wiki/Metamodell

Entwicklung strukturierter Anforderungen

- ein Ansatz zu qualitativ hochwertigen Anforderungen: erste Version erstellen und dann Textqualität schrittweise verbessern
- Alternative: „von Anfang an“ hochwertige Anforderungen zu schreiben
- Dieser Ansatz kann durch Anforderungsschablonen unterstützt werden, die den Satzbau von Anforderungen vorgeben (vorgestellter Ansatz folgt [RS])
- Man beachte, bereits erwähnte Ausdrucksprobleme auch in diesem Ansatz noch relevant

Charakterisierung von Systemaktivitäten

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

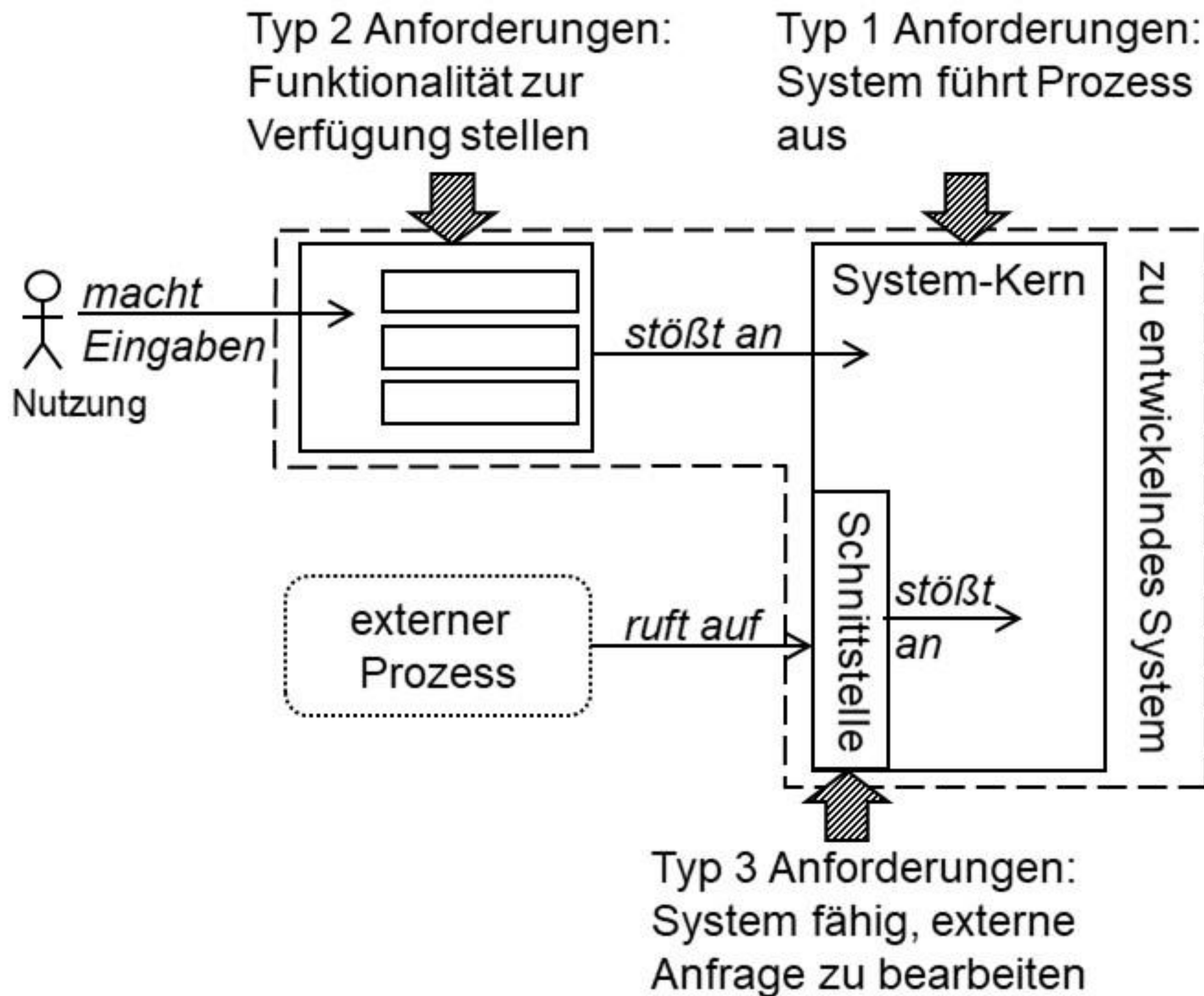
- Selbständige Systemaktivität:

  Das System *führt* den Prozess *selbständig* durch.
- Nutzungsinteraktion:

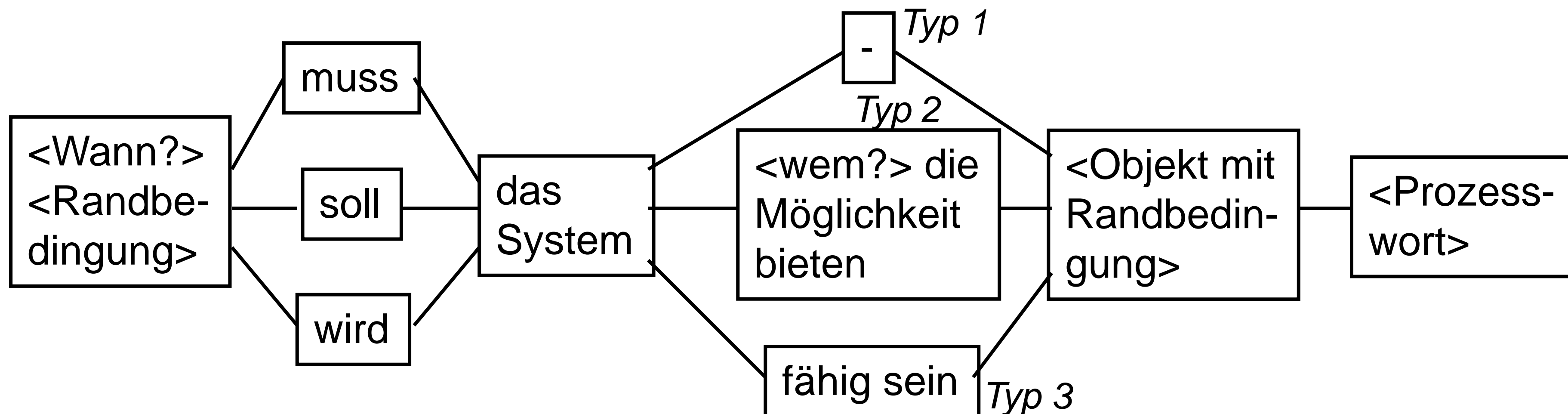
  Das System *stellt* der nutzenden Person die Prozessfunktionalität *zur Verfügung*.
- Schnittstellenanforderung:

  Das System führt einen Prozess in *Abhängigkeit von einem Dritten* (zum Beispiel einem Fremdsystem) aus, ist an sich passiv und wartet auf ein externes Ereignis
- Für jede dieser Systemaktivitäten gibt es eine Schablone
- Frage: Werden Systemaktivitäten so in disjunkte Klassen aufgeteilt?

Visualisierung der Systemaktivitäten



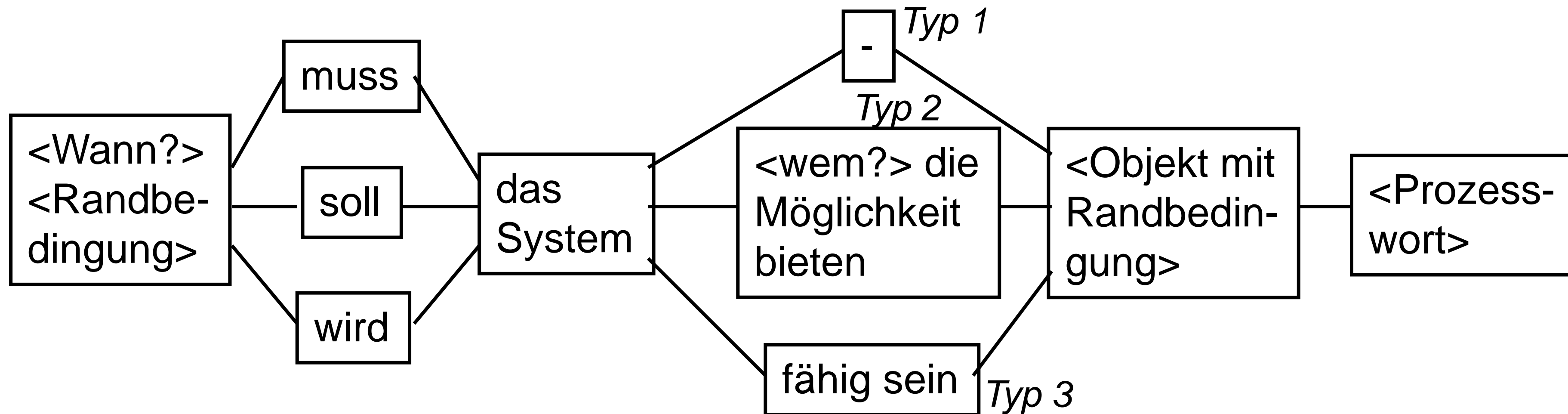
Anforderungsformulierung (Rupp-Schablone)



Typ 1: Selbständige Systemaktivität, System führt Prozess selbständig durch, z. B. Berechnung des bisherigen Aufwandes eines Projekts durch Abfrage aller Teilprojekte und Ergebnisanzeige
Typ 2: Nutzungsinteraktion, System stellt der nutzenden Person die Prozessfunktionalität zur Verfügung, z: B. Verfügbarkeit eines Eingabefeldes, um den Projektdaten einzugeben
Typ 3: Schnittstellenanforderung, d. h. das System führt einen Prozess in Abhängigkeit von einem Dritten (zum Beispiel einem Fremdsystem) aus, ist an sich passiv und wartet auf ein externes Ereignis, z. B. Anfrage einer anderen Bürosoftware nach einer Übersicht über die laufenden Projekte annehmen

Typ 1: Selbständige Systemaktivität

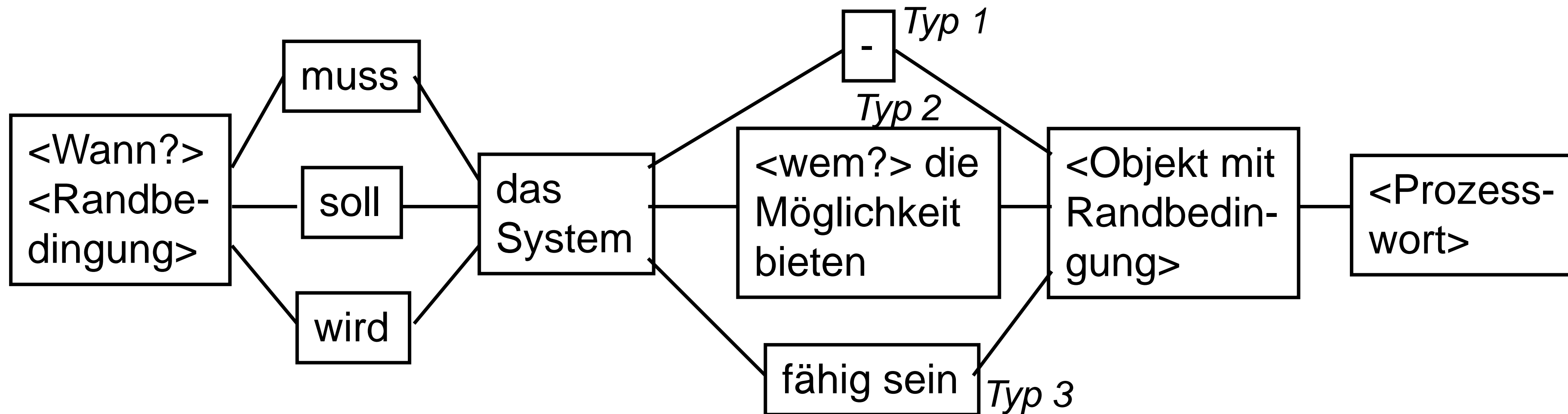
HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES



Nach Abschluss der Eingabe (mit „Return“-Taste oder Bestätigungsknopf) bei der Bearbeitung von Daten muss das System neu eingegebene Daten in seine permanente Datenhaltung übernehmen. [„Daten“ im *Glossar* konkretisieren]

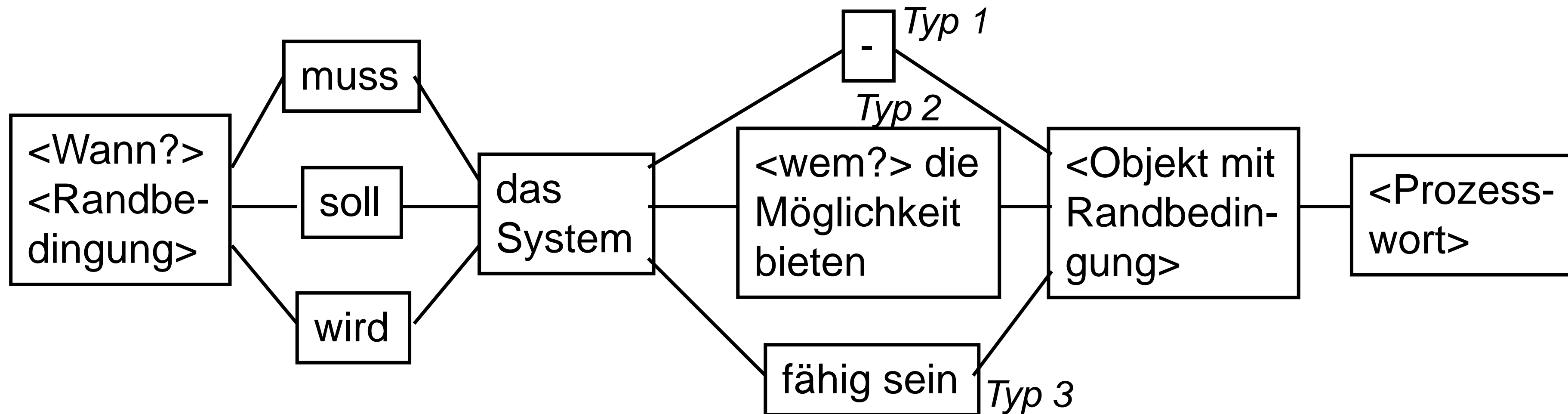
Nach der Eingabe eines neuen Teilprojekts oder einer neuen Projektaufgabe und nach der Aktualisierung des Aufwandes eines Teilprojekts oder einer neuen Projektaufgabe muss das System die Aufwandsangaben auf Plausibilität prüfen.

Typ 2: Nutzungsinteraktion



In der Projektbearbeitung muss das System der nutzenden Person die Möglichkeit bieten, ein neues Projekt mit Projektausgangsdaten anzulegen.
In der Projektbearbeitung muss das System der nutzenden Person die Möglichkeit bieten, jedes Projekt auszuwählen.
Nach der Projektauswahl muss das System der nutzenden Person die Möglichkeit bieten, für existierende Projekte neue Teilprojekte anzulegen.

Typ 3: Schnittstellenanforderung



Nach der Kontaktaufnahme durch die Software „Globalview“ muss das System fähig sein, Anfragen nach den Projektnamen, deren Gesamtaufwänden und Fertigstellungsgraden anzunehmen.

Beispiel: WebService-Schnittstellen werden so beschrieben

(folgt Typ2: Nach der Annahme der Anfrage …)

Vom Aktivitätsdiagramm zur textuellen Anforderung

- Jede Aktion wird mit einer Anforderung oder mehreren Anforderungen beschrieben
- Jede Entscheidung wird mit einer Anforderung oder mehreren Anforderungen beschrieben
- Aus dem Ablauf der zur Aktion oder Entscheidung führt, wird der erste Teil der jeweiligen Anforderung („Wann?“) erzeugt
- Hinweis: Anforderungen zum Beispiel stehen im folgenden Kapitel

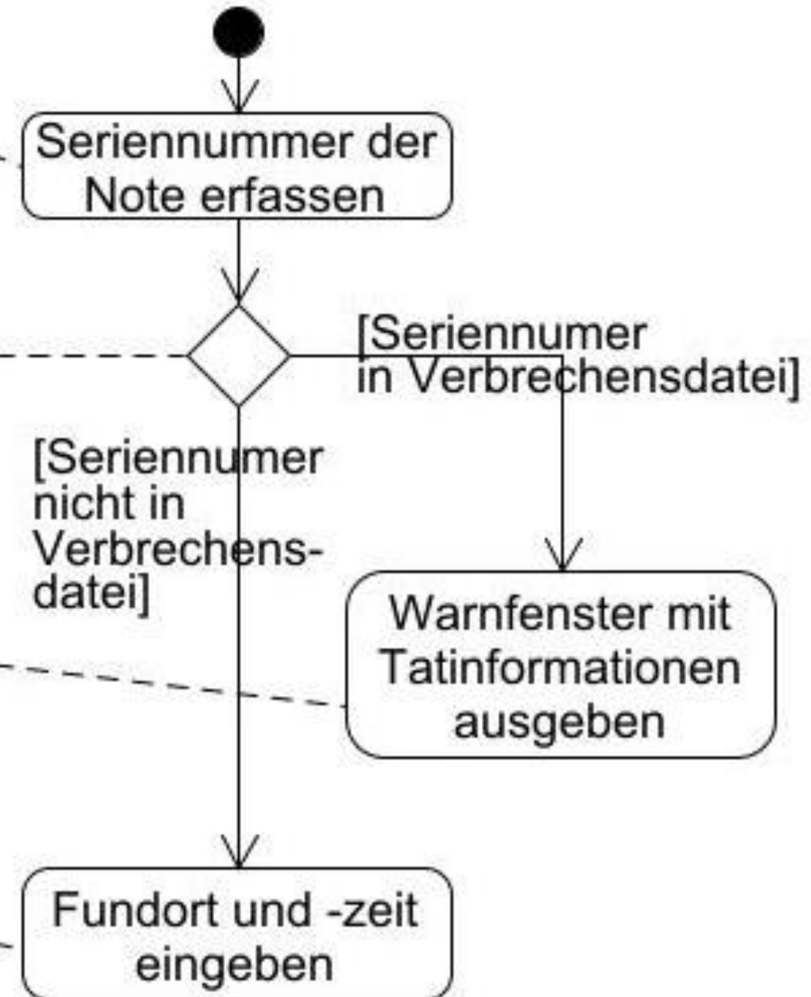
Beispielübersetzung (Fragment)

2: Nachdem das System gestartet wurde muss das System der nutzenden Person die Möglichkeit bieten, die Seriennummer der Banknote einzugeben

1: Nach Eingabe der Banknote, muss das System die eingegebene Seriennummer in der Verbrechensdatei suchen.

1: Wenn die Seriennummer in der Verbrechensdatei gefunden wurde, muss das System ein Warnfenster mit den Tatinformationen ausgeben.

2: Wenn die Seriennummer nicht in der Verbrechensdatei gefunden wurde, muss das System der nutzenden Person die Möglichkeit bieten, den Fundort und den Zeitpunkt des Fundes einzugeben.



Nicht-funktionale Anforderungen (1/2) [sehr kurz]

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

4.5

Video

- Bisher lag der Schwerpunkt auf funktionalen Anforderungen „was muss das System machen"
- *technische Anforderungen:*
  - Hardwareanforderungen
  - Architekturanforderungen
  - Anforderungen an die Programmiersprachen
- *Anforderungen an die Benutzungsschnittstelle*:
  - Form und Funktion von Ein- und Ausgabe-Geräten
  - (gesamter Ergonomie-Bereich)
- *Anforderungen an die Dienstqualität:*
  - DIN EN ISO 66272 unterteilt die Dienstgüte in die fünf Merkmale Zuverlässigkeit, Benutzbarkeit, Effizienz, Änderbarkeit und Übertragbarkeit

Nicht-Funktionale Anforderungen (2/2)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- *Anforderungen an sonstige Lieferbestandteile,* z. B.
  - Systemhandbücher
  - Installationshandbücher
- *Anforderungen an die Durchführung der Entwicklung und Einführung,* z. B.
  - Anforderungen an die Vorgehensweise
  - anzuwendende Standards
  - Hilfsmittel (Tools)
  - Durchführung von Besprechungen,
  - Abnahmetests (fachliche Abnahme, betriebliche Abnahme)
- *rechtlich-vertraglichen Anforderungen*, z. B.
  - Zahlungsmeilensteine
  - Vertragsstrafen
  - Umgang mit Änderungen
  - Eskalationspfade

Varianten der Anforderungsermittlung (1/3)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- Persona: Konkretisierung von Stakeholdern, insbesondere nutzenden Personen als konkrete Individuen
- Bsp.: Lara, 27 Jahre, Wirtschaftsinformatik, 4 Jahre im Unternehmen, Projektleiterin, liebt strukturierte Vorgehensweisen, mag viele Visualisierungen von Zusammenhängen, macht privat einen Origami-Blog, hält als Haustier eine Boa
- Persona helfen in der Analyse tätigen Personen manchmal sich in konkrete Abläufe und Handlungsweisen einzudenken

- Persona werden gerne in kreativen Bereichen, wie Usability und Interaction Design genutzt

Varianten der Anforderungsermittlung (2/3)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- Epic: Beschreibung typischer Arbeitsabläufe späterer nutzenden Personen (klarer Anfang, eindeutiges Ergebnis)

-> ähnlich einsetzbar wie Use Cases, können auch Aktivitätsdiagrammerstellung unterstützen

Variantes der Anforderungsermittlung (3/3)

- User Story (u. a. in Extreme Programming): Fokus auf eine von einer bestimmen Rolle gewünschten Funktionalität
- abstrakt: Als <Stakeholder in folgender Rolle> möchte ich <geforderte Funktionalität> um <gewünschter Nutzen>.
- Als Projektleitung möchte ich den aktuellen Stand an verbrauchten Arbeitsstunden der Arbeitspakete kompakt überblicken, um zu bewerten, ob aktuelle Planungsziele erreicht werden können.

-> User Stories verfeinern Epics und stellen damit Teile von Abläufen von Aktivitätsdiagrammen dar

- User Storys sind alternativ/ergänzend zur vorgestellten Anforderungsanalyse nutzbar

Lastenheft / Pflichtenheft

4.6

- Lastenheft wird vom auftraggebenden Unternehmen (AG) geschrieben
  - welche Funktionalität ist gewünscht
  - welche Randbedingungen (SW/ HW) gibt es
- Pflichtenheft wird vom auftragnehmenden Unternehmen (AN) (Software-Entwicklung) geschrieben
  - welche Funktionalität wird realisiert
  - auf welcher Hardware läuft das System
  - welche SW-Schnittstellen (Versionen) berücksichtigt
- Variante: AG beauftragt AN direkt in Zusammenarbeit Pflichtenheft zu erstellen
  - ein gemeinsames Heft ist sinnvoll
  - Pflichtenheft ist meist (branchenabhängig) zu bezahlen

Lastenheft / Pflichtenheft: möglicher Aufbau

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

0. Administrative Daten: von wem, wann genehmigt, ...
1. Zielbestimmung und Zielgruppen
   - In welcher Umgebung soll System eingesetzt werden?
   - Ziele des Systems, welche Stakeholder betroffen?
2. Funktionale Anforderungen
   - Produktfunktionen (Use Cases, Aktivitätsd., Anforderungen)
   - Produktschnittstellen (a. GUI-Konzept   b. andere SW)
3. Nichtfunktionale Anforderungen
   - Qualitätsanforderungen
   - weitere technische Anforderungen
4. Lieferumfang
5. Abnahmekriterien
6. Anhänge (insbesondere Glossar)

Video

5. Grobdesign

- 5.1 Systemarchitektur
- 5.2 Ableitung von grundlegenden Klassen
- 5.3 Ableitung von Methoden und Kontrollklassen
- 5.4 Validierung mit Sequenzdiagrammen
- 5.5 Überlegungen zur Oberflächenentwicklung

Systemarchitektur

5.1

Festlegen der Randbedingungen bzgl. Hardware, Betriebssystem, verwendeter Software, zu integrierender Systeme

- Vorgabe der Hardware, die Software muss z. B. auf einer Spezialhardware funktionieren
- Vorgabe des Betriebssystems, die Software muss eventuell mit anderer Software auf Systemebene zusammenarbeiten
- Vorgabe der Middleware, die Software wird häufig auf verschiedene Prozesse verteilt, die miteinander kommunizieren müssen
- Vorgaben zu Schnittstellen und Programmiersprachen, die Software soll mit anderer Software kommunizieren und muss dabei deren Schnittstellen berücksichtigen
- Vorgaben zum „Persistenz-Framework“, die Daten der zu erstellenden Software müssen typischerweise langfristig gespeichert werden

Klassenmodellierung für OO-Programmier*innen

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

5.2a

- Generell soll im Grobdesign eine erste Klassenmodellierung stattfinden, die die gesamte geforderte Funktionalität abdeckt
- Hauptaufgabe des Klassenmodells, auch Domain-Model genannt, ist damit die Vollständigkeit
- Danach wird Domain-Model im Feindesign in Richtung effizienter Programmierung, z. B. mit Hilfe von Design-Pattern, optimiert
- in OO erfahrende programmierende Personen (HS OS, 4. Semester), können bereits im Domain-Model sinnvolle Optimierungen (d. h. Nutzung guter Design-Regeln) vornehmen
- Deshalb werden hier UML-Klassendiagramme und Sequenzdiagramme für Personen mit Programmiererfahrung vorgestellt

Modellierungsaufgabe

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- Es soll eine SW zur Verwaltung von mitarbeitenden Personen mit ihren Fähigkeiten erstellt werden.
- Die Software soll Projekte verwalten, denen mitarbeitende Personen zugeordnet und ein Scrum Master aus den mitarbeitenden Personen zugeordnet werden können.
- Mitarbeitende Personen können in verschiedenen Projekten mitarbeiten, dazu wird festgelegt, von wann bis wann sie zu welchem Prozentanteil mitarbeiten.
- (Achtung, dies ist keine sinnvolle Anforderungsanalyse)

- wichtiger Hinweis: Die UML und damit Klassendiagramme sind programmiersprachenunabhängig, deshalb gibt es auch Teile von Java, die nicht in UML (ohne Erweiterungen) darstellbar sind [und andersherum]

Erinnerung: Java-Grundregeln für Klassen

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- Klassenname in Einzahl (Nomen oder Nominalisierend: Mitarbeitend)
- Objektvariablen (= Instanzvariablen) sind immer private; bei Vererbung auch protected möglich
- gibt immer parameterlosen Konstruktor
- gibt für jede Objektvariable get- und set-Methode
- letzten beiden Regeln werden von vielen Java-Frameworks, auch Java selbst bei XML-Nutzung, benötigt
- gibt immer toString()-Methode zur Objektvisualisierung
- gibt (fast) immer equals()- und hashCode()-Methode
- alle genannten Konstruktoren und Methoden sind generierbar
- Sie halten sich an Java-Coding-Guidelines; Einstieg dazu über

  http://home.edvsz.hs-osnabrueck.de/skleuker/querschnittlich/CodingGuidelinesUndGlossar.pdf

Klasse Mitarbeitend (1/3)

```
public class Mitarbeitend {
  private int id;
  private String name;
  private static int idCount = 1000;
```

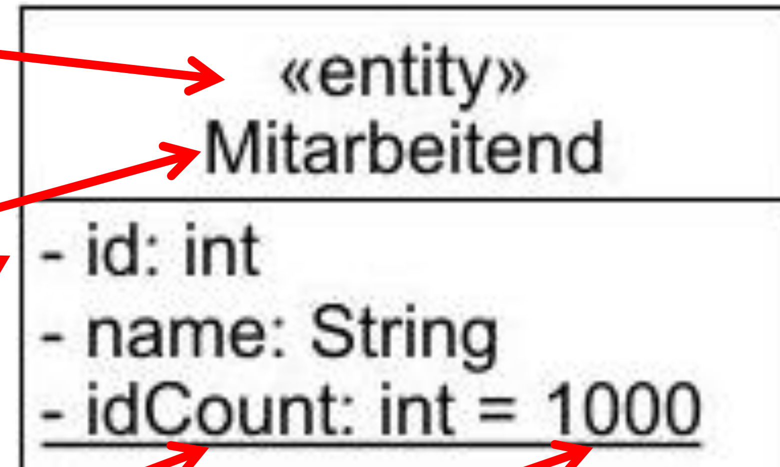
in französischen Anführungsstrichen stehen optionale Stereotypen; diese bietet die UML als Markierungs- und Erweiterungsmöglichkeit; sind für Klassendiagramme nicht vorgegeben

Klassenname (evtl. Paket davor)

Objektvariaben mit <Sichtbarkeit> <Name>: <Typ>

Klassenvariablen sind unterstrichen

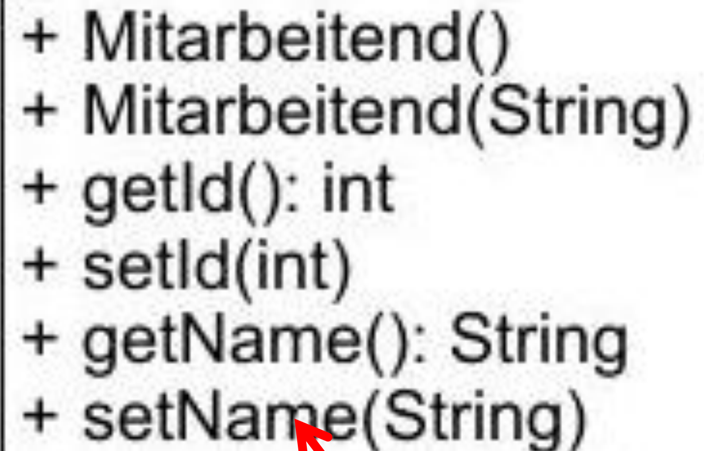
Startwerte können für alle Variablen angegeben werden

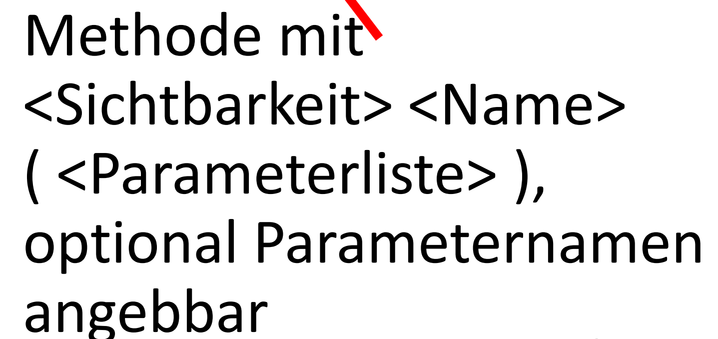


Klasse Mitarbeitend (2/3)

```
public Mitarbeitend() {
  this.id = Mitarbeitend.idCount++;
}

public Mitarbeitend(String name) {
  this();
  this.name = name;
}

public int getId() {return id;}

public void setId(int id) {
  this.id = id;
}

public String getName() {
  return this.name;
}

public void setName(String name) {
  this.name = name;
}
```

Konstruktor mit <Sichtbarkeit> <Name> (<Parameterliste>)





Klasse Mitarbeitend (3/3)

```
  public static int wertIdCount() {
     return Mitarbeitend.idCount;
  }
}
```

Sichtbarkeiten:

+: public

-: private

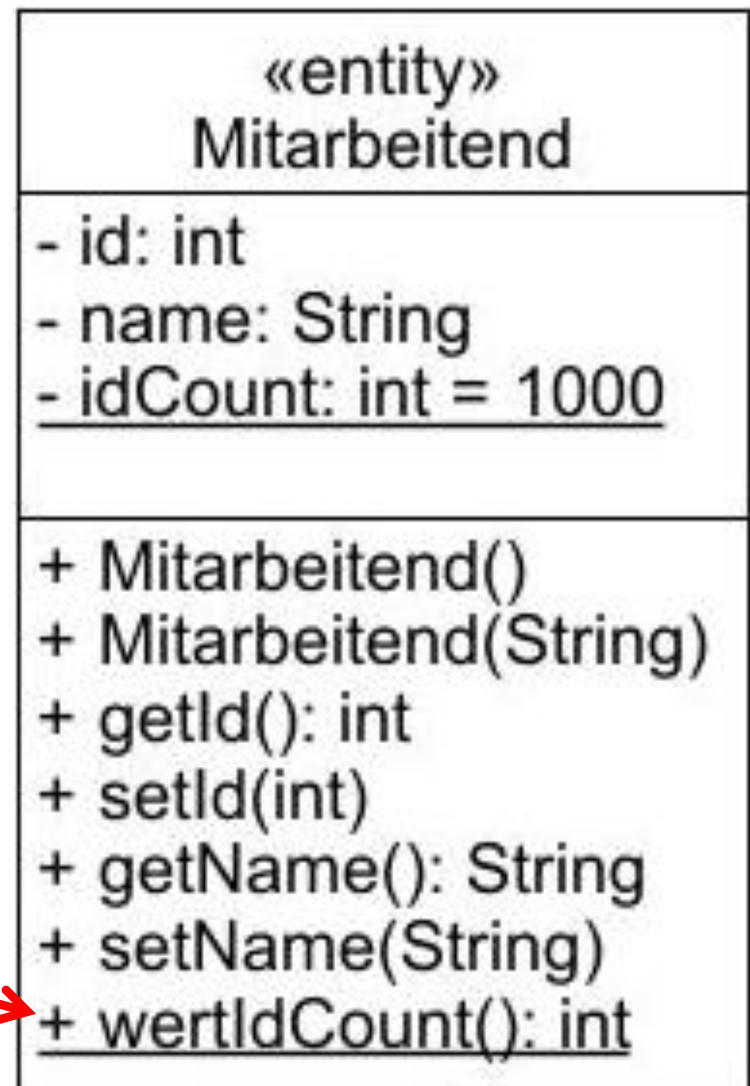
#: protected

~ : (nicht genau package-protected wie in Java)

Rückgabetyp void weglassbar

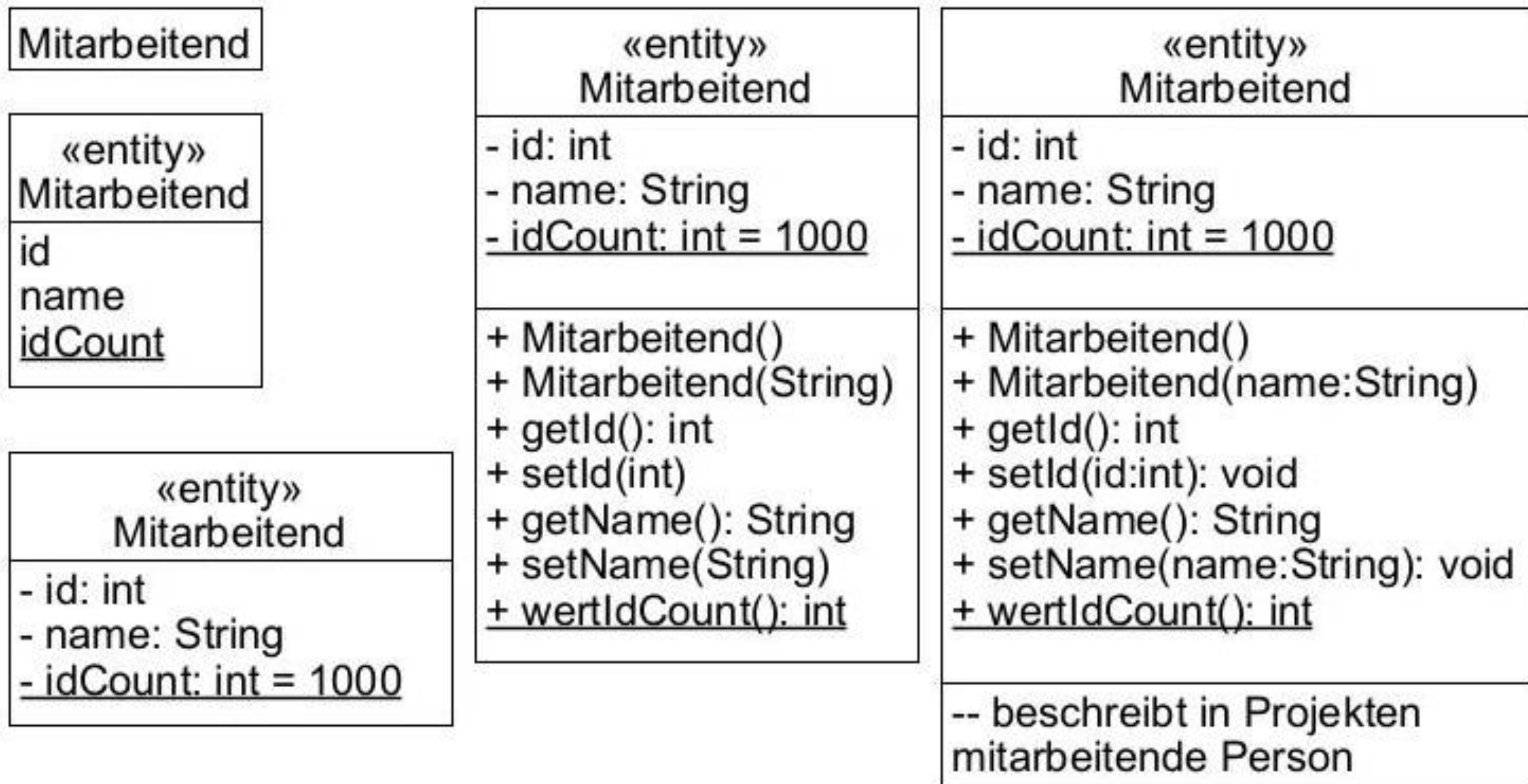
Klassenmethoden sind unterstrichen

mit <Sichtbarkeit> <Name>
(<Parameterliste>)



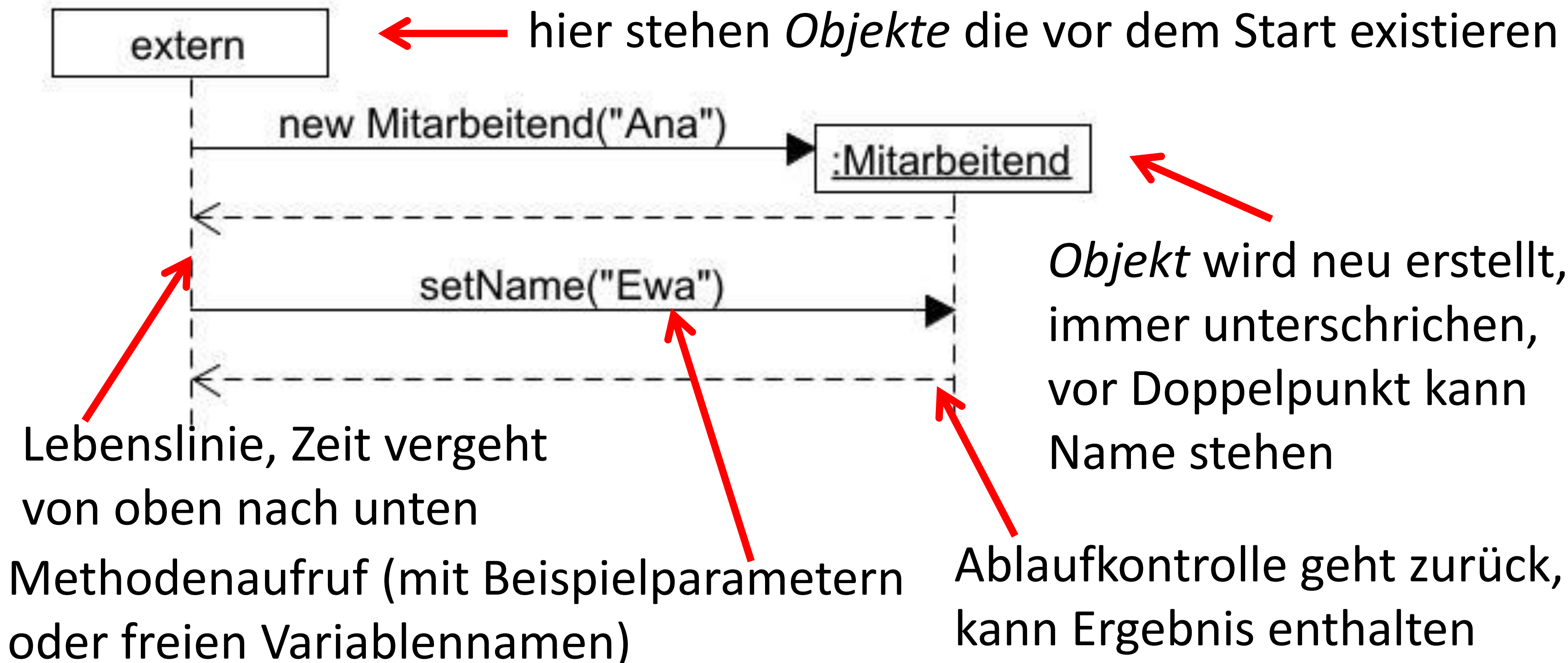
Inkrementelle Entwicklung mit UML

- generell können fast alle Informationen weggelassen und später ergänzt werden
- wird Klasse in anderen Klassendiagrammen gezeigt, wird auch oft nur der Kasten gezeigt



Dynamische Modellierung mit Sequenzdiagrammen

- Klassendiagramme sind statisch und zeigen „nur“ Aufbau
- Beispielabläufe mit Sequenzdiagrammen darstellbar
- Beispiel: jemand/irgendein Objekt erzeugt Mitarbeitend und ändert den Namen



Algorithmen mit Sequenzdiagrammen

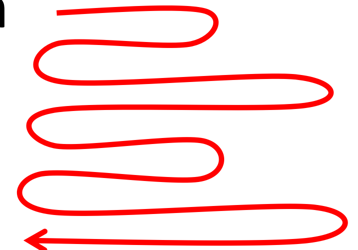
HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- Sequenzdiagramm zeigt Vertauschen von Namen

extern
m1:Mitarbeitend
m2:Mitarbeitend
getName()
n1
getName()
n2
setName(n2)
setName(n1)

diese Objekte existieren beim Diagrammstart, haben Namen (hier unnötig)

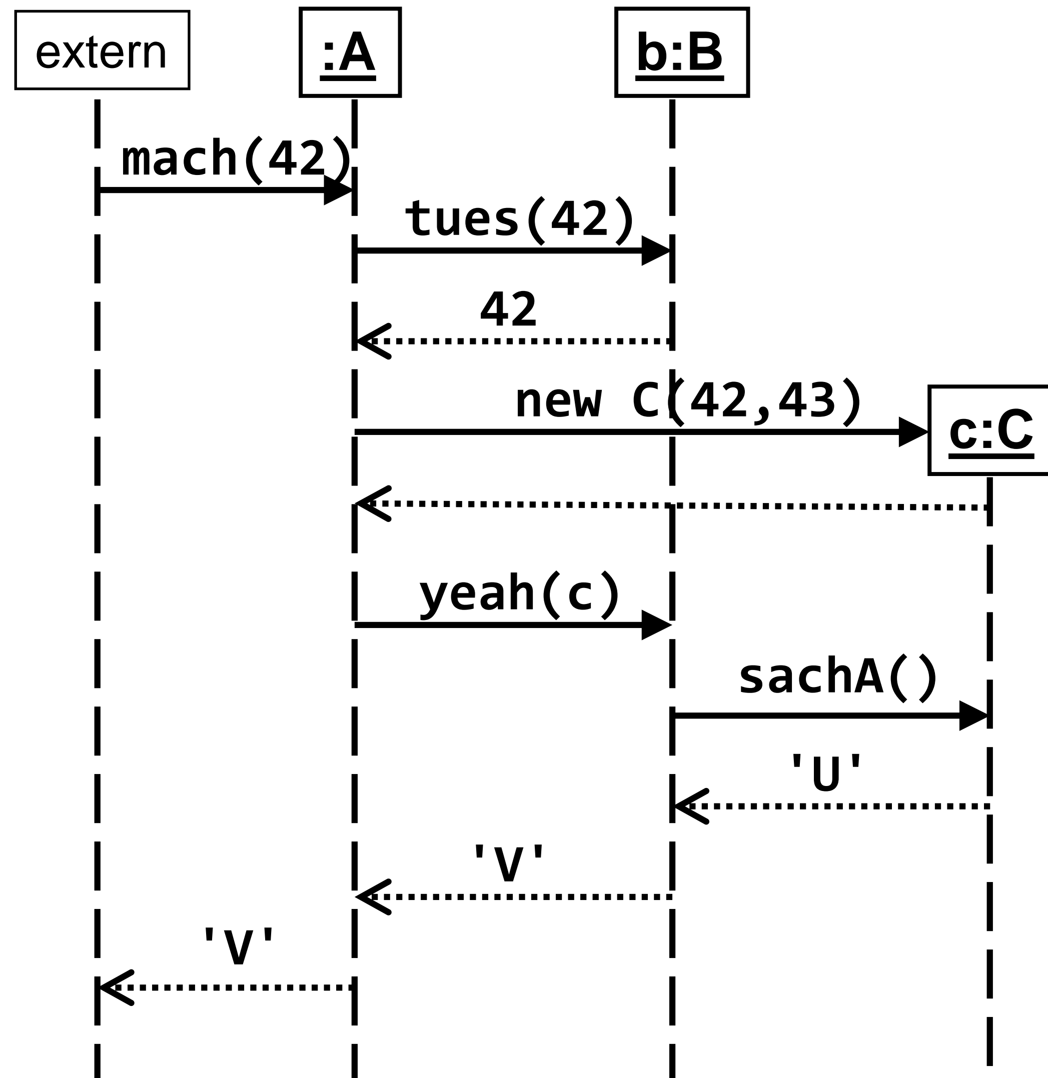
man kann Pfeilen immer durchgehend folgen



“extern” nicht Teil der UML, schließt aber Diagramme ab

Zusammenhang: Programm und Sequenzdiagramm

```
public class A {
   private B b= new B();
   private C c;
   public char mach(int x){
    int t= b.tues(x);
    c= new C(t,t+1);
    return b.yeah(c);
   }
}
public class B {
    public int tues(int x){
        return x%255;
    }

    public char yeah(C c){
        char e=c.sachA();
        return (char) (e+1);
    }
}
```



Mitarbeitend-Objekt hat Sammlung von Fähigkeiten

Video

- Faehigkeit ist Enumeration
- Umsetzung in Java:

```
public enum Faehigkeit {
  JAVA, C, GO, MASTER, PRODUCTOWNER
}
```

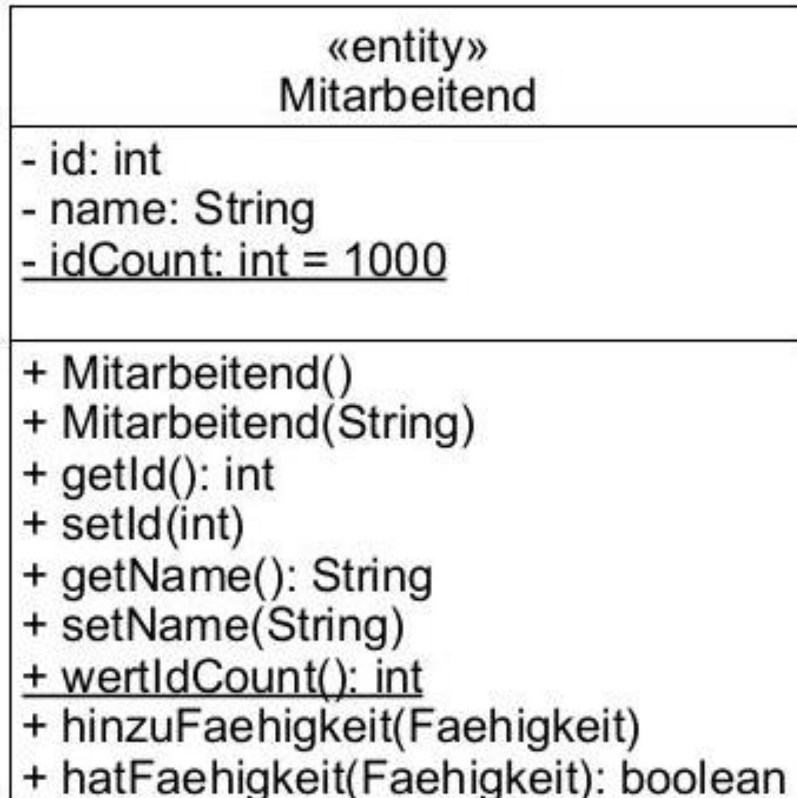
- in Mitarbeitend:

```
private Set<Faehigkeit> faehigkeiten;

public void hinzuFaehigkeit(Faehigkeit f) {
  this.faehigkeiten.add(f);
}

public boolean hatFaehigkeit(Faehigkeit f) {
  return this.faehigkeiten.contains(f);
}
```

Sammlungen in Klassendiagrammen

gerichtete Assoziation, Klasse hat Objektvariable von Typ anderer Klasse

Objektvariable von Mitarbeitend private faehigkeiten



-faehigkeiten

*

*

«enumeration»
Faehigkeit

Java
C
Go
Master
Productowner

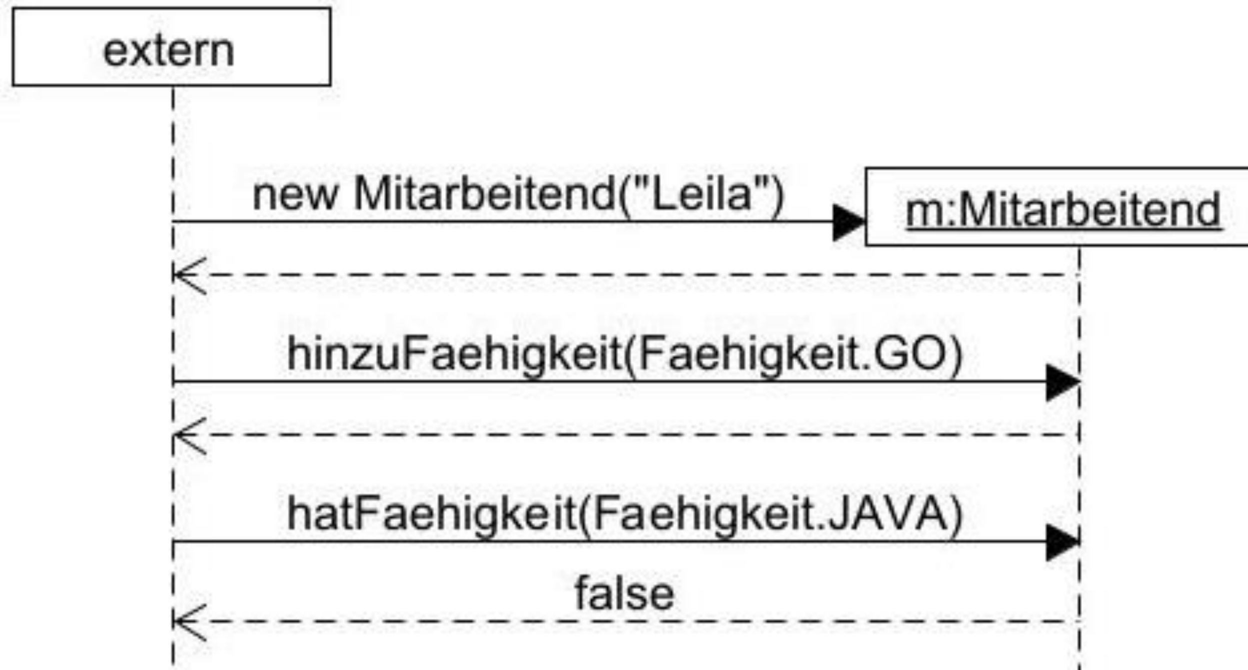
Multiplizitäten
0 keines
1 genau eines
* beliebig viele
0..1 max. eines
3..* mindestens 3

Aufzählungs-werte angeben

Assoziation genauer

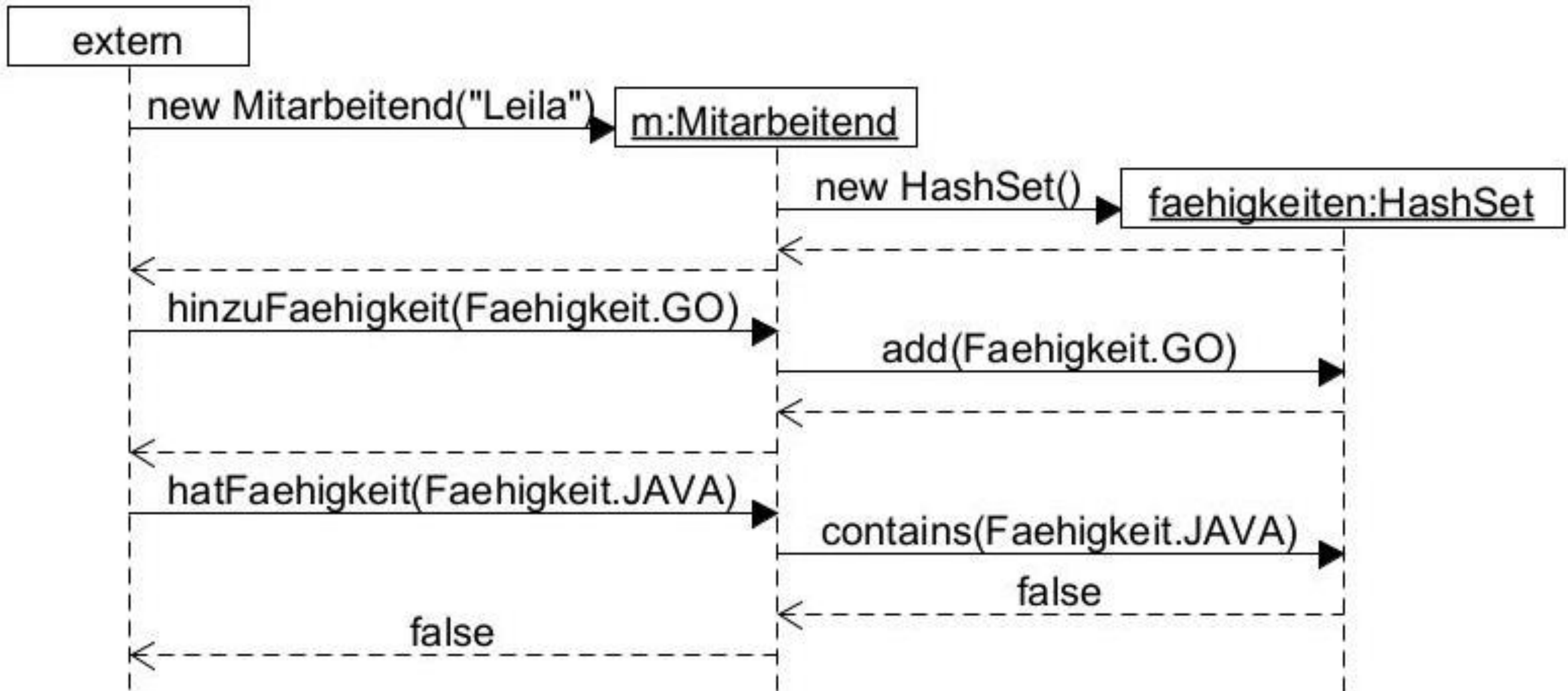
- Pfeil gibt an, dass nur Mitarbeitend-Objekte ihre Fähigkeiten kennen, nicht anders herum
- ohne Pfeilspitze unterspezifiziert, bzw. bidirektional
- * rechts: zu jedem Mitarbeitend-Objekt gehören beliebig viele Faehigkeiten, die in der Variablen faehigkeiten stehen
- * links: jedes Faehigkeits-Objekt kann in beliebig vielen Mitarbeitend-Objekten vorkommen (dies sieht man nicht im Code, ist aber Teil der Modellierung; ist damit Randbedingung)
- ohne weitere Angaben ist Art der Sammlung bzw. Collection (List, Map, Set, MultiSet) unterspezifiziert
- -faehigkeiten steht auf der rechten Seite, *nicht* in der Mitte!

neues Mitarbeitend-Objekt mit Faehigkeiten



- in Sequenzdiagrammen nur „wichtige“ Klassen für Verständnis
- deshalb ist HashSet-Objekt hier nicht sichtbar
- meist werden solche Collections weggelassen
- natürlich können alle Objekte eingezeichnet werden

neues Mitarbeiten-Objekt mit Faehigkeiten - genauer



- hier wurde Set-Objekt zur Veranschaulichung eingetragen
- (werden wir in der Veranstaltung nicht machen)

Wer erstellt Mitarbeitend-Objekte

- zumindest bei Entitäten soll es nur eine Klasse geben, die Objekte erzeugt
- typischerweise Controller- oder Verwaltungsklasse
- Controller ist einzige Möglichkeit für CRUD
- Mitarbeitend-Objekt zu erzeugen (CREATE)
- Mitarbeitend-Objekt (über Schlüssel) zu finden (READ)
- Mitarbeitend-Objekt zu verändern (UPDATE)
- Mitarbeitend-Objekt zu löschen (DELETE)
- alle Veränderungen und Befragungen von Mitarbeitend-Objekten, hier zu Fähigkeiten, findet über diese Klasse statt
- (ab jetzt get- und set- sowie Java-übliche Methoden weggelassen)

MitarbeitendController in Java (1/2)

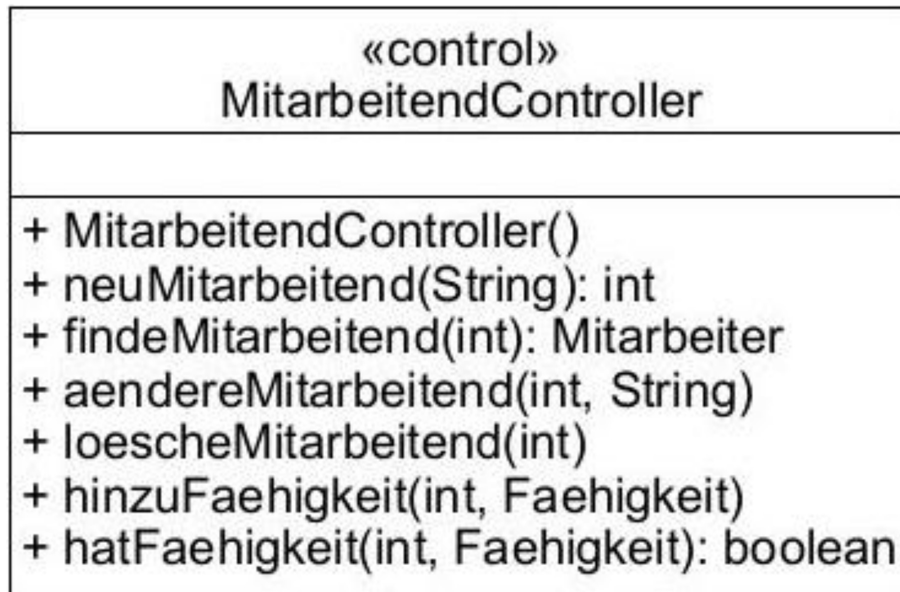
HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

```
public class MitarbeitendController {
  private Map<Integer,Mitarbeitend> mitarbeitende;

  public MitarbeitendController() {
    this.mitarbeitende = new HashMap<>();
  }

  public int neuMitarbeitend(String name) {
    Mitarbeitend tmp = new Mitarbeitend(name);
    this.mitarbeitende.put(tmp.getId(), tmp);
    return tmp.getId();
  }

  public Mitarbeitend findeMitarbeitend(int id) {
    return this.mitarbeitende.get(id);
  }

  public Mitarbeitend loescheMitarbeit(int id) {
    return this.mitarbeitende.remove(id);
  }
```

MitarbeitendController in Java (2/2)

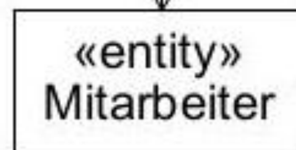
```java
public void aendereMitarbeitend(int id, String name) {
  Mitarbeitend tmp = this.findeMitarbeitend(id);
  if (tmp != null) {
    tmp.setName(name);
  }
}

public void hinzuFaehigkeit(int id, Faehigkeit f) {
  Mitarbeitend tmp = this.findeMitarbeitend(id);
  if (tmp != null) {
    tmp.hinzuFaehigkeit(f);
  }
}

public boolean hatFaehigkeit(int id, Faehigkeit f) {
  Mitarbeitend tmp = this.findeMitarbeitend(id);
  return tmp != null && tmp.hatFaehigkeit(f);
}
```

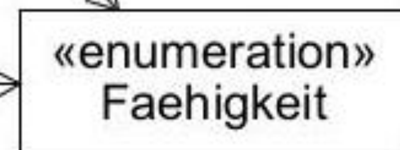
Modellierung: MitarbeitendController



"nutzt"-Beziehung (Klasse kommt im Code vor, gibt aber keine Objektvariable)

jedes Mitarbeitend in genau einem Controller

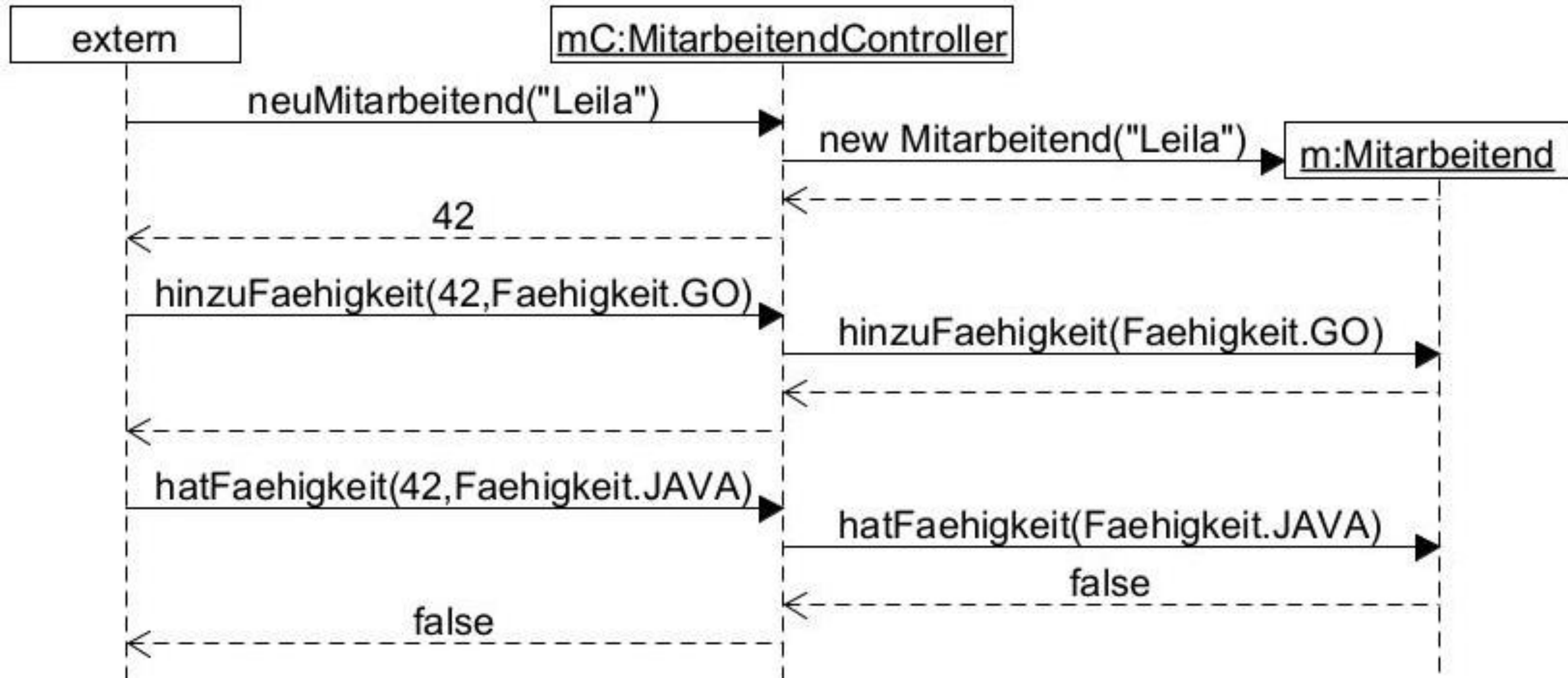




Objektvariable vom Typ Sammlung in MitarbeitendController

Mitarbeitend-Objekt mit Fähigkeiten anlegen

- typisch: Weiterleitung (Delegation) von Controller-Aufruf an Entität



- Sequenzdiagramme können mit sehr kurzen Fragmenten Sachverhalte zeigen; es kann auch sinnvoll sein längere detaillierte Abläufe zu visualisieren

Einschub: Programmzeilen des Grauens

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

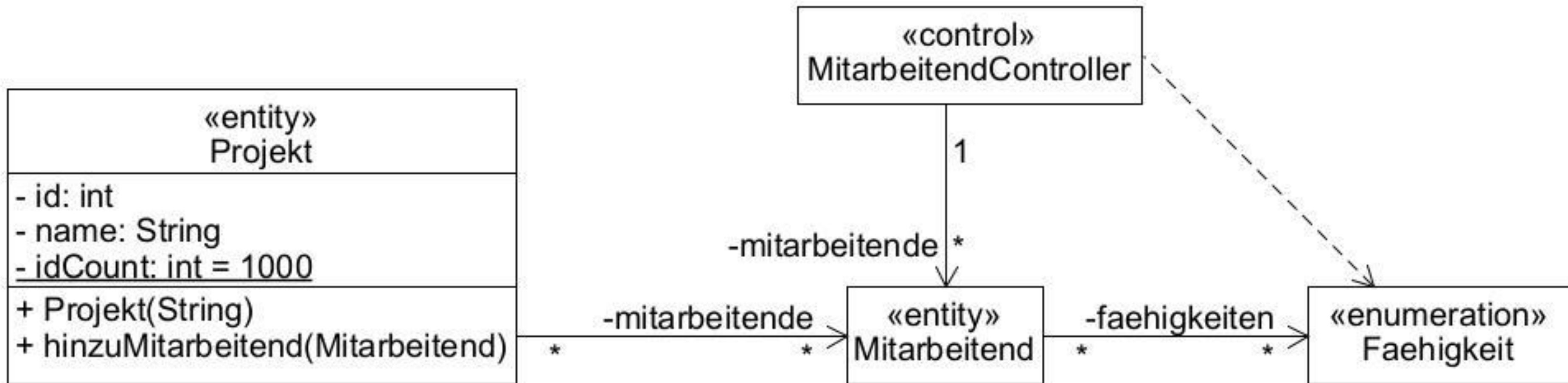
- Nie, nie Objektvariablen mit get oder find holen und dann bearbeiten; Bearbeitung immer durch Controller
- OP: Herz herausoperieren, an Uni-Klinik schicken, dort Herz korrigieren, zurück schicken, Herz wieder einsetzen

```
Mitarbeitend m = mitarbeitendController.findeMitarbeitend(42);
Set<Faehigkeit> sf = m.getFaehigkeiten();
sf.add(Faehigkeit.GO);
m.setFaehigkeiten(sf); // schlecht und ohnehin ueberfluessig
```

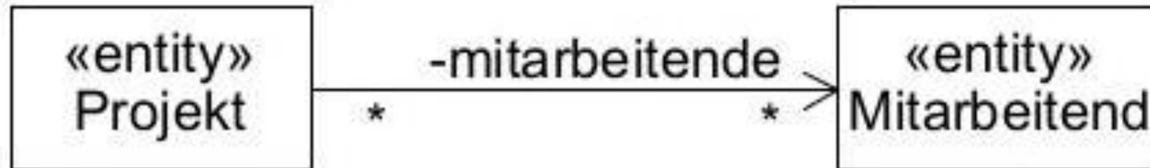
- wenn Sie sowas sehen, Standardfrage: „Wenn Du gerne programmierst, warum lernst Du es nicht"

Projekte mit beliebig vielen Mitarbeitend-Objekten

- gleiche Objektvariablennamen erlaubt, muss aber nicht sein
- muss nicht alle CRUD-Methoden geben

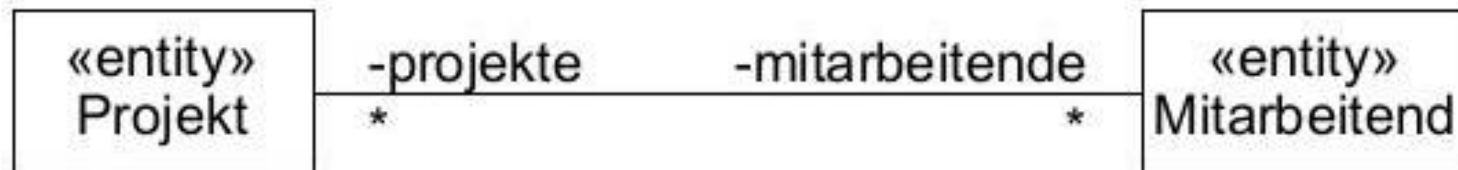


Design-Entscheidung über Modellierung hinaus



- Design-Entscheidung: Projekt kennt seine Mitarbeitend-Objekte; sollen zu Mitarbeitend-Objekt alle Projekte bestimmt werden, muss über alle Projekte iteriert werden

- wenn Projekte von Mitarbeitend-Objekten zu suchen sehr wichtig, dann Assoziation umdrehen

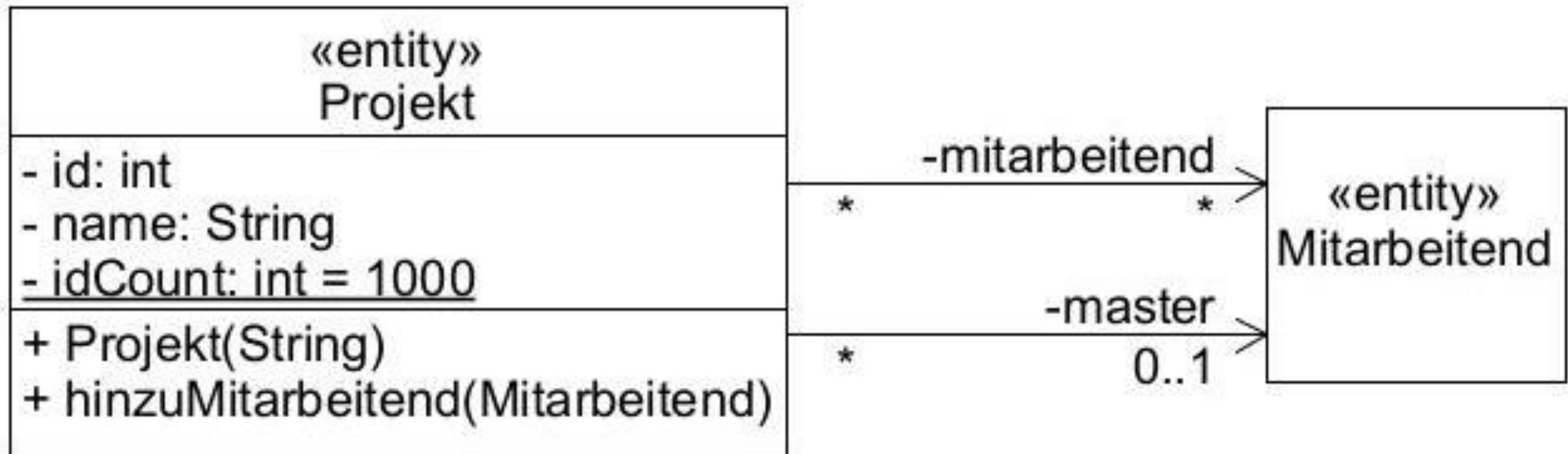


- wenn beide Richtungen sehr wichtig, dann bidirektional (möglichst vermeiden, da später fehleranfällig; nicht immer vermeidbar)

Jedes Projekt kann einen Scrum-Master haben

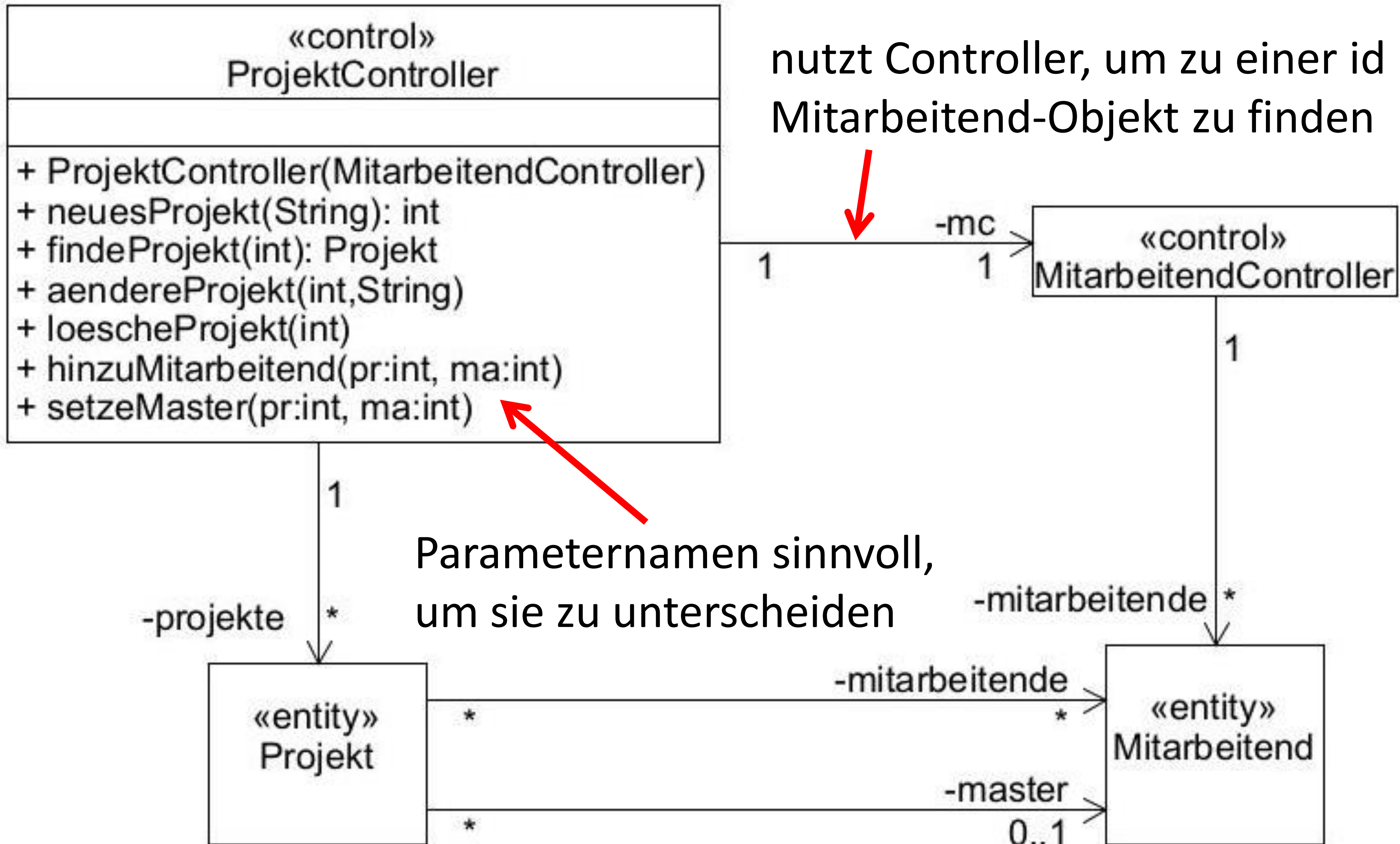
HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- kann mehrere Assoziationen zwischen zwei Klassen geben



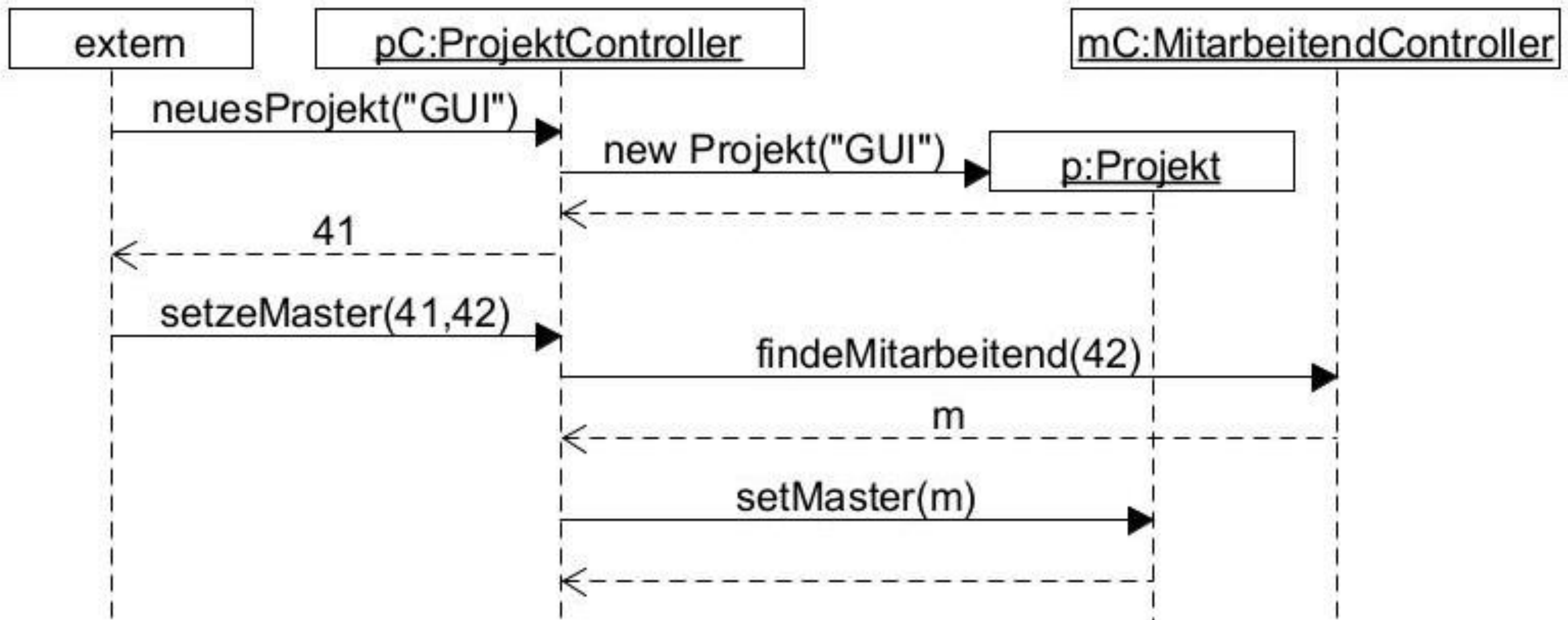
- Erinnerung: existierende Methoden setMaster() und getMaster() nicht mehr angegeben
- Frage wo geprüft wird, ob Mitarbeitend-Objekt Fähigkeit „MASTER“ hat, bleibt offen

ProjektController



neues Projekt mit Master erzeugen

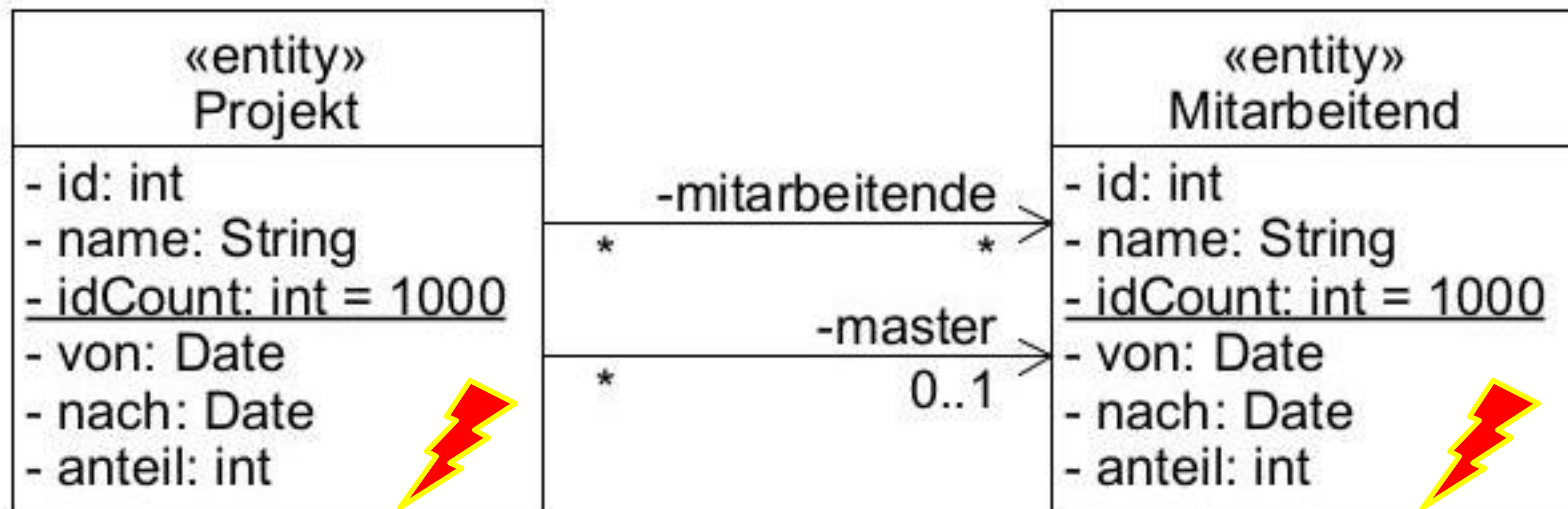
HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES



später sehen wir, dass auch in Sequenzdiagrammen geprüft werden kann, ob m == null gilt

Erweiterung: Mitarbeitend-Objekt anteilig zuordnen

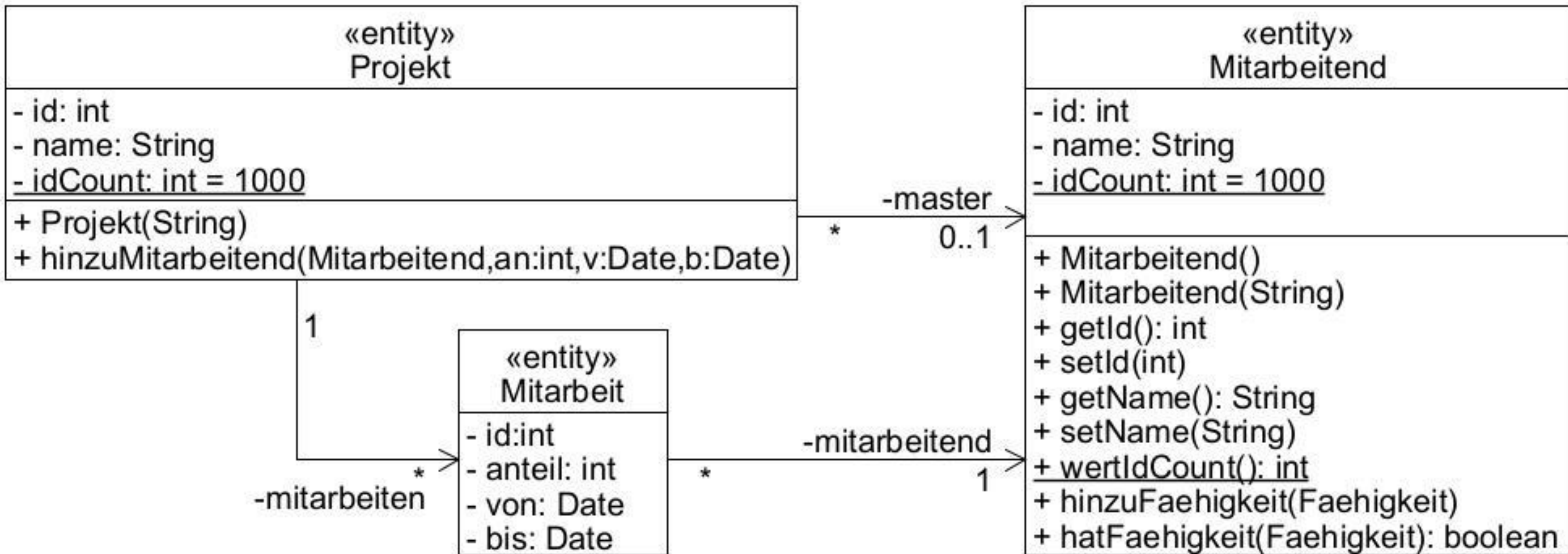
- Jede mitarbeitende Person arbeitet von einem Datum bis einem Datum zu einem bestimmten Prozentanteil in einem Projekt
- weder auf einer, noch auf beiden Seiten macht folgendes Sinn:



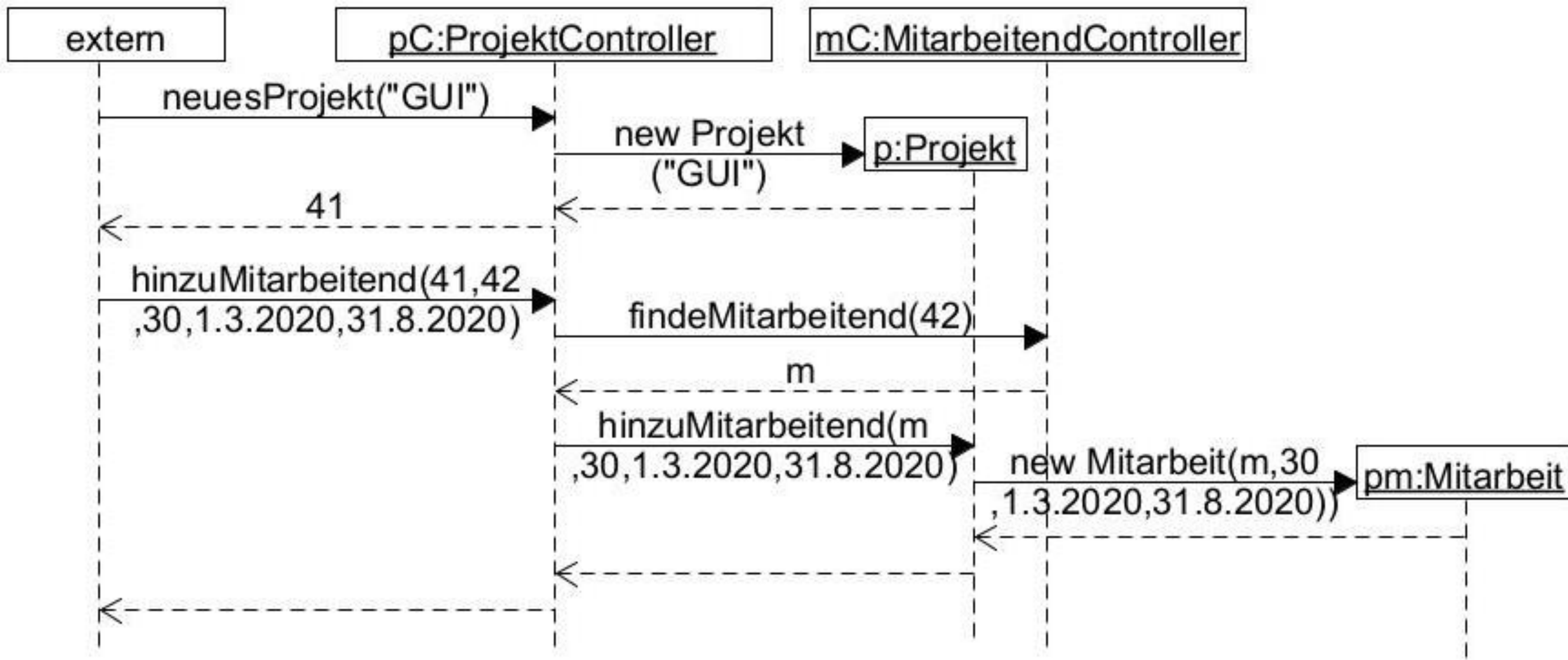
- entweder: jede mitarbeitende Person eines Projekts muss zum gleichen Datum mit gleichen Anteil starten und beenden
- oder: jedes Projekt einer mitarbeitenden Person muss zum gleichen Datum mit gleichen Anteil starten und beenden

Standardlösung: Koppelentität

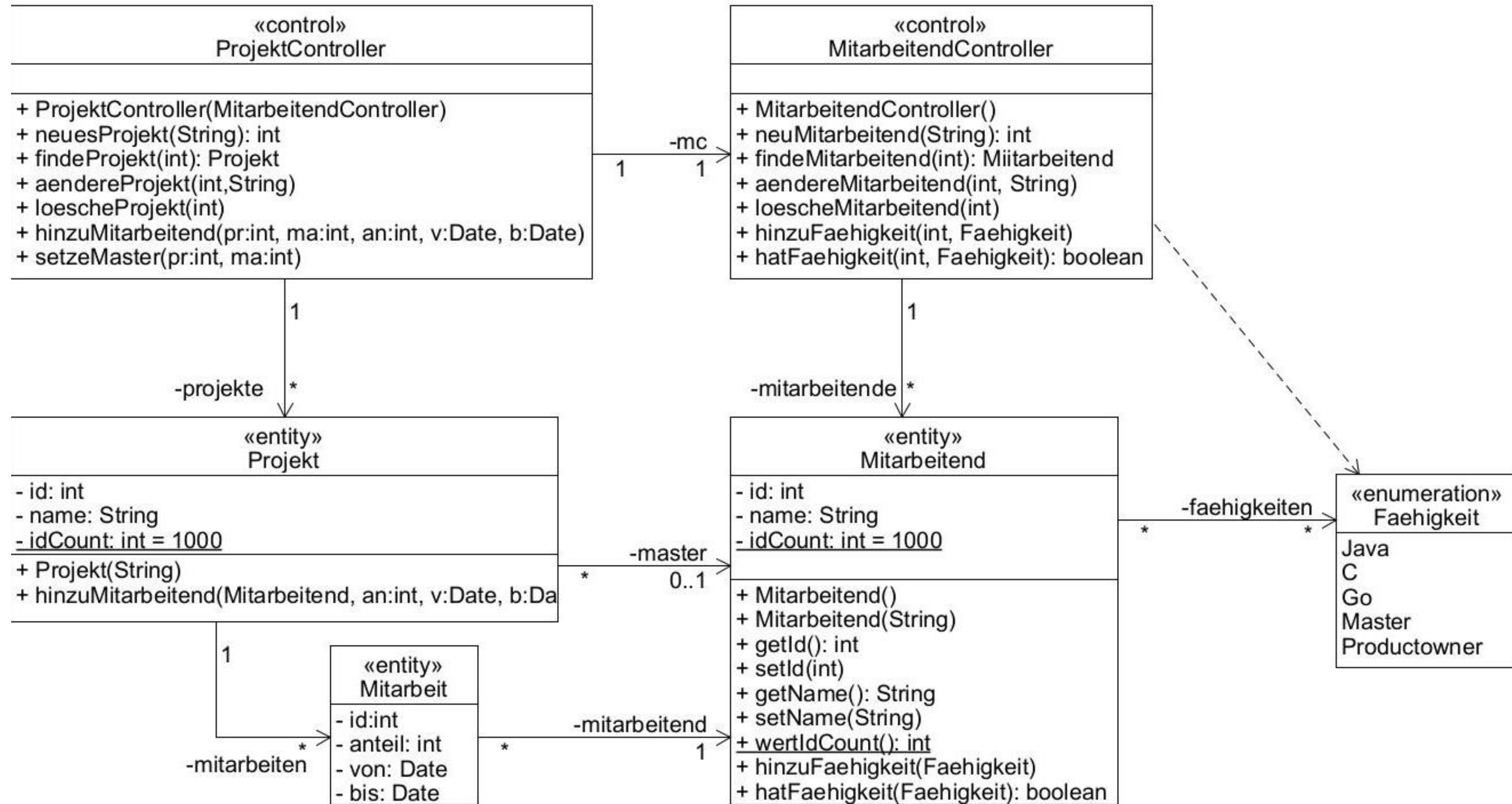
- Erinnerung: Übersetzung von M:N-Beziehungen von ER-Diagrammen in Tabellen



Mitarbeitend-Objekt zum Projekt hinzufuegen



Zwischenstand zum Zoomen

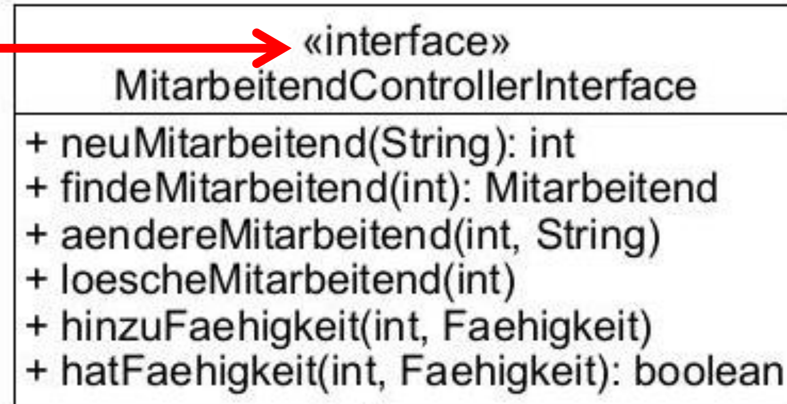


Flexibilisierung mit Interfaces

- Konzept der bisherigen Controller ok, allerdings bis jetzt rein lokale Datenhaltung
- Realität: Daten befinden sich in einer Datenbank
- Controller nutzt Datenbankverbindung, um Entitätsobjekte zu verwalten (CRUD)
- DB-Verwaltung wird typischerweise von eigener SW übernommen; z. B. objekt-relationale Mapper für relationale Datenbanken
- Java-Standardlösung: JPA (s. Software-Architektur, 5. Semester)
- schön wäre, wenn einfach zwischen verschiedenen Lösungen umgeschaltet werden könnte
- Ansatz: nur Methoden spezifizieren (also abstract) und verschiedene Implementierungen anbieten

Interface in UML

Stereotyp <<interface>>
alle Methoden abstract



realisiert-Pfeil, gestrichelt mit offenen Dreieck als Pfeilspitze

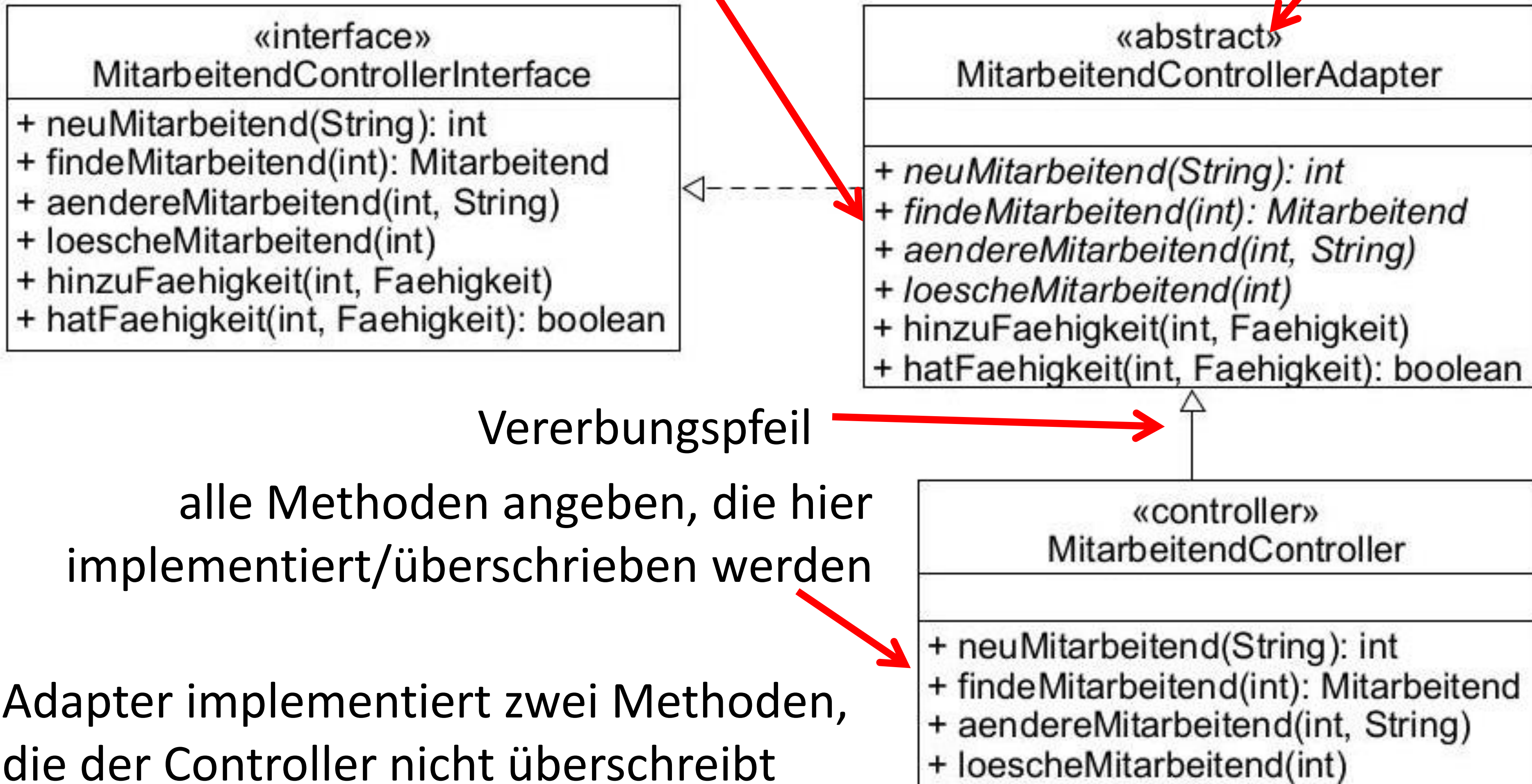
«control»
MitarbeitendController

+ MitarbeitendController()
+ neuerMitarbeitend(String): int
+ findeMitarbeitend(int): Mitarbeitend
+ aendereMitarbeitend(int, String)
+ loescheMitarbeitend(int)
+ hinzuFaehigkeit(int, Faehigkeit)
+ hatFaehigkeit(int, Faehigkeit): boolean

«control»
MitarbeitendControllerDB

+ MitarbeitendController()
+ neuMitarbeitend(String): int
+ findeMitarbeitend(int): Mitarbeitend
+ aendereMitarbeitend(int, String)
+ loescheMitarbeitend(int)
+ hinzuFaehigkeit(int, Faehigkeit)
+ hatFaehigkeit(int, Faehigkeit): boolean

Teilimplementierung



Adapter implementiert zwei Methoden, die der Controller nicht überschreibt

Zwischenfazit

- Beispiel zeigt einen systematischen Weg zur Erstellung eines Klassendiagramms
- Sequenzdiagramme veranschaulichen die Dynamik, wer was wann wo aufruft
- Klassendiagramme entstehen oft an Whiteboards mit vielen Fotos für Zwischenergebnisse , wischen, streichen, markieren, …

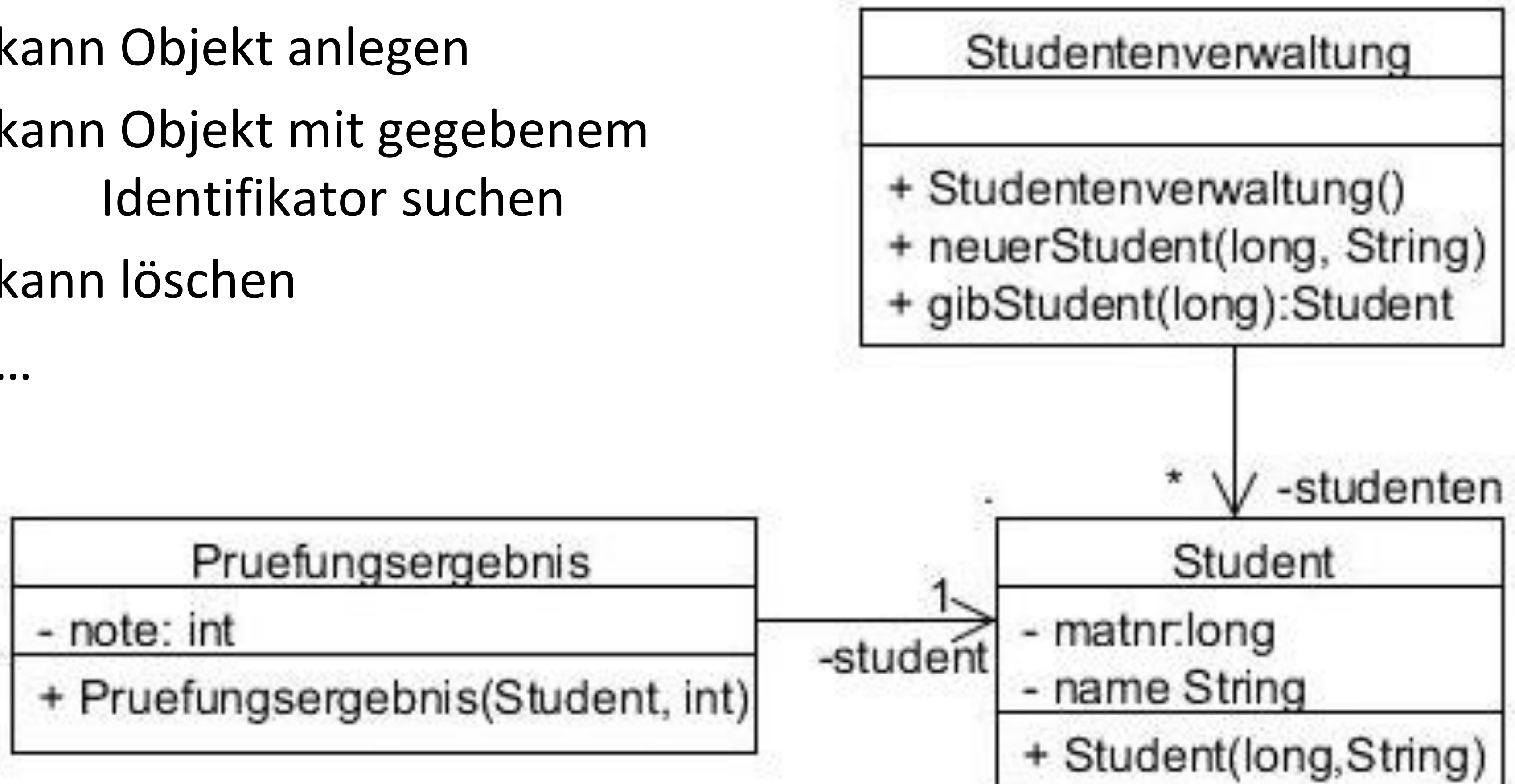
- Beispiel zeigt eine sinnvolle Lösung, aber weitere Themen
  - es gibt Varianten bei den Rückgaben, gerade null ist diskutabel (-> Java kennt Optional (später); generell Ergebnisklasse(n) sinnvoll)
  - was passiert bei Ausnahmen
  - wohin mit Konstanten (z. B. Hilfsklassen, alle können zugreifen)
  - …

Beispiel für Design-Idee (1/5)

Video

Wenn Objektsammlungen benötigt, gibt es häufig eine Verwaltungsklasse (hier mal Verwaltung statt engl. Controller):

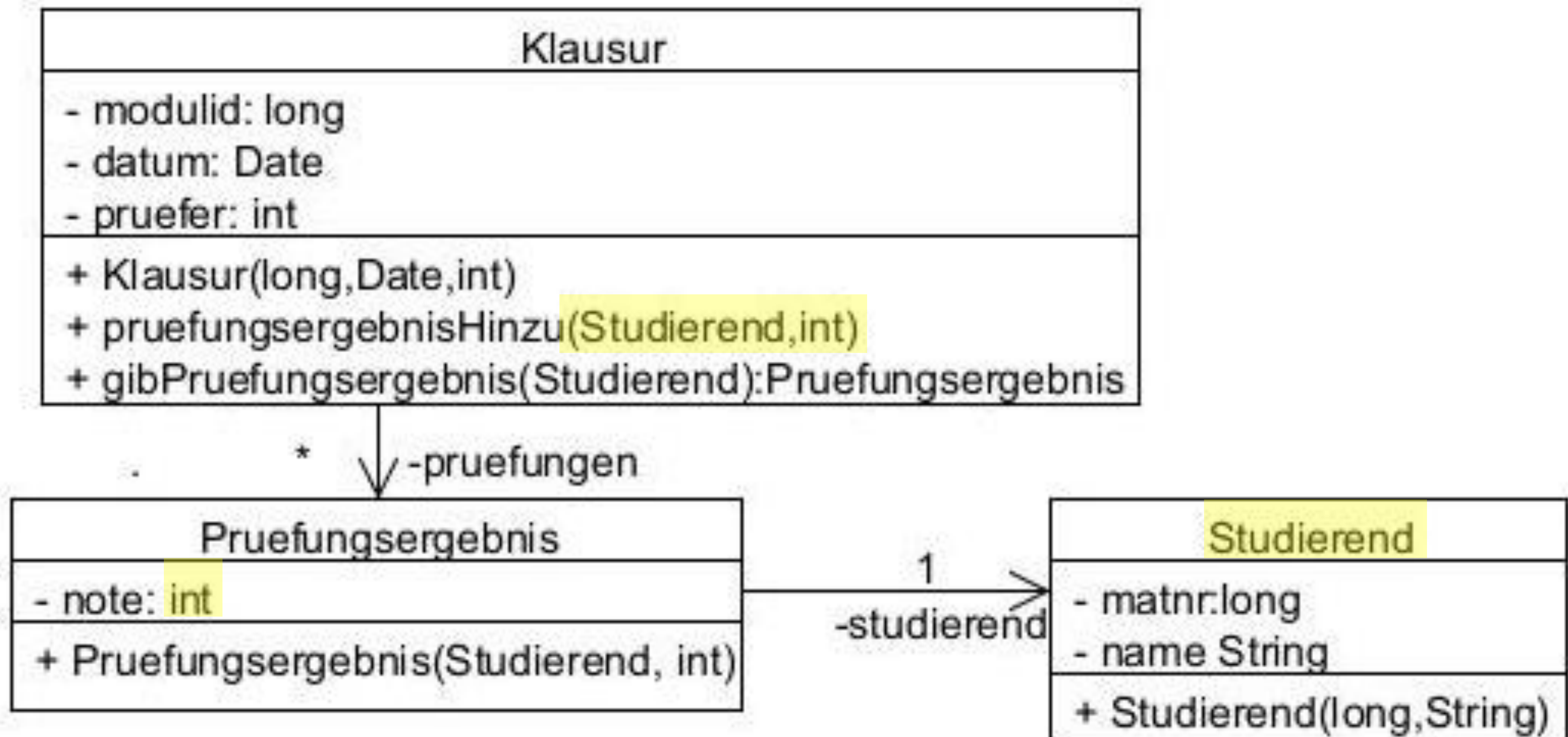
- kann Objekt anlegen
- kann Objekt mit gegebenem Identifikator suchen
- kann löschen
- …



Beispiel für Design-Idee (2/5)

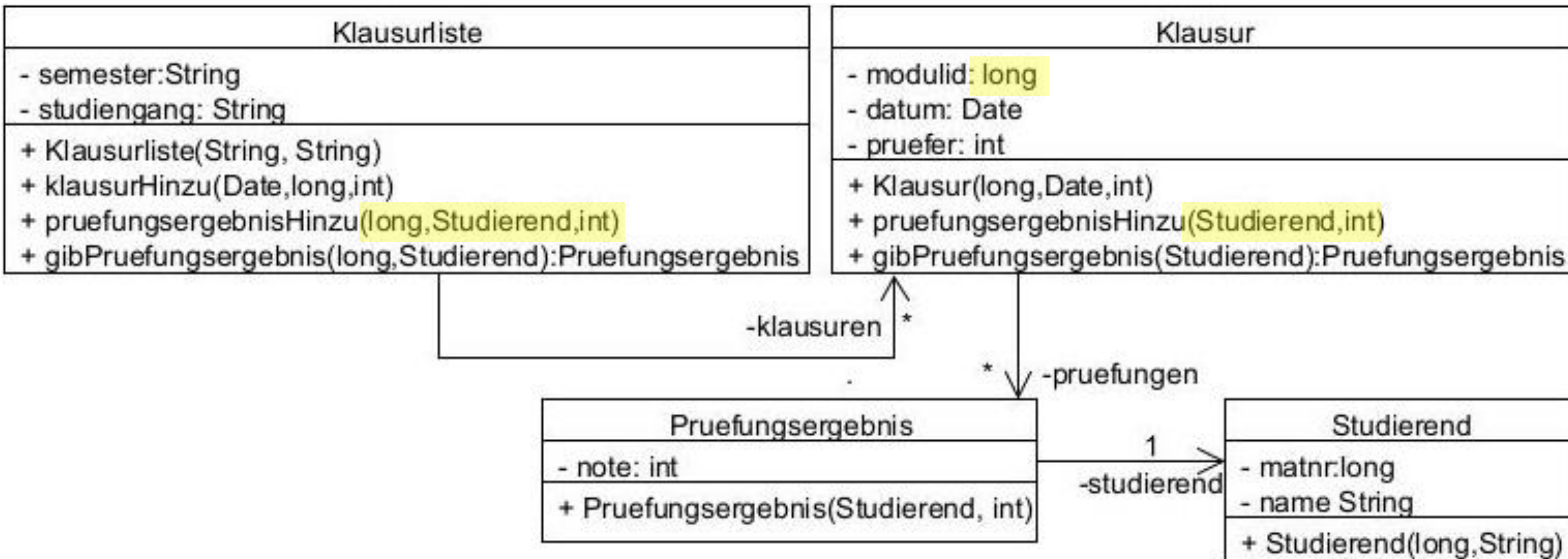
Objektsammlungen können auch Teil anderer Objekte sein, die bieten wieder: anlegen, suchen, ändern, löschen

- Klausur bekommt Studierend und Note um Pruefungsergebnis zu erzeugen und dann zu verwalten



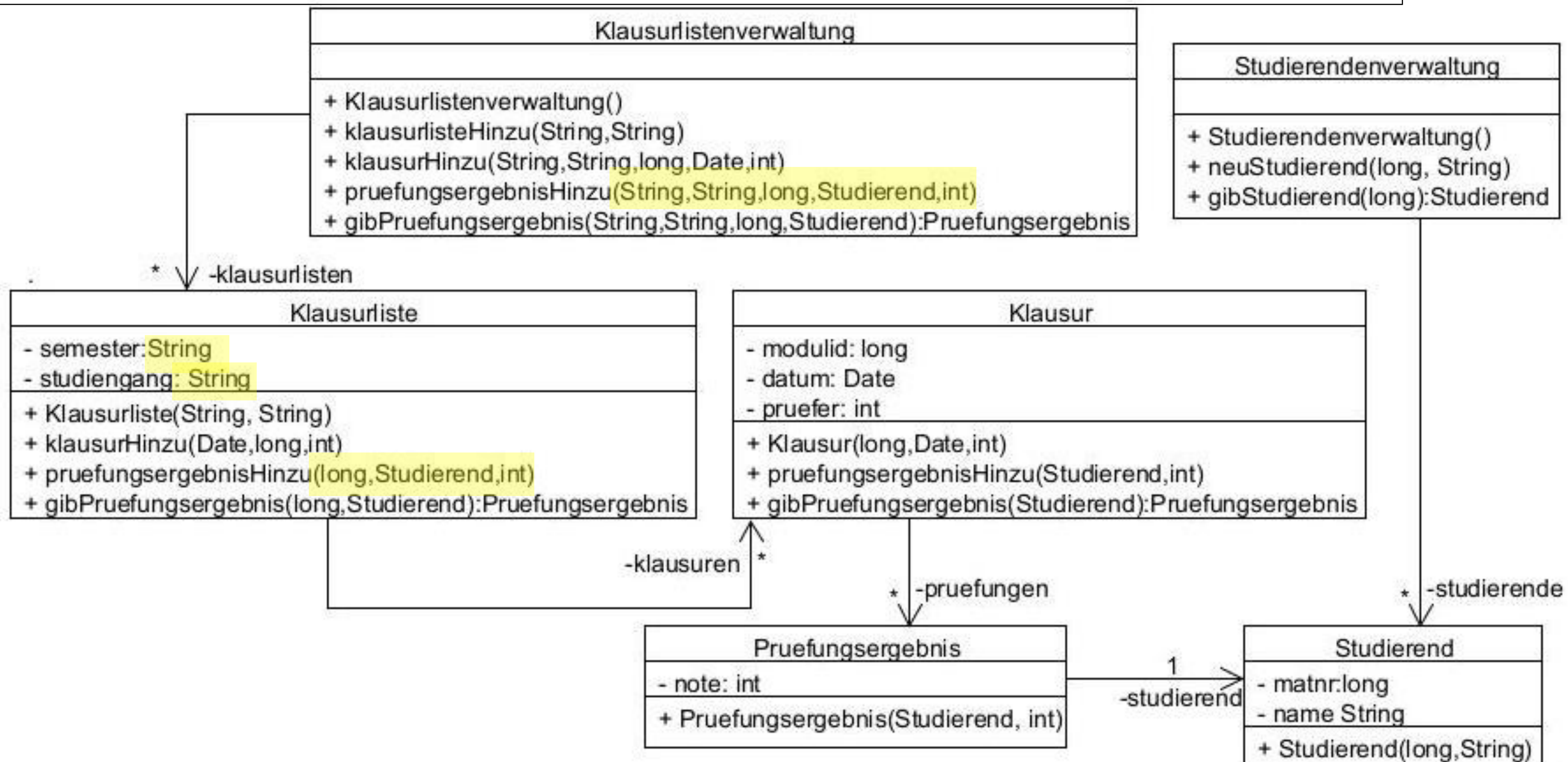
Beispiel für Design-Idee (3/5)

Idee fortgesetzt, man beachte zusätzlichen Parameter



- Klausurliste kann Klausur anlegen und verwalten
- Klausurliste kann Klausur mitteilen ein Pruefungsergebnis anzulegen

Beispiel für Design-Idee (4/5)



- Klausurlistenverwaltung kann Klausurliste anlegen
- Klausurlistenverwaltung kann Klausurliste mitteilen, ein Pruefungsergebnis anzulegen (über Klausur)

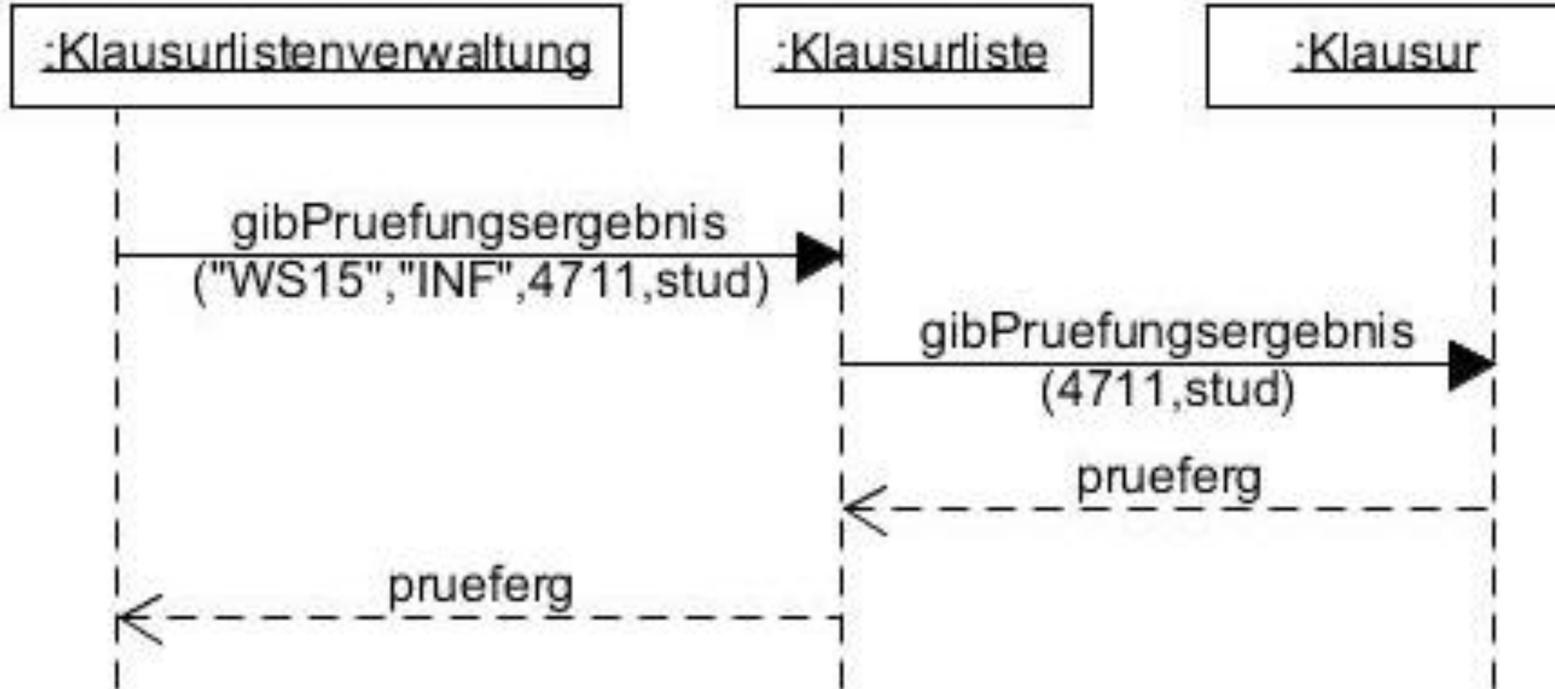
Beispiel für Design-Idee (5/5)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- Ausblick: Objekterzeugung und erstes Prüfungsergebnis

```
Studierendenverwaltung sv = new Studierendenverwaltung()
sv.neuStudierend(42, "Ronja");
Studierend stud = sv.gibStudierend(42);

KlausurlistenVerwaltung kv = new Klausurlistenverwaltung();
kv.klausurlisteHinzu("WS19", "Inf");
kv.klausurHinzu("WS19", "Inf" ,4711 ,"23.12.15" ,0);
kv.pruefungsergebnisHinzu("WS19", "Inf", 4711, stud ,170);
// letzte Methode intern:
//   kv sucht passende Klausurliste kli für ("WS19", "Inf")
//   kli sucht passende Klausur kla für ("Inf", 4711)
//   kla erzeugt neues Prüfungsergebnis und fügt es kla hinzu
```

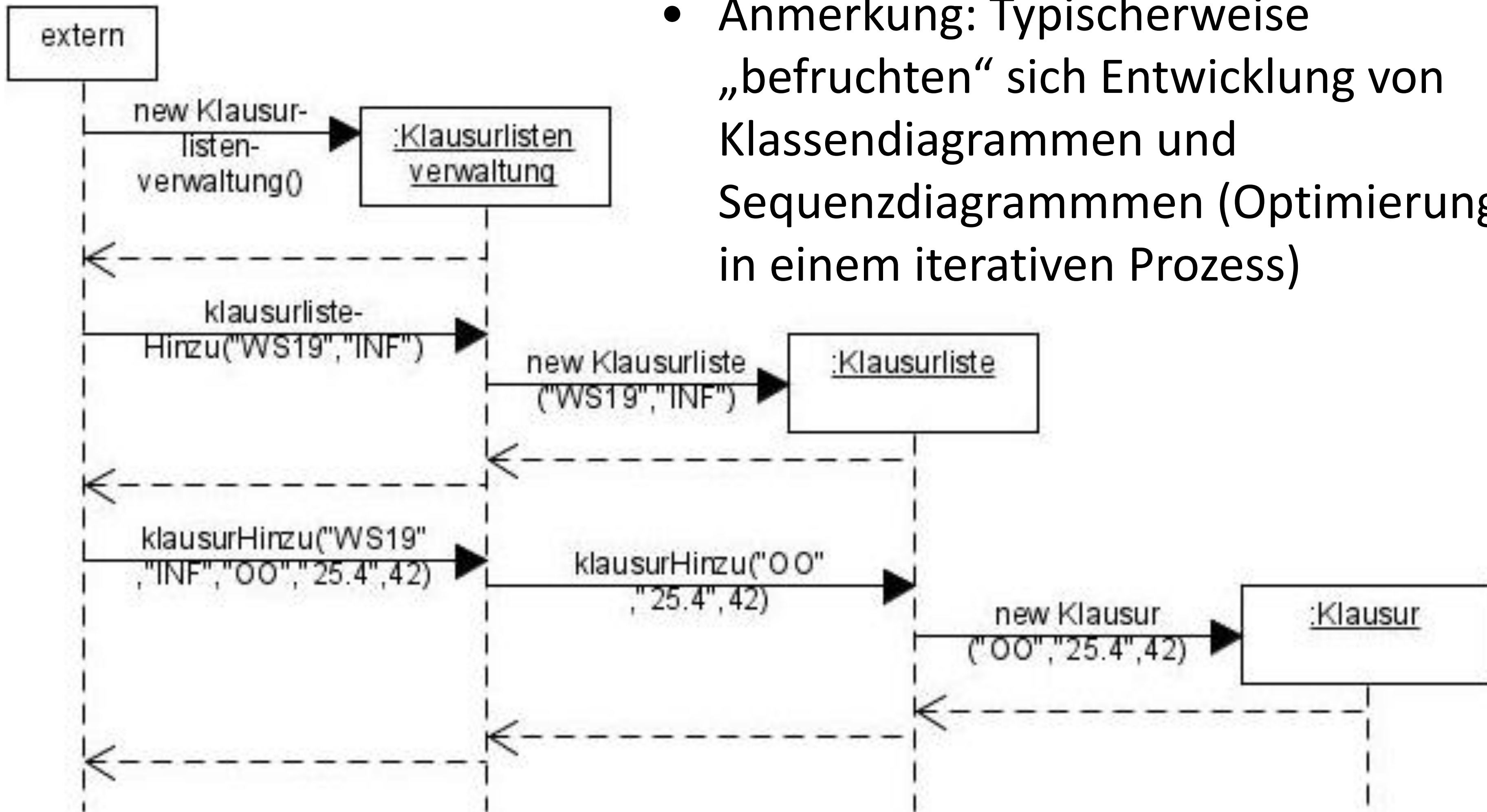
Typisches Sequenzdiagramm



- Objekte in Kopfzeile existieren (woher uninteressant)
- z. B. Klausur-Objekt hat Methode gibPruefungsergebnis(.,.)
- Parameter konkret (4711) oder abstrakt (stud) angebbar, gleiches für Ergebnisse (Rückgabewerte)
- hier interne Methode klausurSuchen(.,.) weggelassen

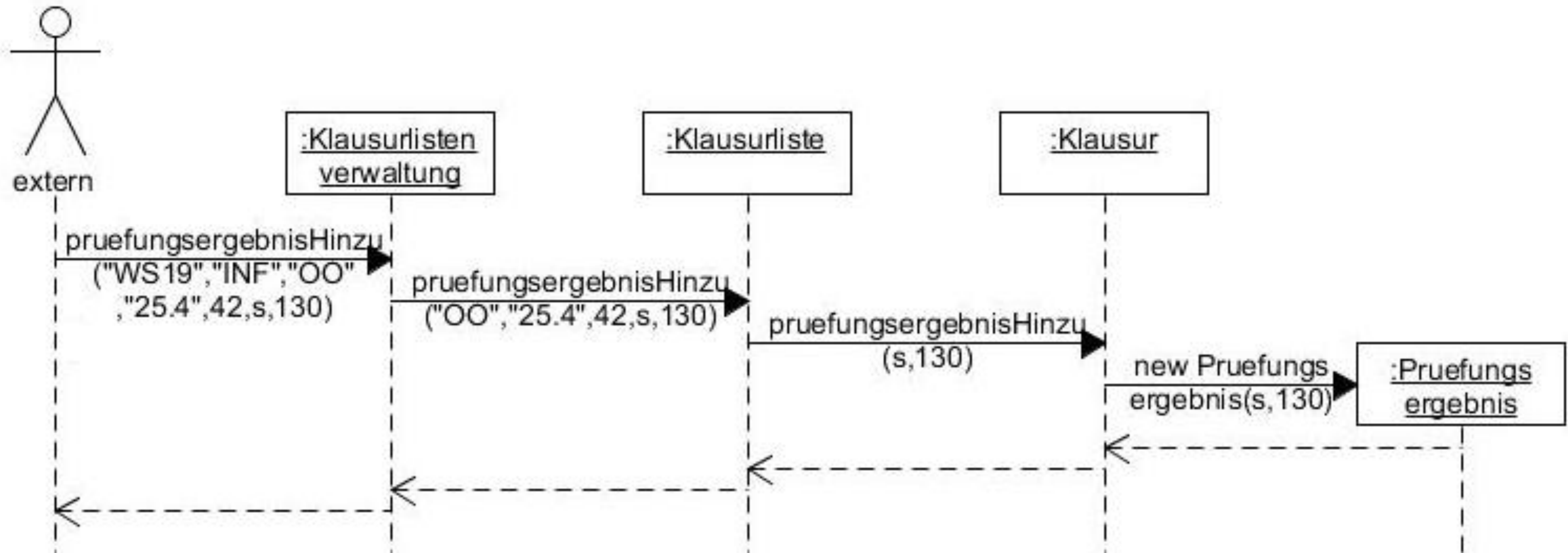
Beispiel: Initialisierung

- Anmerkung: Typischerweise „befruchten“ sich Entwicklung von Klassendiagrammen und Sequenzdiagrammmen (Optimierung in einem iterativen Prozess)



Beispiel: Anstoß der Funktionalität

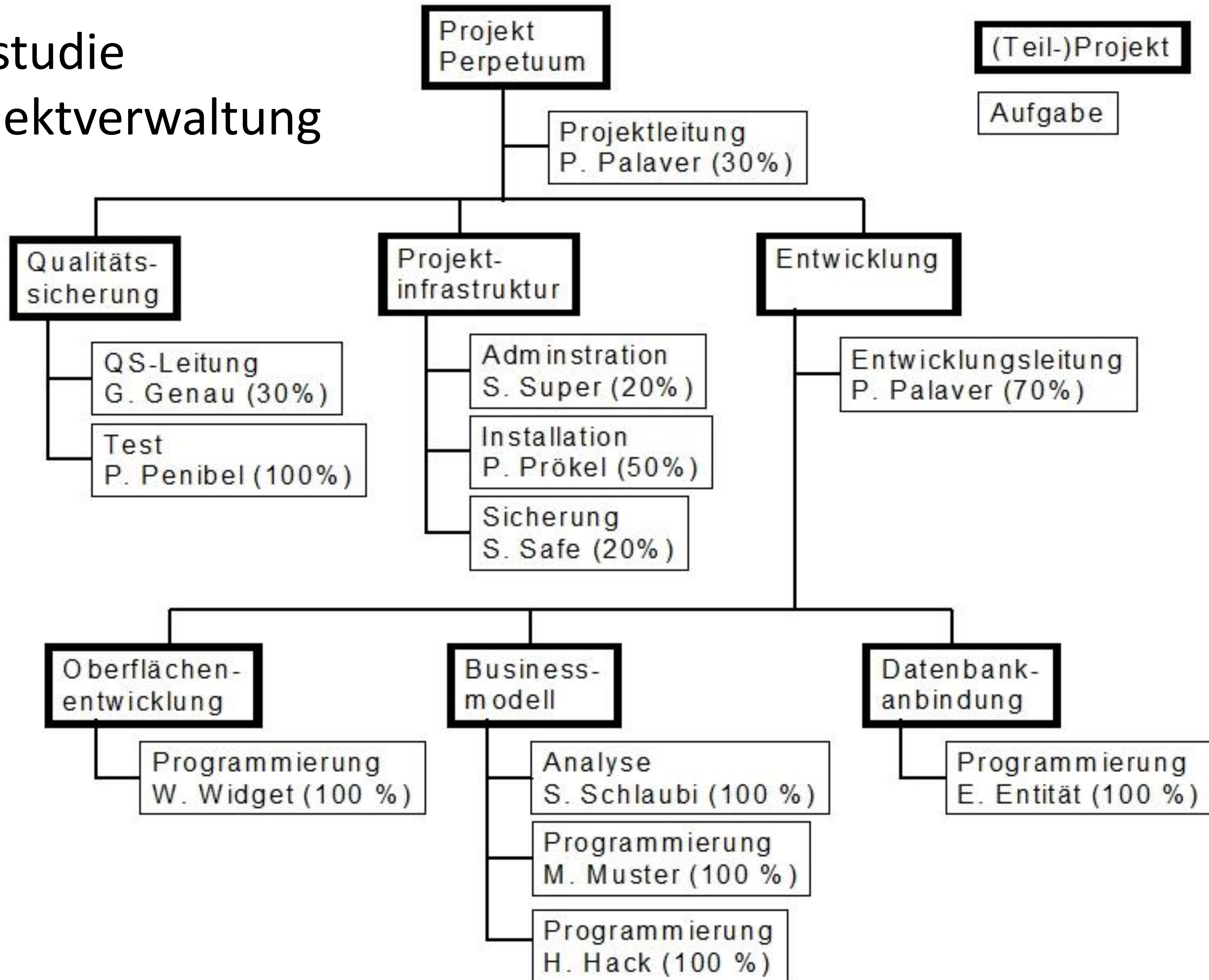
- Ablauf zeigt wieder die konsequente Delegation
- Verwaltung erhält Auftrag, nutzt teile der Parameter Zielobjekt zu bestimmen und gibt Aufruf mit restlichen Parametern an Zielobjekt weiter



Beispiel: Projektstrukturplan

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

5.2 Fallstudie Projektverwaltung



Erste Iteration: Klassen finden

- Aktivitätsdiagramme werden durch Anforderungen konkretisiert
- Text der Anforderungen ist Grundlage zum Finden erster Klassen
- Im Text werden Objekte identifiziert; sind Individuen, die durch Eigenschaften (Exemplarvariablen) und angebotene Funktionalität charakterisiert werden
- grober Ansatz: Nomen in Anforderungen und *Glossar ansehen*; können Objekte oder Eigenschaften sein
- Adjektive können auf Eigenschaften hindeuten
- Informationen in Klassen gesammelt; Klassen beschreiben Struktur, die jedes Objekt hat
- verwandter Begriff: Domain Model

Analyse der Anforderungen – Ausschnitt 1. Iteration

A1.1: In der Projektbearbeitung muss das System der nutzenden Person die Möglichkeit bieten, ein neues Projekt mit Projektausgangsdaten anzulegen.

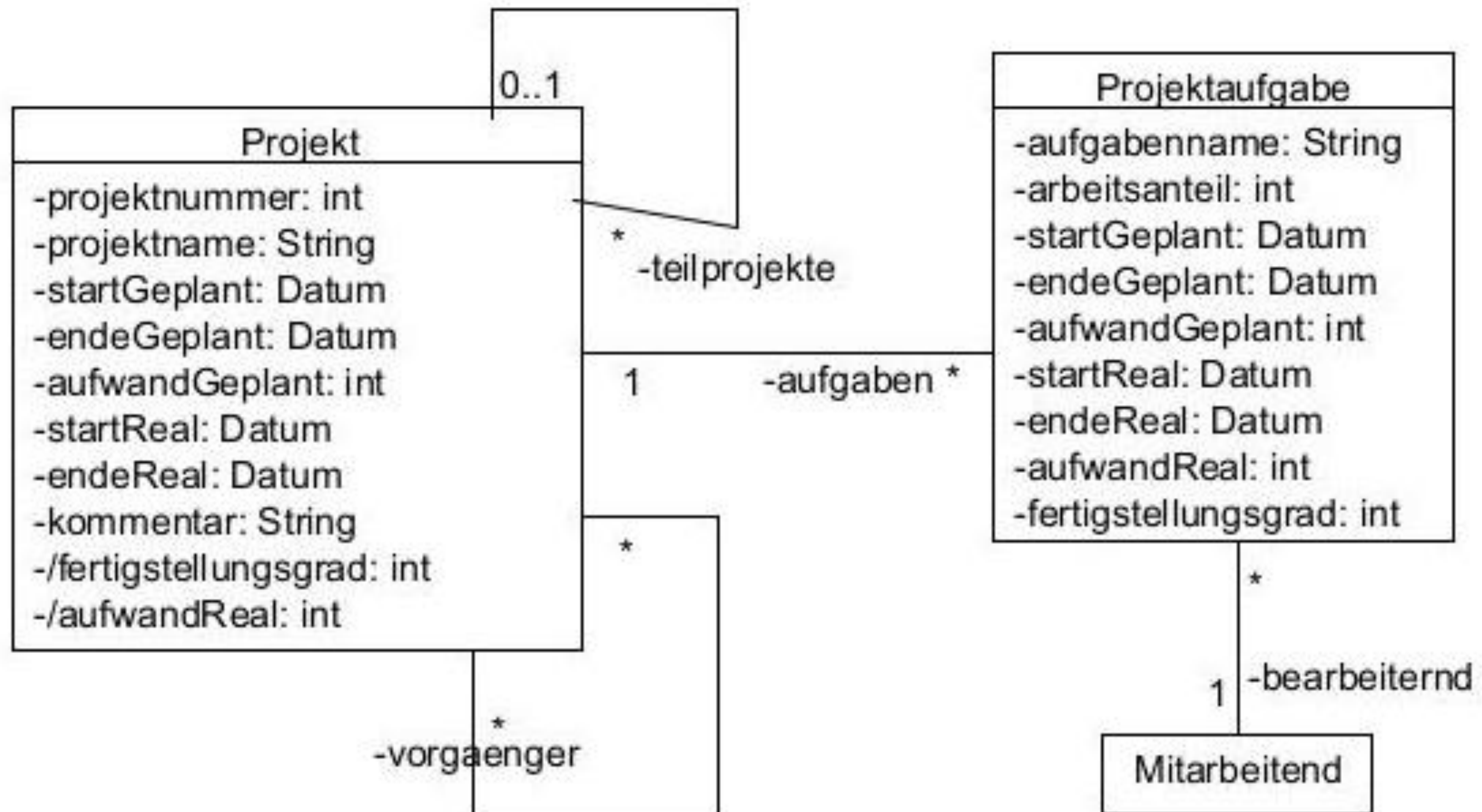
- Glossar Projektausgangsdaten: automatisch vergebene eindeutige Projektnummer, Projektname, geplanter Start- und Endtermin, geplanter Aufwand
- *gefunden: Klasse Projekt mit Exemplarvariablen Projektnummer, Projektname, geplanter Starttermin, geplanter Endtermin, geplanter Aufwand*

A1.2: Nach Abschluss der Eingabe (mit „Return"-Taste oder Bestätigungsknopf) bei der Bearbeitung von Daten muss das System neu eingegebene Daten in seine permanente Datenhaltung übernehmen.

A1.3: In der Projektbearbeitung muss das System der nutzenden Person die Möglichkeit bieten, jedes Projekt auszuwählen.

- gefunden: keine Klassen oder Exemplarvariablen (Funktionalität später)

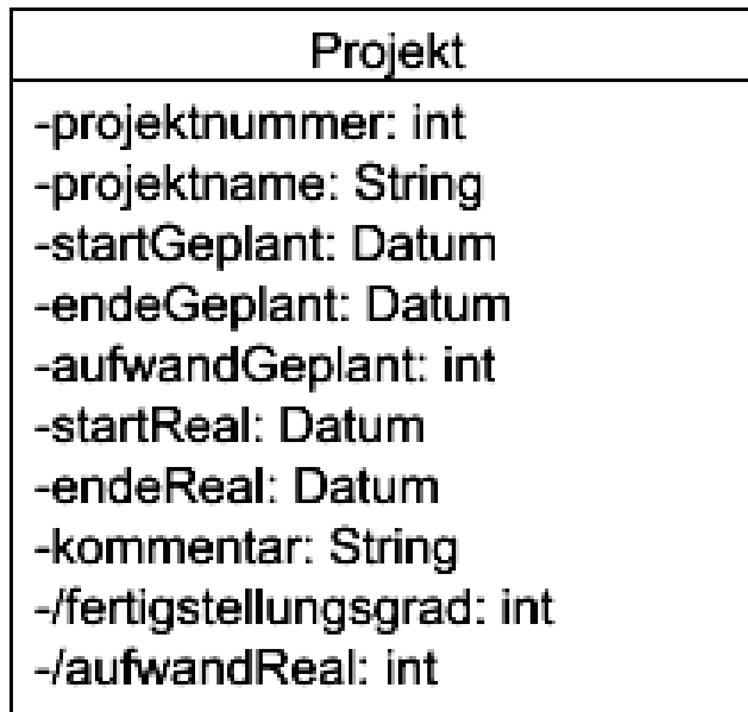
UML-Notation

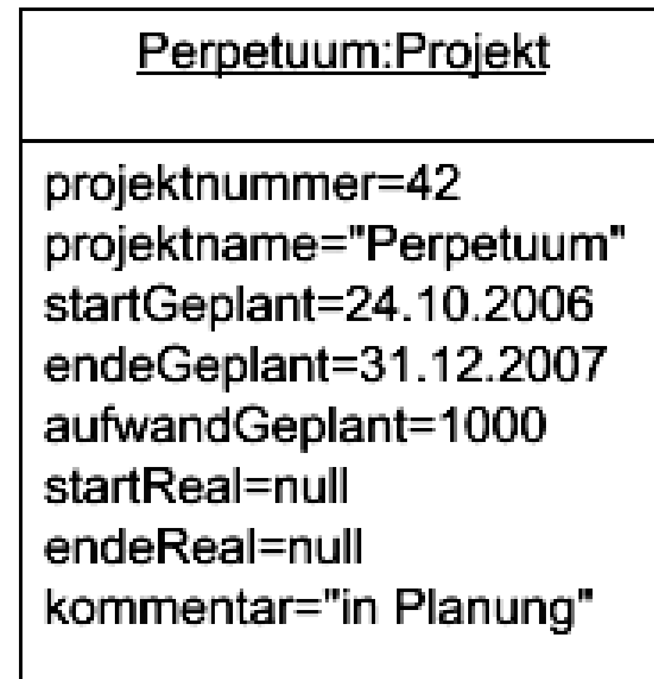


- / bedeutet abgeleitet, d. h. kann aus anderen Modelinformationen berechnet werden (meist in Modellen weggelassen)

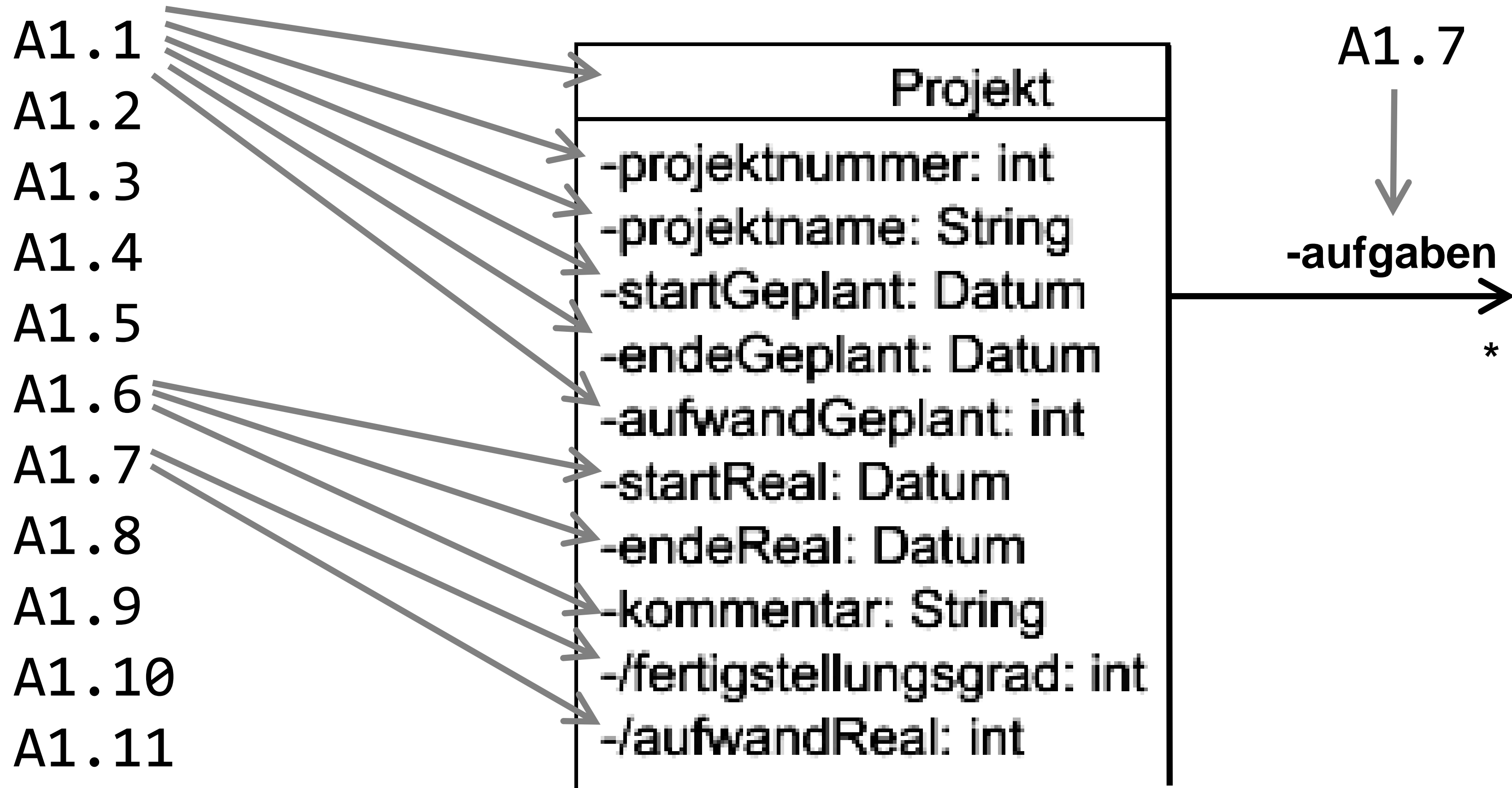
Zusammenhang Klasse und Objekt

- Objekte lassen sich auch in der UML darstellen
- Kasten mit unterstrichenem „:<Klassenname>“
- vor Doppelpunkt optional Objektname
- Objekte werden nicht im Klassendiagramm dargestellt (aber in Sequenzdiagrammen, dann ohne Objektvariablen)





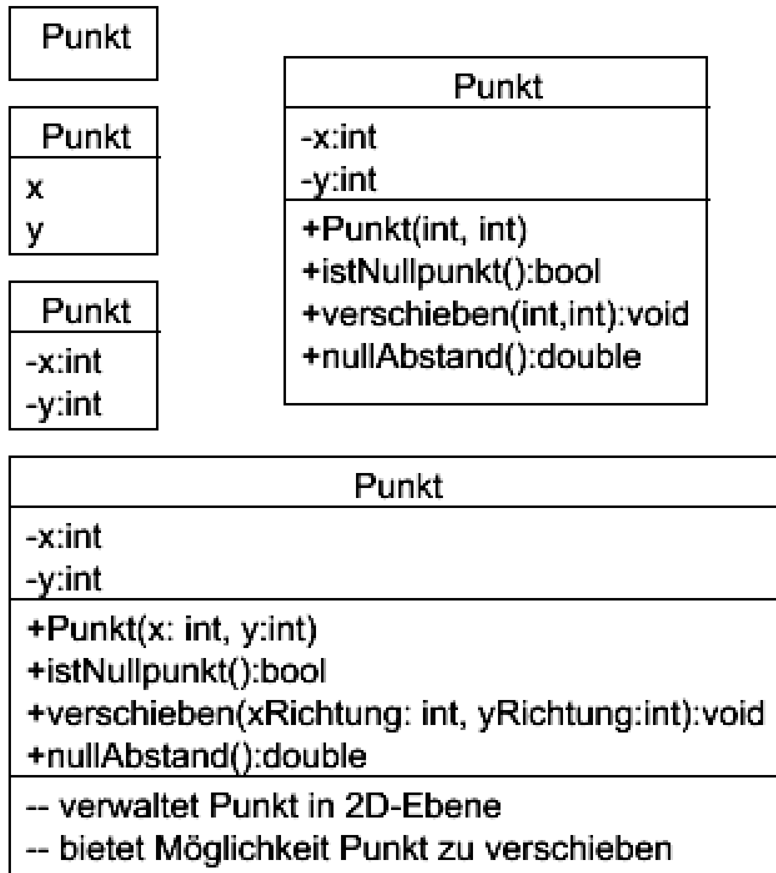
Tracing-Information (was wo) festhalten

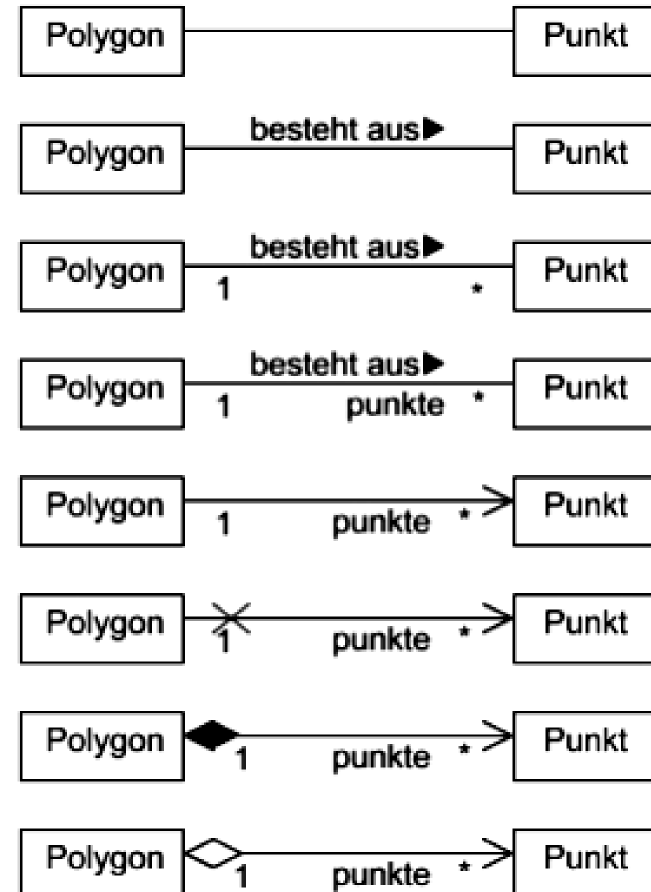


- Zuordnung welche Anforderung wie (ganz, teilweise) in welchen UML-Elementen umgesetzt
- (besser in einem Tool oder Text)

UML unterstützt iteratives Vorgehen

- UML-Teile weggelassen bzw. ausblenden, abhängig von notwendigen bzw. vorhandenen Teilinformationen
- Je implementierungsnäher desto detaillierter





2. Iteration: Methoden suchen

5.3

- Methoden stehen für Funktionalität, die ein Objekt anbietet; typisch: Zustand (d. h.) Exemplarvariable ändern, Ergebnis basierend auf Exemplarvariablen berechnen
- Ansatz 1: Analysiere Verben im Text
- Ansatz 2: Aus Use Cases lässt sich häufig eine Steuerungsklasse (Koordinationsklasse) ableiten
- folgende Anforderungen an die Klassenformulierung müssen beachtet werden:
  - Klassen übernehmen jeweils eine Aufgabe und besitzen genau die zur Durchführung benötigten Methoden und die für die Methoden benötigten Exemplarvariablen
  - Klassen sollen möglichst wenig andere Klassen kennen, wodurch die Schnittstellenanzahl gering gehalten wird
- (Hinweis: unser Projektverwaltungsbeispiel ist datenlastig, deshalb wenige Methoden)

Beispiel: zweite Analyse der Anforderungen

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

A1.3: In der Projektbearbeitung muss das System die Möglichkeit bieten, jedes Projekt auszuwählen.

- *Steuerungsklasse Projektverwaltung*
- *Exemplarvariablen: alle Projekte und selektiertes Projekt*
- *Projektauswahl ist set-Methode*

A1.4: Nach der Projektauswahl muss das System der nutzenden Person die Möglichkeit bieten, für existierende Projekte neue Teilprojekte anzulegen.

- *Wie bei Mengen von Werten üblich, wird meistens eine add- und eine delete-Methode gefordert. In diesem Fall nur teilprojektHinzufuegen(Projekt): void*

A1.7: Nach der Projektauswahl muss das System der nutzenden Person die Möglichkeit bieten, neue Projektaufgaben mit dem Aufgabennamen, dem geplanten Start- und Endtermin, dem Arbeitsanteil der mitarbeitenden Person und dem geplanten Aufwand zu definieren.

- *Projekt hat Methode aufgabeHinzufuegen(Projektaufgabe): void*
- *Konstruktor Aufgabe(String, Datum, Datum, int, int)*

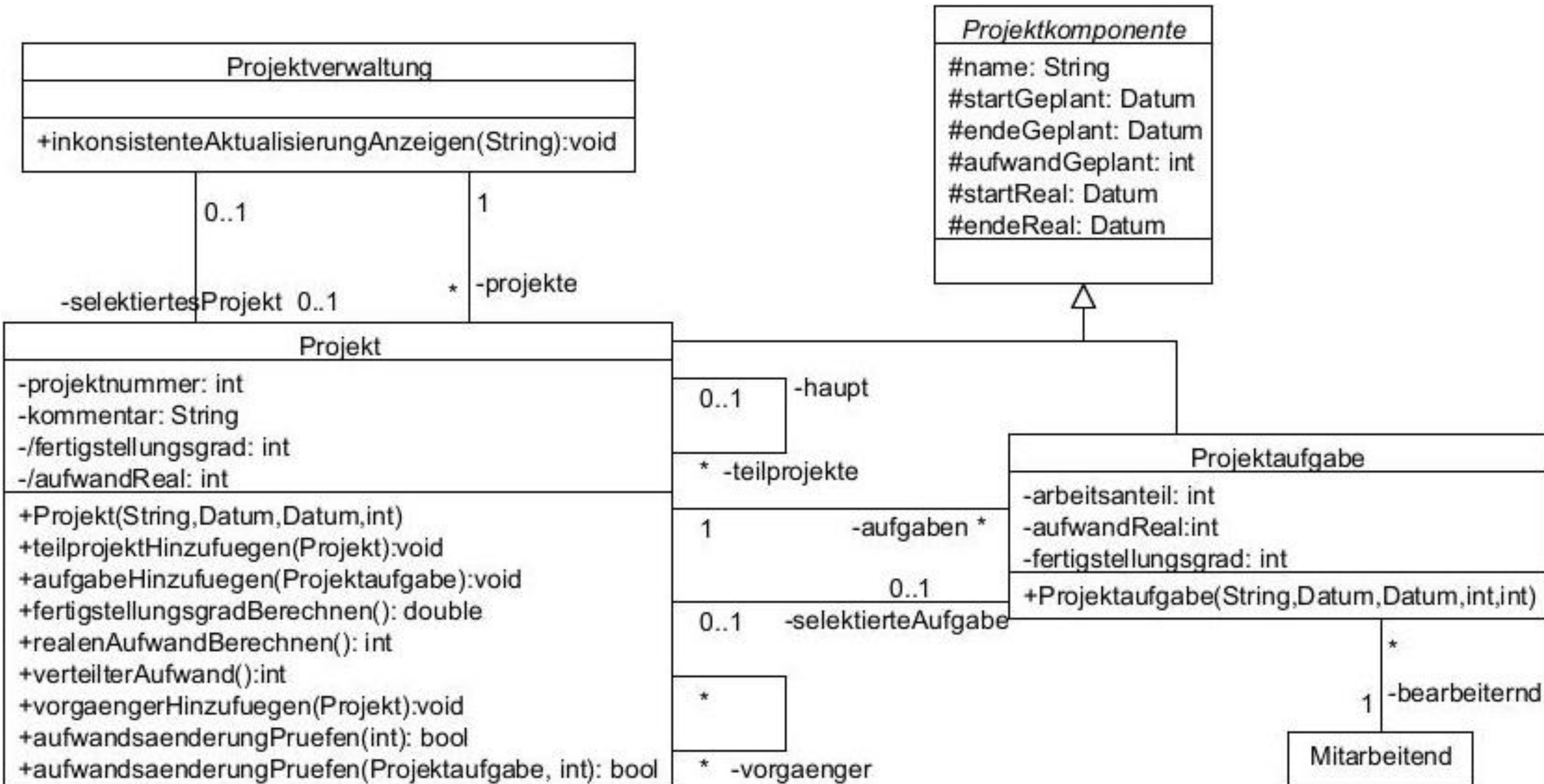
Klassendiagramm

Vererbung

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- Analysemodell wird auf erste Optimierungen geprüft
- Wenn Objekte verschiedener Klassen große Gemeinsamkeiten haben, kann Vererbung genutzt werden
- Variante 1: Abstrakte Klasse mit möglichen Exemplarvariablen, einigen implementierten und mindestens einer nicht-implementierten Methode
- Variante 2: Interface ausschließlich mit abstrakten Methoden (haben später noch Bedeutung)
- Vererbung reduziert den Codierungsaufwand
- Vererbung erschwert Wiederverwendung
- Vererbung ist Hilfsmittel nicht Ziel der Objektorientierung
- Liskovsches Prinzip für überschreibende Methoden der erbenden Klassen berücksichtigen:
  - Vorbedingung gleich oder abschwächen
  - Nachbedingungen gleich oder verstärken

Beispiel: Vererbung



nächster Schritt: Prüfen, wo statt Projekt und Projektaufgabe Projektkomponente stehen kann (Abstrahierung)

Klassen: von Analyse zum Design

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- hier steht zunächst Analyseklassenmodell im Vordergrund, dass meist nicht genauso implementiert wird
- Klassenmodell wird schrittweise in Richtung „sinnvoll programmierbar“ umgebaut
- in „sinnvoll“ gehen Erfahrungen und Randbedingungen ein (z. B. Web-Applikation)
- Erfahrungen zum guten Design werden u. a. mit Design-Pattern dokumentiert (wichtig, aber später)
- mit Design-Erfahrungen wird erstes Klassenmodell bei Erstellung besser (gibt dann nur ein zentrales Klassenmodell)

Validierung mit Sequenzdiagrammen

5.4

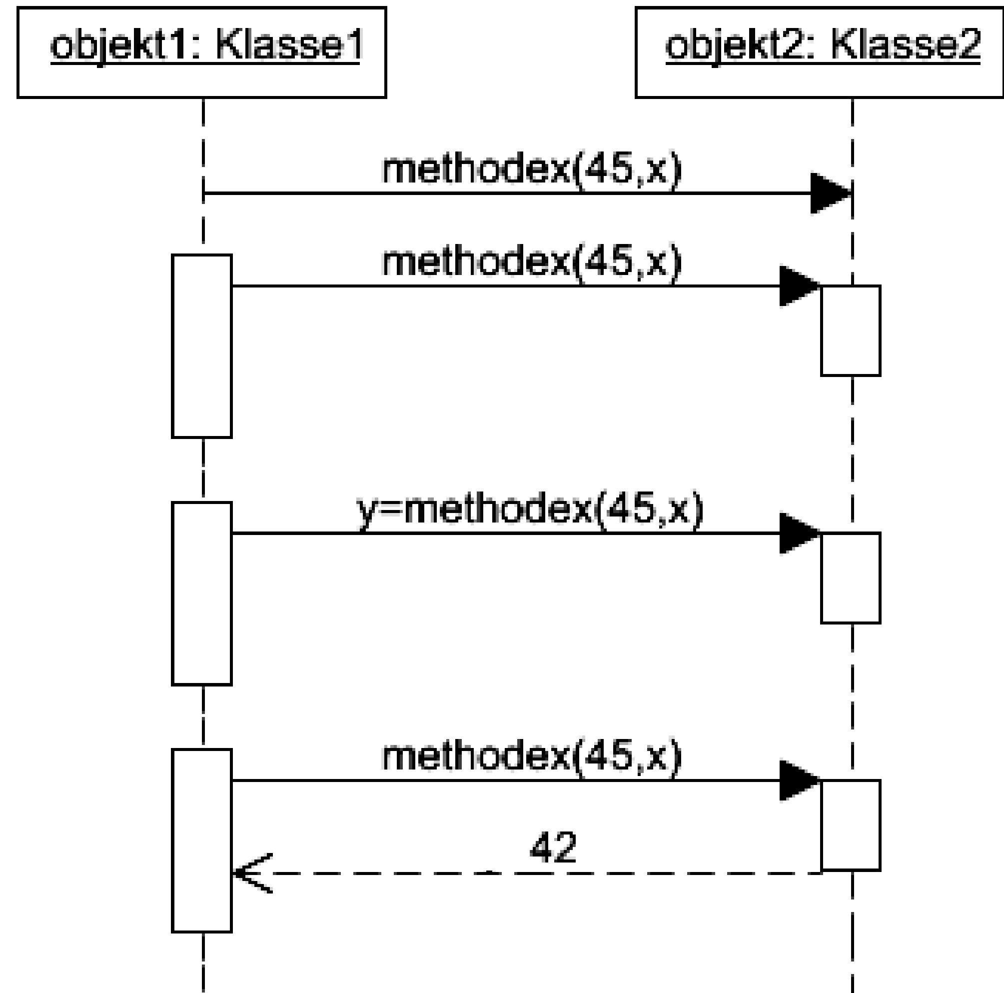
- Sequenzdiagramme beschreiben, wie Objekte bei anderen Objekten Methoden aufrufen
- Mit Hilfe des erreichten Modells kann man mit Sequenzdiagrammen validieren, ob die im Aktivitätsdiagramm beschriebenen Abläufe möglich sind
- Sequenzdiagramme in der klassischen Form beschreiben damit Beispielabläufe

Darstellungsvarianten in Sequenzdiagrammen

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- rechte Seite zeigt verschiedene Darstellungsmöglichkeiten eines Methodenaufrufs
- Rückgabewerte werden weggelassen, wenn nur Ablauf wichtig
- Aktivitätsbalken (optional) verdeutlicht, dass Objekt aktiv ist (rechnet, wartet)
- visualisiert in Klasse 1 die Zeile

  y = objekt2.methodex(45,x);
- letzte Variante meist am intuitivsten (in VL genutzt ohne Aktivitätsbalken)



Iterative Entwicklung und Validierung

Video

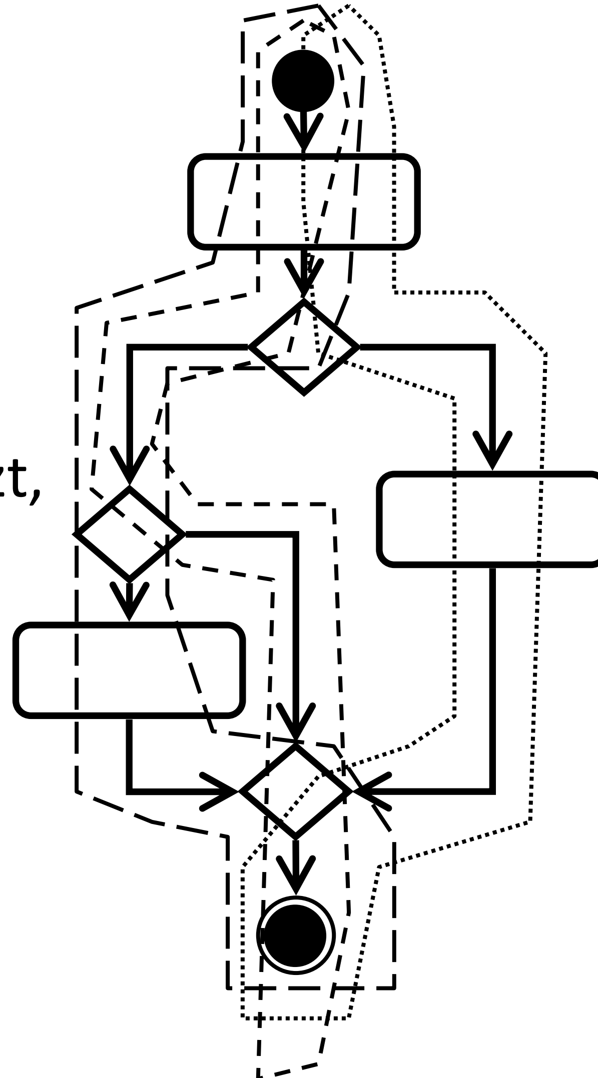
Beispielablauf

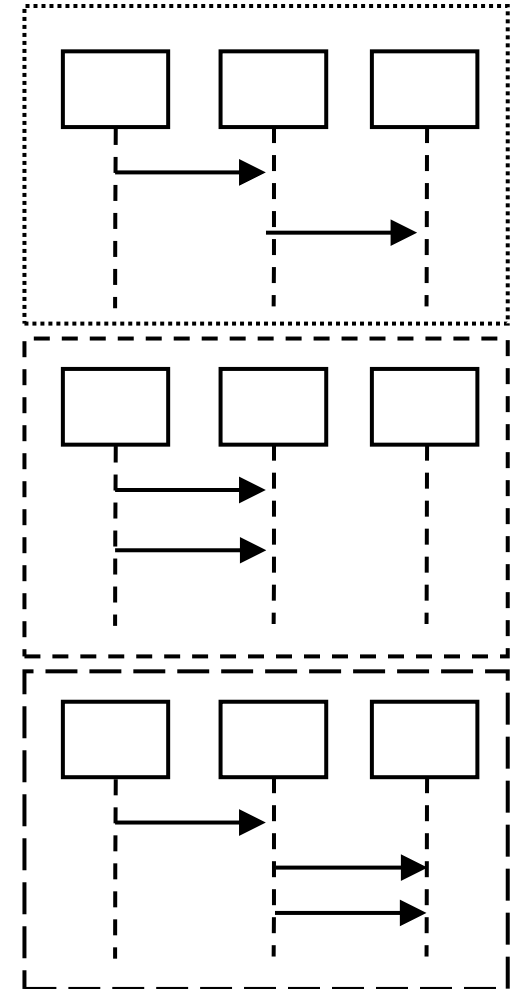
- Ableitung von Methodennamen
- Zeichnen eines kleinen Sequenzdiagramms mit dieser Methode; feststellen, ob weitere Methoden benötigt
- Ergänzung von Methodenparametern
- Ergänzung des Sequenzdiagramms um Parameter; feststellen, ob weitere Methoden benötigt

- Falls kein Sequenzdiagramm herleitbar, auf Ursachenforschung gehen (Modellfehler?)
- Optimales Ziel: Mögliche Durchläufe durch Aktivitätsdiagramme werden abgedeckt

Zusammenhang zwischen Aktivitäts- und Sequenzdiagrammen

für jeden möglichen Durchlauf durch das Aktivitätsdiagramm wird ein Sequenzdiagramm, evtl. zusammengesetzt, erstellt



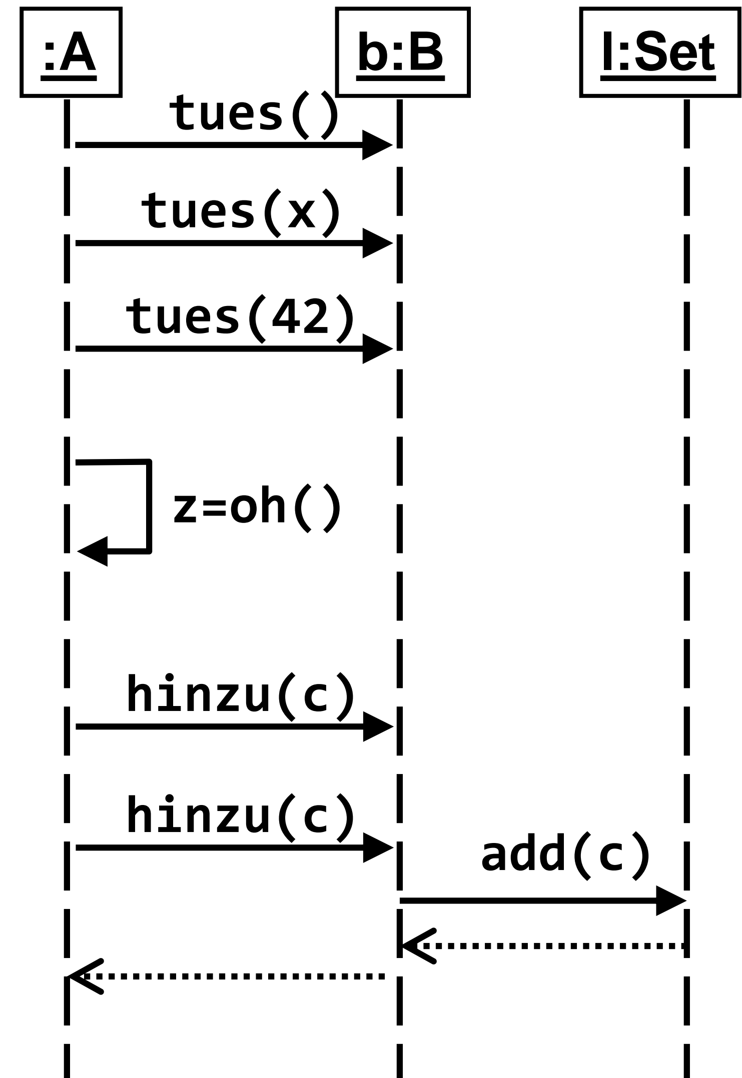


Iterative Entwicklung eines Sequenzdiagramms

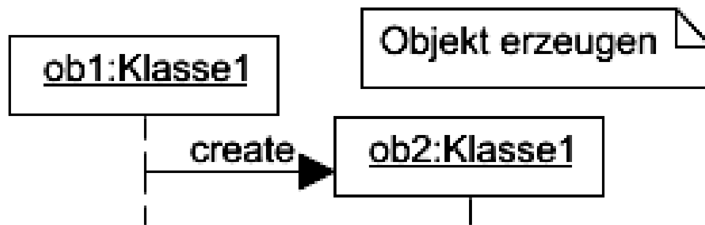
HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

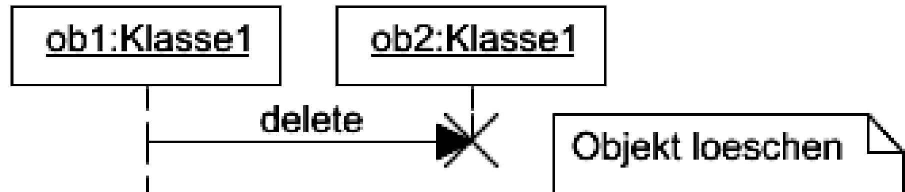
- generell: zunächst unterspezifiziert,
- dann Parameter verfeinern
- abstrakter Ablauf (x) oder konkreter Beispielablauf (mit Werten)
- Ergänzung interner Berechnungen, z. B. in A **`z = this.oh();`**
- interne Collections meist nicht dargestellt
- Darstellung aber möglich, in B:

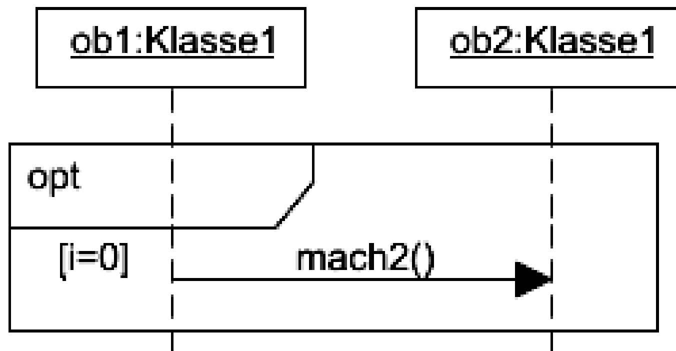
```
public void hinzu(C c){
  l.add(c);
}
```

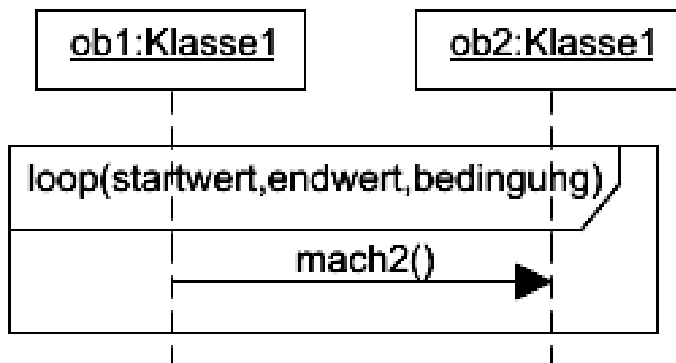


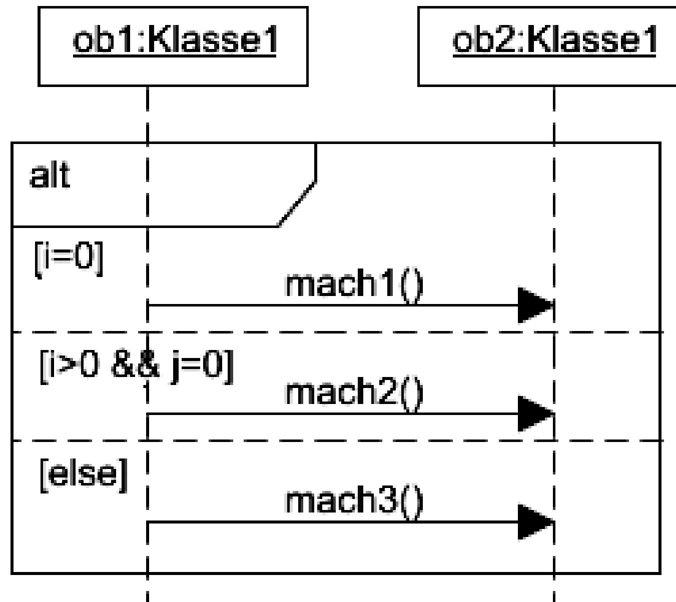
Highlevel-Sequenzdiagramme (nur Ausblick)



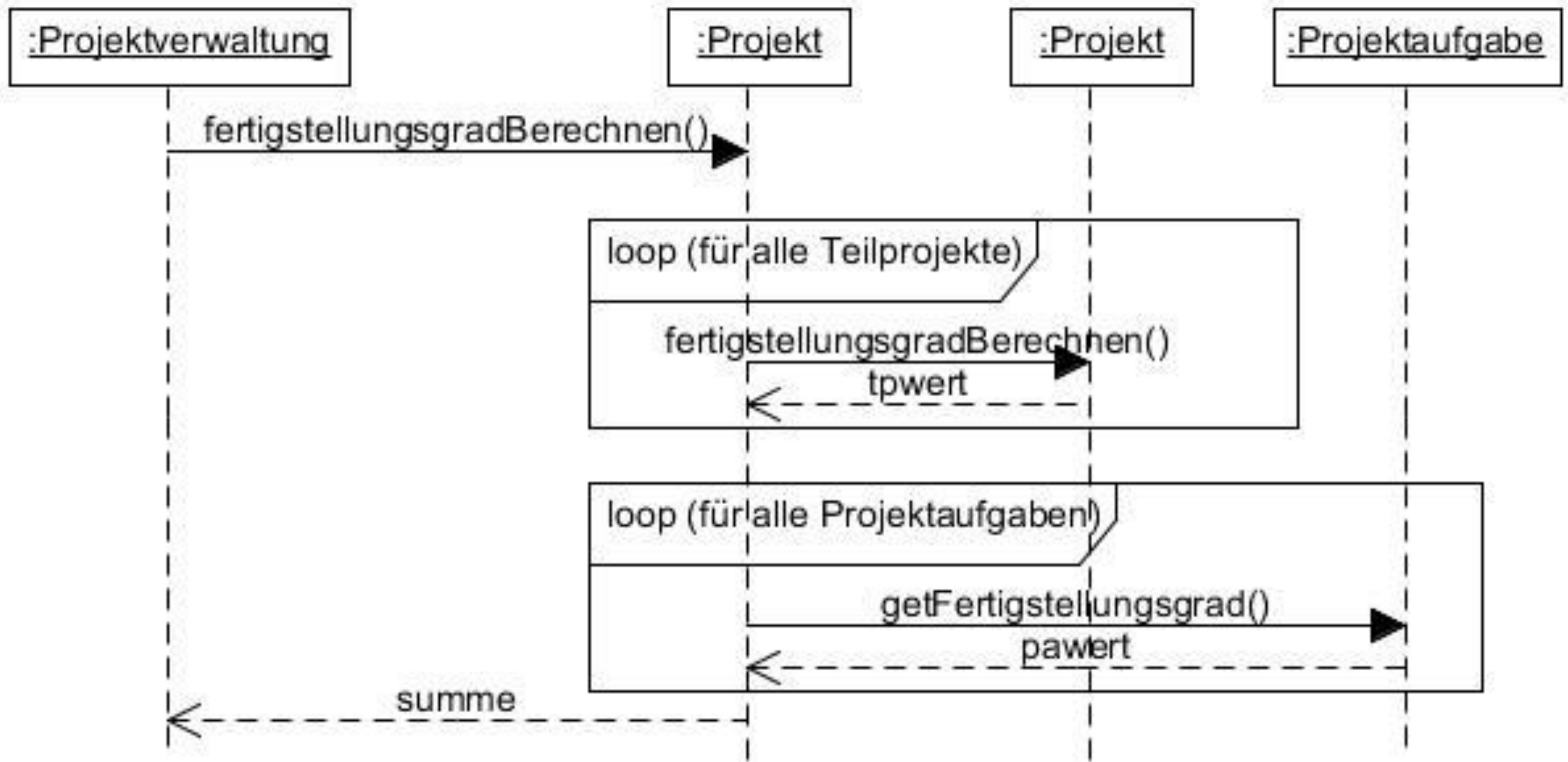






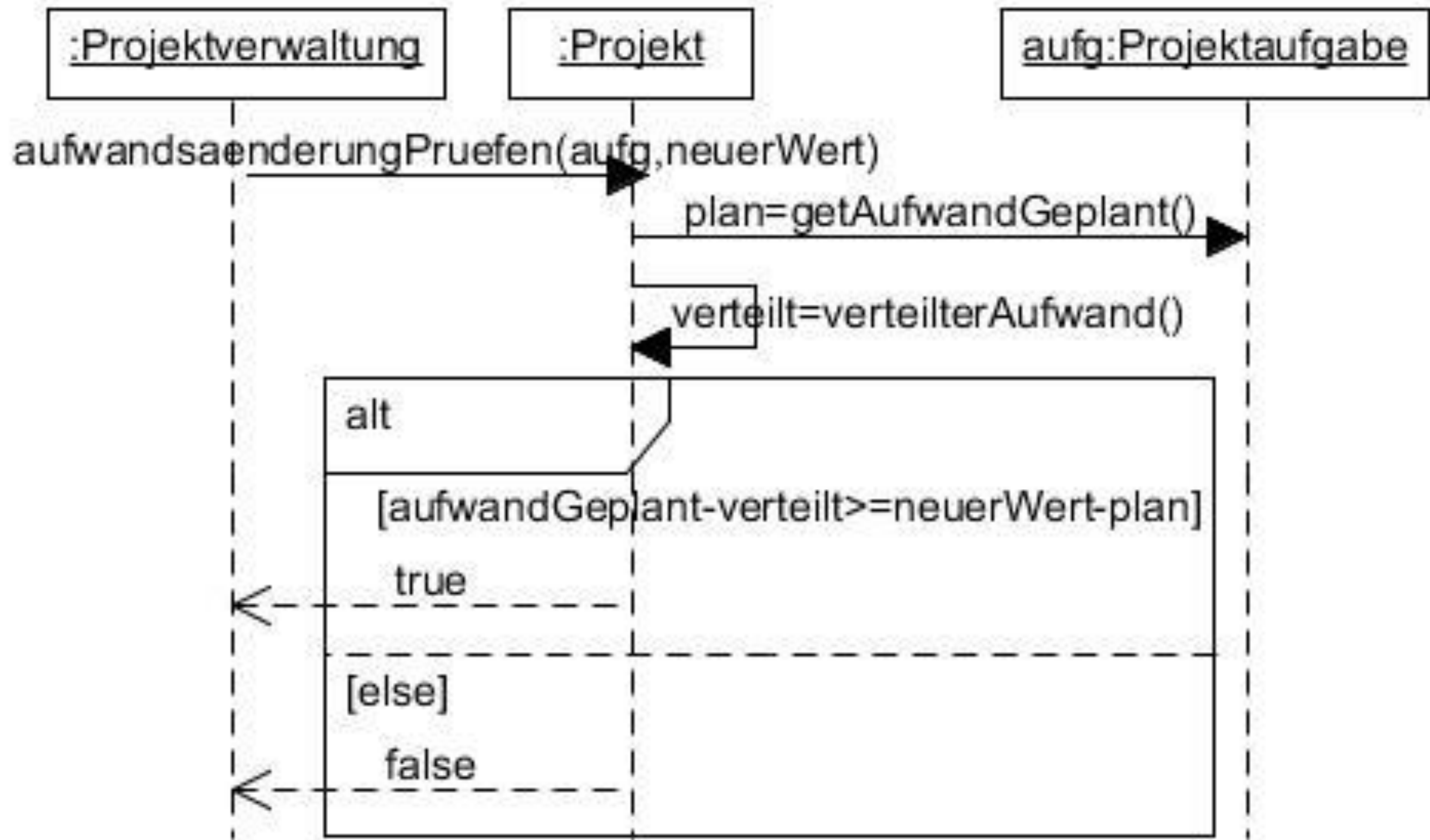


Beispiel: Fertigstellungsgrad berechnen



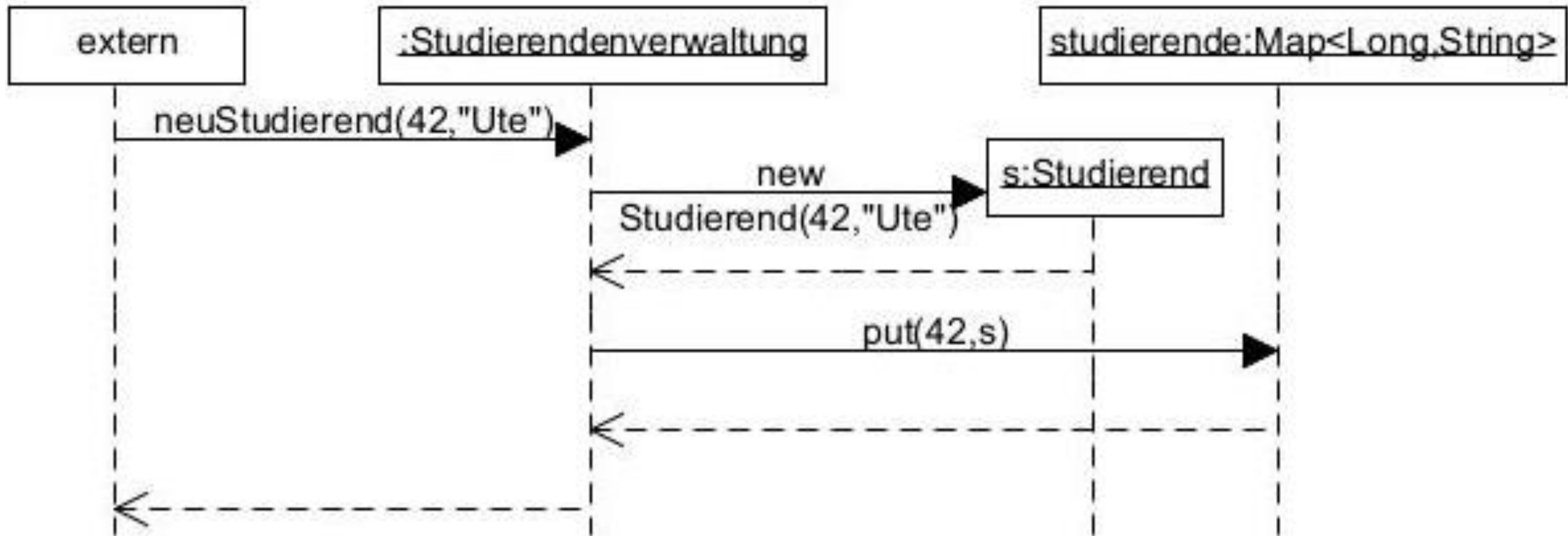
Beispiel: Prüfung Aufwandsänderung Projektaufgabe

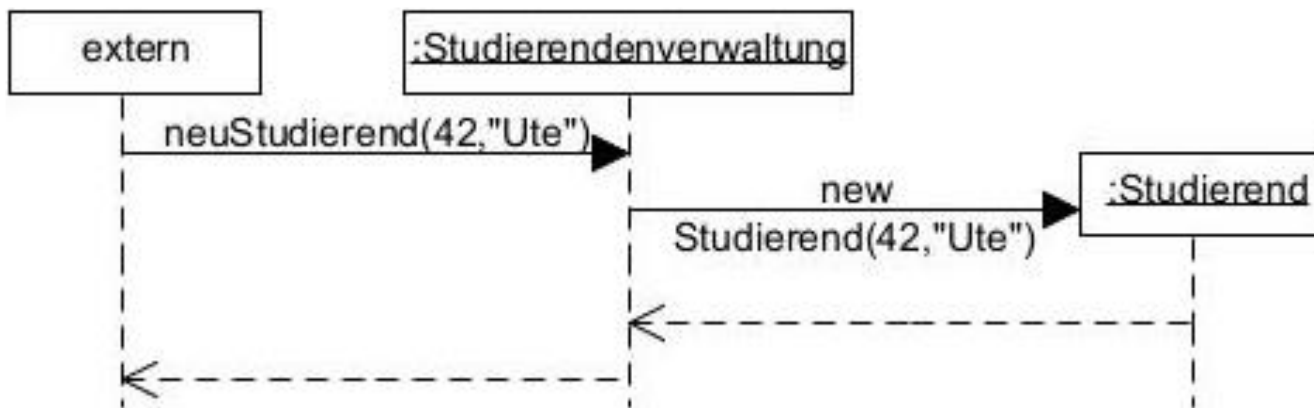
HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES



Sequenzdiagramm – Detailgrad (1/3)

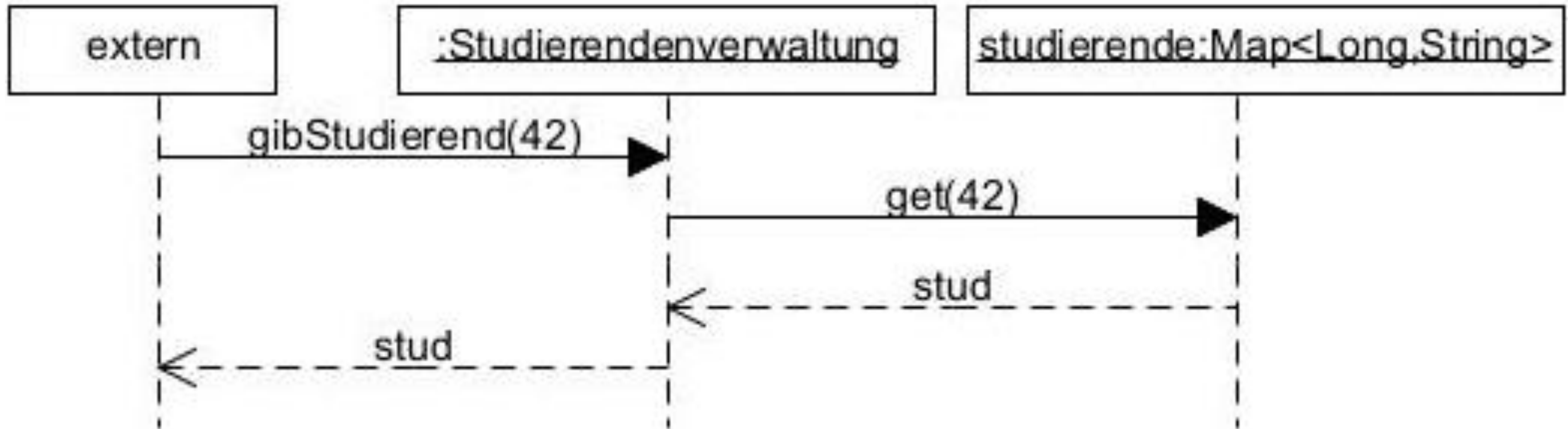
- man kann alle Objekte einzeichnen oder unwichtige weglassen

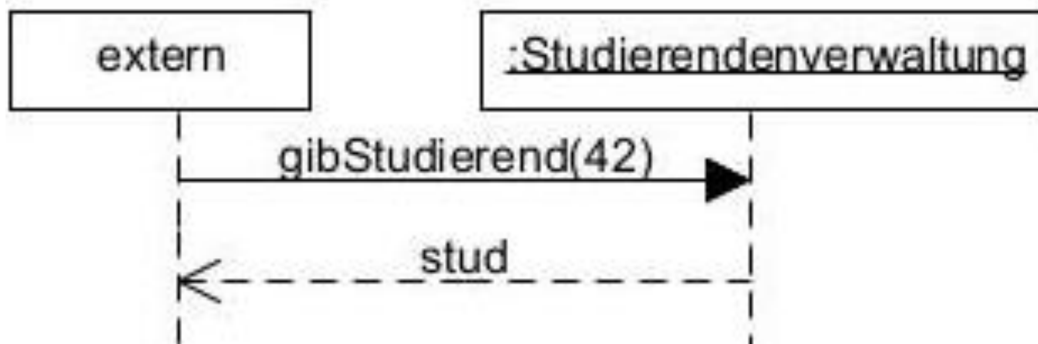




Sequenzdiagramm – Detailgrad (2/3)

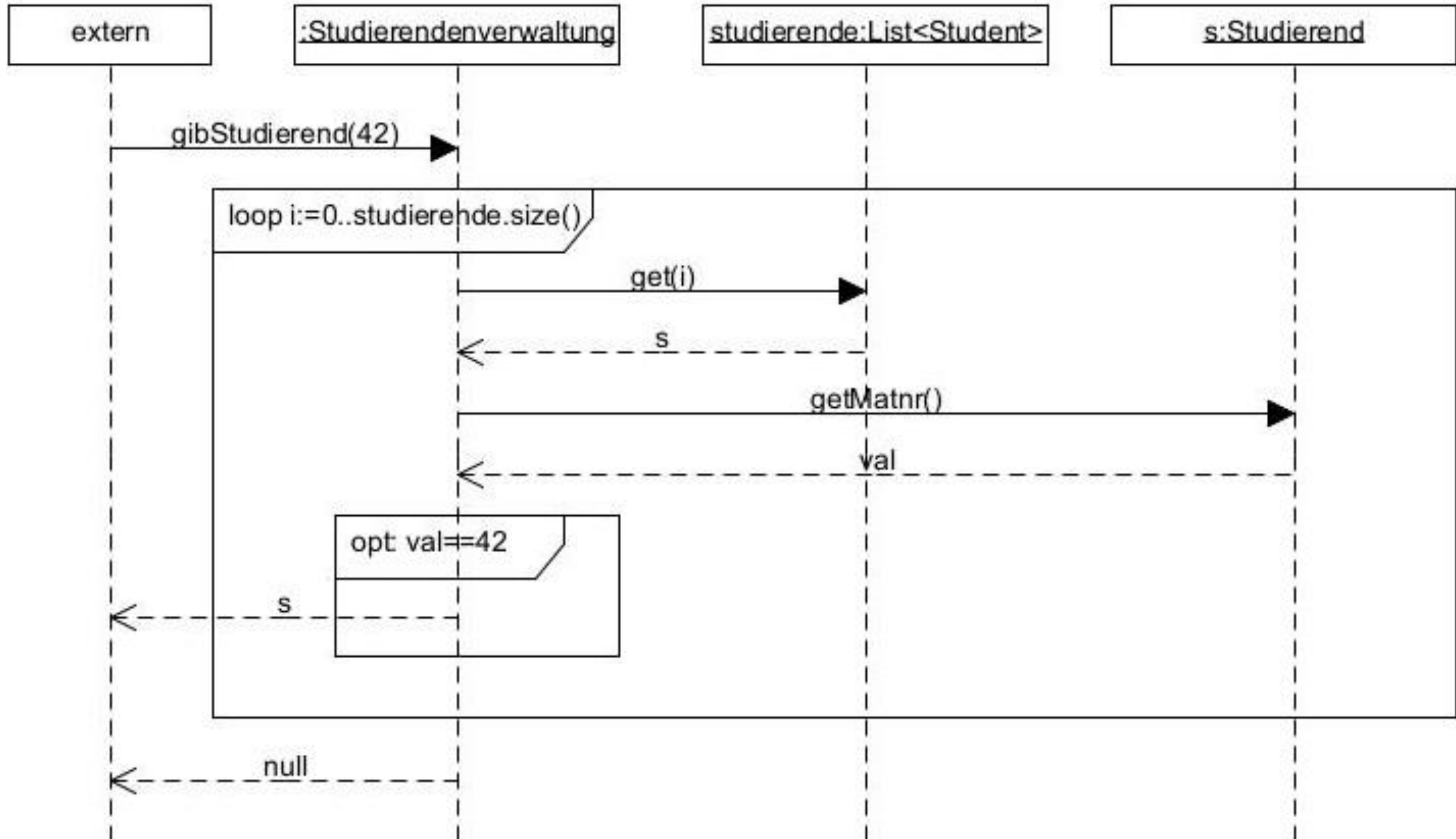
- man kann alle Objekte einzeichnen oder unwichtige weglassen



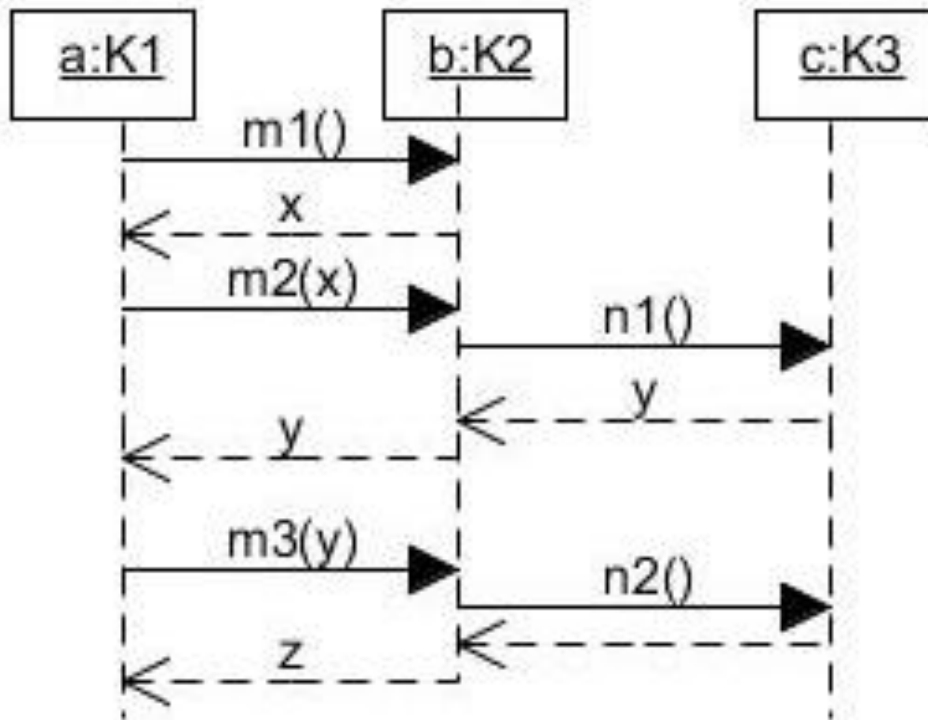


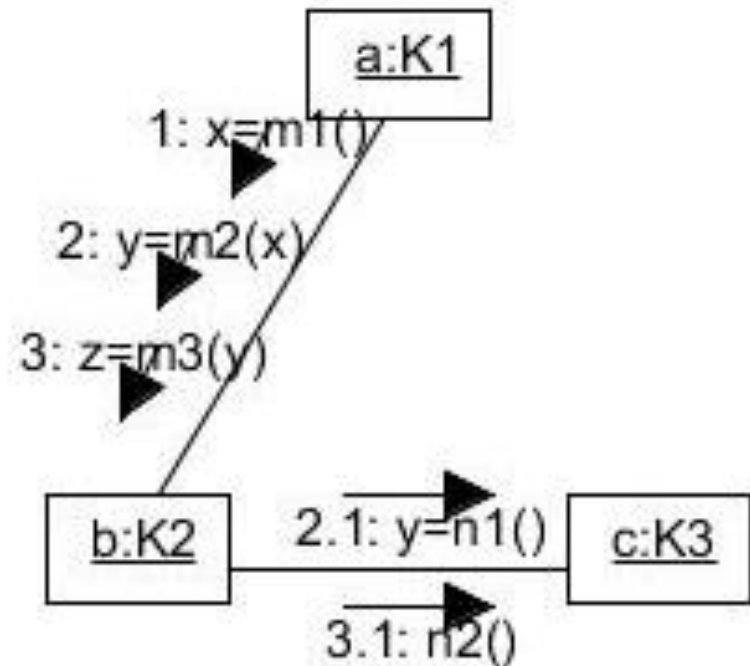
Sequenzdiagramm – Detailgrad (3/3)

- theoretisch: kann man Methoden detailliert zeigen



Sequenzdiagramm und Kommunikationsdiagramm





- gleiches Ausdrucksvermögen wie einfache Sequenzdiagramme
- Zusammenspiel der Objekte wird deutlicher
- interne Berechnung 2.1, 2.2 (ggfls. 2.1.1, 2.1.1.1)

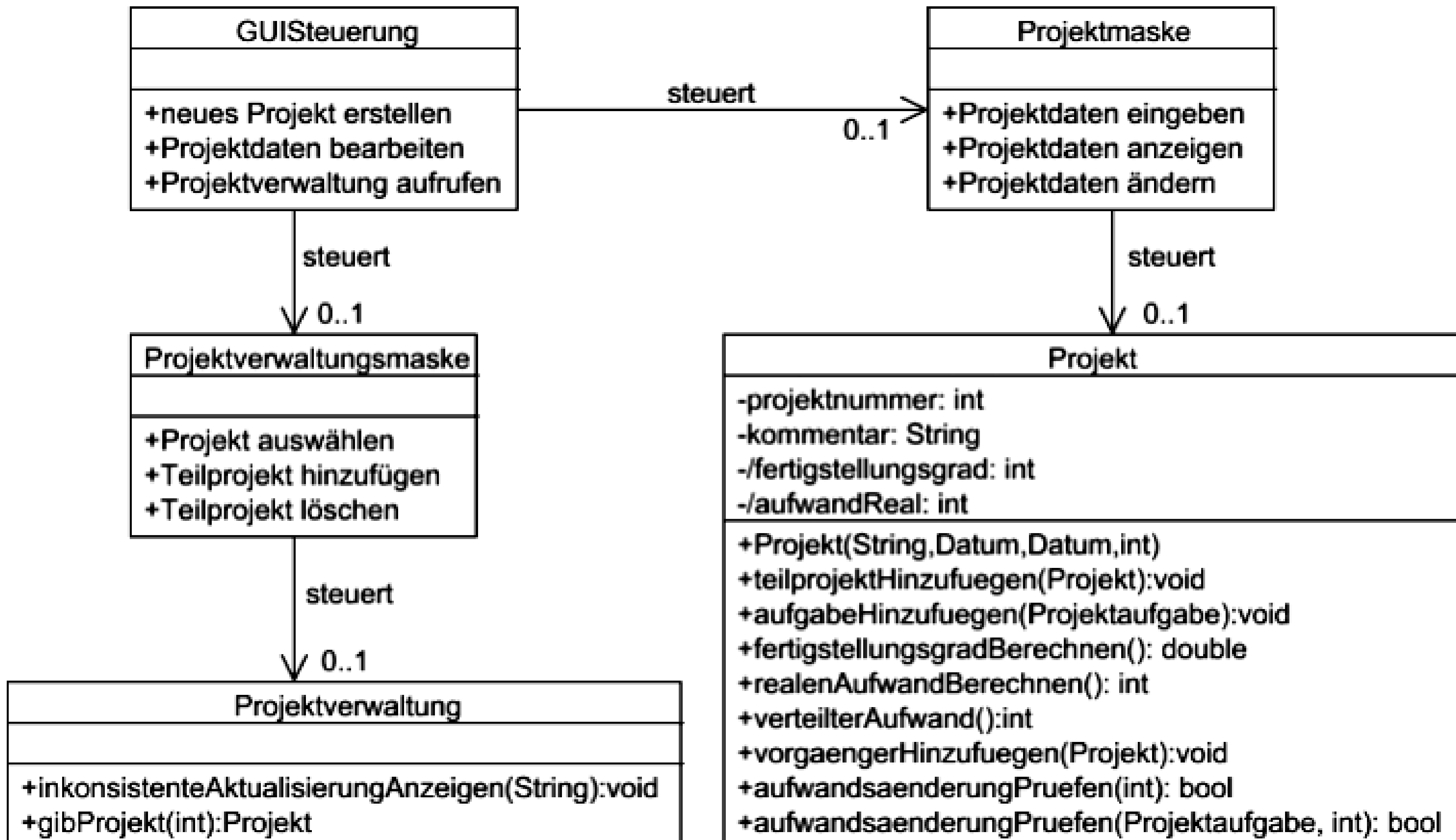
GUI-Modellierung

5.5

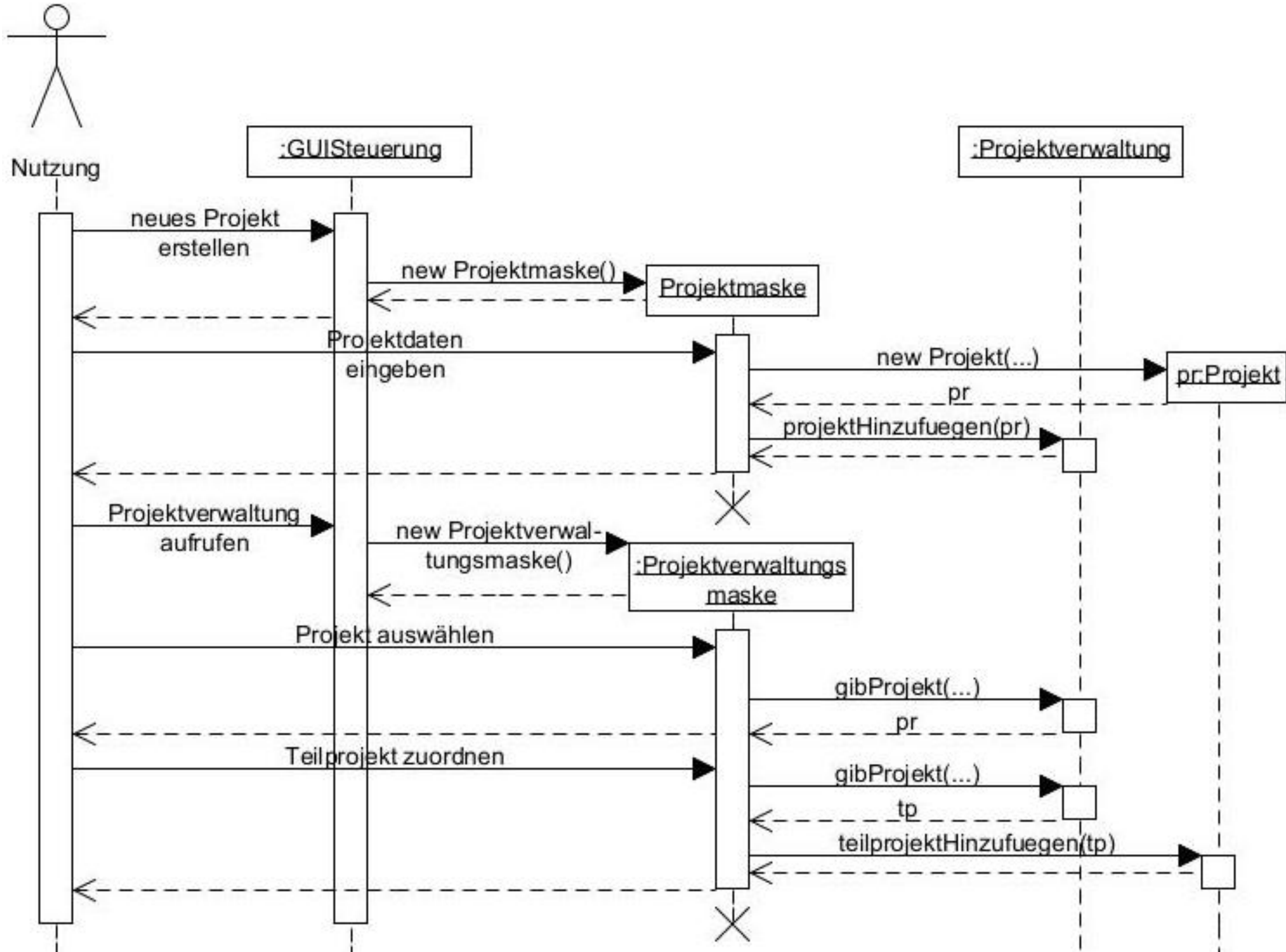
Video

- fachlich hängt Oberfläche (GUI, Graphical User Interface) eng mit unterliegendem Geschäftsklassenmodell (bisher behandelt) zusammen
- möglicher Ansatz: „Mache alle Modellanteile an der Oberfläche sichtbar, die eine nutzende Person ändern oder für dessen Inhalte er sich interessieren kann."
- Variante: mache ersten GUI-Prototyp und halte bei Ein- und Ausgaben fest, welche Modellinformationen sichtbar sein sollen
- GUI-Prototyp gut mit auftraggebenden Personen diskutierbar
- Hinweis: Thema Softwareergonomie

Erweiterung mit Boundary-Klassen



Sequenzdiagramm mit Nutzungsdialog



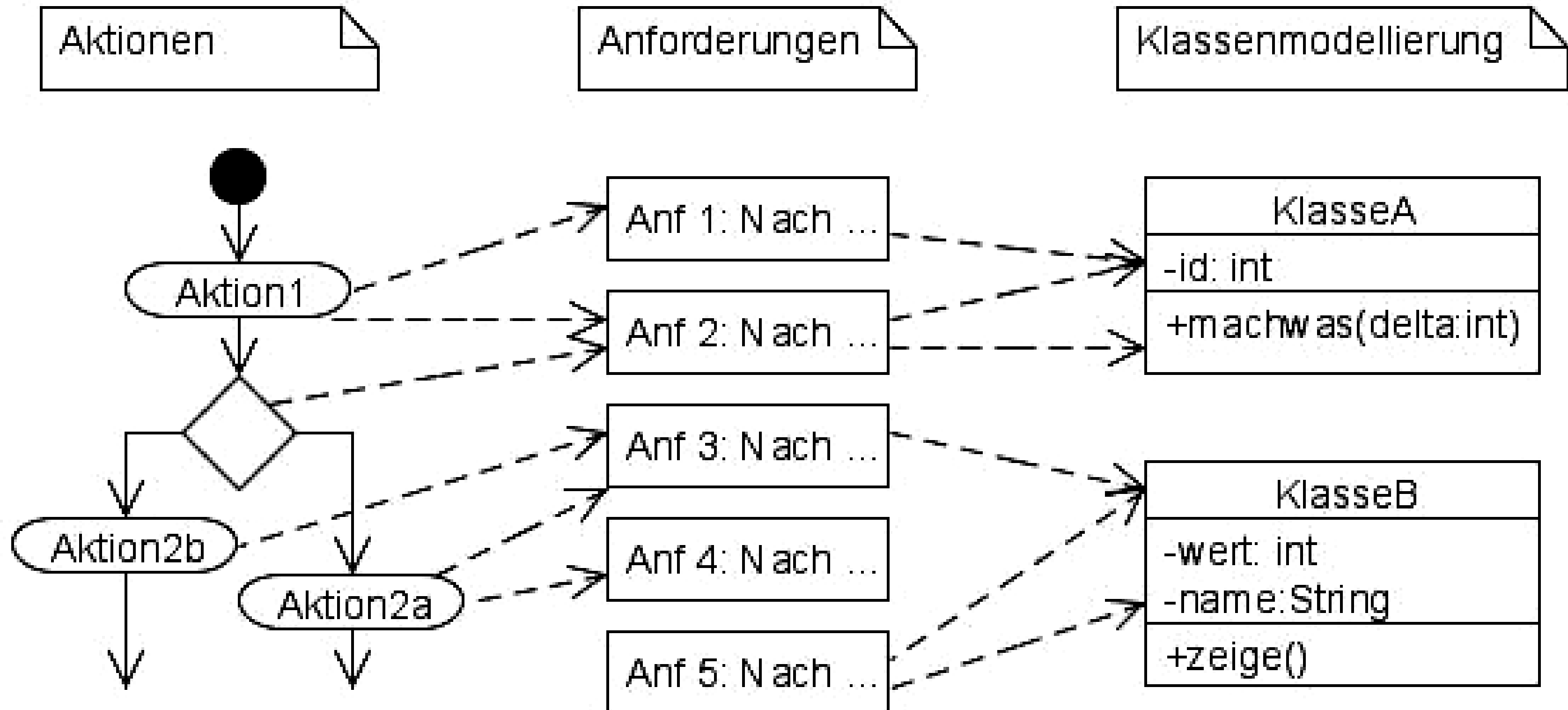
Anforderungsverfolgung

Typische Fragen:

- Werden alle Anforderungen umgesetzt?
- Wo werden Anforderungen umgesetzt?
- Gibt es Spezifikationsanteile, die nicht aus Anforderungen abgeleitet sind?
- Woher kommt eine Klasse, eine Methode, ein Parameter?
- Was passiert, wenn ich eine Anforderung oder eine Klasse ändere?

- Generell werden die Fragen wesentlich komplexer zu beantworten, wenn Software später umgebaut oder erweitert wird

Anforderungsverfolgung - Beispielzusammenhänge



HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

6. Vom Klassendiagramm zum Programm

- 6.1 CASE-Werkzeuge
- 6.2 Übersetzung einzelner Klassen
- 6.3 Übersetzung von Assoziationen
- 6.4 Spezielle Arten der Objektzugehörigkeit
- 6.5 Aufbau einer Software-Architektur
- 6.6 Weitere Schritte zum lauffähigen Programm

Analyse des Ist-Standes

- bekannter Weg: Wünsche des auftraggebenden Unternehmens, Anforderungsformulierung, Analyse-Modell
- Analysemodell kann realisiert werden, aber:
  - Klassen kaum für Wiederverwendung geeignet
  - Programme meist nur aufwändig erweiterbar
  - viele unterschiedliche Lösungen zu gleichartigen Problemen
- deshalb: fortgeschrittene Designtechniken studieren
- aber: um fortgeschrittenes Design zu verstehen, muss man die Umsetzung von Klassendiagrammen in Programme kennen (dieses Kapitel)
- aber: um fortgeschrittenes Design zu verstehen, muss man einige OO-Programme geschrieben haben

UML-Toolsuiten / CASE-Werkzeuge

6.1

Theorie:

- UML-Werkzeuge unterstützen die automatische Umsetzung von Klassendiagrammen in Programmgerüste (Skelette)
- entwickelnde Personen müssen die Gerüste mit Code füllen
- viele Werkzeuge unterstützen Roundtrip-Engineering, d.h. Änderungen im Code werden auch zurück in das Designmodell übernommen (wenn man Randbedingungen beachtet)
- Roundtrip beinhaltet auch Reverse-Engineering

Praxis:

- sehr gute kommerzielle Werkzeuge; allerdings muss man für Effizienz Suite von Werkzeugen nutzen; d. h. auf deren Entwicklungsweg einlassen
- ordentliche nicht kommerzielle Ansätze für Teilgebiete; allerdings Verknüpfung von Werkzeugen wird aufwändig

Übersetzung einfacher Diagramme (1/4)

6.2

Mitarbeitend
- minr:int - mitarbeitendzaehler:int - nachname:String - vorname:String
+ getMinr():int + getMitarbeitendzaehler():int + getNachname():String + getVorname():String + setMinr(in minr:int):void + setMitarbeitendzaehler(in mitarbeitendzaehler:int):void + setNachname(in nachname:String):void + setVorname(in vorname: String):void

Anmerkung: auch bei Realisierung kann vereinbart werden, dass get- und set-Methoden in Übersichten weggelassen (und damit als gegeben angenommen) werden

Klassenmethoden sind unterstrichen

Übersetzung einfacher Diagramme (2/4)

```
public class Mitarbeitend {
   /**
    * @uml.property  name="minr"
    */
   private int minr;
   /**
    * Getter of the property <tt>minr</tt>
    * @return  Returns the minr.
    * @uml.property  name="minr"
    */
   public int getMinr() {
       return minr;
   }
   /**
    * Setter of the property <tt>minr</tt>
    * @param minr  The minr to set.
    * @uml.property  name="minr"
    */
   public void setMinr(int minr) {
       this.minr = minr;
   }
```

Übersetzung einfacher Diagramme (3/4)

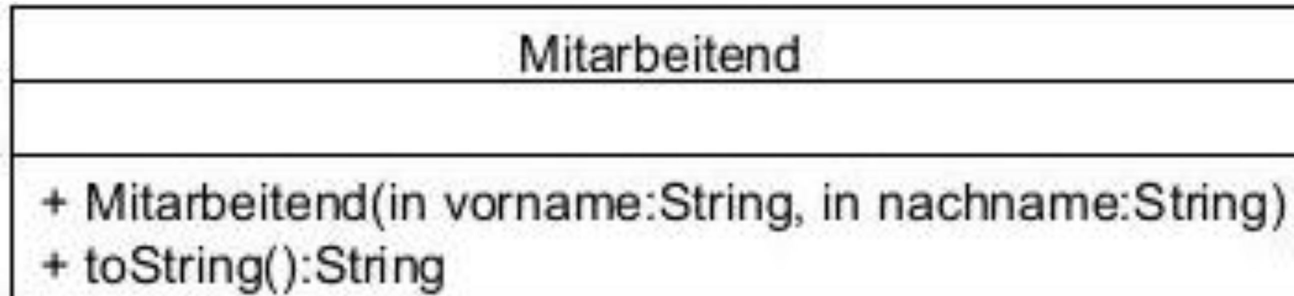
```
private String vorname = "";
public String getVorname() {
   return vorname;
}

public void setVorname(String vorname) {
  this.vorname = vorname;
}

private String nachname = "";
public String getNachname() {
  return nachname;
}

public void setNachname(String nachname) {
    this.nachname = nachname;
}
```

Übersetzung einfacher Diagramme (4/4)

```
    private static int mitarbeitendzaehler;

    public static int getMitarbeitendzaehler() {
        return Mitarbeitend.mitarbeitendzaehler;
    }

    public static void setMitarbeitendzaehler
                                   (int mitarbeitendzaehler) {
        Mitarbeitend.mitarbeitendzaehler
                                  = mitarbeitendzaehler;
    }
}
```

[- - - -] = evtl. notwendige Korrekturen, bei CASE-Werkzeug

Notwendige Code-Ergänzung durch Entwicklung



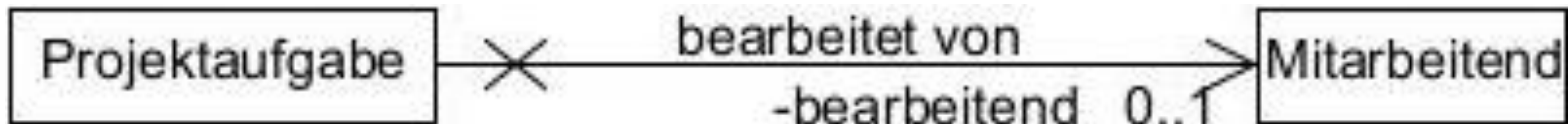

```
public Mitarbeitend(String vorname, String nachname){
   this.vorname = vorname;
   this.nachname = nachname;
   this.minr = Mitarbeitend.mitarbeitendzaehler++;
}
@Override
public String toString() {
  return minr + ": " + this.vorname + " " + this.nachname;
}
```

[gestrichelter Kasten] = von entwickelnden Personen ergänzt

Umgang mit Assoziationen im Design

6.3 Video

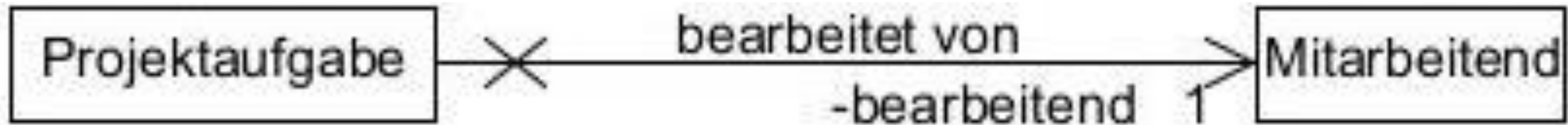
- Assoziationen zunächst nur Strich mit Namen (und Multiplizitäten)
- für Implementierung jede Assoziation konkretisieren (Richtung der Navigierbarkeit, Multiplizitäten sind Pflicht)




```
public class Projektaufgabe {
  /** werkzeugspezifische Kommentare weggelassen
   */
  private Mitarbeitend bearbeitend;
  public Mitarbeitend getBearbeitend() {
    return this.bearbeitend;
  }
  public void setBearbeitend(Mitarbeitend bearbeitend) {
    this.bearbeitend = bearbeitend;
  }
}
```

Multiplizität 1

- Objekreferenz darf nie `null` sein

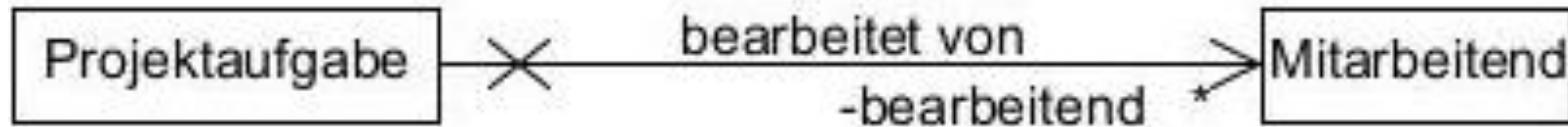



```
private Mitarbeitend bearbeitend = new Mitarbeitend();
```

- oder im Konstruktor setzen
- man sieht, default-Konstruktoren sind auch hier hilfreich; deshalb, wenn irgendwie möglich definieren
- Gleiche Problematik mit der Werte-Existenz, bei Multiplizität 1..n

Multiplizität n (1/2)

- Umsetzung als Collection (Sammlung, Container)



- Umsetzung hängt von Art der Collection ab
  - sollen Daten geordnet sein
  - sind doppelte erlaubt
  - gibt es spezielle Zuordnung key -> value
- entwickelnde Person muss zur Typwahl spätere Nutzung kennen
- eine Umsetzung für 1..*

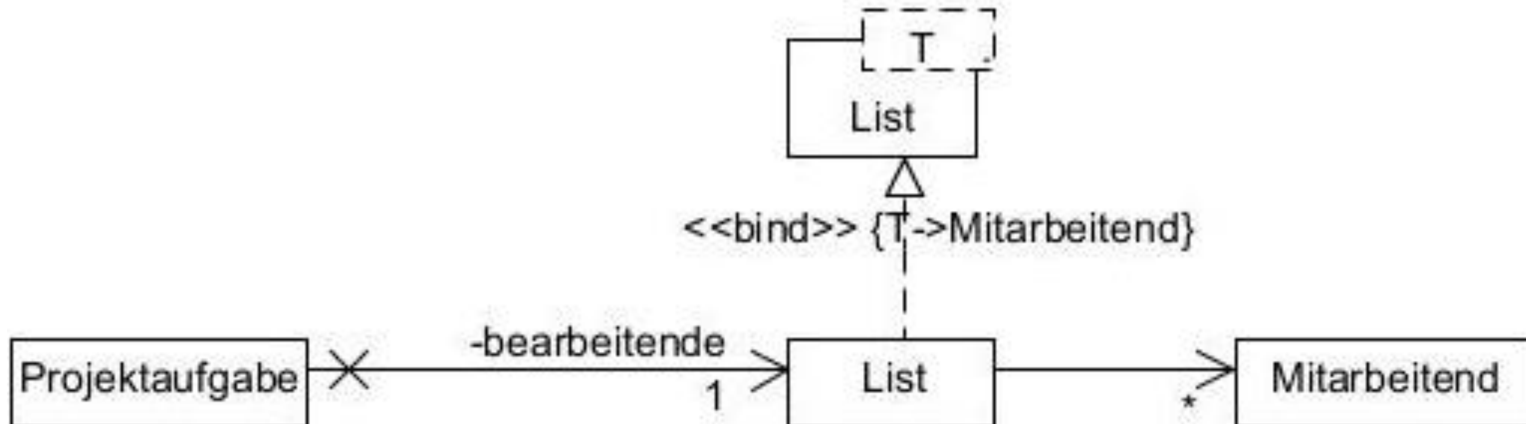
```
import java.util.List;
import java.util.ArrayList;
public class Projektaufgabe {
private List<Mitarbeitend> bearbeitend = new ArrayList<>();
```

- bitte, bitte in Java nicht alles mit ArrayList realisieren (!!!)
- Multiplizität 0..7 als Array umsetzbar

Multiplizität n (2/2)

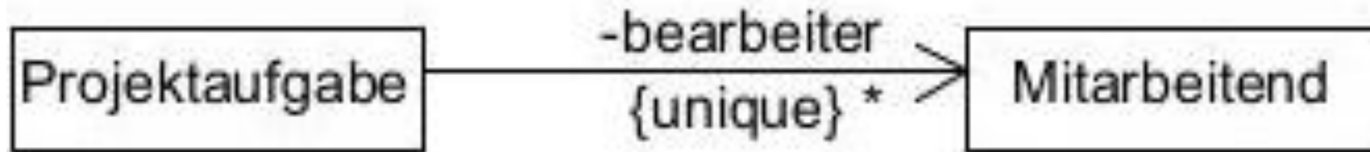
HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- Zum Codefragment der letzten Zeile passt besser folgendes Klassendiagramm



- Hinweis: Standardhilfsklassen z. B. aus der Java-Klassenbibliothek oder der C++-STL werden typischerweise in Klassendiagrammen nicht aufgeführt
- Anmerkung: man sieht die UML-Notation für generische (oder parametrisierte) Klassen
- UML-Werkzeuge unterscheiden sich bei der Generierung und beim Reverse-Engineering beim Umgang mit Collections

Collections in UML

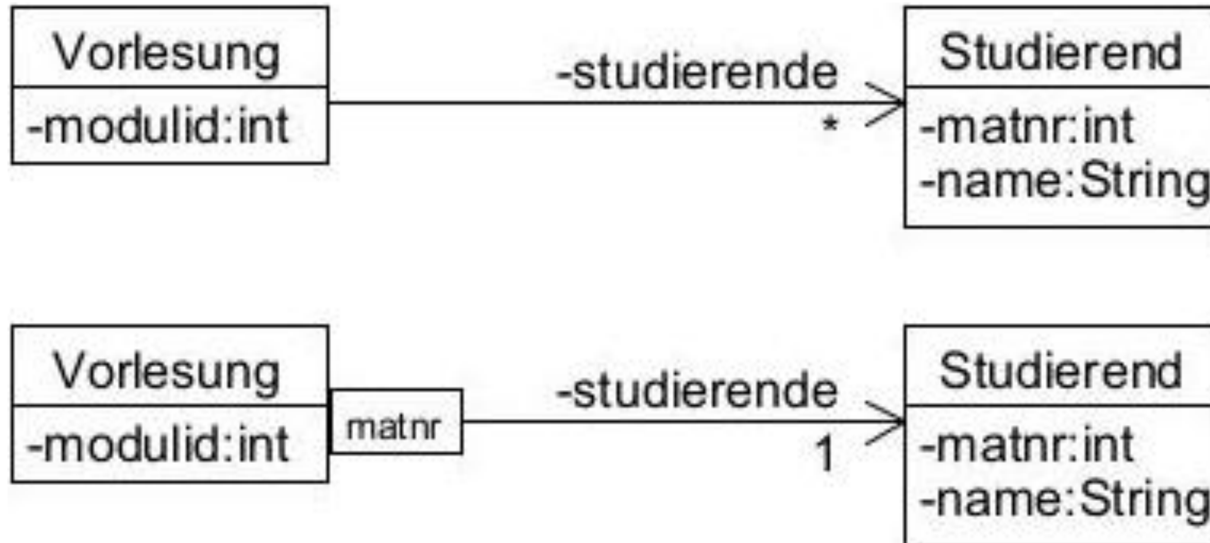


- Constraints (Randbedingungen) stehen in geschweiften Klammern (weitere Möglichkeiten -> Object Constraint Language, OCL)
- unique: eindeutig, nur einmal
- ordered: geordnet, sortiert oder Reihenfolge beibehaltend
- unique: Set
- ordered: List
- notunique, unordered: MultiSet
- Default ohne Angabe ist: {unique, unordered}

Collections in der Programmierung

- Jede OO-Programmiersprache hat große Sammlung an Umsetzungen von Collections
- UML lässt meist trotz Constraints verschiedene Umsetzungen zu
- Java: Beispielumsetzungen für Set
  - HashSet: generell recht schnelles Einfügen und Löschen
  - TreeSet: garantiert log(n) für Basisfunktionalität, nutzt Ordnung der Elemente (Interface Comparable<>)
  - LinkedSet: behält beim Iterieren die Reihenfolge der Eintragungen ein (ordered)
  - org.apache.commons.collections.list.SetUniqueList: Liste mit eindeutigen (unique) Einträgen
  - …

Qualifizierte Assoziationen



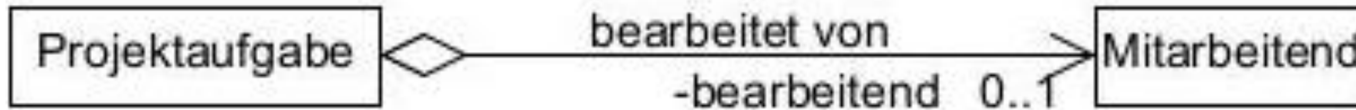
- qualifizierendes Attribut als Teil der Assoziation angegeben
- steht typischerweise für Map (Dictionary)

```
private Map<Integer,Studierend> studierende
```

- zu jeder der Vorlesung bekannten Matrikelnummer gehört genau ein Studierend-Objekt
- andere Multiplizitäten (0..1, *) möglich

Arten der Zugehörigkeit (Aggregation 1/2)

- nicht exklusiver Teil eines Objekts (Mitarbeitend-Objekt kann bei mehreren Projektaufgaben bearbeitende Person sein)



in C++: `Mitarbeitend *bearbeitend;`

```
Mitarbeitend* Projektaufgabe::getBearbeitend(){
    return bearbeitend;}
```

oder `Mitarbeitend bearbeitend;`

```
Mitarbeitend& Projektaufgabe::getBearbeitend(){
    return bearbeitend;}
```

Realisierungsproblem: Projektaufgabe kann Namen der bearbeitenden Person ändern

```
bearbeitend->setNachname("Meier");
```

- Kann als Verstoß gegen Kapselung (Geheimnisprinzip) angesehen werden
- Designansatz: Klasse erhält Interface, die Methoden enthält, die bestimmte andere Klassen nutzen können

Arten der Zugehörigkeit (Aggregation 2/2)

- Designansatz: Verhindern unerwünschten Zugriffs durch Interface (generell gute Idee !)

Projektaufgabe

0..1
-bearbeitend

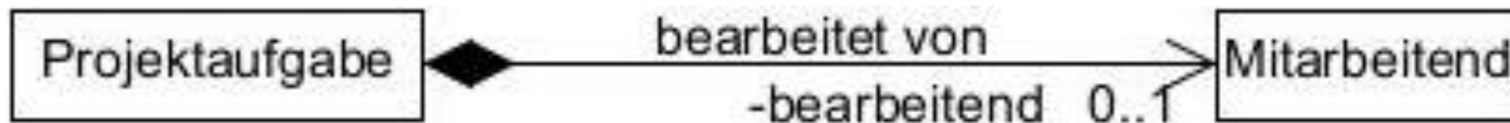
<<interface>>
Aufgabenbearbeitend
+ getMinr():int
+ getNachname():String
+ getVorname():String

Mitarbeitend
- minr:int
- mitarbeitendzaehler:int
- nachname:String
- vorname String
+ Mitarbeitend()
+ Mitarbeitend (in vorname:String, in nachname:String)
+ getMinr():int
+ getMitarbeitendzaehler():int
+ getNachname():String
+ getVorname():String
+ setMinr(in minr:int):void
+ setMitarbeitendzaehler(in mitarbeitendzaehler:int):void
+ setNachname(in nachname:String):void
+ setVorname(in vorname: String):void
+ toString():String

Kurzdarstellung Interfacerealisierer:

Arten der Zugehörigkeit (Komposition 1/2)

- Konkretisierung der Zugehörigkeit: existenzabhängiges Teil, Exemplarvariable gehört ausschließlich zum Objekt (Mitarbeitend-Objekt kann [unrealistisch] nur exakt eine Projektaufgabe bearbeiten; niemand anderes nutzt dieses Objekt)



- Realisierung in C++

```
Mitarbeitend bearbeitend;
Mitarbeitend Projektaufgabe::getBearbeitend (){
  return bearbeitend;
}
```

- Bei Rückgabe wird Kopie des Objekts erstellt und zurückgegeben

Arten der Zugehörigkeit (Komposition 2/2)

- Java arbeitet nur mit Referenzen (unschöne Ausnahme sind Elementartypen), wie realisiert man

```
@Override  // in Mitarbeitend.java
public Mitarbeitend clone(){ // echte Kopie
    Mitarbeitend ergebnis = new Mitarbeitend();
    ergebnis.minr = minr;
    ergebnis.nachname = nachname; //Strings sind
    ergebnis.vorname = vorname;   //Konstanten
    return ergebnis;
}
// in Projektaufgabe
public Mitarbeitend getBearbeitend() {
    return this.bearbeitend.clone();
}
```

- Variante: bei Realisierung überall doll aufpassen

Kurzzeitige Klassennutzungen

- Klassen nutzen andere Klassen nicht nur für Exemplar- (und Klassen-) Variablen
- Klassen erzeugen Objekte anderer Klassen lokal in Methoden, um diese weiter zu reichen

```
public class Projektverwaltung {
   private Projekt selektiertesProjekt;
   public void projektaufgabeErgaenzen(String name){
       Projektaufgabe pa = new Projektaufgabe(name);
       selektiertesProjekt.aufgabeHinzufuegen(pa);
   }
```

- Klassen nutzen Klassenmethoden anderer Klassen
- In der UML gibt es hierfür den „Nutzt“-Pfeil

- Wenn zu viele Pfeile im Diagramm, dann mehrere Diagramme mit gleichen Klassen zu unterschiedlichen Themen
- UML-Werkzeuge unterstützen Analyse von Abhängigkeiten

Erstellen einer Softwarearchitektur

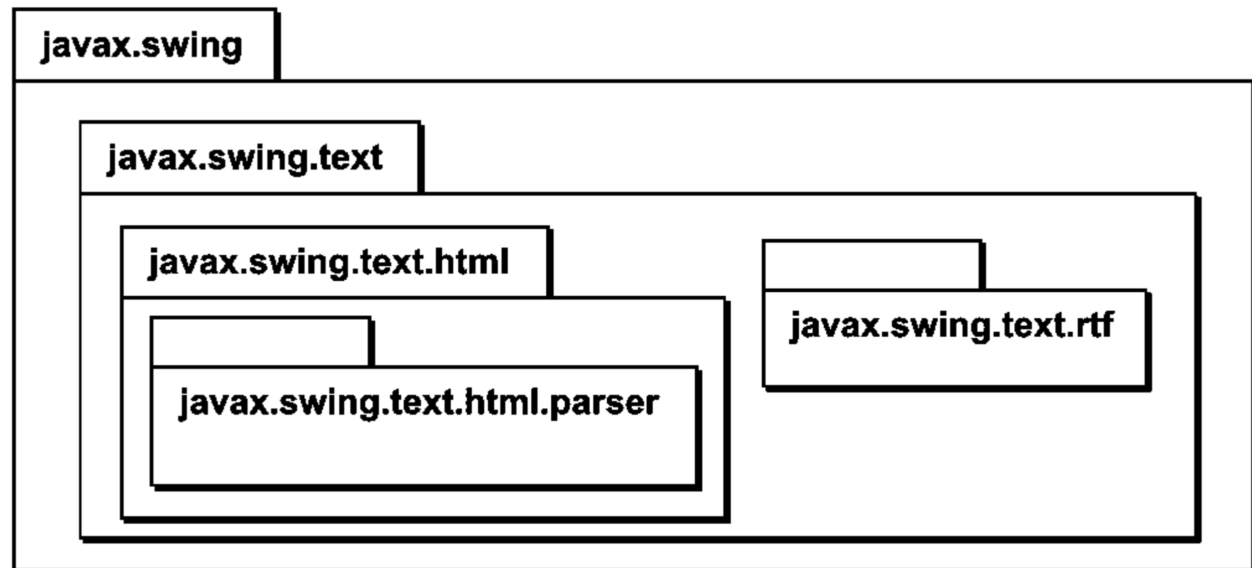
6.5

- Ziel des Design ist ein Modell, welches das Analysemodell vollständig unter Berücksichtigung implementierungsspezifischer Randbedingungen umsetzt
- allgemeine Randbedingungen: Es gibt ingenieurmäßige Erfahrungen zum gutem Aufbau eines Klassensystems; dieses wird auch SW-Architektur genannt
- Ziele für die Architektur
  - Performance
  - Wartbarkeit
  - Erweiterbarkeit
  - Verständlichkeit
  - schnell realisierbar
  - Minimierung von Risiken

Systematische Entwicklung komplexer Systeme

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

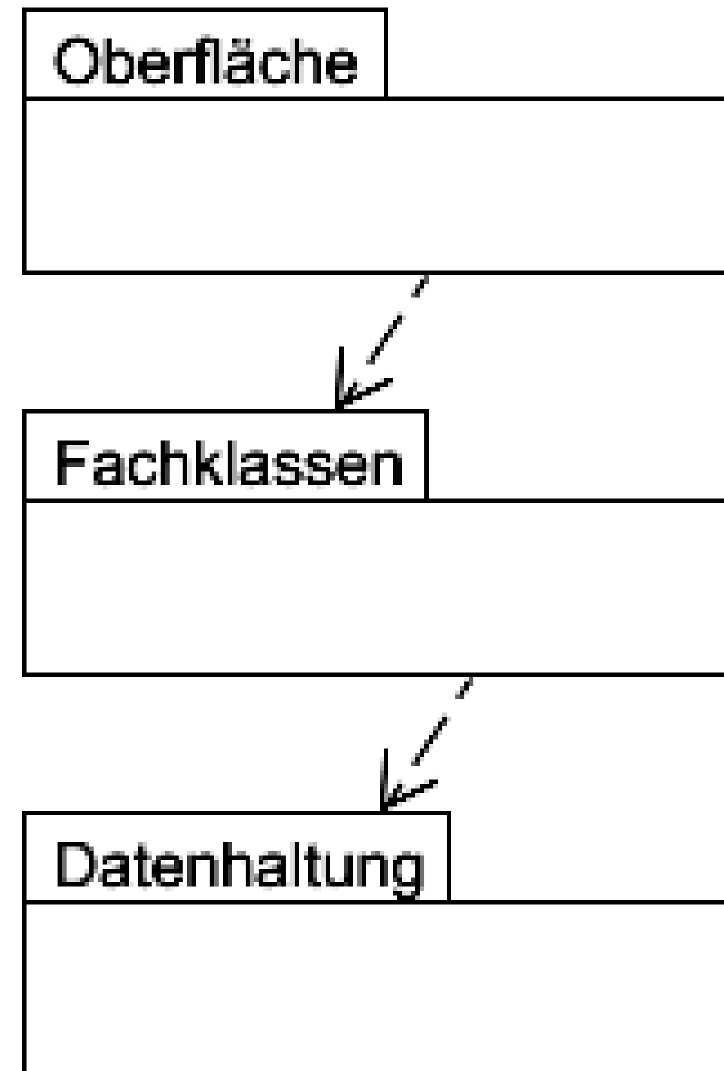
- Für große Systeme entstehen viele Klassen; bei guten Entwurf:
- Klassen die eng zusammenhängen (gemeinsame Aufgabengebiete)
- Klassen, die nicht oder nur schwach zusammenhängen (Verknüpfung von Aufgabengebieten)
- Strukturierung durch SW-Pakete; Pakete können wieder Pakete enthalten



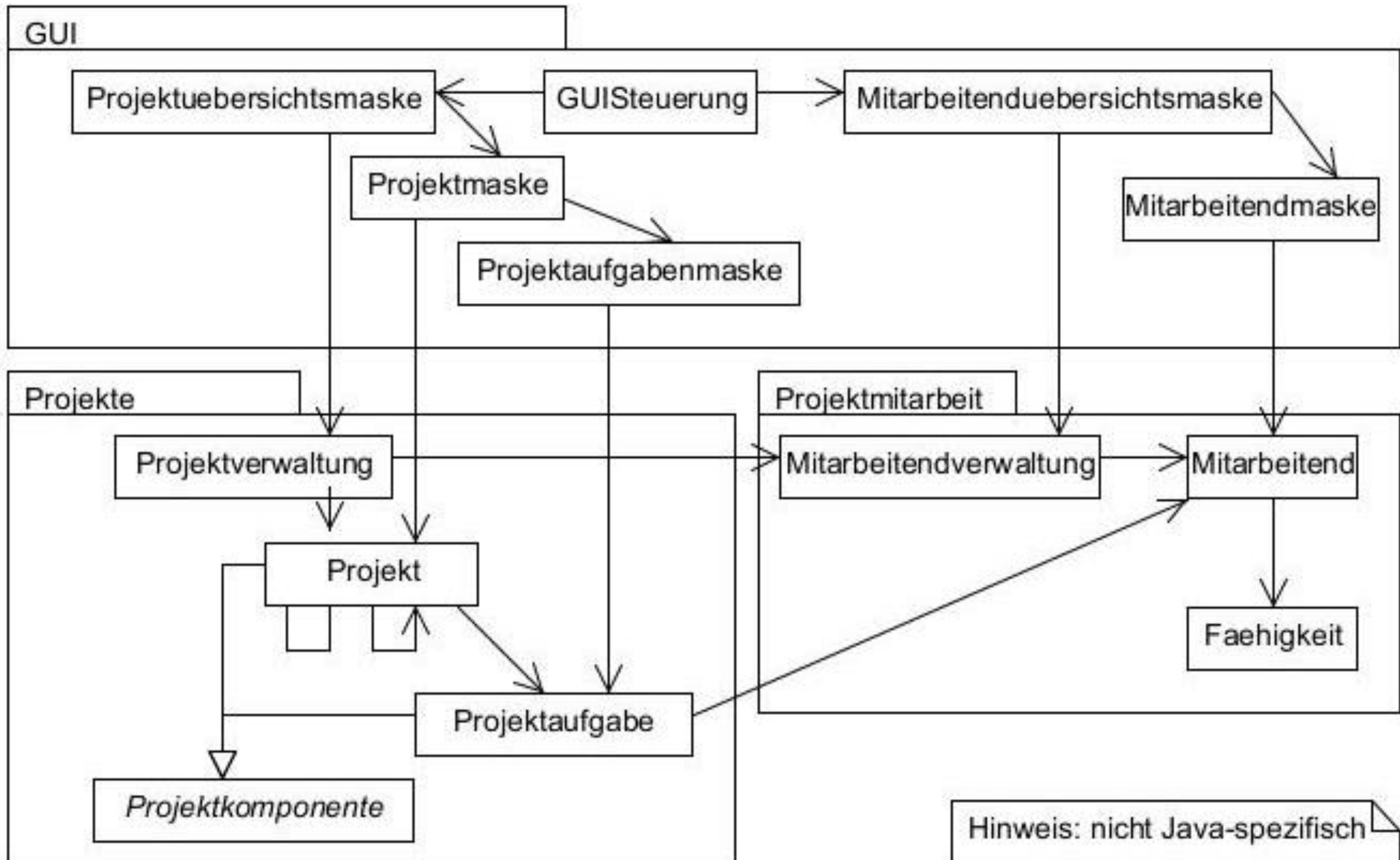
Typische 3-Schichten-SW-Architektur

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- Ziel: Klassen eines oberen Pakets greifen nur auf Klassen eines unteren Paketes zu (UML-“nutzt“-Pfeil)
- Änderungen der oberen Schicht beeinflussen untere Schichten nicht
- Analysemodell liefert typischerweise nur Fachklassen
- Datenhaltung steht für Persistenz
- typisch Varianten von 2 bis 5 Schichten
- Klassen in Schicht sollten gleichen Abstraktionsgrad haben
- Schicht in englisch „tier“
- obere und untere Schichten können stark von speziellen Anforderungen abhängen (später)

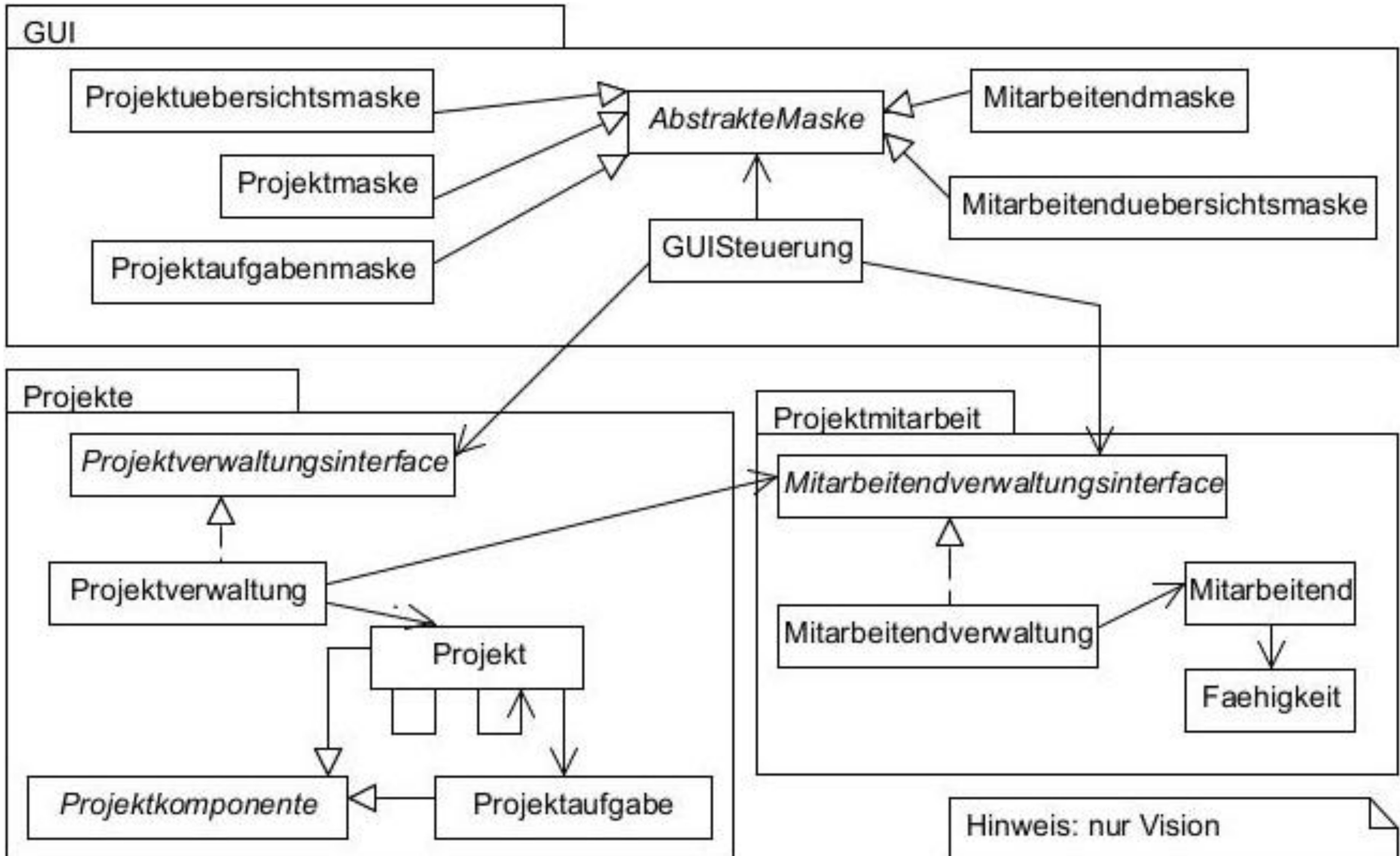


Beispiel: grobe Paketierung (eine Variante)

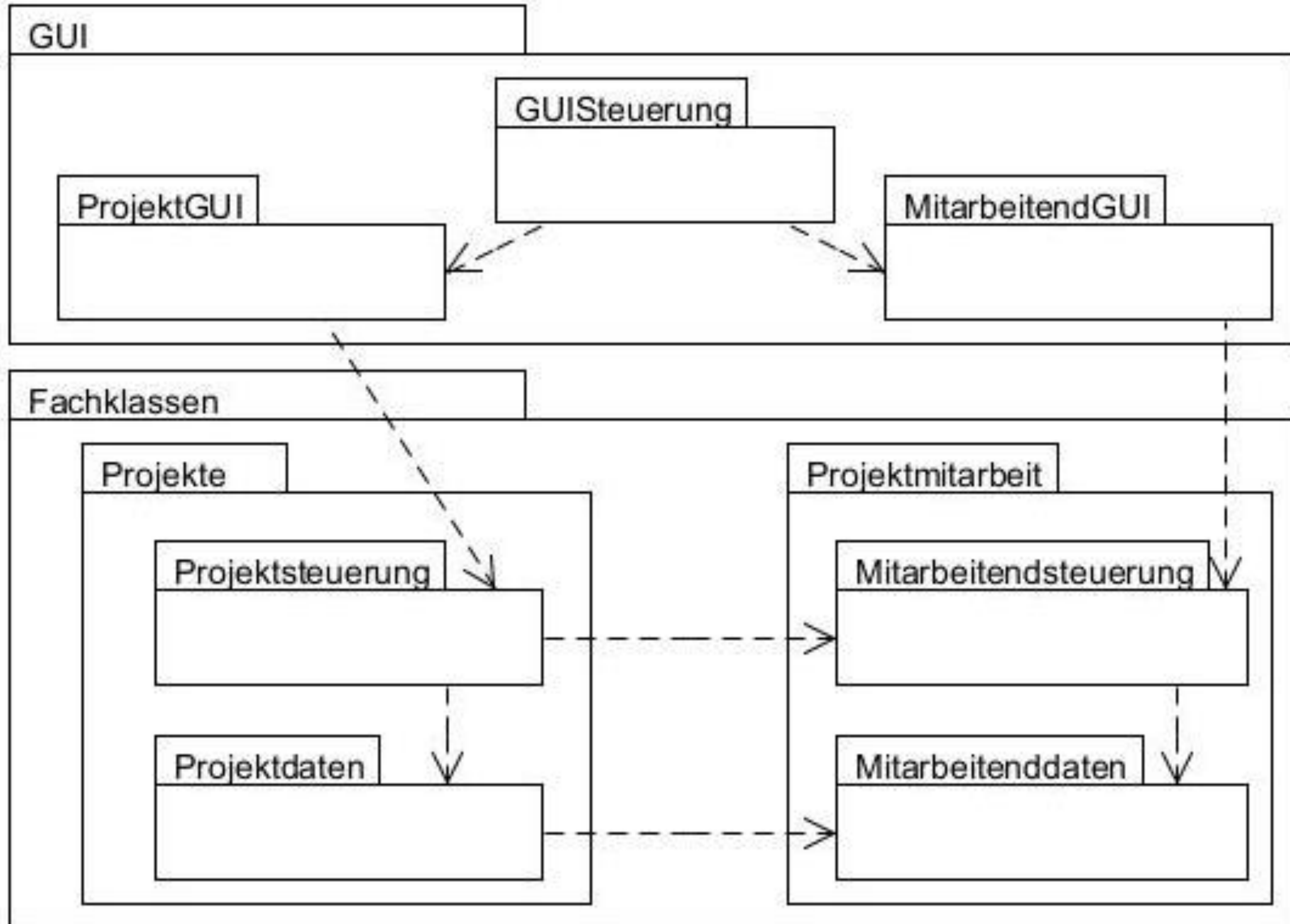


- Anmerkung: Datenverwaltung noch nicht konzipiert

Beispiel: grobe Paketierung (zweite Variante)



Forderung: azyklische Abhängigkeitsstruktur



Umsetzung von Paketen in Java und C++

```
package fachklassen.projektdaten;
import fachklassen.projektmitarbeit.Mitarbeitend;
public class Projektaufgabe {
   private Mitarbeitend bearbeitend;
   /* ... */
}
```

```
#include "Mitarbeitend.h" //evtl. mit Dateibaum
using namespace Fachklassen::Projektmitarbeit;
namespace Fachklassen{
   namespace Projektdaten{
       class Projektaufgabe{
             private:
                 Mitarbeitend *bearbeitend; // ...
       };
   }
}
```

Paketnamen und Strukturierungsmöglichkeiten

- gibt in Programmiersprachen Regeln für Paketnamen
- Beispiel: Firma mit Webseite meineFirma.de
- Paketnamen beginnen immer mit de.meineFirma

- Pakete orientieren sich an Architekturstilen
- Beispiel: Boundary – Control – Entity
- man kann Pakete z. B. auch nach Use Cases ordnen
- Interfaces können in anderen Paketen getrennt von Implementierung stehen

- ein oder mehrere Pakete werden in Java als jar-Datei ausgeliefert

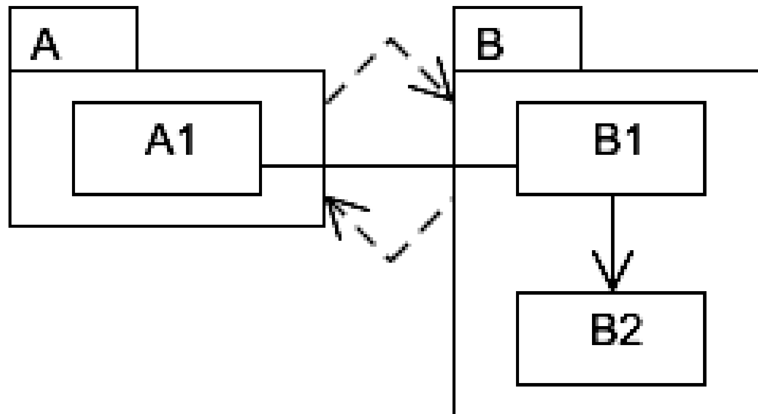
Paketabhängigkeiten optimieren

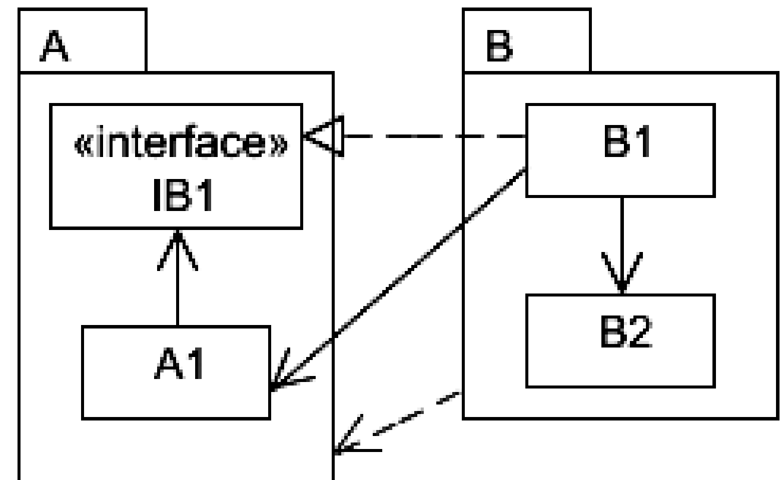
HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- Ziel ist es, dass Klassen sehr eng zusammenhängen; es weniger Klassen gibt, die eng zusammenhängen und viele Klassen und Pakete, die nur lose gekoppelt sind
- Möglichst bidirektionale oder zyklische Abhängigkeiten vermeiden
- Bei Paketen können Zyklen teilweise durch die Verschiebung von Klassen aufgelöst werden
- Wenig globale Pakete (sinnvoll für projektspezifische Typen (z. B. Enumerations), Konstanten, Utility-Klassen und Ausnahmen)
- Es gibt viele Designansätze, Abhängigkeiten zu verringern bzw. ihre Richtung zu ändern

Trick: Abhängigkeit umdrehen

Video





- generell können Interfaces häufiger in anderen Paketen liegen, als ihre implementieren Klassen
- Java-Klassenbibliothek Swing fordert häufig die Interface-Implementierung bei der Ereignisbehandlung

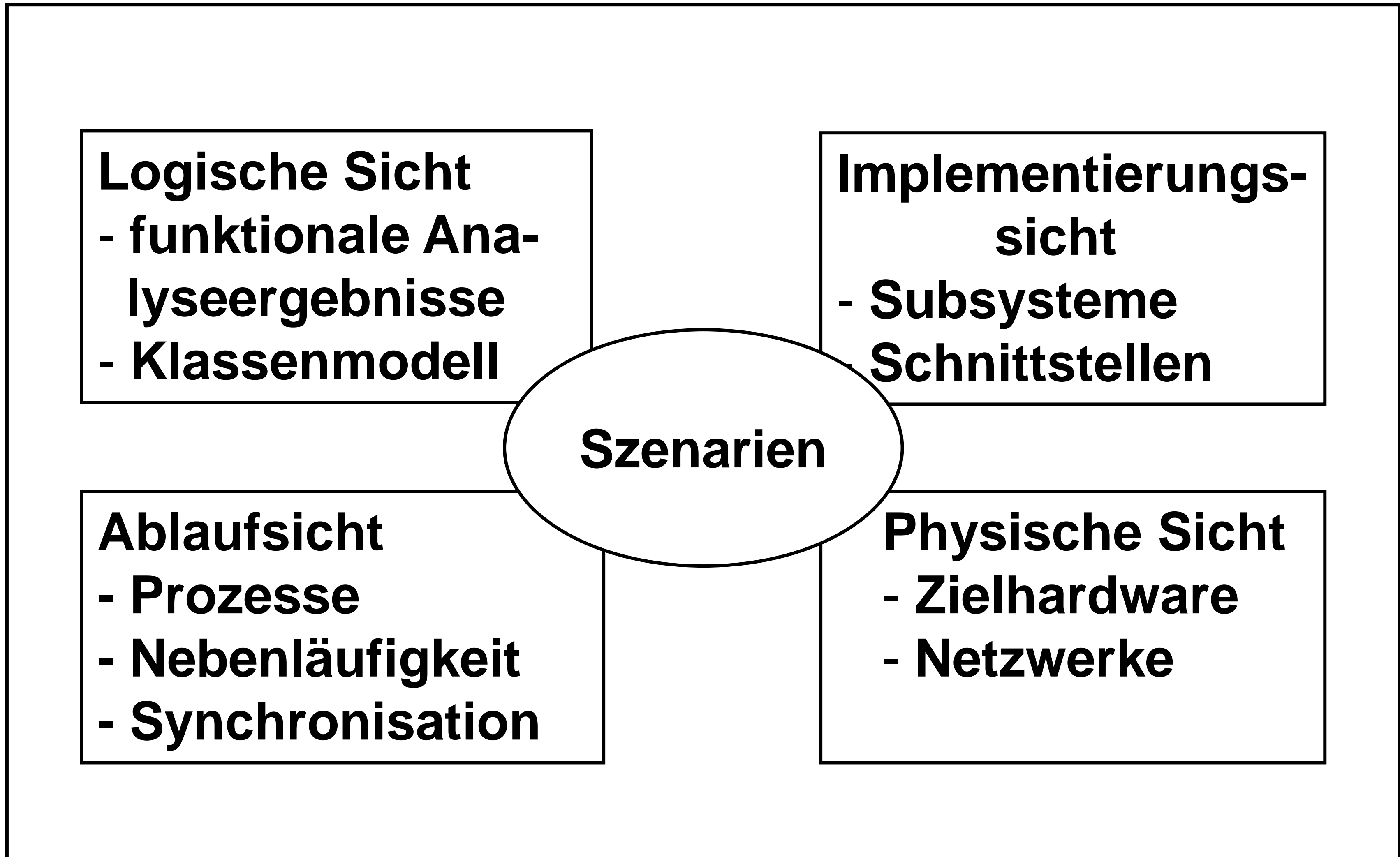
Architektursichten

6.6

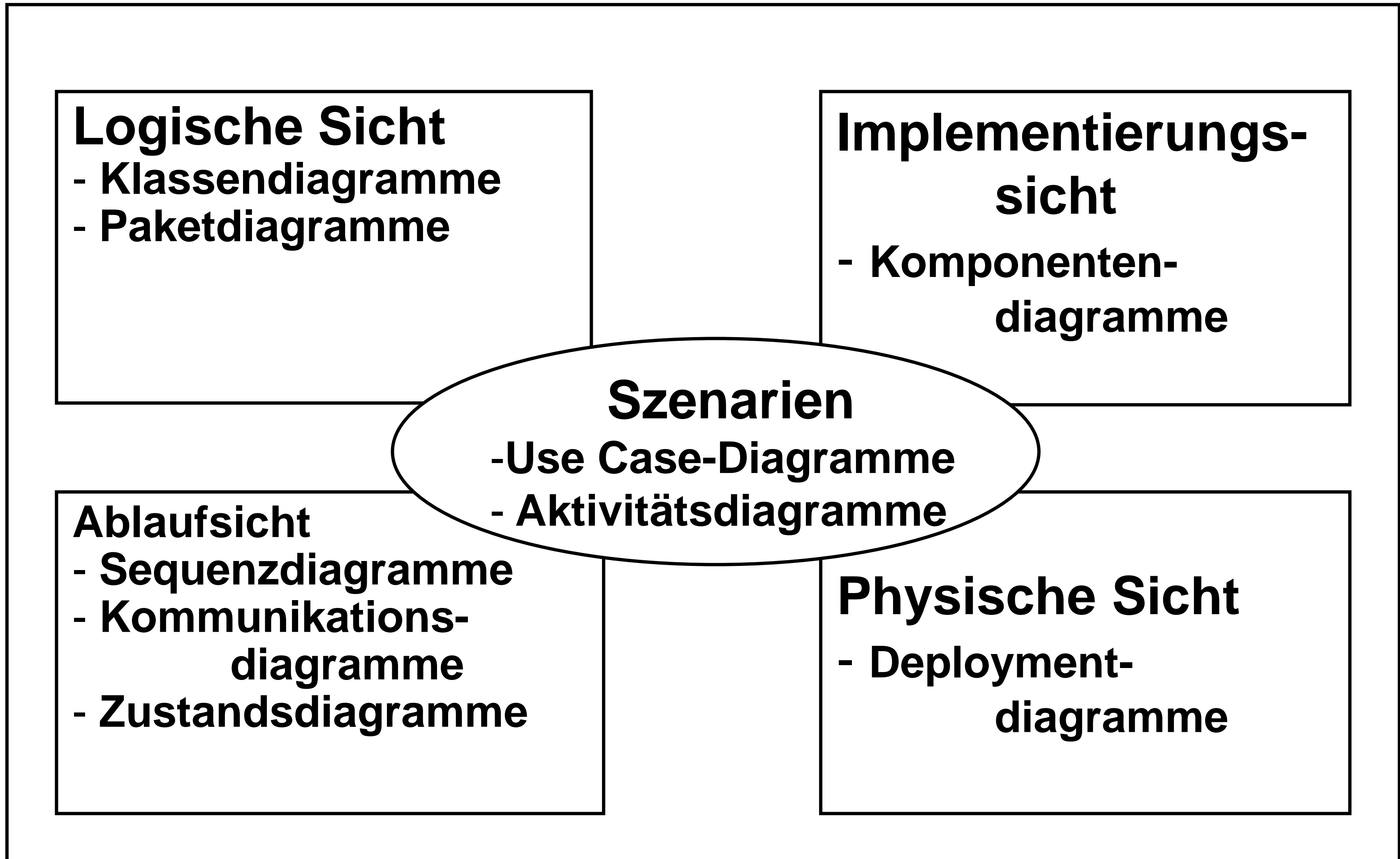
- Paket- und Klassendiagramme geben einen guten Überblick über die Ergebnisse des statischen SW-Entwurfs
- Es gibt aber mehr Sichten (Betrachtungsweisen), die für eine vollständige SW-Architektur relevant sind
- z. B. wurde die HW des zu entwickelnden Systems noch nicht berücksichtigt
- Diese Sichten müssen zu einem System führen; deshalb müssen sich Sichten überlappen
- z. B. eigenes Diagramm mit Übersicht über eingesetzte Hardware und Vernetzung; dazu Information, welche SW wo laufen soll

4+1 Sichten

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES



4+1 Sichten mit (Teilen der) UML



Ablaufsicht

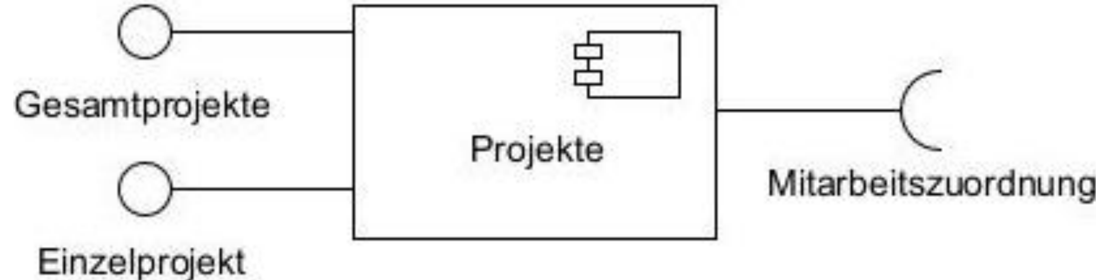
- wichtig für verteilte Systeme; bzw. Systeme, die verteilt (auch auf einem Rechner) laufen
- Festlegen der Prozesse
- Festlegen etwaiger Threads
- so genannte aktive Klassen; werden Objekte dieser Klassen gestartet, so starten sie als eigenständige Prozesse/Threads

- Prozessverhalten u. a. durch Sequenzdiagramme beschreibbar
- (später etwas mehr; gibt eigenes Modul dazu)

Implementierungssicht

- Das Designmodell muss physikalisch realisiert werden; es muss eine (ausführbare) Datei entstehen
- Pakete fassen als Komponenten realisiert Klassen zusammen
- häufig werden weitere Dateien benötigt: Bilder, Skripte zur Anbindung weiterer Software, Installationsdateien
- Komponenten sind austauschbare Bausteine eines Systems mit Schnittstellen für andere Komponenten
- Typisch ist, dass eine Komponente die Übersetzung einer Datei ist
- Typisch ist, dass eine Komponente die Übersetzung eines Pakets ist; in Java .jar-Datei

Komponentendiagramm



<<component>>
Projekte

<<provided Interfaces>>
Gesamtprojekte
Einzelpojekt
<<required interfaces>>
Mitarbeitszuordnung

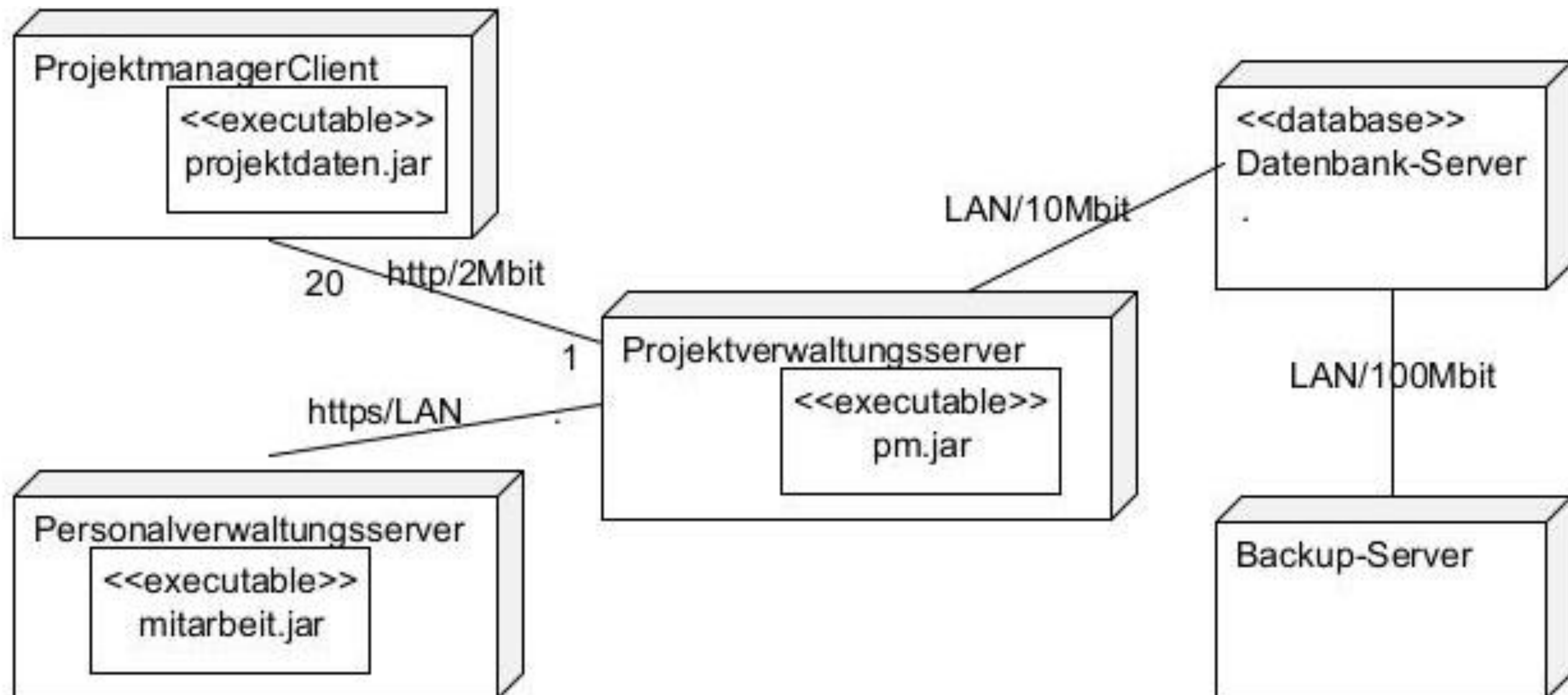
<<realization>>
Projektverwaltung
Projekt
Projektaufgabe
Projektkomponente
Einzelprojekt
Gesamtprojekte

<<artifacts>>
pm.jar
gruenesProjekt.gif
gelbesProjekt.gif
rotesProjekt.gif

- Bilder zeigen zwei alternative Darstellungen
- Komponenten bieten Schnittstellen(realisierungen) (Kreis) und benötigen Schnittstellen(realisierungen) (Halbkreis)
- Komponenten können über Schnittstellen in Diagrammen verknüpft werden
- in die Komponenten können die zugehörigen Klassen eingezeichnet werden
- Ports erlauben den Zugriff auf bestimmten Teil der Klassen und Interfaces (nicht im Diagramm)

Physische Sicht: vorgegebene HW mit Vernetzung

- Einsatz- und Verteilungsdiagramm (deployment diagram)
- Knoten steht für physisch vorhandene Einheit, die über Rechenleistung oder/und Speicher verfügt
- <<executable>> (ausführbare Datei) und <<artifact>> (Datei) müssen zur HW-Beschreibung nicht angegeben werden



Java Module (1/7)

Video

- vor Java 9: alle genutzten Bibliotheken (.jar, .zip) im Classpath eingebunden
- Klassen und Pakete können doppelt sein oder sich überlappen, Auswahl abhängig von Reihenfolgen im Classpath
- neuer Ansatz: Gruppen von Paketen in Modulen vereinigen

Vorteile:

- keine zirkulären Abhängigkeiten erlaubt, bessere Struktur
- ein Paket kann nur von einem Modul im Module-Path angeboten werden (sonst Fehler)
- aus Modulen kann eigene Java-Applikation gebaut werden, die nur notwenige Module der JRE enthält (statt immer vollständige JRE (oder JDK) auszuliefern)

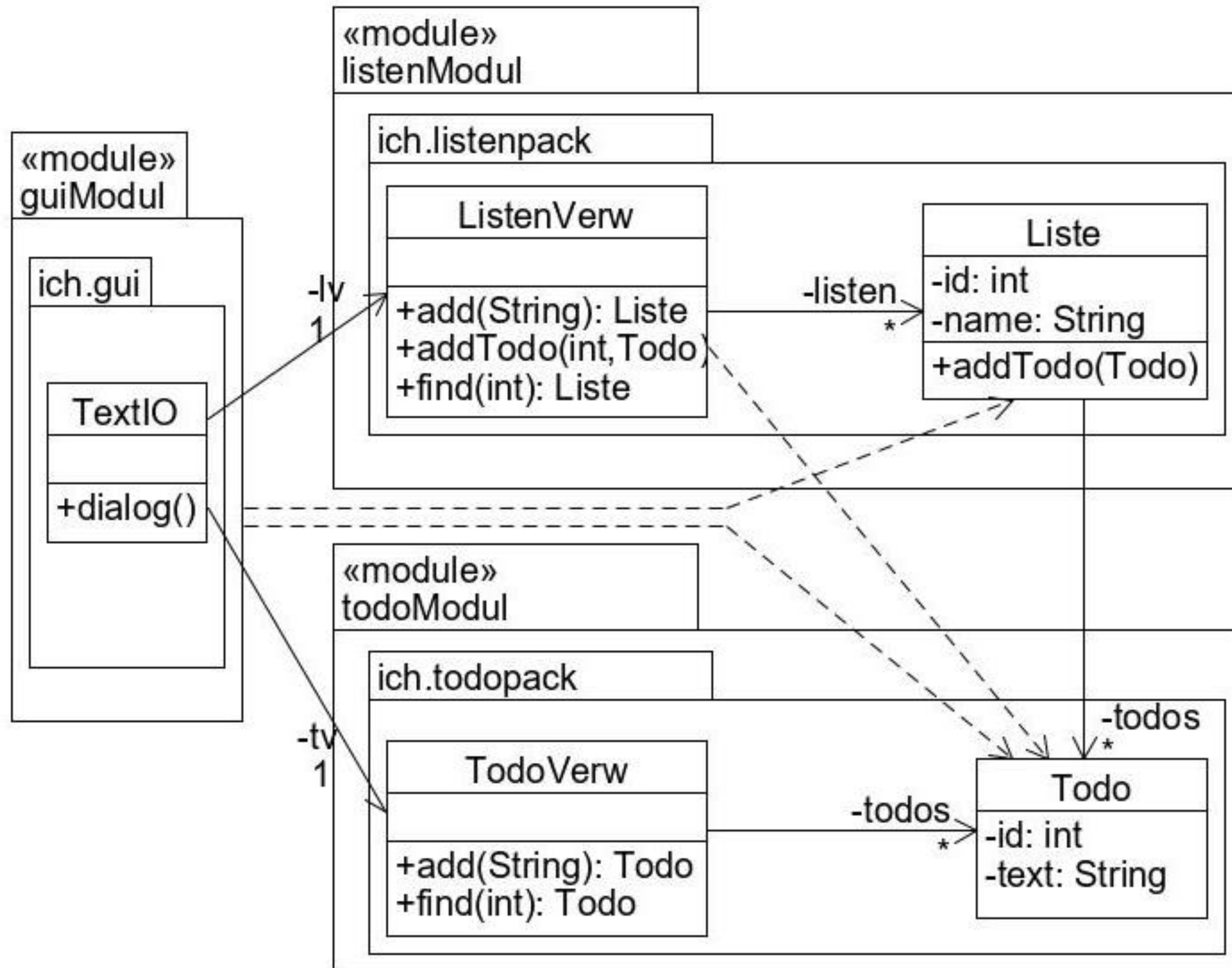
Nachteile später

Java Module (2/7) – Modul Deskriptor module-info.java

```
// optionales „open“ ermoeglicht Reflection
open module de.hs-osnabrueck.meinProjekt.modxy {
  // Inhalt fuer alle nutzbar
  exports de.hs-osnabrueck.meinProjekt.paket1;

  // Unterpakete, wenn nach aussen sichtbar, sind anzugeben
  exports de.hs-osnabrueck.meinProjekt.paket1.subpaket2;

  // explizit festlegbar, welche Module zugreifen duerfen
  exports de.modulSpecial to anderesMod1, de.anderesMod2;

  // benoetigte Module angeben
  requires blubb.anderesModul;

  // damit andere genutzte Module dieses Moduls auch nutzen
  //koennen
  requires transitive blubb.modulInAnderesModulBenoetigt
}
```

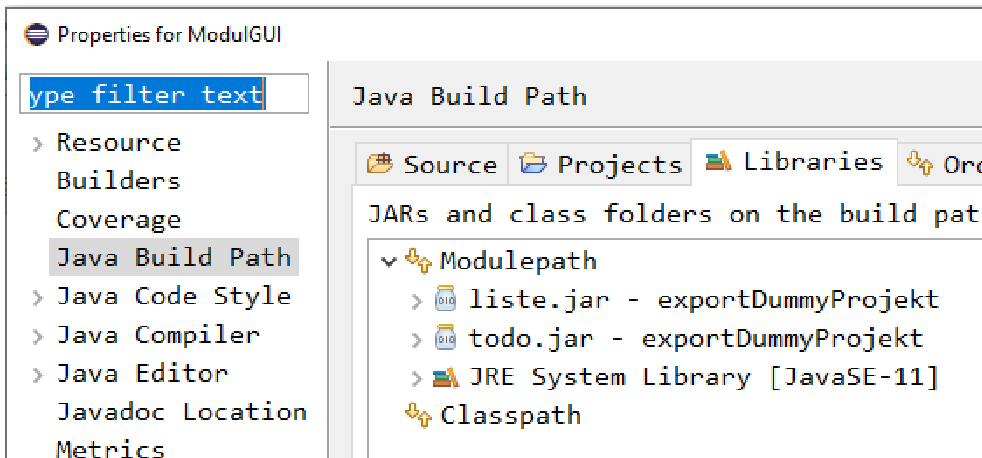
Java Module (3/7) – Beispiel Klassendiagramm

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES



Java Module (4/7) – in Eclipse

- jedes Modul als eigenes Projekt
- jedes Modul ein Deskriptor modul-info.java
- Module als Jar-Dateien exportiert
- Module werden im Module-Path eingebunden




```
ModulGUI
  src
    ich.gui
      EinUndAusgabe.java
      TextIO.java
    module-info.java
  JRE System Library
  Referenced Libraries
    liste.jar - exportDummyProjekt
    todo.jar - exportDummyProjekt
ModulListe
  src
    ich.listenpack
      Liste.java
      ListenVerw.java
    module-info.java
  JRE System Library
  Referenced Libraries
    todo.jar - exportDummyProjekt
ModulTodo
  JRE System Library
  src
    ich.todopack
      Todo.java
      TodoVerw.java
    module-info.java
```

Java Module (5/7) – Modul-Deskriptoren

```
open module guiModul {
  requires listenModul;
}


open module listenModul {
  requires transitive todoModul;
  exports ich.listenpack;
}


open module todoModul {
  exports ich.todopack;  // to listenModul, guiModul;
}
```

Java Module (6/7) – Module Arten

- Java selbst in Module aufgeteilt, java.base automatisch eingetragen
- weitere Elemente der Java-Bibliothek müssen angegeben werden, z. B. requires java.sql;

Zugriffe

Modulepath

Application Explicit Modules
- Module mit Modul-Deskriptor
- Variante:open, alles für Reflection freigegeben

Automatic Modules
- klassische Jars ohne Modul-Deskriptor
- exportiert alle Pakete
- importiert alle Pakete anderer Module

Classpath

Unnamed Modules
- klassische Jars ohne Modul-Deskriptor
- werden zusammen als ein Modul angesehen
- exportiert alle Pakete

```
requires dbunit;
requires
```

Name of automatic module 'dbunit' is unstable, it is derived from the module's file name.

Java Module (7/7) – kritische Analyse

- ab Java 9 müssen alle Klassen in Modulen enthalten sein
- damit wäre Inkompatibilität mit Java 8 riesig
- Trick: „alte“ Pakete gehören implizit zu einem Default-Modul (unnamed module); unklar wie lang diese Lösung existiert; im Module-Path werden alte Jars zu automatic modules

- viele Werkzeuge nutzen „Innereien“ der JVM, z. B. Reflection
- z. B. JPA, automatische Generierung und Nutzung von Tabellen zu fast beliebigen Java-Klassen
- diese Nutzung ist per Default in Java 9 ausgeschaltet, muss über „open“ ermöglicht werden (auch beim VM-Start konfigurierbar)
- viele Frameworks und Bibliotheken laufen immer (noch) nicht mit Java ab Version 9 zusammen
- Fazit: keine klare Empfehlung, neues Projekt mit Modulen zu machen

Video

8. Optimierung des Designmodells

- 8.1 Design im Kleinen
- 8.2 Model View Controller
- 8.3 Vorstellung einiger GoF-Pattern
- 8.4 Abschlussbemerkungen zu Pattern
- 8.5 Patternorientierte Konzepte in der Programmierung

Zentrale Aufgabe: von Analyse zum Design (1/2)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- Analyse der Klassen:
  - haben sie klar definierte Aufgabe
  - können Klassen vereinigt werden
  - sollten Klassen aufgeteilt werden
  - welche Optimierungen sind aus Design-Sicht möglich? (zentrale Frage, untersuchen wir weiter)
- Exemplar- und Klassenvariablen müssen Typen haben
- Variablen und Methoden brauchen Sichtbarkeiten
- Methoden brauchen Rückgabe- und Parametertypen (Signaturen); in Java und C++ spielen Ausnahmen eine Rolle

Zentrale Aufgabe: von Analyse zum Design (2/2)

- für Assoziationen
  - Multiplizitäten beachten
  - über mögliche Richtungen nachdenken
  - Art der Zugehörigkeit klären
- GUI-Klassen und persistente Datenhaltung einbauen
- Anmerkung 1: Übergang von Analyse zu Design ist durch Iterationen (Verfeinerungen) fließend
- Anmerkung 2: Die vorgestellten Regeln sind häufig 90-10 Regeln (in 90% müssen sie angewandt werden, bei Verstößen muss man argumentieren können, warum)

Einschub: Coding-Guidelines

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- sehr wichtiges Hilfsmittel, damit alle Code lesen können
- auf den Folien wird für Kompaktheit teilweise drauf verzichtet

Praktikum: minimale Regeln

- Alle Imports ausschreiben ~~`import java.util.*`~~
- Eine Variante von Einrückungen (Eclipse-Stil)
- Objektvariablen und Objektmethoden vorne mit **`this.`**

```
public int getMatnr() {
  return this.matnr;
}
```

- Bei if und Schleifen immer Block mit geschweiften Klammern
- Keine Klasse im Default-Package („ganz oben“)
- Alle Namen sind intuitiv lesbar für andere Leute
- Pro Zeile nur ein Befehl
- Java-übliche CamelCase-Notation

Einfache Basisregeln

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

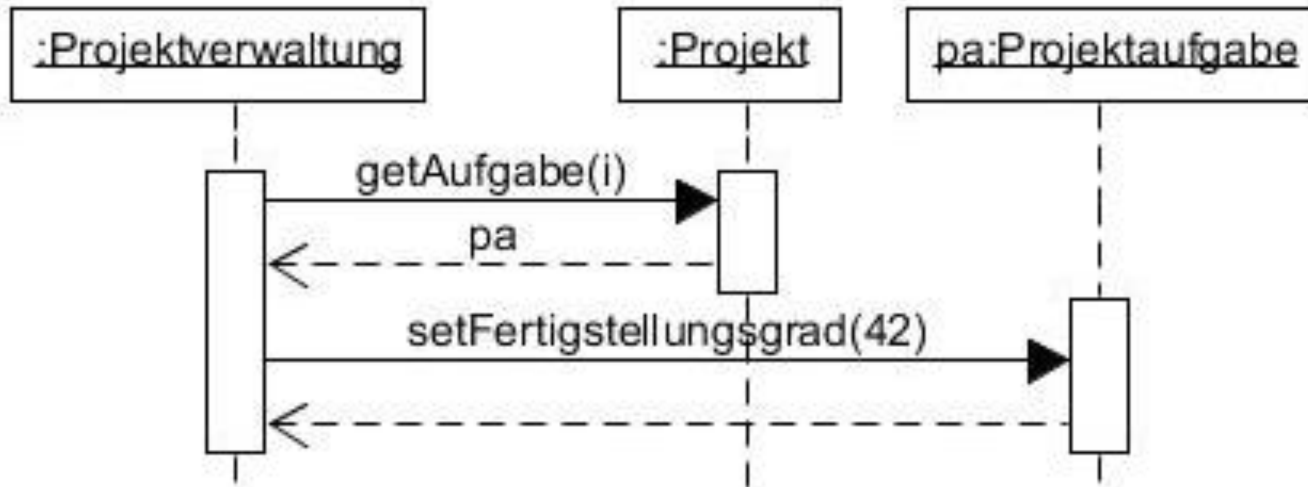
8.1

- KISS = Keep It Simple Stupid, man soll die einfachst mögliche Realisierung wählen, die das Problem vollständig löst und gut nachvollziehbar ist (kein „Quick and Dirty“, sondern eine klare Entscheidung für einen einfachen Entwicklungsstil)

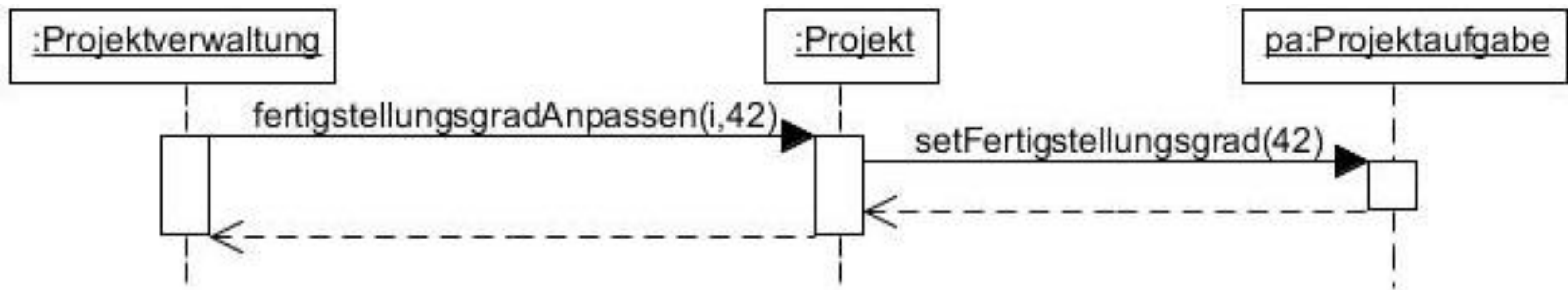
- YAGNI = You Ain’t Gonna Need It, keine Verallgemeinerungen entwickeln, die das Design für theoretisch in der Zukunft vielleicht gewünschte Erweiterungen vereinfachen

Keine allwissenden Klassen

ändere Fertigstellungsgrad einer Projektaufgabe

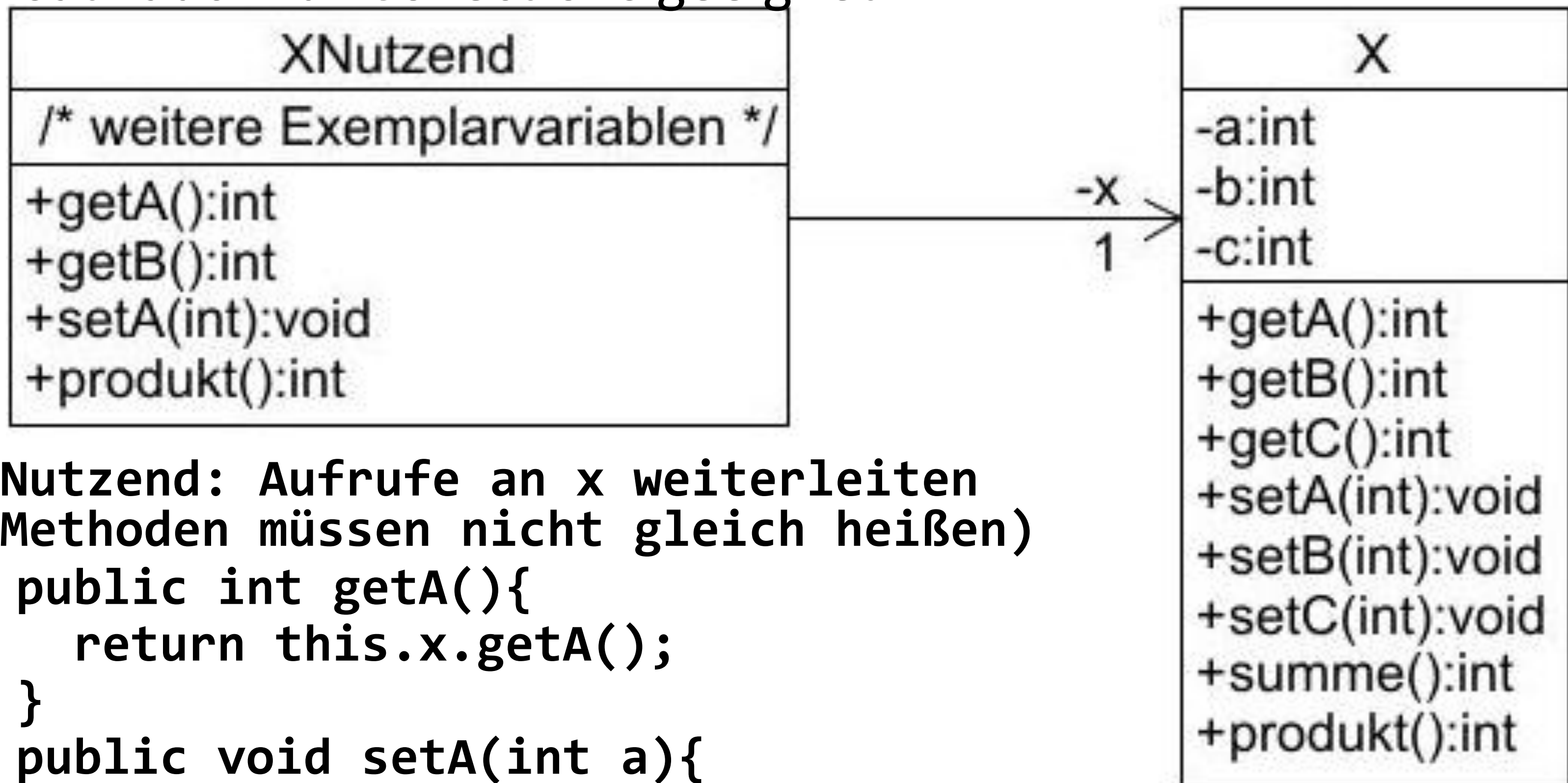


besser:



„Verpacken“ von Exemplarvariablen (Aggregation)

- Generell kann man für Exemplarvariablen vom Typ X statt einer get-Methode alle Methoden von X anbieten, die man an die Exemplarvariable weiterleiten will.
- Ansatz auch für Collections geeignet



- **XNutzend: Aufrufe an x weiterleiten (Methoden müssen nicht gleich heißen)**

```
public int getA(){
  return this.x.getA();
}
public void setA(int a){
  this.x.setA(a);
}
```

Erinnerung: Bedeutung von Schnittstellen

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

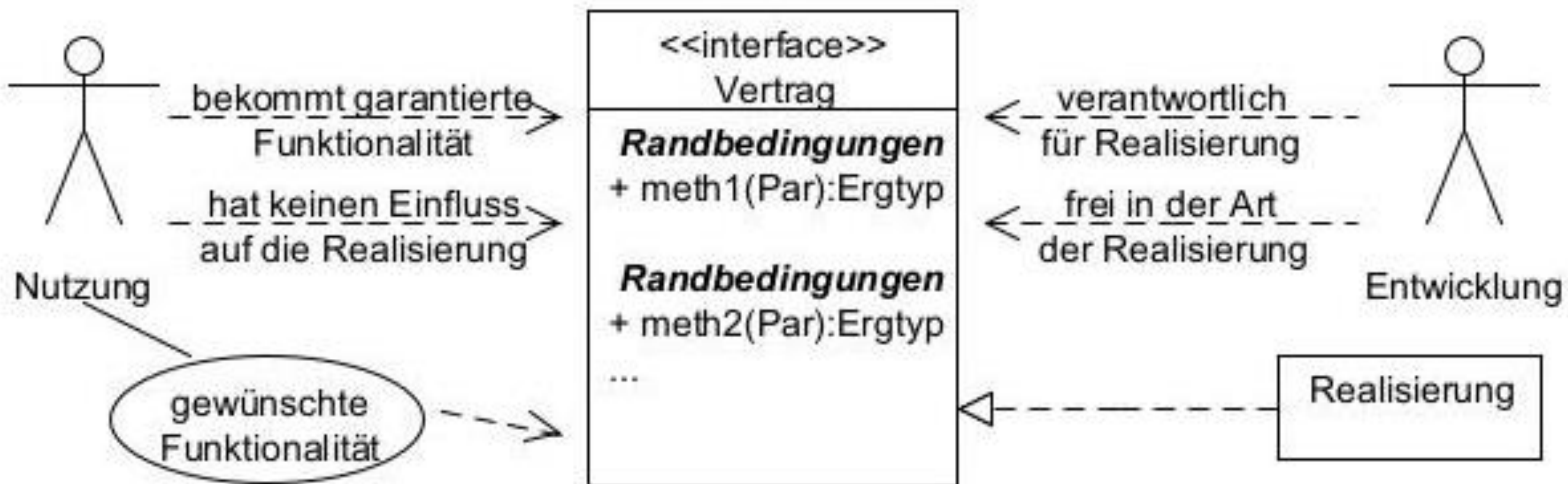
- Schnittstellen sind zentrales Element des *Design by Contract*
- vorgegebene Aufgabe: Implementiere mir folgende Funktionalität ... beschrieben durch
  - Vorbedingung
  - Signatur <Sichtbarkeit> <Methodenname>(<Parameter>)...
  - Nachbedingung
- entwickelnde Person realisiert OO-Programm (Details sind frei)
- entwickelnde Person garantiert, dass Schnittstelle (oder Fassade) gewünschte Funktionalität liefert
- generell sollte man bei Vererbungen und Implementierungen die am wenigsten spezielle benötigte Klasse nutzen; deshalb

  `List<Projektaufgaben> aufgaben` und nicht

  `ArrayList<Projektaufgaben> aufgaben` im Code

zentrale Folie: Design by Contract

- abstrakte Klasse stellt einen Vertrag dar
- Realisierer garantiert die gewünschte Funktionalität
- nutzende Person kann konkretes Objekt mit Funktionalität erhalten
- wie die Realisierung aussieht, ist allein Sache der realisierenden Person



Grundidee von Design-Pattern

8.2

Damit nicht alle Klassen eng miteinander gekoppelt sind, gibt es Ansätze:

- die Aufgaben einer Klasse von der Verwaltung der Klassen, die Informationen dieser Klasse benötigen, zu trennen
- die Erzeugung von Objekten möglichst flexibel zu gestalten
- Interfaces zur Trennung von Implementierung und angebotenen Methoden einzusetzen

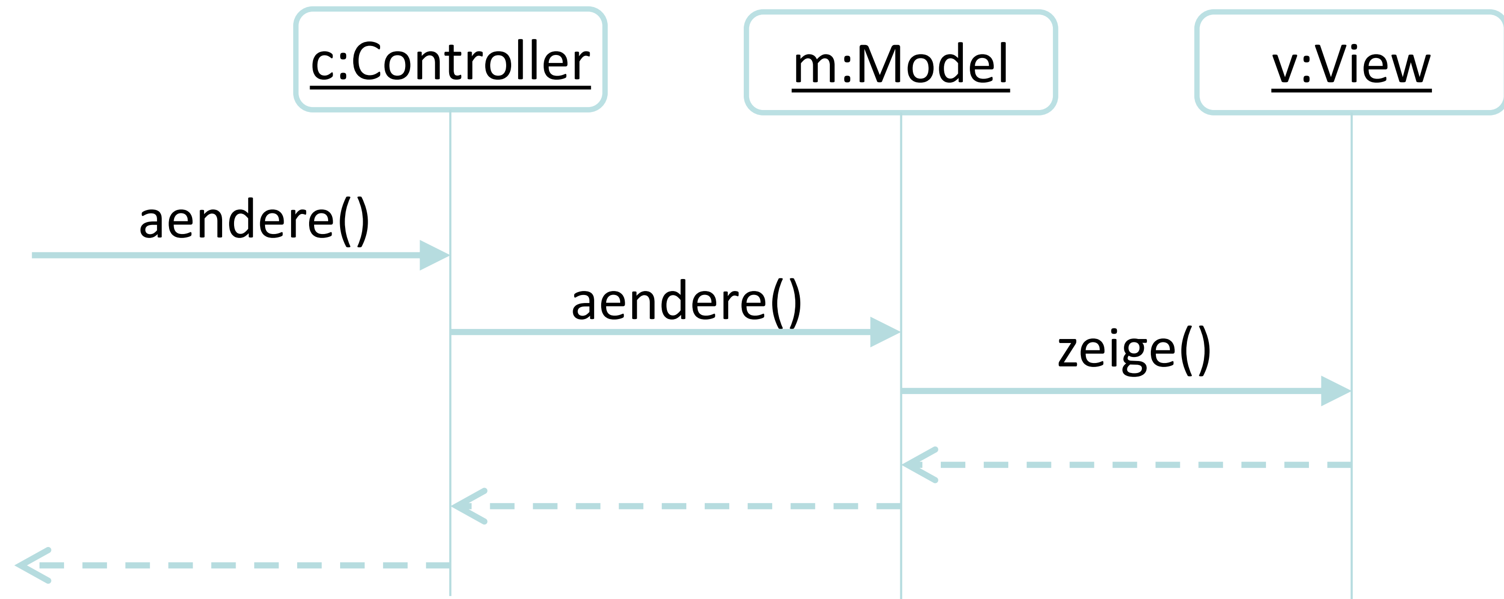
- Hierzu werden so genannte Design-Pattern eingesetzt, die für einen bestimmten Aufgabentyp eine flexible Lösung vorschlagen
- oft zitiert: E. Gamma, R. Helm, R. Johnson, J. Vlissides, Entwurfsmuster, Addison-Wesley, 2004 (Gang of Four [GoF]-Buch, hier neuere Auflage)

Model-View-Controller

Video

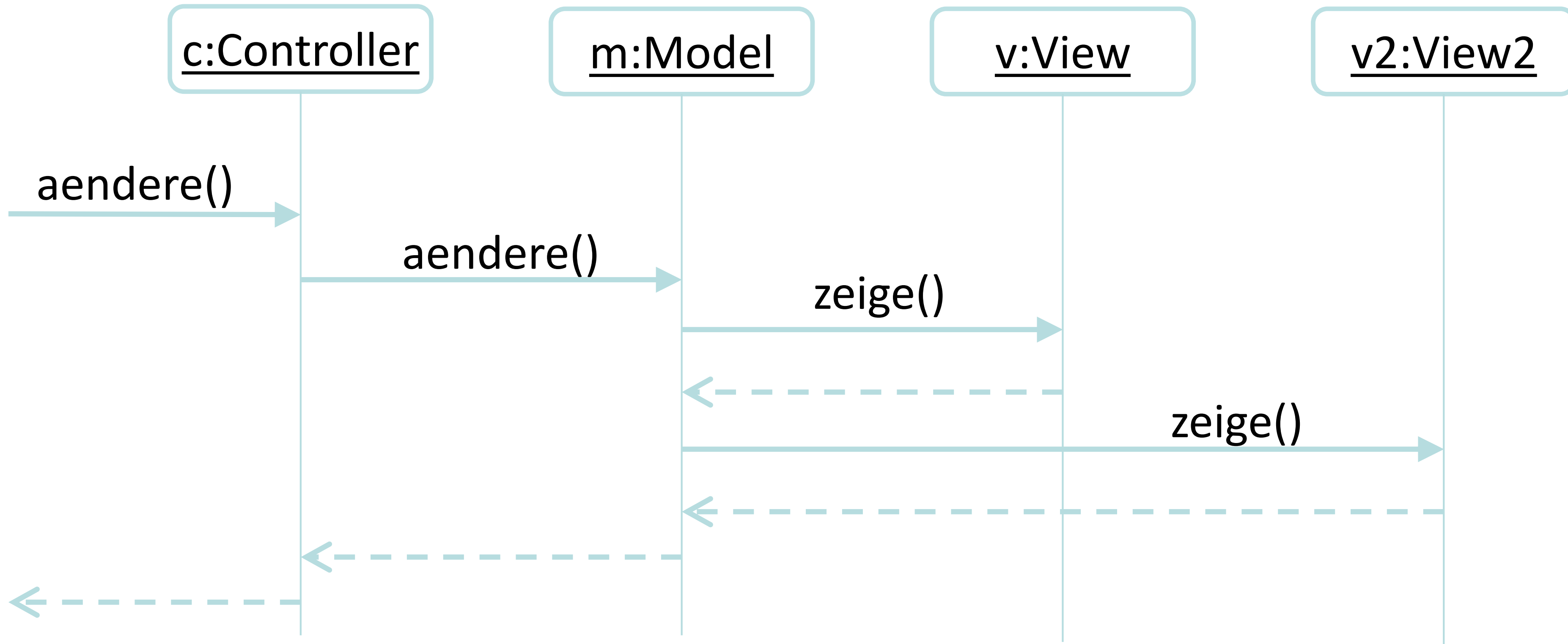
- Typisch für graphische Oberflächen ist, dass es Objekte zur Eingabe gibt, die zur Bearbeitung der eigentlichen Inhaltsklasse führen, die dann eventuell zu Änderung der Anzeige führen
- Die Aufteilung in die drei genannten Aufgaben führt zum Model-View-Controller (MVC)-Ansatz
- MVC wurde zuerst in Smalltalk Ende der 80'er des vorigen Jahrhunderts eingesetzt:
  - Model: Zustandsinformation der Komponente (Inhaltsklasse)
  - View: Beobachter des Zustands, um diesen darzustellen; es kann viele Views geben
  - Controller: Legt das Verhalten der Komponente auf Benutzungseingaben fest

MVC – einfacher Kommunikationsablauf




```
// Variante: Model kennt View
// Erzeugung
View v = new View();
Model m = new Model(v);
Controller c = new Controller(m);
```

MVC: was bei mehreren Views

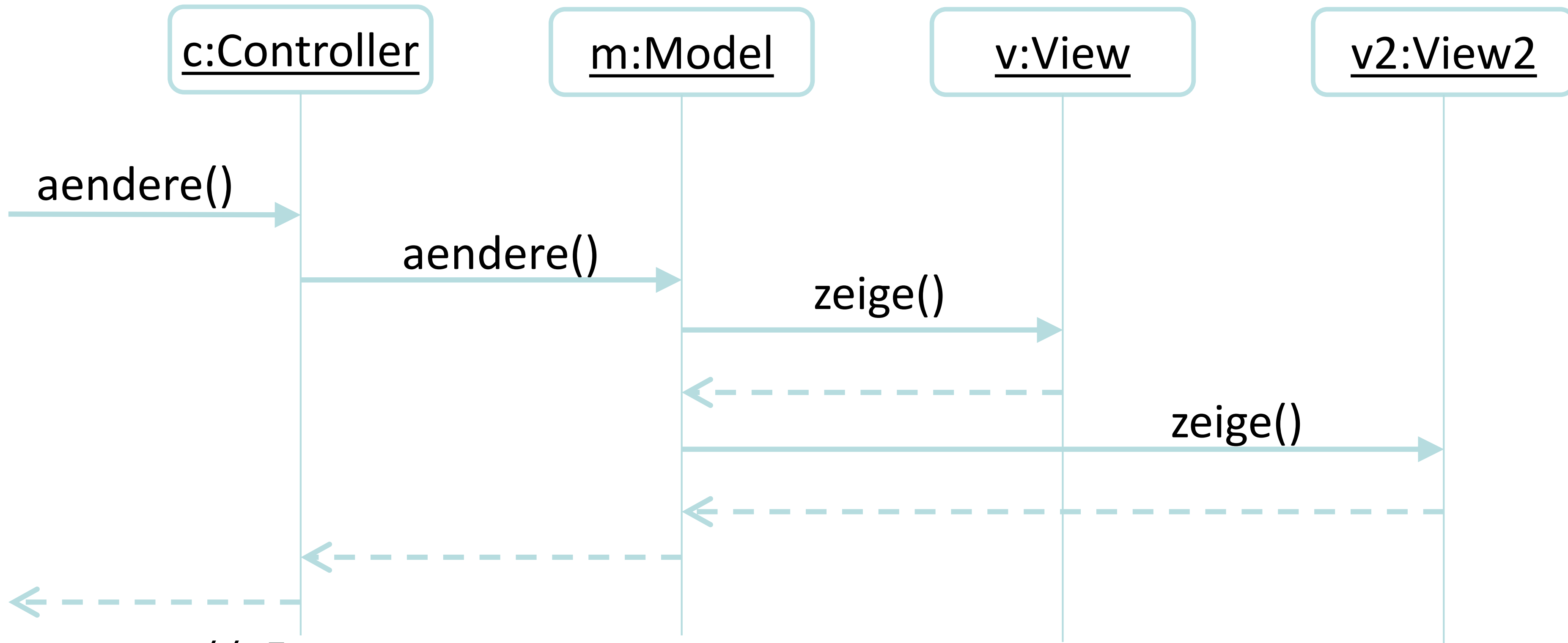



```
// Erzeugung
View v = new View();
View2 v2 = new View2();
Model m = new Model(v, v2);  // ???
Controller c = new Controller(m);
```

generereller Ablauf gut, aber Erstellung hölzern

MVC: mehrere Views

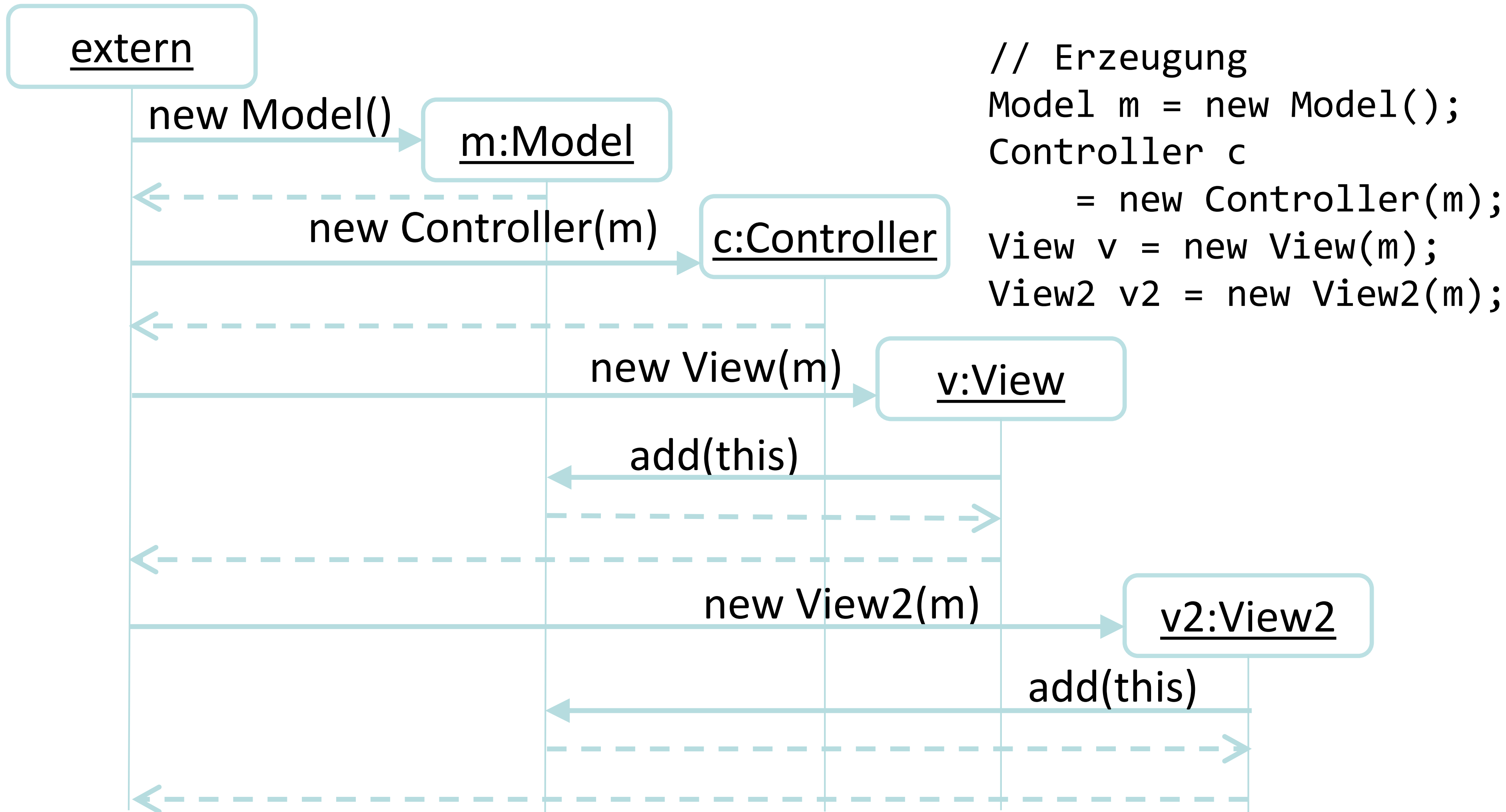
HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES




```
// Erzeugung
View v = new View();
View2 v2 = new View2();
Model m = new Model();
m.add(v);
m.add(v2);
Controller c = new Controller(m);
```

besser, aber View vom Model getrennt, neue Daten immer als Parameter von zeige()

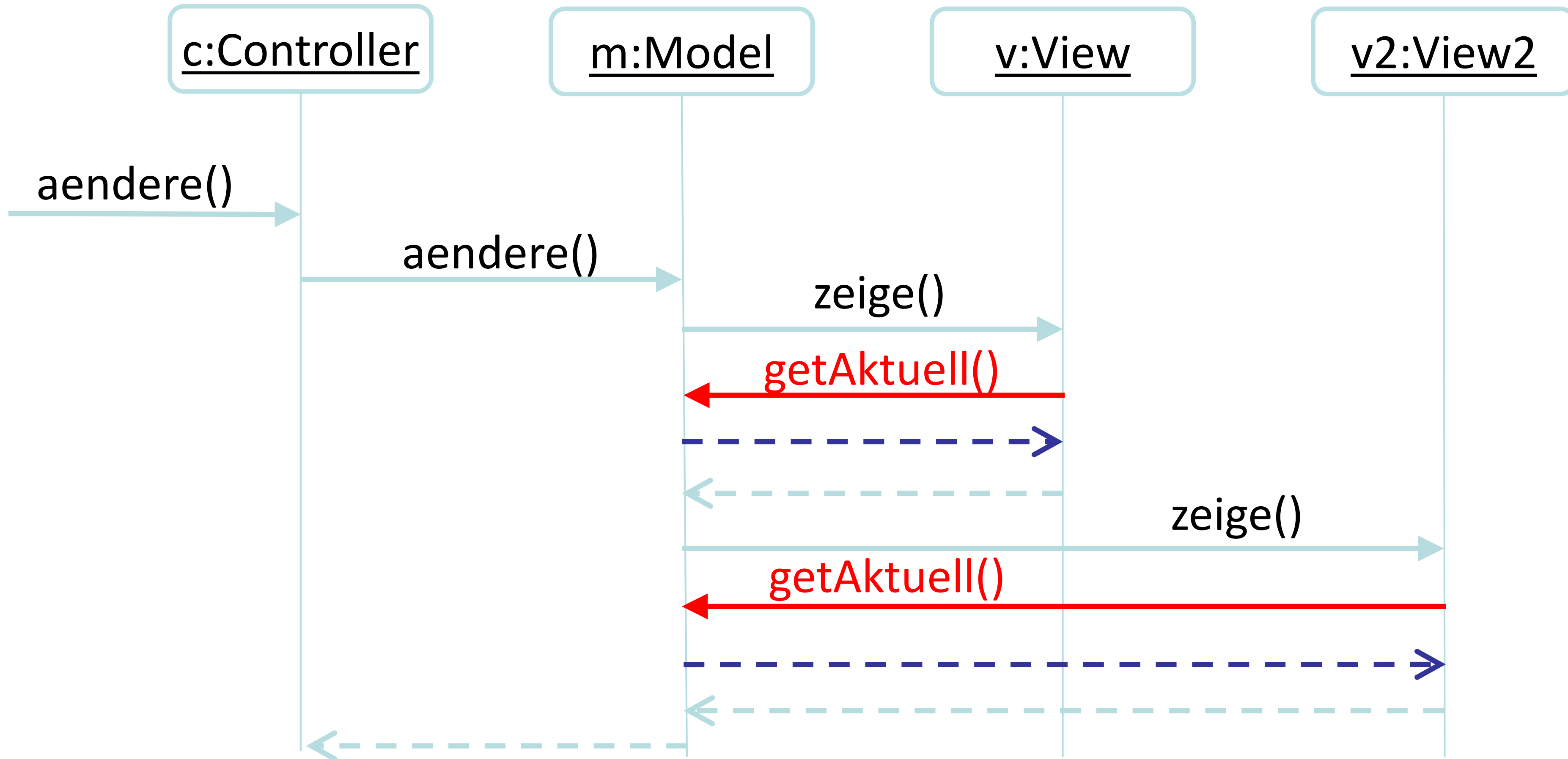
MVC: Model hält Sammlung angeschlossener Views




```
// Erzeugung
Model m = new Model();
Controller c
    = new Controller(m);
View v = new View(m);
View2 v2 = new View2(m);
```

MVC: Model hält Sammlung angeschlossener Views

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES



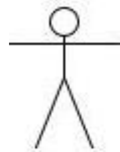
Java-Beispiel zum MVC (1/7)

Ich bin der View

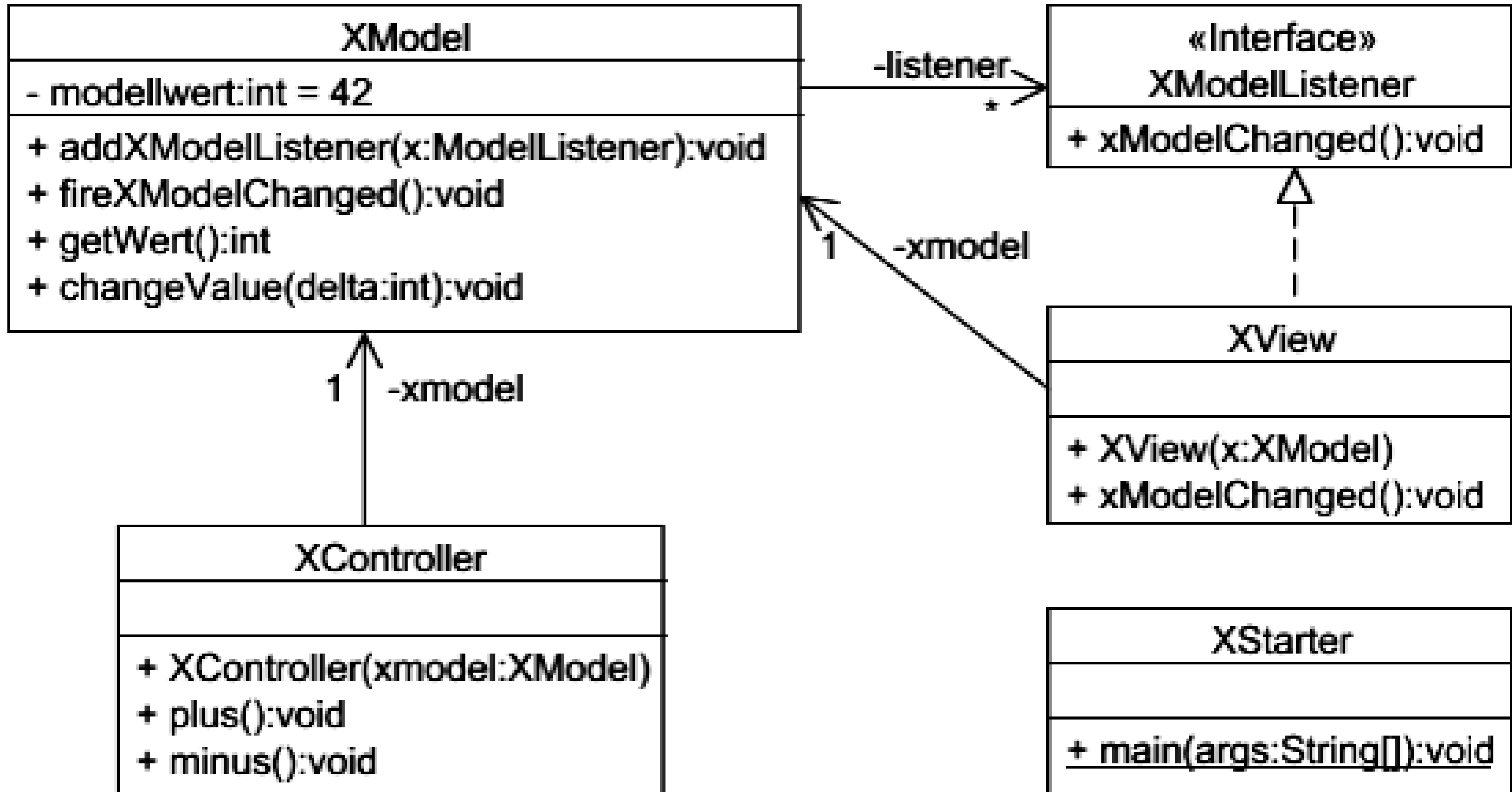
Modellwert: 43

Ich bin der Controller

plus minus



Java-Beispiel zum MVC (2/7)



Java-Beispiel zum MVC (3/7)


```
public class XModel{
  private List<XModelListener>listener = new ArrayList<>();
  private int modellwert = 42;

  public void addXModelListener(XModelListener x){
    this.listener.add(x); //Verwaltung der Listener des Modells
  }

  public int getWert(){ //Auslesen der Modellinhalte
    return this.modellwert;
  }

  public void changeValue(int delta){ //Veränderung des Modells
    this.modellwert += delta;
    this.fireXModelChanged(); // alle informieren
  }

  private void fireXModelChanged(){
    for(XModelListener x: this.listener)
      x.xModelChanged();
  }
}
```

Java-Beispiel zum MVC (4/7)

```
public class XView extends JFrame implements XModelListener{
  private XModel xmodel;
  private JLabel jlabel = new JLabel("Modellwert:    ");
  public XView(XModel x){
    super("Ich bin der View");
    this.xmodel = x;
    this.xmodel.addXModelListener(this);
    //Rest Swing für Anzeige
    super.getContentPane().add(jlabel);
    super.setDefaultCloseOperation(EXIT_ON_CLOSE);
    super.setSize(250, 60);
    super.setLocation(0, 0);
    super.setVisible(true);
  }
  @Override
  public void xModelChanged() {
    this.jlabel.setText("Modellwert: "+this.xmodel.getWert());
  }
}
```

Java-Beispiel zum MVC (5/7)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

```
import java.awt.FlowLayout;
import java.awt.event.*; // hier zur Abkuerzung, in echten
import javax.swing.*;    // Projekten diesen * vermeiden

public class XController extends JFrame{
  private XModel xmodel;

  public XController(XModel x){
    super("Ich bin der Controller");
    this.xmodel = x;
    super.getContentPane().setLayout(new FlowLayout());
    JButton plus = new JButton("plus");
    super.getContentPane().add(plus);
    plus.addActionListener(new ActionListener(){
      @Override
      public void actionPerformed(ActionEvent e){
        xmodel.changeValue(1);
      }}
    );
```

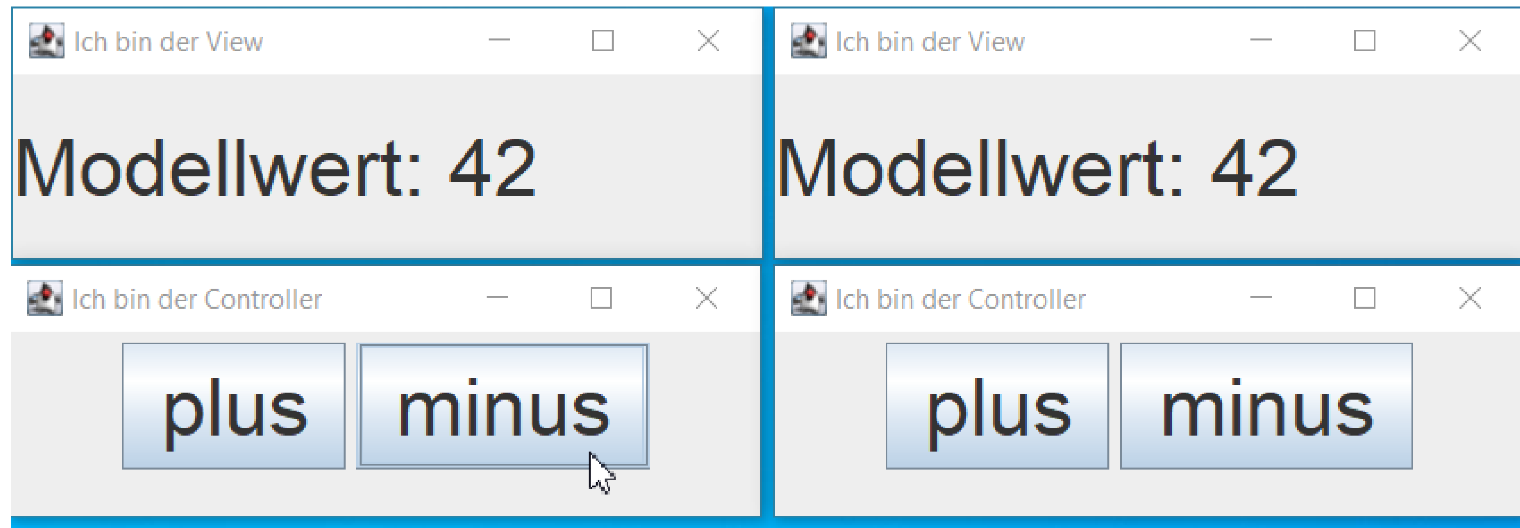
Java-Beispiel zum MVC (6/7)

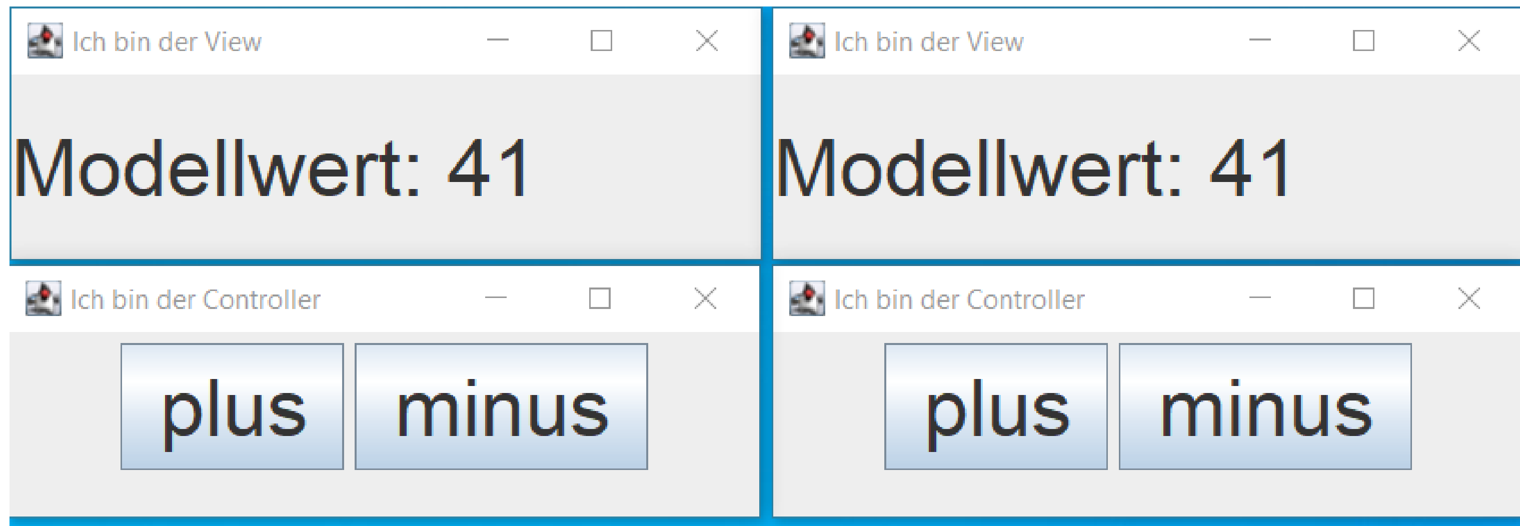

```
    JButton minus = new JButton("minus");
    super.getContentPane().add(minus);
    minus.addActionListener(new ActionListener(){
      @Override
      public void actionPerformed(ActionEvent e){
        xmodel.changeValue(-1);
      }}
    );
    super.setDefaultCloseOperation(EXIT_ON_CLOSE);
    super.setSize(250, 60);
    super.setLocation(0, 90);
    super.setVisible(true);
  }
}
```

Java-Beispiel zum MVC (7/7)

```
public interface XModelListener {
  public void xModelChanged();
  /* Anmerkung: alternativ kann man auch geänderte
     Werte als Parameter übertragen */
}

public class XStarter {
  public static void main(String[] args) {
    XModel x = new XModel();
    new XView(x);
    new XController(x);
  }
}
```

Mehrere Views – mehrere Controller – ein Model



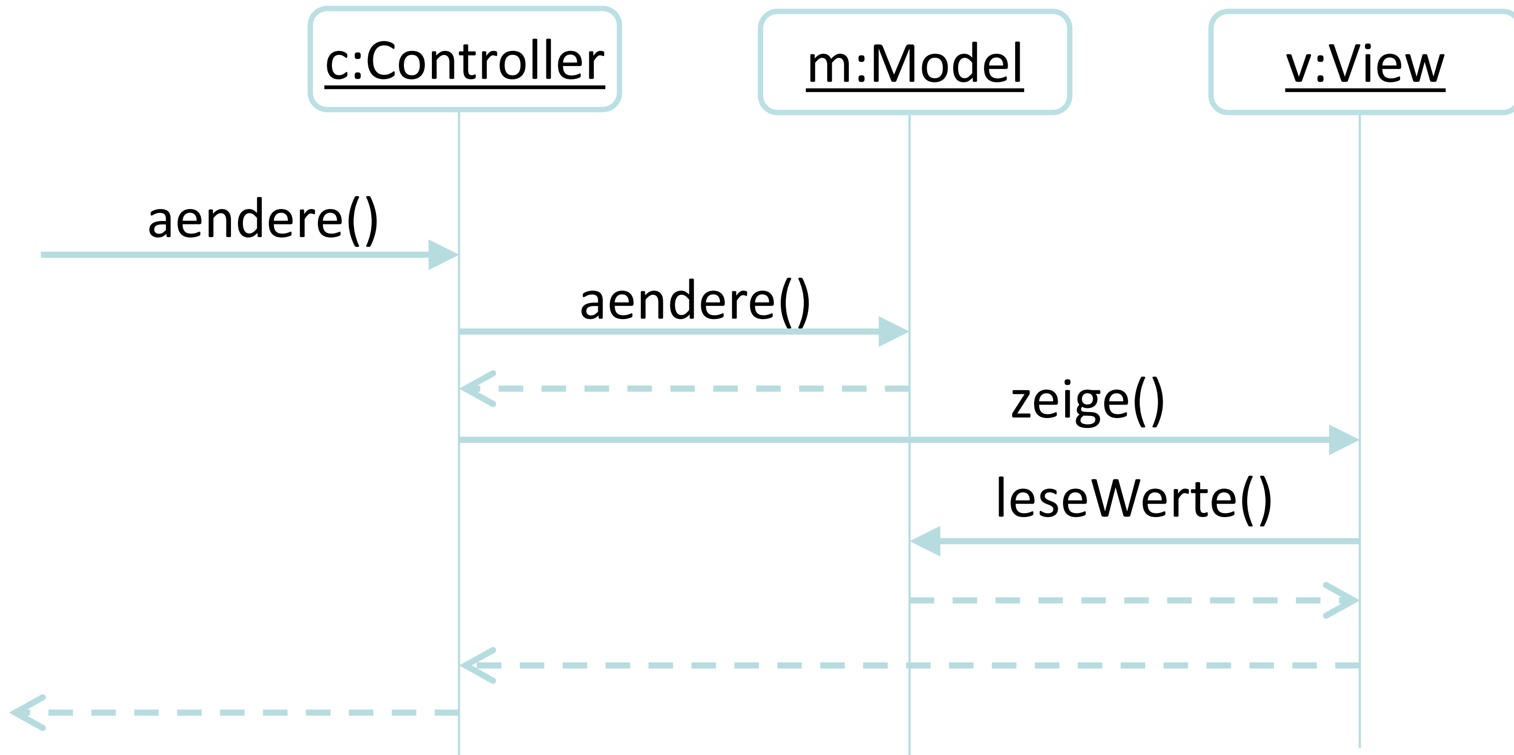


Pattern-Varianten

Pattern schlagen eine mögliche Lösung vor; kann in Projekten variiert werden

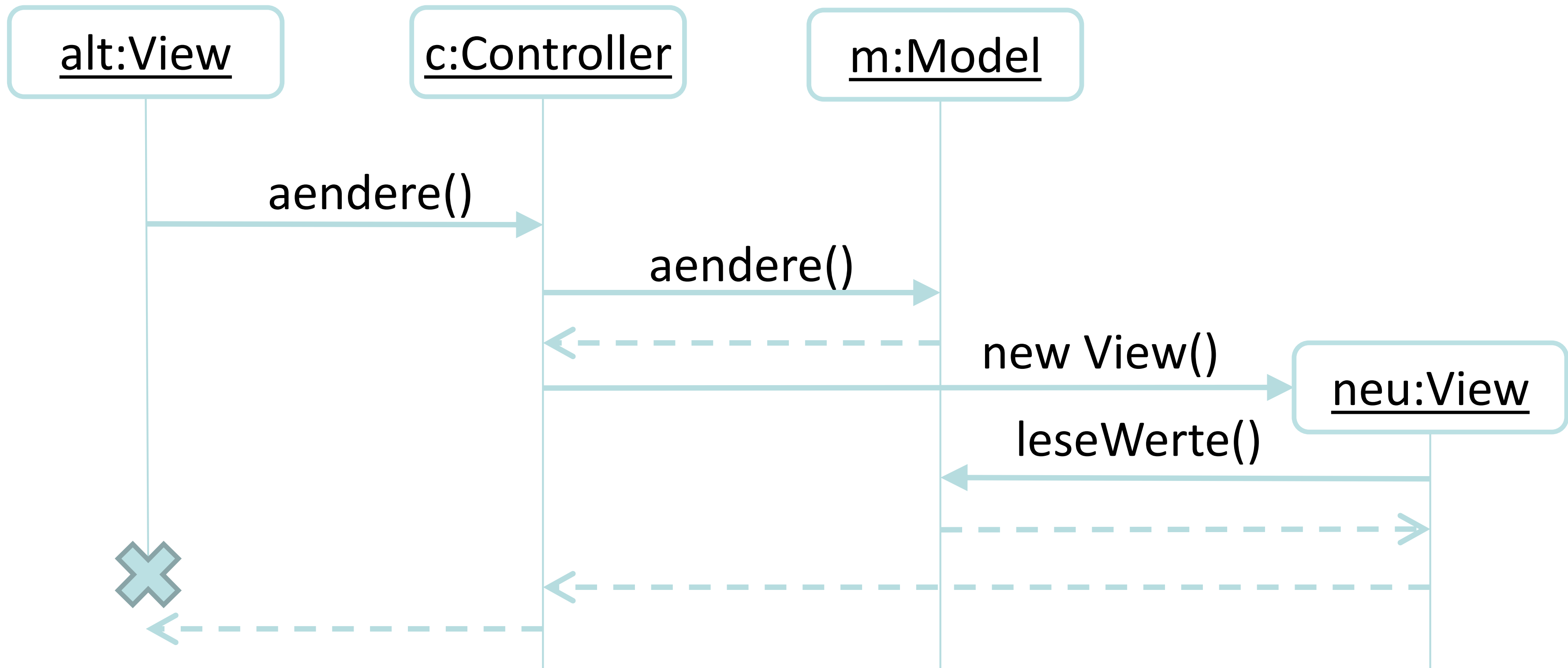
- Interface weglassen, wenn nur eine View-Art
- Aufteilung auch sinnvoll, wenn nur ein View existiert (klare Aufgabentrennung)
- wenn Controller und View eng verknüpft, können sie vereinigt werden, z. B. GUI-Elemente in Java-Swing
- Listenerverwaltung kann vom Model in Controller verlegt werden
- auch ohne Listen ist MVC-Aufteilung sinnvoll

Ablaufvariante: Controller managt alles




```
// Erzeugung
Model m = new Model();
View v = new View(m);
Controller c = new Controller(m, v);
```

Variante der Ablaufvariante: Controller managt alles




```
// Internet
// Web-Seite (View) ruft Controller auf
// Controller ändert Model
// Controller erzeugt neuen View
// View berechnet aus Model neue Web-Seite,
//    Web-Seite beinhaltet Verbindung zum Controller
```

MVC als Design-Konzept

- Kommunikationswege hängen von konkreter Umsetzung ab
- Viele Varianten:
  - Model-Delegate (Controller und View zusammen)
  - Model-View-ViewModel (eigenes Model für Darstellung)
  - Model-View-Presenter
  - Model-View-Adapter

- eine Umsetzung: Controller steuert Änderungen des Modells, Modell teilt allen Views mit, dass eine Änderung aufgetreten ist
- folgende Folien: eine Verknüpfungsmöglichkeit in MVC

Ansatz Observer-Observable

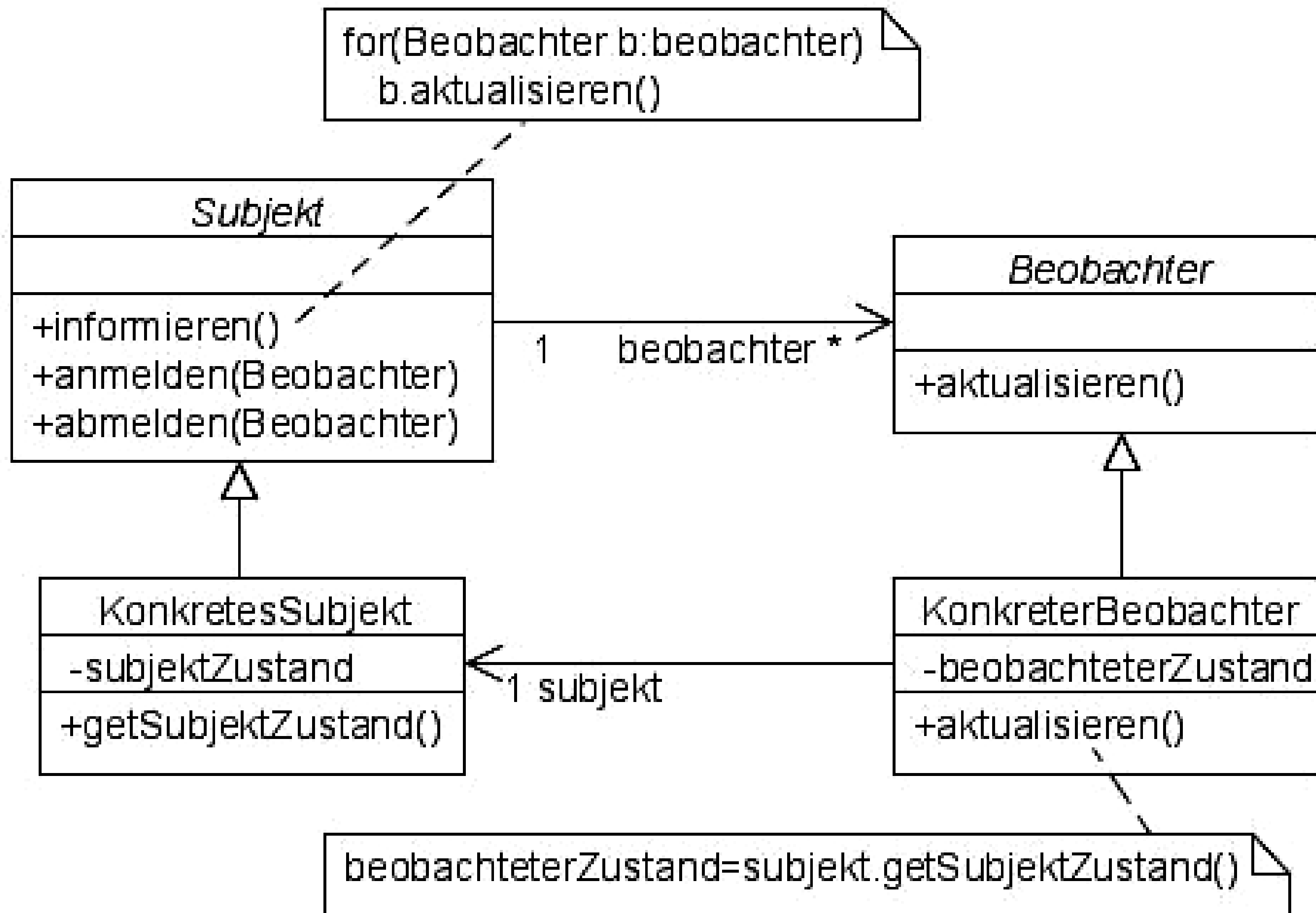
8.3 Video

- Es gibt Subjekte für deren Zustand sich viele interessieren (z. B. Nachrichtenkanäle)
- Die Subjekte bieten die Möglichkeit, dass sich Interessenten anmelden (z. B. Kanal abonnieren)
- Bei jeder Subjektzustandsänderung werden Interessenten informiert (neue Nachrichten)

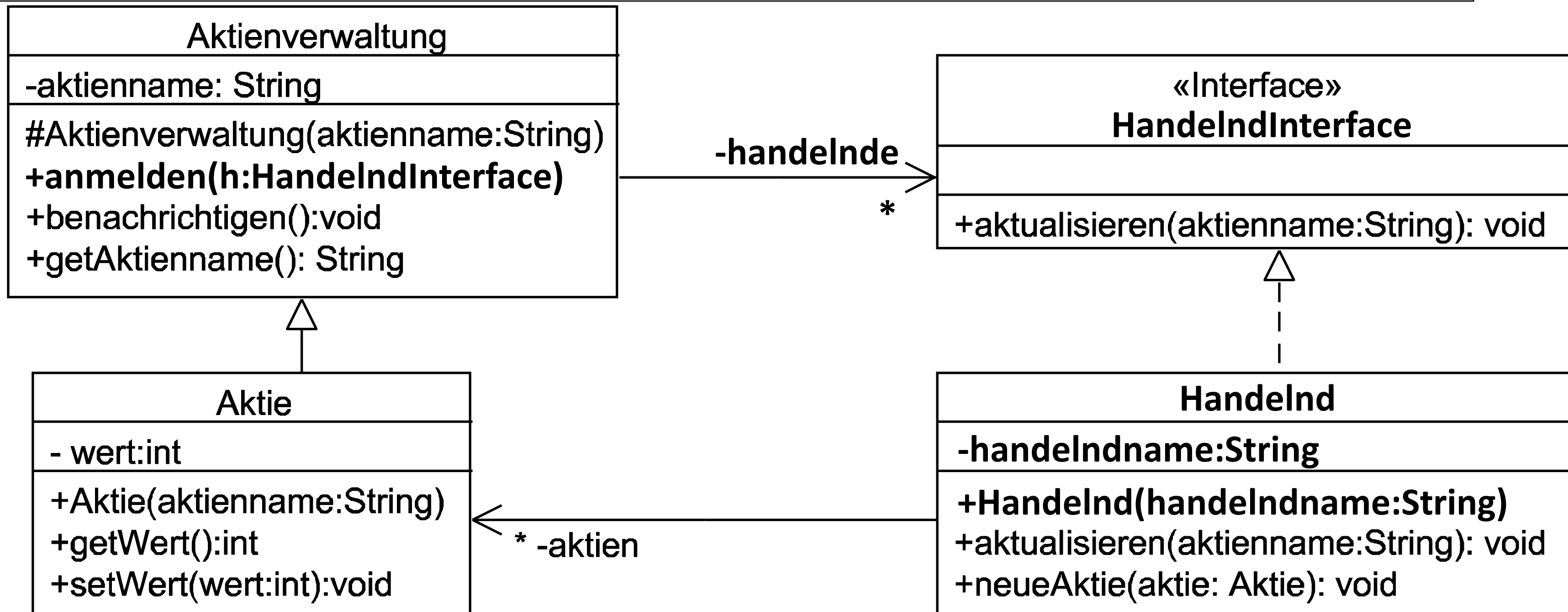
- Interessent muss sich bei Subjekt anmelden
- Damit obiges Objekt weiß, wie Interessent angesprochen werden soll, muss Interessent Schnittstelle realisieren

- Hinweis: Enge Verwandtschaft zur hier vorgestellten Model-View-Controller-Variante

Beobachter (Observer – Observable)



Beobachter – Beispielaufgabe (1/5)



Gegeben sei obiges Klassendiagramm, das die Nutzung des Observer-Pattern zeigt. Dabei interessieren sich Aktien handelnde Personen für Aktienkurse und können sich bei Aktien anmelden, die ihnen zuvor mit neueAktie übergeben wurden. Falls sich der Wert dieser Aktien ändert, werden alle interessierten Handelnden benachrichtigt, welche Aktie (ihr Name) sich geändert hat. Aktien haben einen eindeutigen Aktiennamen.

Beobachter – Beispielaufgabe (2/5)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

```
import java.util.ArrayList;
import java.util.List;
public class Aktienverwaltung {
  private String aktienname;
  private List<HandelndInterface> handelnde
                     = new ArrayList<>();

  protected Aktienverwaltung(String aktienname) {
    this.aktienname = aktienname;
  }

  public void anmelden(HandelndInterface h){
    this.handelnde.add(h);
  }

  public void benachrichtigen(){
    for(HandelndInterface h: this.handelnde)
      h.aktualisieren(aktienname);
  }

  public String getAktienname(){
    return this.aktienname;
  }
}
```

Beobachter – Beispielaufgabe (3/5)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

```
public class Aktie extends Aktienverwaltung {

  private int wert=42;
  public Aktie(String aktienname){
    super(aktienname);
  }

  public int getWert() {
    return this.wert;
  }

  public void setWert(int wert) {
    this.wert = wert;
    super.benachrichtigen();
  }

  @Override
  public String toString(){
    return super.getAktienname();
  }
}
```

Beobachter – Beispielaufgabe (4/5)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

```
public interface HandelndInterface {
  public void aktualisieren(String aktienname);
}

import java.util.ArrayList;
import java.util.List;
public class Handelnd implements HandelndInterface {

  private String handelndname;
  private List<Aktie> aktien = new ArrayList<>();

  public Handelnd(String handelndname) {
    this.handelndname = handelndname;
  }

  public void neueAktie(Aktie a){
    this.aktien.add(a);
    a.anmelden(this);
  }
```

Beobachter – Beispielaufgabe (5/5)

```
  public void aktualisieren(String aktienname) {
    System.out.println(handelndname
            + " hat neuen Wert für " + aktienname + ": "
            + this.holeAktienWert(aktienname) );
  }

  //alternativ beim Aktualisieren Wert mitschicken
  private int holeAktienWert(String aktienname){
    for(Aktie a: this.aktien)
      if(a.getAktienname().equals(aktienname)) {
        return a.getWert();
      }
    //nie erreichen
    assert(false); // Java, ist nicht JUnit!
    return 0;
  }

  @Override
  public String toString(){
    return this.handelndname;
  }
}
```

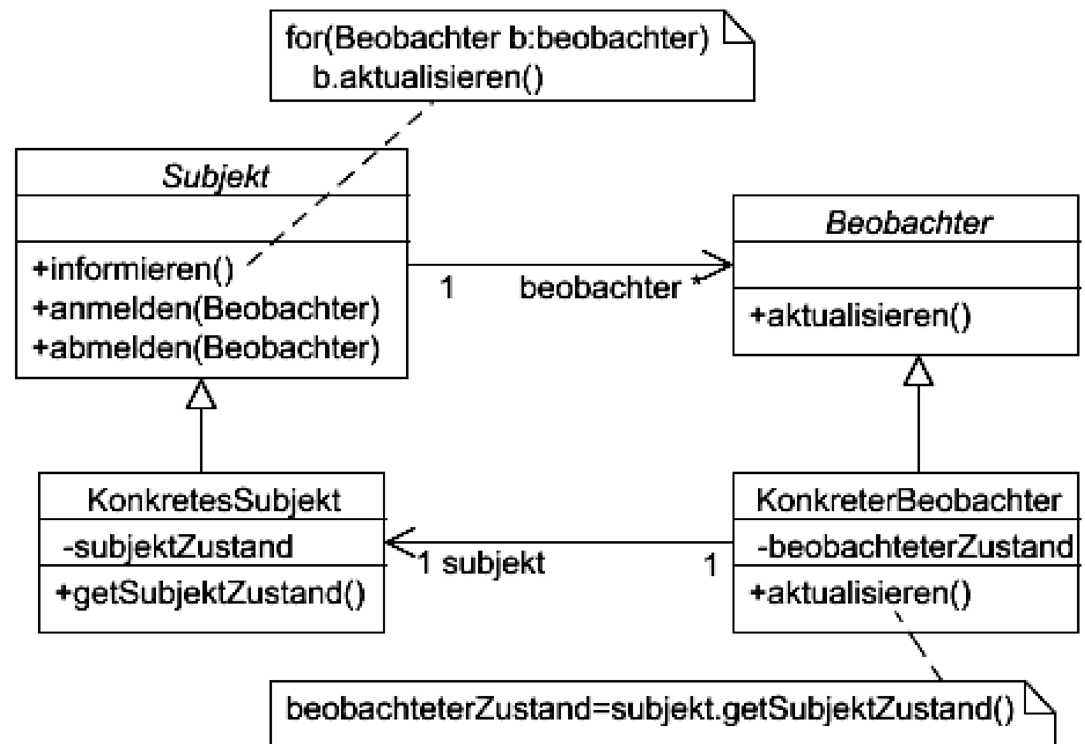
Pattern und Varianten

- Für fast jedes Pattern gibt es Varianten, die abhängig von Randbedingungen sinnvoller sein können

Bsp.: Wertänderung mit aktualisieren() übertragen

Bsp.: Java hat keine Mehrfachvererbung

- Subjekt wird Interface
- Listenverwaltung in Hilfsklasse
- Konkretes Subjekt delegiert Listen-aufgaben an Objekt der Hilfsklasse



Adapter - Problem

Video

Szenario:

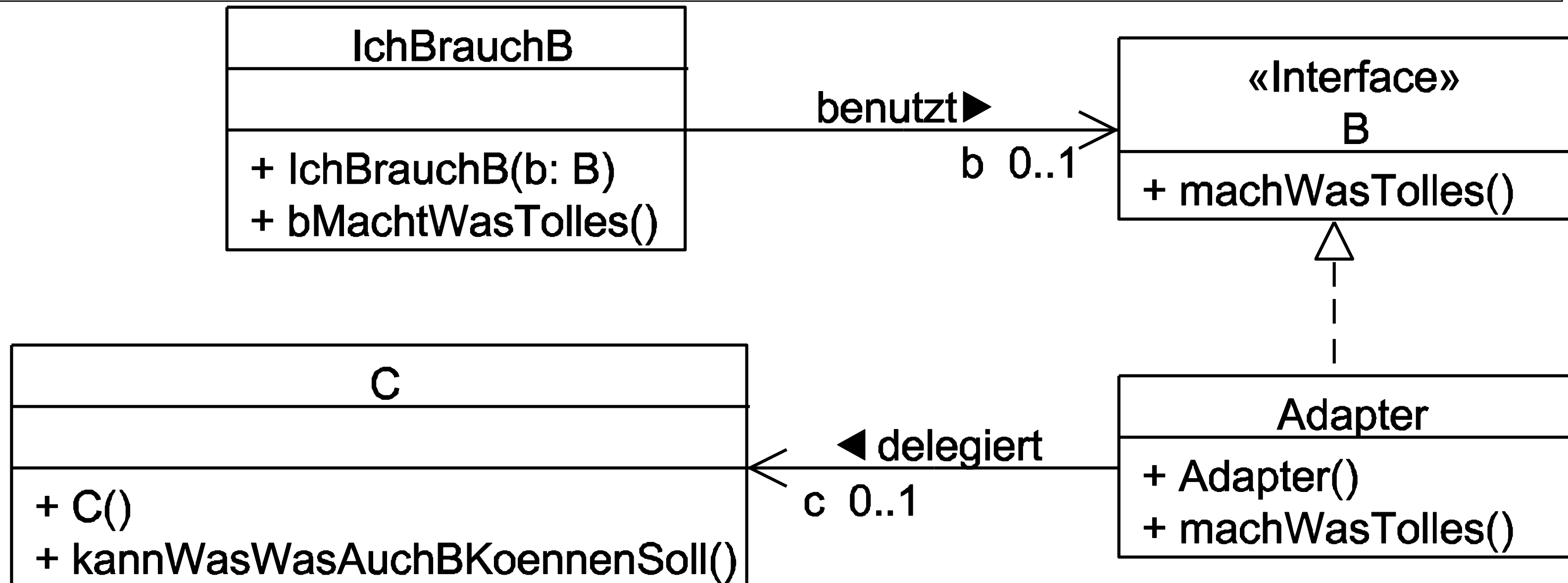
- Klasse IchBrauchB benötigt ein Objekt der Klasse B, genauer spezielle Funktionalität (Methode) der Klasse B
- Wir haben bereits eine Klasse C, die die von IchBrauchB von B geforderte Funktionalität anbietet
- C bietet die gewünschte Funktionalität unter dem falschen Methodennamen an, da C Teil einer komplexen Klassenstruktur ist, kann C nicht verändert werden

Lösung:

- Schreibe Adapterklasse, die sich wie B verhält (von B erbt bzw. Interface B implementiert) und Objekt der Klasse C aggregiert
- Adapter leitet Aufruf der von IchBrauchB gewünschten Funktionalität an C weiter

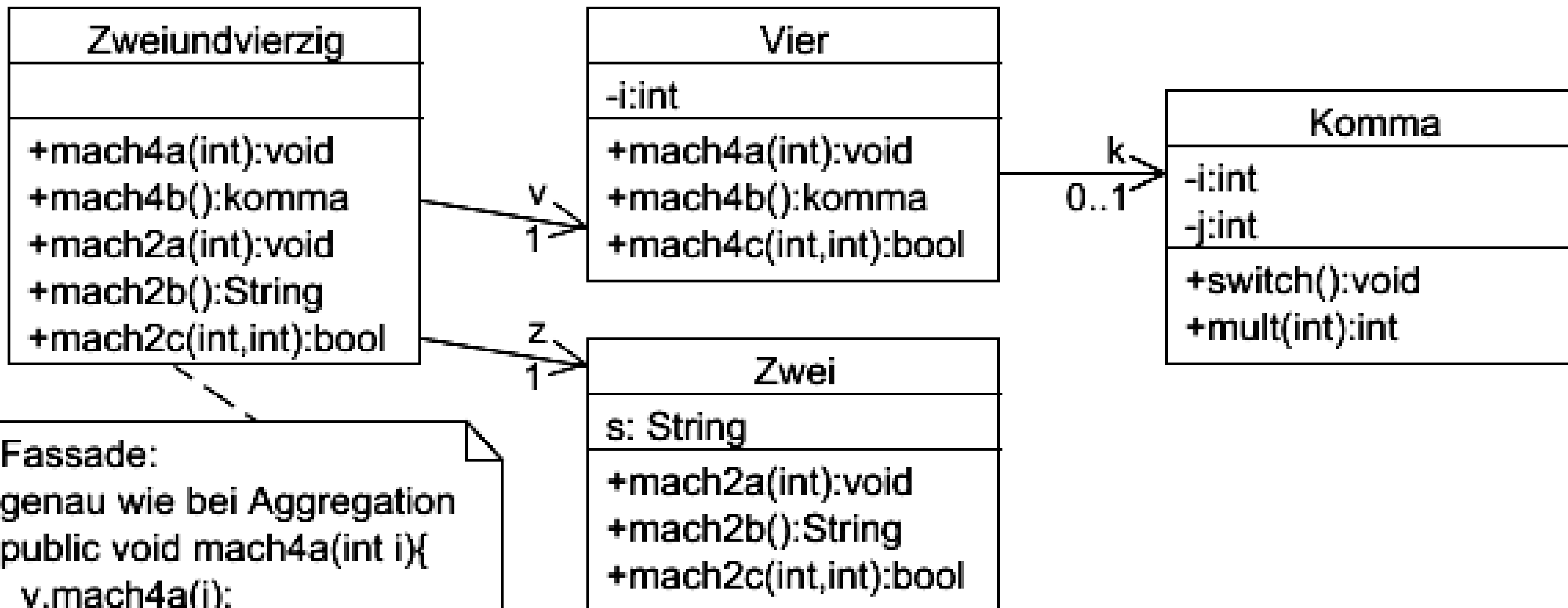
Adapter - Lösung

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES




```
public class Adapter implements B{
  private C c = null;
  ...
  public Adapter(){ this.c = new C();}
  ...
  @Override
  public ... machWasTolles(){
    return this.c.kannWasWasAuchBKoennenSoll();
  }
}
```

Fassade nach außen



- Generell sollen Klassen eng zusammen-hängend sein, z. B. Methoden können nicht auf mehrere Klassen verteilt werden
- anderen Nutzungen möchte man nur eine einfache externe Sicht bieten, deshalb liefern zusammenhängende Klassen häufiger eine Fassadenklasse („davorgeklatscht“) nach außen

Einsatzmöglichkeiten von Sichtbarkeiten

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- Standard-OO-Programmierung: Exemplarvariablen private [oder protected], Exemplarmethoden public (analog für Klassenvariablen und –methoden)

- In Spezialfällen können Sichtbarkeiten geändert werden, Beispiel:
  - Im gesamten System gibt es ein Objekt, mit dem die Verbindung zu anderen Systemen aufgebaut wird
  - Wird das Objekt das erste Mal benötigt, wird es erzeugt, bei weiteren Anfragen werden Referenzen auf dieses identische Objekt zurück gegeben
    - Objekt muss in Klassenvariable gespeichert werden
    - Nutzungen dürfen keine Konstruktoren aufrufen, da es sonst verschiedene Objekte gibt (Konstruktoren werden private)
    - Zugriff auf das Objekt über Klassenmethoden

Singleton (1/3)

```
public class Singleton {
  private int x = 0;
  private int y = 0;
  private static Singleton pkt = null; //für einziges
                                       //Exemplar

  private Singleton(int x, int y){
    this.x = x;
    this.y = y;
  }

  public static Singleton getPunkt(){
    if (Singleton.pkt == null) { // ein einziges Mal erzeugen
      Singleton.pkt = new Singleton(6, 42);
    }
    return Singleton.pkt;
  }
```

Singleton (2/3)

```
  @Override
  public Singleton clone(){
    //echtes Kopieren verhindern
    return this;
  }

  public void ausgeben(){
    System.out.print("[" + this.x + "," + this.y + "]");
  }

  public void verschieben(int dx, int dy){
    this.x += dx;
    this.y += dy;
  }
}
```

Singleton (3/3)

```
public class Main {
  public static void main(String[] s){
    Singleton p1 = Singleton.getPunkt();
    Singleton p2 = Singleton.getPunkt();
    // Singleton sing = new Singleton();
    // error: constructor not visible
    p1.ausgeben();
    p2.ausgeben();
    if(p1 == p2) {
        System.out.println("\n identisch");
    }
    p1.verschieben(3, 5);
    p1.ausgeben();
    p2.ausgeben();
    Singleton p3 = p1.clone();
    if(p2 == p3) {
        System.out.println("\n identisch");
    }
  }
}
```

```
[6,42][6,42]
identisch
[9,47][9,47]
identisch
```

Decorator (1/9)

Video

- gegeben ist eine Klasse mit Methoden; diese gegebenen Methoden sollen ergänzt/verändert werden
- Beispiel: Protokolliere was wird wann ausgeführt (Logging)
- Ansatz: gegeben einfache Klasse

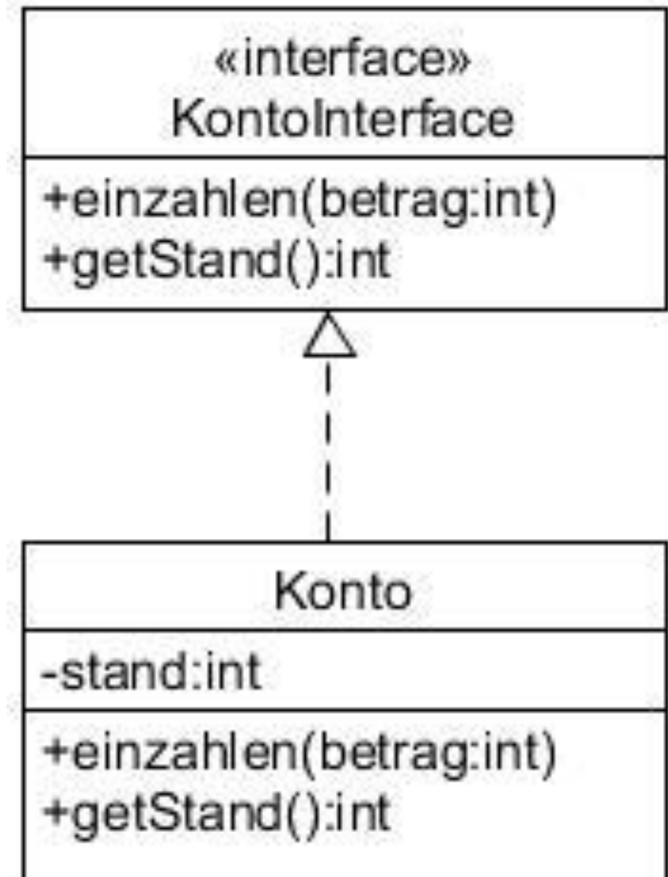
Decorator (2/9)

```
public class Konto {
    private int stand;

    public void einzahlen(int betrag) {
        this.stand += betrag;
    }

    public int getStand() {
        return this.stand;
    }

    public String toString() {
        return "Konto{" + "stand=" + this.stand + '}';
    }
}
```

Decorator (3/9)

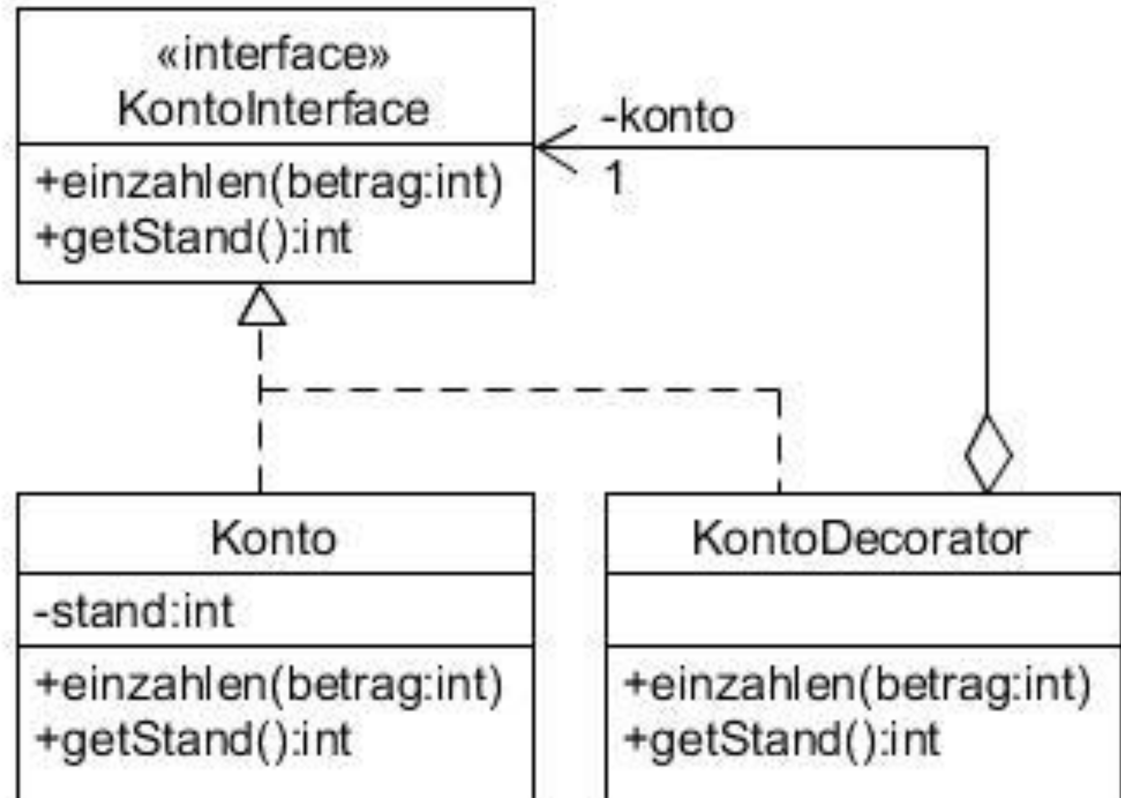
- ergänze Interface

```
public interface KontoInterface {

    void einzahlen(int betrag);

    int getStand();

}


public class Konto
    implements KontoInterface { …
```



Decorator (4/9)

- ergänze neue Klasse (Decorator) die das Interface realisiert und ein Objekt der Klasse als Exemplarvariable hält
  - Idee: delegiere Aufrufe an diese Exemplarvariable und ergänze drum herum neue Funktionalität
  - flexibler: Exemplarvariable nutzt Interface-Typ



Decorator (5/9)

```java
public class KontoDecorator implements KontoInterface {

    private KontoInterface konto;

    public KontoDecorator(KontoInterface konto){
        this.konto = konto;
    }

    @Override
    public void einzahlen(int betrag) {
        System.out.println("vor einzahlen");
        this.konto.einzahlen(betrag);
        System.out.println("nach einzahlen");
    }

    @Override
    public int getStand() {
        System.out.println("vor getStand");
        int ergebnis = this.konto.getStand();
        System.out.println("nach getStand");
        return ergebnis;
    }
}
```

Decorator (6/9)

```
public static void main(String[] args) {
    KontoInterface k = new Konto();
    KontoInterface kd = new KontoDecorator(k);
    kd.einzahlen(42);
    System.out.println("Stand: " + kd.getStand());
    System.out.println("Konto: " + k);
}
```

```
vor einzahlen
nach einzahlen
vor getStand
nach getStand
Stand: 42
Konto:
Konto{stand=42}
```

Decorator (7/9) – etwas mehr Effekt (1/2)


```
public class KontoDecorator2 implements KontoInterface{

    private KontoInterface konto;
    private int schutz; // meine Privatgebuehr

    public KontoDecorator2(KontoInterface konto){
        this.konto = konto;
    }

    @Override
    public void einzahlen(int betrag) {
        System.out.println("vor einzahlen");
        this.schutz += 4;
        this.konto.einzahlen(betrag - 4);
        System.out.println("nach einzahlen");
    }
```

Decorator (8/9) – etwas mehr Effekt (2/2)

```
    @Override
    public int getStand() {
        System.out.println("vor getStand");
        int ergebnis = this.konto.getStand();
        System.out.println("nach getStand");
        return ergebnis + this.schutz;
    }

}
```

Decorator (9/9) – sind verknüpfbar

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

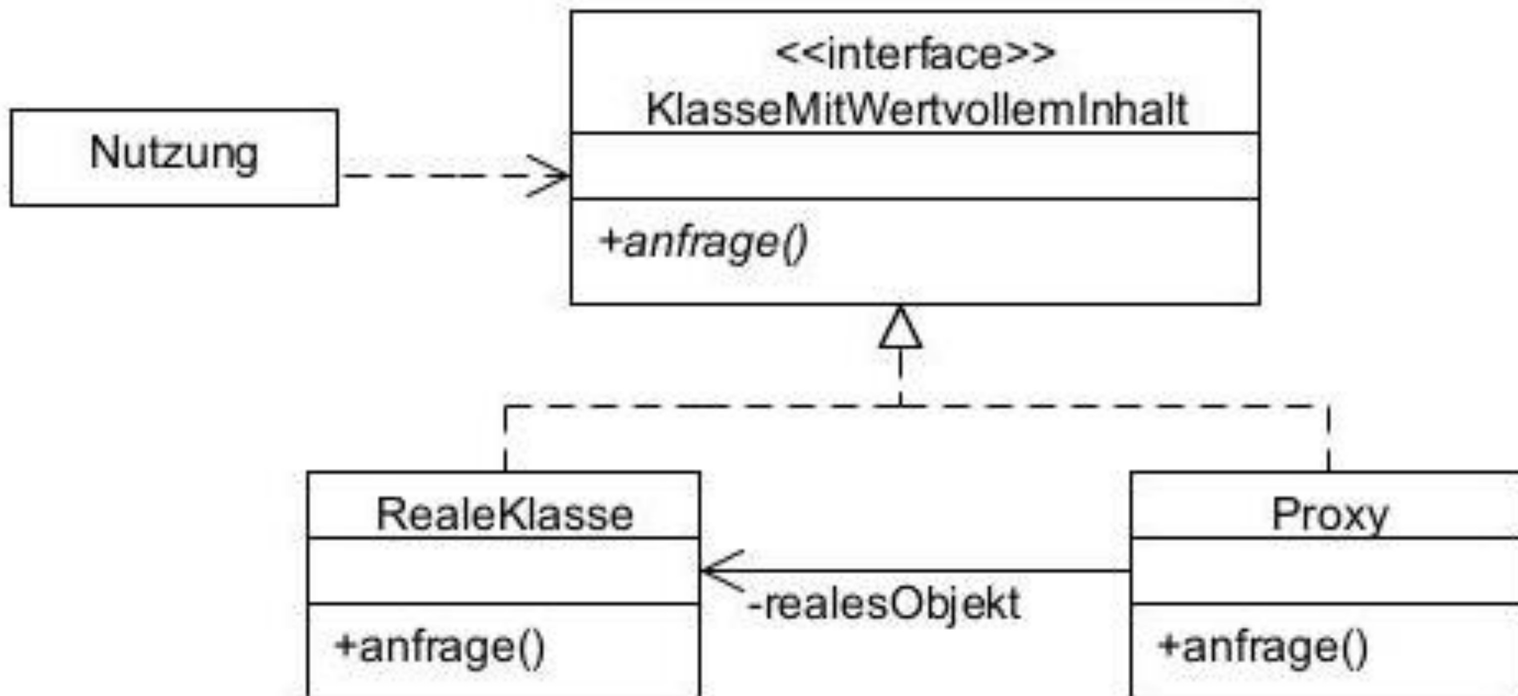
```
public static void main(String[] args) {
    KontoInterface k = new Konto();
    KontoInterface ktmp = new KontoDecorator2(k);
    KontoInterface kd = new KontoDecorator2(ktmp);
    kd.einzahlen(42);
    System.out.println("Stand: " + kd.getStand());
    System.out.println("Konto: " + k);
}
```

```
vor einzahlen
vor einzahlen
nach einzahlen
nach einzahlen
vor getStand
vor getStand
nach getStand
nach getStand
Stand: 42
Konto: Konto{stand=34}
```

Proxy

Video

- Beim Proxy (oder Stellvertreter)-Pattern wird der Zugriff auf eine „wertvolle“ Ressource durch eine vorgeschaltete Klasse gesteuert
- Nutzungen des Proxys nutzen diesen wie die eigentliche Klasse



Proxy – Implementierungsmöglichkeit (1/3)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

```
public interface KlasseMitWertvollemInhalt {
  public int anfrage(String details);
}
```

```
public class RealeKlasse implements
   KlasseMitWertvollemInhalt {

  private Verbindung verbindung;

  public RealeKlasse(String verbindungsdaten){
    this.verbindung = new Verbindung(verbindungsdaten);
  }

  @Override
  public int anfrage(String details) {
    return this.verbindung.befragen(details);
  }
}
```

Proxy – Implementierungsmöglichkeit (2/3)

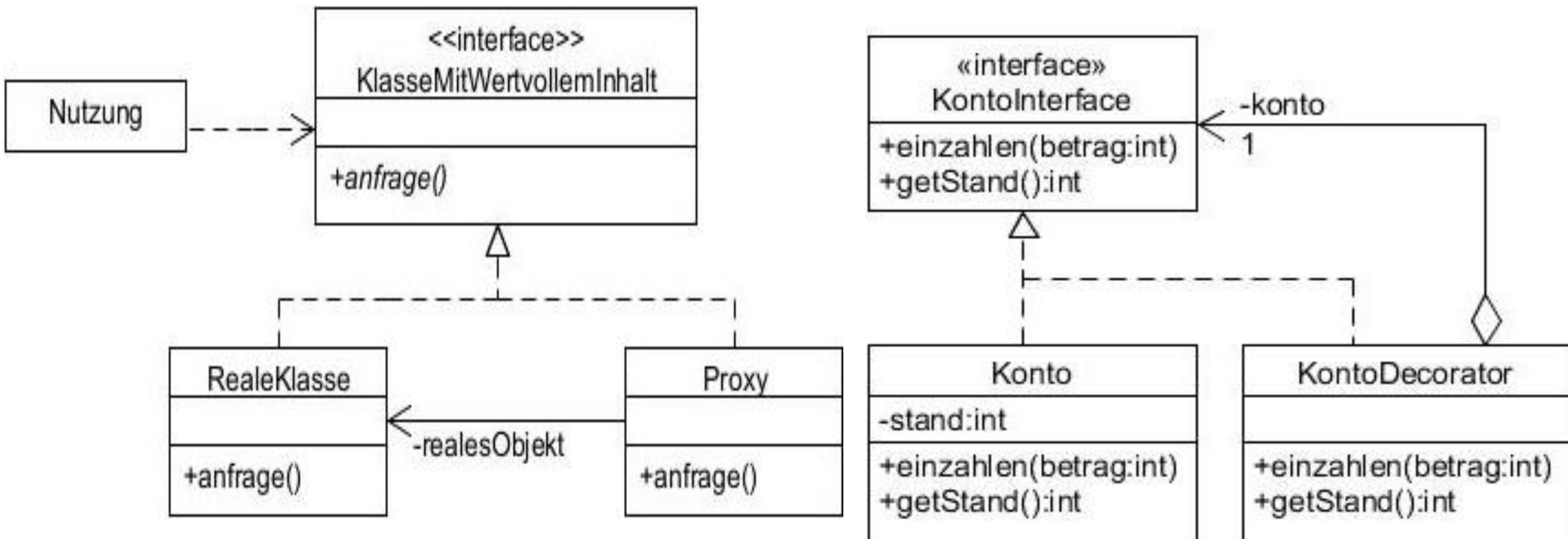
```
public class Proxy implements KlasseMitWertvollemInhalt {

  //hier Variante mit Singleton (gibt Alternativen)
  private static RealeKlasse realesObjekt;

  public Proxy(){
    if(Proxy.realesObjekt == null){
      Proxy.realesObjekt = new RealeKlasse("Spezialinfos");
    }
  }

  public int anfrage(String details) {
    // hier nur Weiterleitung
    // Varianten: Protokollierung, Cache, …
    return Proxy.realesObjekt.anfrage(details);
  }
}
```

Proxy – Implementierungsmöglichkeit (3/3)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

```
public class Nutzend {

  public int proxyNutzen(String anfrage){
    KlasseMitWertvollemInhalt k = new Proxy();
    return k.anfrage(anfrage);
  }

  public static void main(String[] s){
    //etwas sinnlos, zu Testzwecken
    Nutzend n = new Nutzend();
    System.out.println(n.proxyNutzen("gib41"));
  }
}
```

Proxy, Decorator – Verwandt, aber anderer Einsatz (1/2)



Proxy, Decorator – Verwandt, aber anderer Einsatz (2/2)

- gemeinsam: Verhalten einer existierenden Klasse wird verändert
- beide sind zur Erweiterung der Funktionalität nutzbar

aber:

- Proxy-Schwerpunkt liegt auf der Kontrolle des Objektzugriffs
- Objekt wird oft im Proxy erzeugt
- Verbindung wird zu Compile-Zeit bereits festgelegt

- Decorator fügt Funktionalität zu existierendem Objekt hinzu
- Objekt wird injiziert (Übergabe Konstruktor oder mit set)
- Verbindung wird erst zur Laufzeit hergestellt

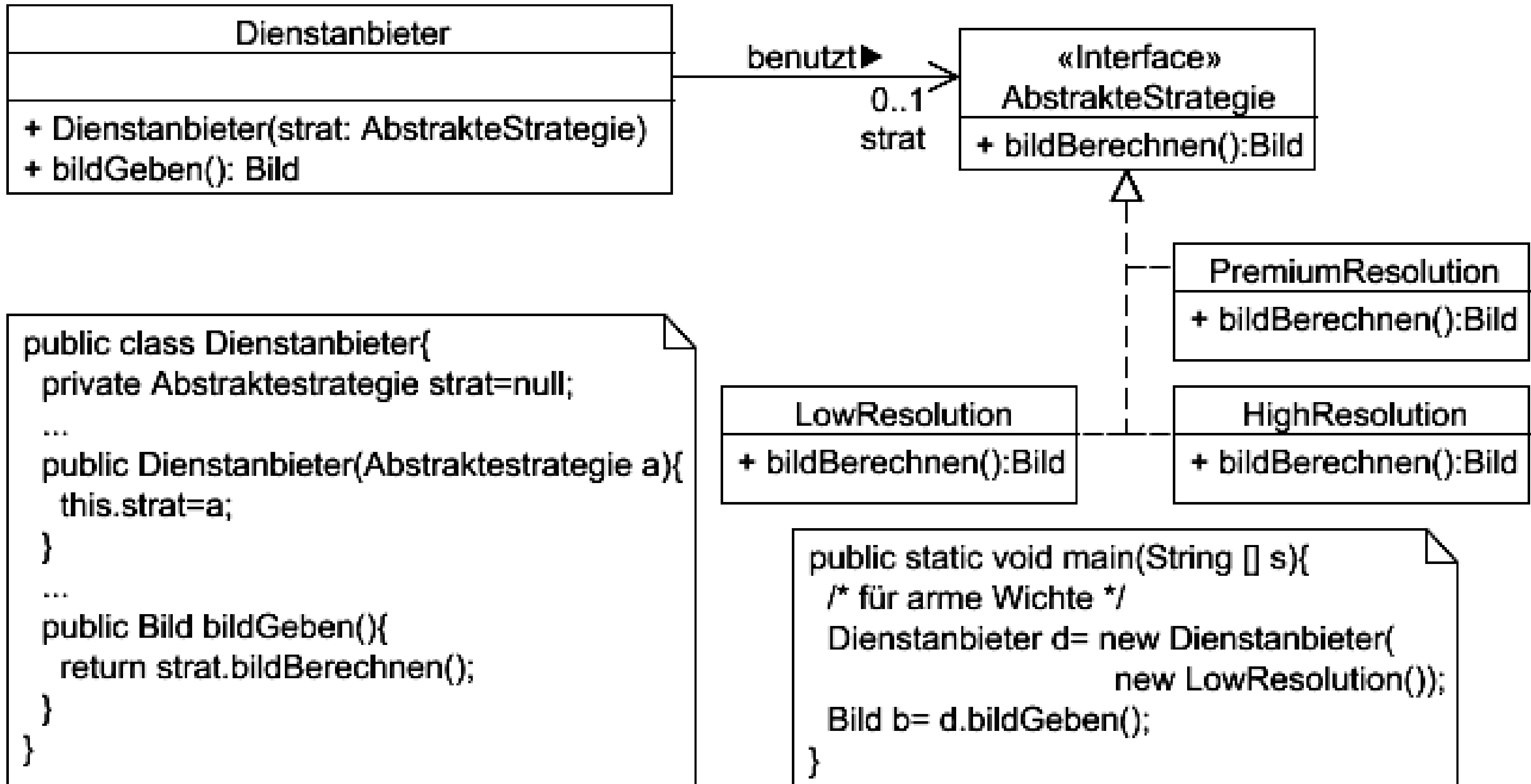
Strategy - Problem

- Für eine Methode gibt es verschiedene Möglichkeiten sie zu implementieren
- Die Wahl der Implementierungsart soll leicht verändert werden können

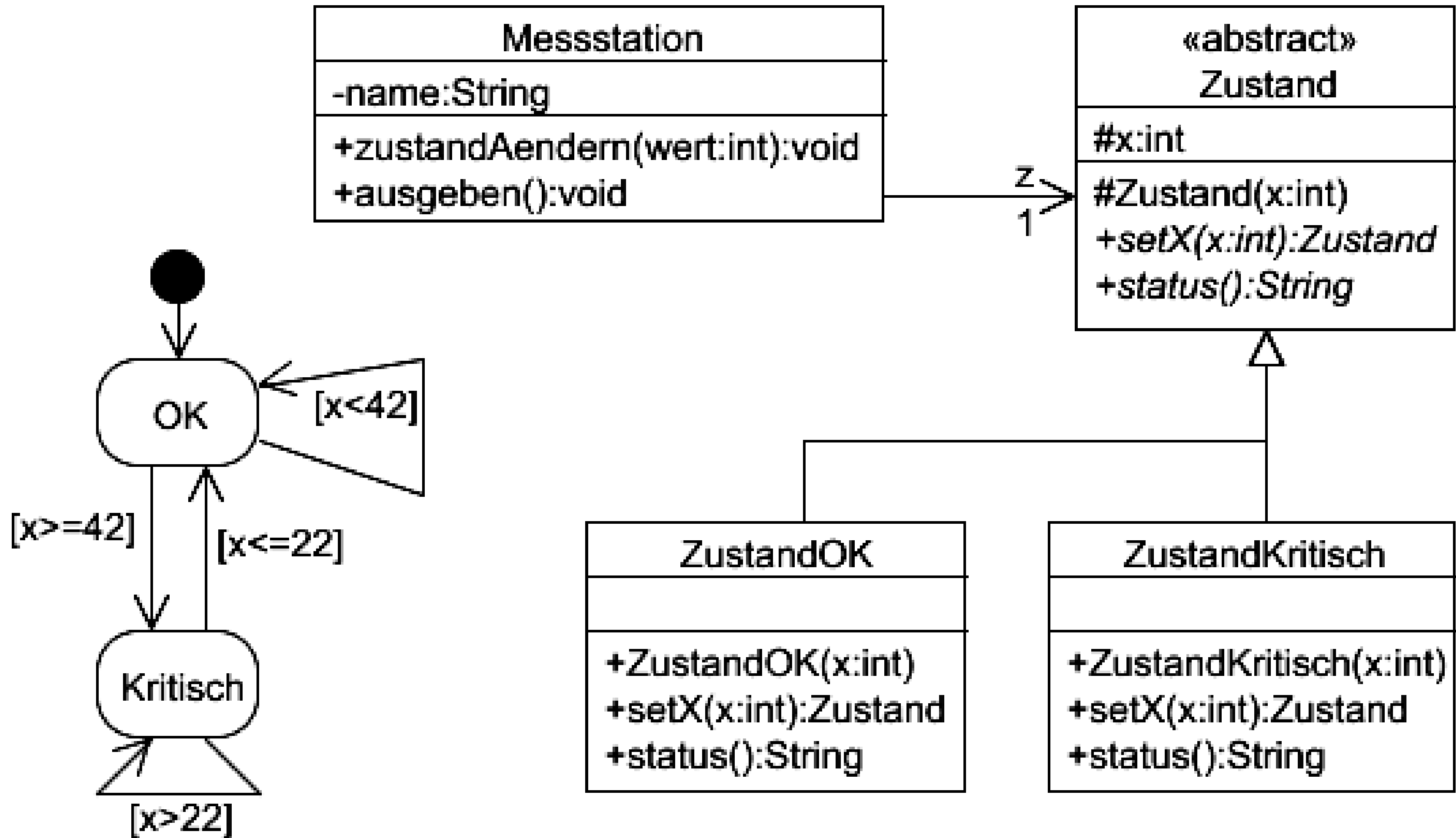
Einsatzszenarien

- Prototypische Implementierung soll später leicht ausgetauscht werden können
- Wahl der effizientesten Methode hängt von weiteren Randbedingungen ab (z. B. suchen / sortieren)
- Ausführungsart der Methode soll zur Laufzeit geändert werden können (z. B. nutzende Person zahlt für einen Dienst und bekommt statt Werbe- Detailinformationen)

Strategy - Lösungsbeispiel



State-Pattern (eine eigene Variante)



State-Pattern – Implementierungsauszug (1/3)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

```
public abstract class Zustand {

  protected int x;

  public abstract Zustand setX(int x);
  public abstract String status();
  protected Zustand(int x){
      this.x = x;
   }
}
```

- Jede zustandsverändernde Methode (hier setX) führt Änderungen aus und gibt Folgezustand zurück
- Zustand könnte auch veränderbarer Parameter sein

State-Pattern – Implementierungsauszug (2/3)

```java
public class ZustandOK extends Zustand{

   public ZustandOK(int x) {
        super(x);
   }

   @Override
   public Zustand setX(int x) {
        super.x = x;
        if(x >= 42) {
          return new ZustandKritisch(x);
        }
        return this;
   }

   @Override
   public String status() {return "alles ok";}
}
```

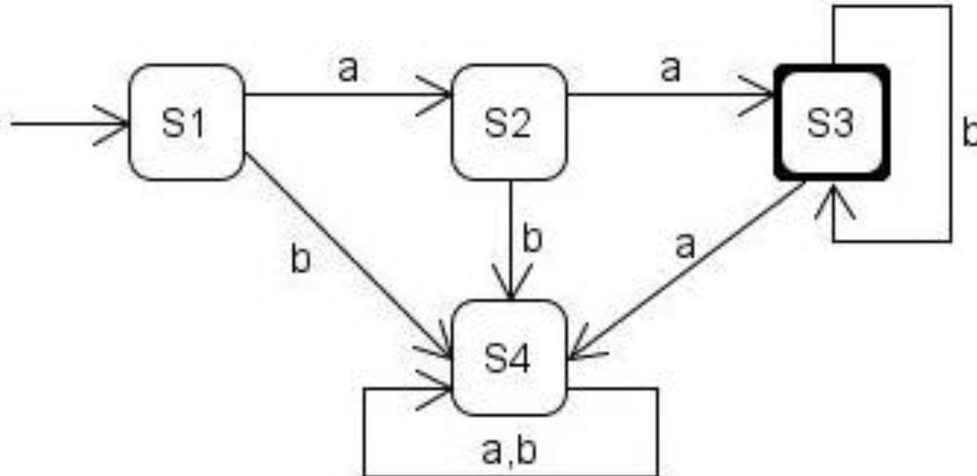
State-Pattern – Implementierungsauszug (3/3)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

```
public class Messstation {
  private String standort = "City";
  private Zustand z = new ZustandOK(0);

  public void zustandAendern(int wert){
    this.z = this.z.setX(wert);
  }

  public void ausgeben(){
    System.out.println(this.standort
                    + " Zustand: " + this.z.status());
  }
}
```

Umsetzung klassischer endlicher Automaten

- Automat mit Startzustand S1, Menge von Endzuständen {S3} und Eingabezeichen a, b; akzeptiert Sprache aab*



Zustand	Zeichen	Folge-zustand
S1	a	S2
S1	b	S4
S2	a	S3
S2	b	S4
S3	a	S4
S3	b	S3
S4	a	S4
S4	b	S4

```
public class S3 implements Zustand {

  @Override
  public Zustand a() {
    return new S4();
  }

  @Override
  public Zustand b() {
    return this;
  }
```

Command-Pattern

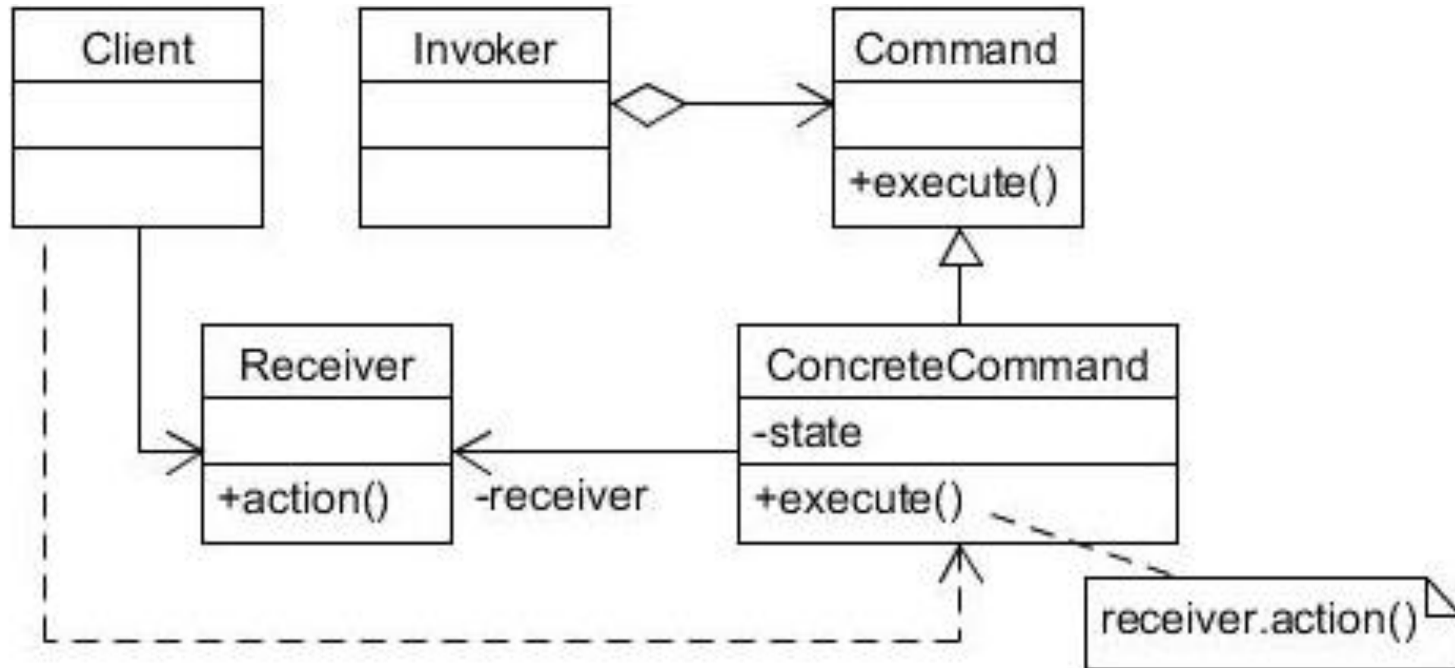
Video

- Problem: unterschiedliche Aktionen werden zentral ausgeführt und verwaltet
- Ansatz: Stecke detaillierte Ausführung in ein (Command-) Objekt; diese haben gemeinsames Interface
- Command-Objekte kennen Details der Ausführung
- Steuerung dann einfach änder- und erweiterbar

- Beispiel: Kleiner Taschenrechner mit + und – und einem Zwischenspeicher für einen Wert, der dann aufaddiert oder subtrahiert werden kann

M	**M+**	**M-**	
7	8	9	+
4	5	6	-
1	2	3	
0			

Bild aus der Literatur



- Command ist abstrakt, zeigt Ausführungsoperation
- ConcreteCommand ist Umsetzung für Receiver
- Receiver führt Operation aus
- Invoker kennt Commands, startet Ausführung
- Client erzeugt ConcreteCommand und setzt Receiver

Beispiel 1/13 : Rechner 1/2

```
package business;

public class Rechner {

   private int anzeige;
   private int speicher;

   public int getAnzeige() {
       return this.anzeige;
   }

   public void setAnzeige(int anzeige) {
       this.anzeige = anzeige;
   }

   public int getSpeicher() {
       return this.speicher;
   }

   public void setSpeicher(int speicher) {
       this.speicher = speicher;
   }
```

Beispiel 2/13 : Rechner 2/2

```
    public void addieren(int wert) {
        this.anzeige += wert;
    }

    public void subtrahieren(int wert) {
        this.anzeige -= wert;
    }

    public void speichern(){
        this.speicher = this.anzeige;
    }

    public void speicherAddieren(){
        this.anzeige += this.speicher;
    }

    public void speicherSubtrahieren(){
        this.anzeige -= this.speicher;
    }

    @Override
    public String toString(){
      return "Speicher: "+ this.speicher +"  Wert: "
                         + this.anzeige;
    }
}
```

Beispiel 3/13 : Klassischer Dialog 1/2

```
package io;
import business.Rechner;

public class Dialog {
  private Rechner rechner = new Rechner();

  public void dialog() {
    EinUndAusgabe ea = new EinUndAusgabe();
    int eingabe = -1;
    while (eingabe != 0) {
      System.out.println(  "(0) Programm beenden\n"
          + "(1) addieren\n" + "(2) subtrahieren\n"
          + "(3) Anzeige in Speicher\n"
          + "(4) Speicher addieren\n"
          + "(5) Speicher subtrahieren");
      eingabe = ea.leseInteger();
      switch (eingabe) {
        case 1: {
          System.out.print("Wert eingeben: ");
          this.rechner.addieren(ea.leseInteger());
          break;
        }
```

Beispiel 4/13 : Klassischer Dialog 2/2

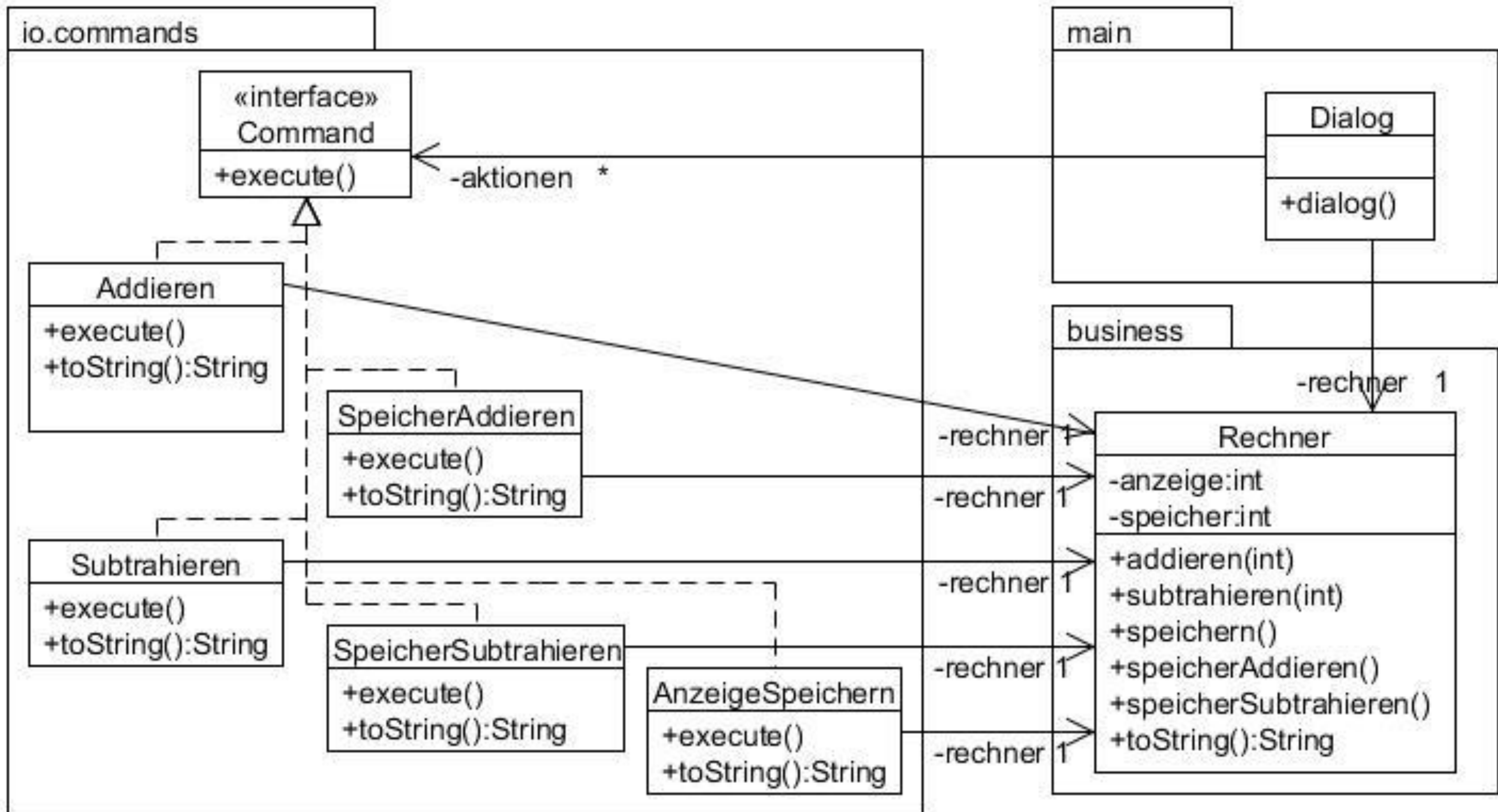
```
      case 2: {
        System.out.print("Wert eingeben: ");
        this.rechner.subtrahieren(ea.leseInteger());
        break;
      }
      case 3: {
        this.rechner.speichern();
        break;
      }
      case 4: {
        this.rechner.speicherAddieren();
        break;
      }
      case 5: {
        this.rechner.speicherSubtrahieren();
        break;
      }
    }
    System.out.println(this.rechner);
  }
}
```

Beispiel 5/13 : Funktioniert immerhin

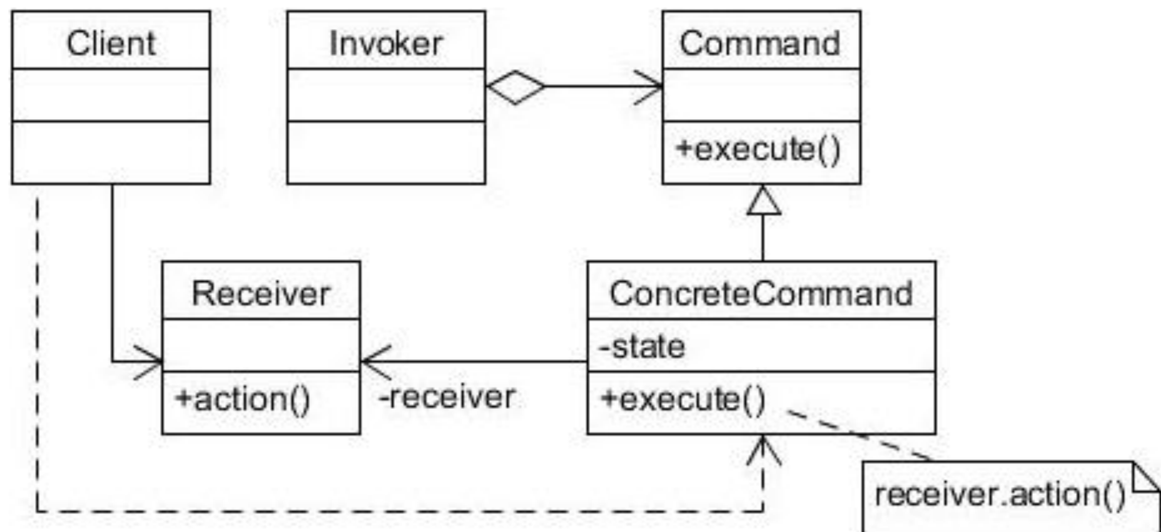
HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

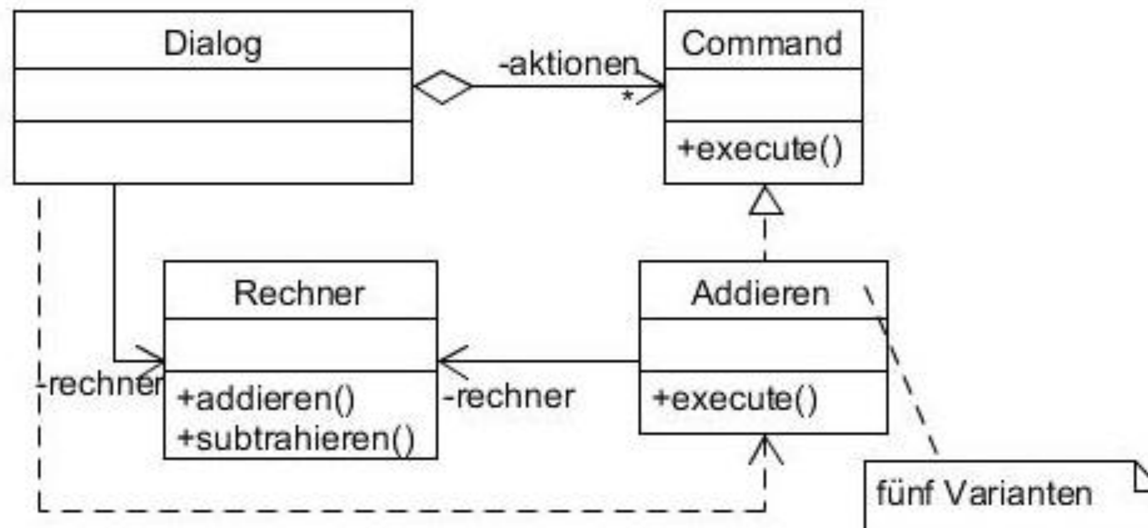
```
(0) Programm beenden
(1) addieren
(2) subtrahieren
(3) Anzeige in Speicher
(4) Speicher addieren
(5) Speicher subtrahieren
1
Wert eingeben: 43
Speicher: 0  Wert: 43
(2) subtrahieren
2
Wert eingeben: 1
Speicher: 0  Wert: 42
(3) Anzeige in Speicher
3
Speicher: 42  Wert: 42
(4) Speicher addieren
4
Speicher: 42  Wert: 84
(5) Speicher subtrahieren
5
Speicher: 42  Wert: 42
(0) Programm beenden
0
Speicher: 42  Wert: 42
```

Beispiel 6/13 : Ansatz: Steuerungsklassen



Beispiel 7/13 : Pattern-Nutzung





Beispiel 8/13 : Umsetzung 1/3

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

```
package io.commands;
public interface Command {
  public void execute();
}
```

```
package io.commands;

import main.EinUndAusgabe;
import business.Rechner;

public class Addieren implements Command {
  private Rechner rechner;

  public Addieren(Rechner rechner){
    this.rechner = rechner;
  }

  @Override
  public void execute() {
    System.out.print("Wert eingeben: ");
    this.rechner.addieren(new EinUndAusgabe().leseInt());
  }

  @Override
  public String toString(){return "addieren";}
}
```

typischerweise werden Zusatzinformationen benötigt

eigentliche Ausführung

Beispiel 9/13 : Umsetzung 2/3 (Varianten -> Praktikum)

```
package main;

import java.util.HashMap;
import java.util.Map;
import business.Rechner;

public class Dialog {

  private Rechner rechner = new Rechner();
  private Map<Integer,Command> aktionen = new HashMap<>();

  public Dialog(){
    this.aktionen.put(1, new Addieren(this.rechner));
    this.aktionen.put(2, new Subtrahieren(this.rechner));
    this.aktionen.put(3, new AnzeigeSpeichern(this.rechner));
    this.aktionen.put(4, new SpeicherAddieren(this.rechner));
    this.aktionen.put(5
                  , new SpeicherSubtrahieren(this.rechner));
  }
```

Beispiel 10/13 : Umsetzung 3/3

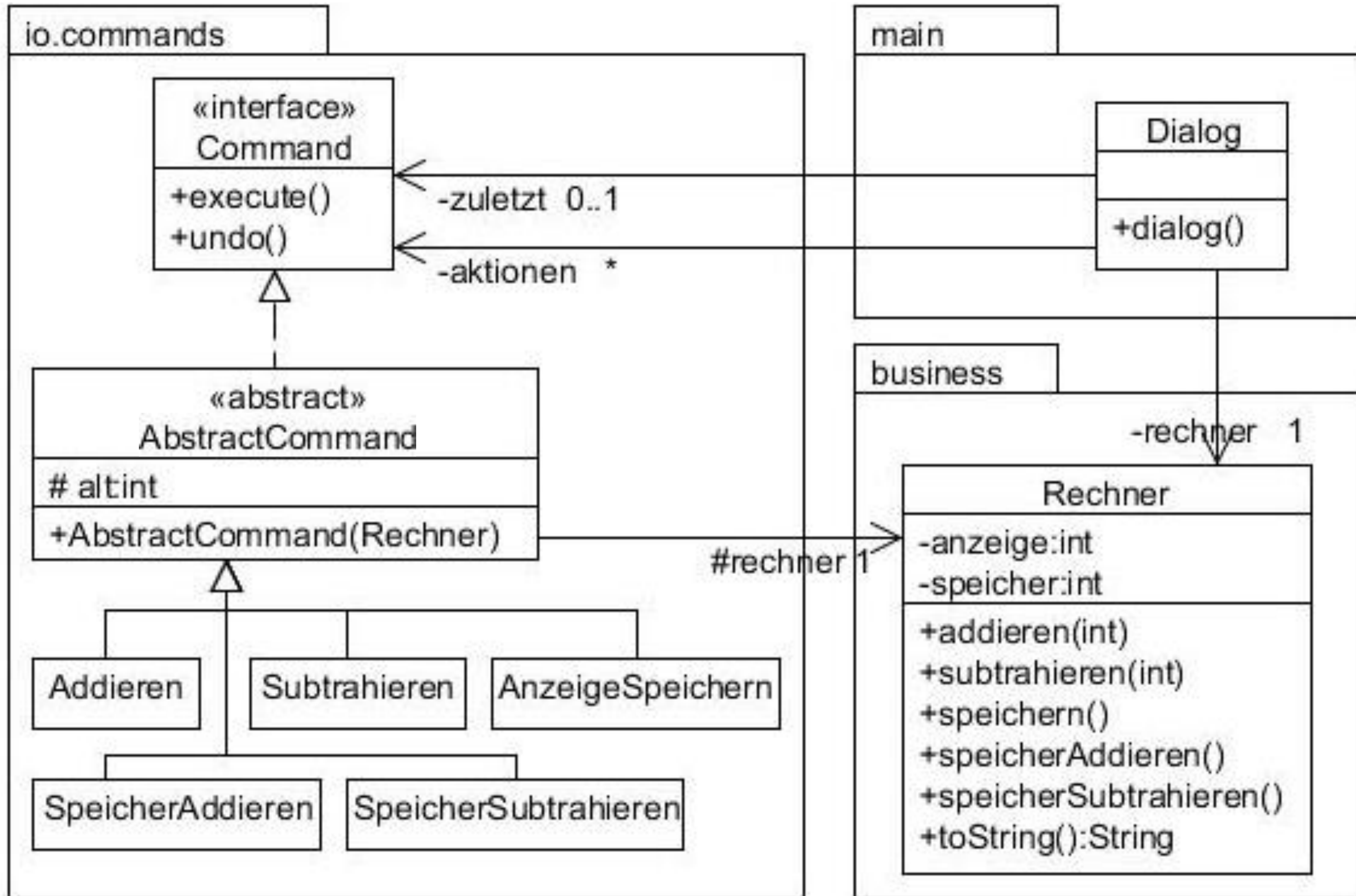
```
public void dialog() {
   EinUndAusgabe ea = new EinUndAusgabe();
   int eingabe = -1;
   while (eingabe != 0) {
     System.out.println("(0) Programm beenden");
     for(int tmp:this.aktionen.keySet()){
       System.out.println("(" + tmp + ") "
                                + this.aktionen.get(tmp));
     }
     eingabe = ea.leseInteger();
     Command com = this.aktionen.get(eingabe);
     if(com != null){
       com.execute();
     }
     System.out.println(this.rechner);
   }
}
```

Beispiel 11/13 : Undo

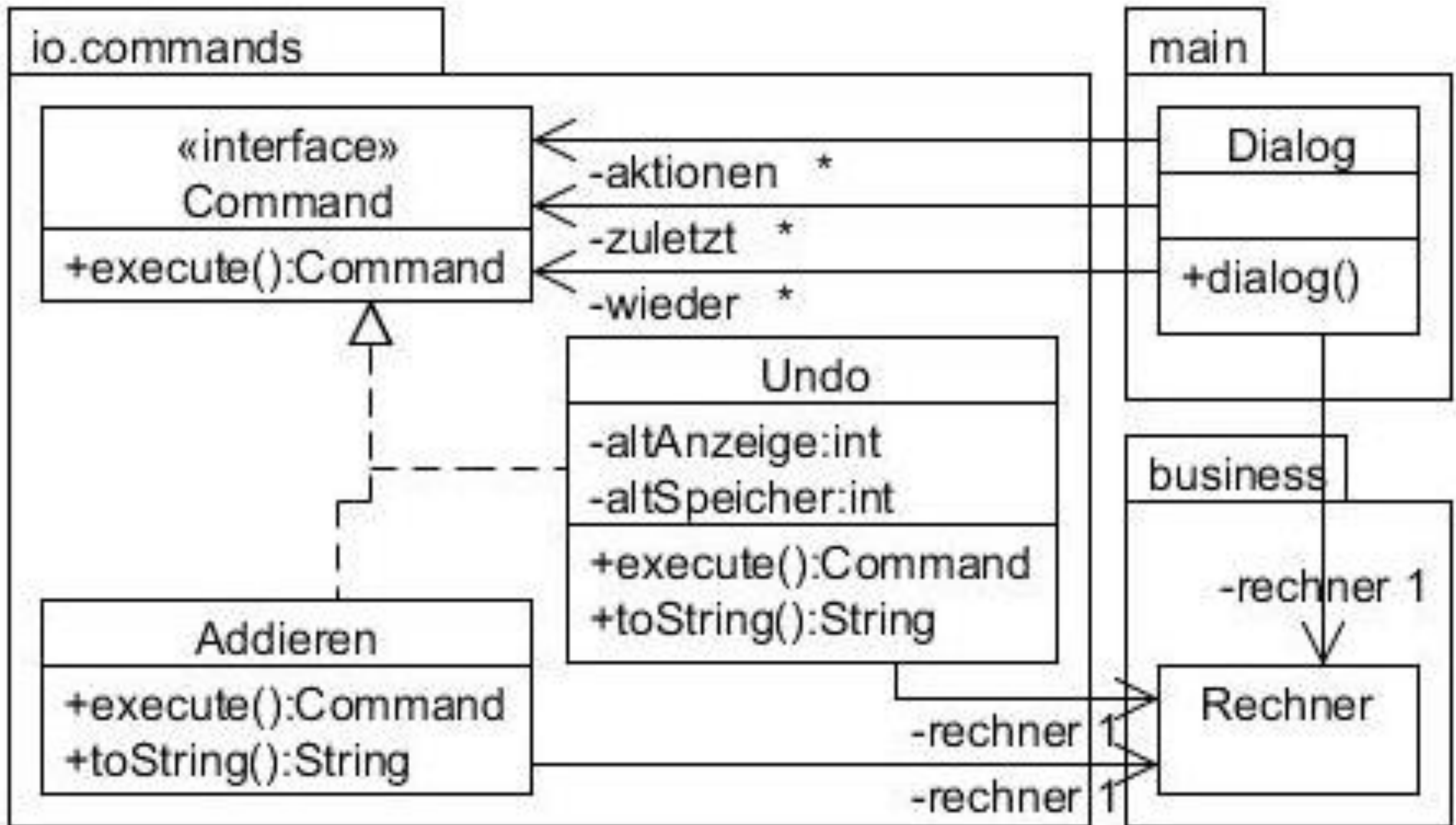
- Command-Pattern eignet sich sehr gut, Aktionen wieder rückgängig zu machen
- es müssen alle Änderungen der Aktion bekannt und reversibel sein

- gibt verschiedene Varianten
  - Ansatz 1: jedes Command-Objekt hat undo-Methode und wird gespeichert [nächste Folien]
  - Ansatz 2: es gibt eigenes Undo-Command-Objekt als Ergebnis von execute()
  - Ansatz 3: Undo- und Command-Objekte haben keine gemeinsame Klasse / Interface
  - …

Beispiel 12/13 : Variante Undo-Methode



Beispiel 13/13 : Variante Undo-Objekte (Skizze)



Fazit Command-Pattern

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- generell oft bei Steuerungen einsetzbar
- oft gut für Undo- und Redo geeignet

- meist individuelle Varianten des Patterns sinnvoll
- (in UML-Diagrammen oft zusätzliche Klasse, die auf Command zugreifen kann)

- Command-Klassen müssen einfach an benötigte Informationen kommen können; wird dies kompliziert, ist der Pattern-Einsatz nicht sinnvoll

Visitor Pattern (1/5) - Idee

Video

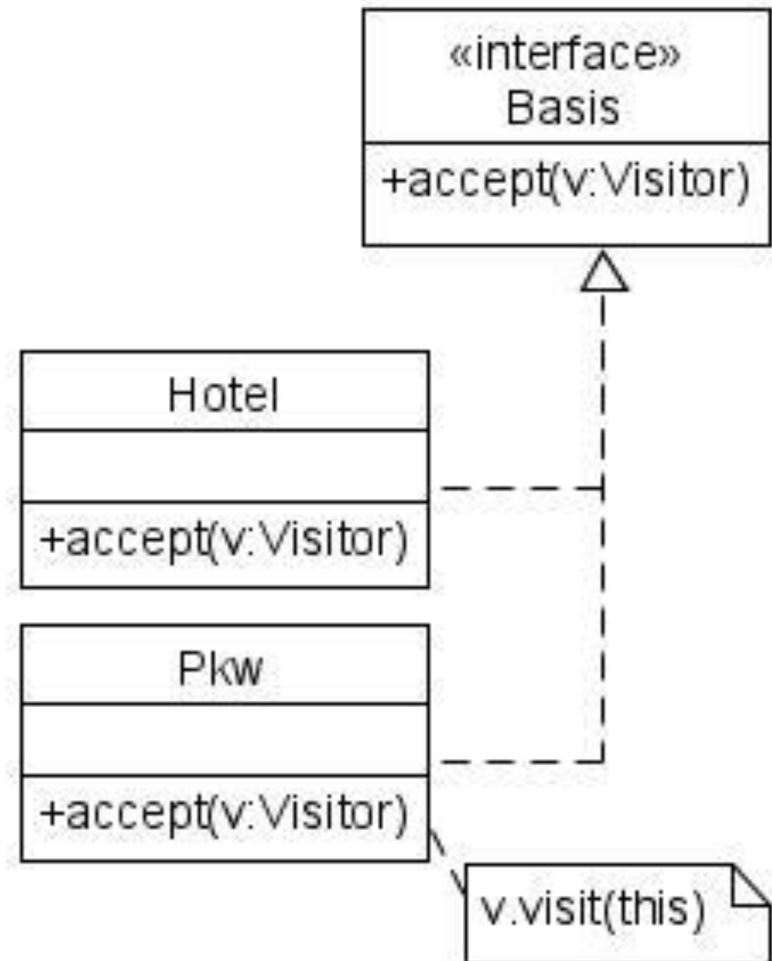
- Es gibt eine zentrale Aufgabe zur Verarbeitung mehrerer Objekte unterschiedlicher Klassen
- Diese Klassen anzupassen ist aufwändig, Gefahr von Copy & Paste
- Verarbeitung soll an einer Stelle passieren, um Synergien zu nutzen und leicht auf Änderungen reagieren zu können

- Beispiel: In einer Reiseverwaltung werden Reisen aus unterschiedlichen Bausteinen, wie Hotel- und Mietwagen-Reservierungen zusammengestellt, für alle Fach-Entitäten soll es Umwandlungsmöglichkeiten nach XML und JSON geben

Visitor Pattern (2/5) - Ansatz

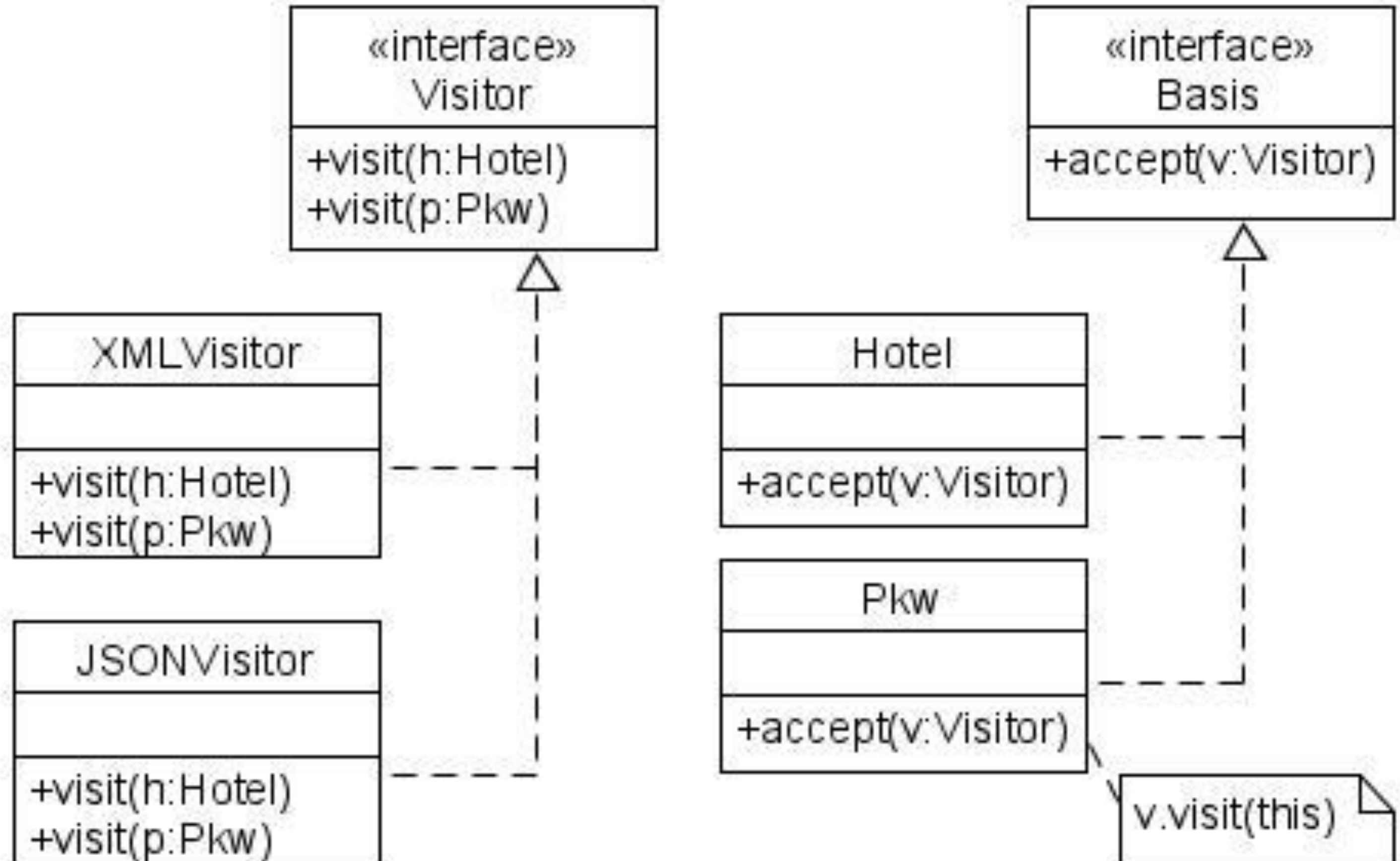
- Neues Interface für Zugriff eines Visitors
- In der Methode wird visit()-Methode des Visitors mit Parameter this aufgerufen
- Visitor kann damit auf besuchtes Objekt zugreifen

- Hinweis: Rückgabeparameter weggelassen, sind aufgabenindividuell zu definieren (cast bei Object notwendig)



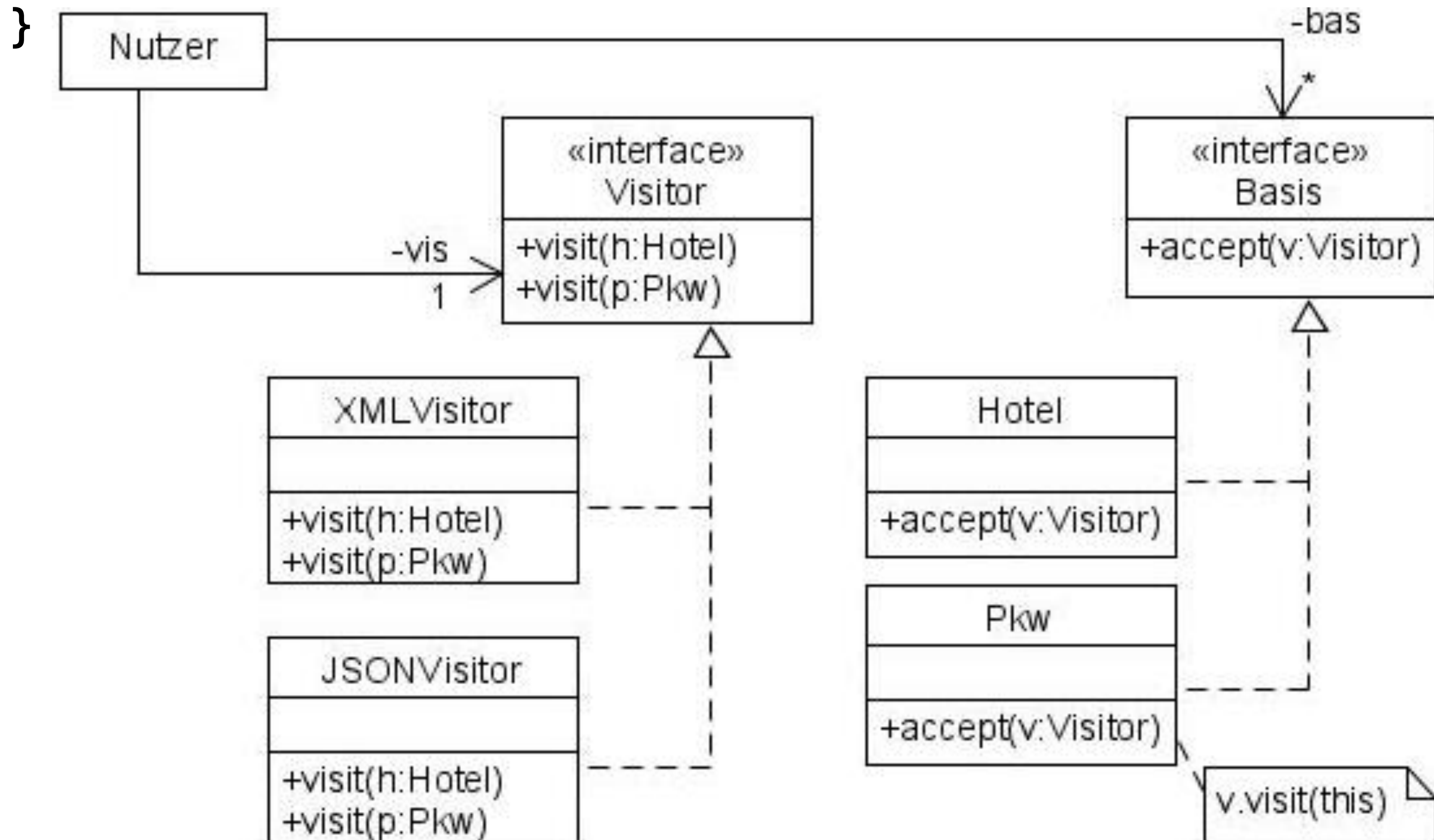
Visitor Pattern (3/5) - Umsetzung

- Statische Polymorphie, für jede besuchte Klasse eigene Methode (mit eigener Rückgabe)



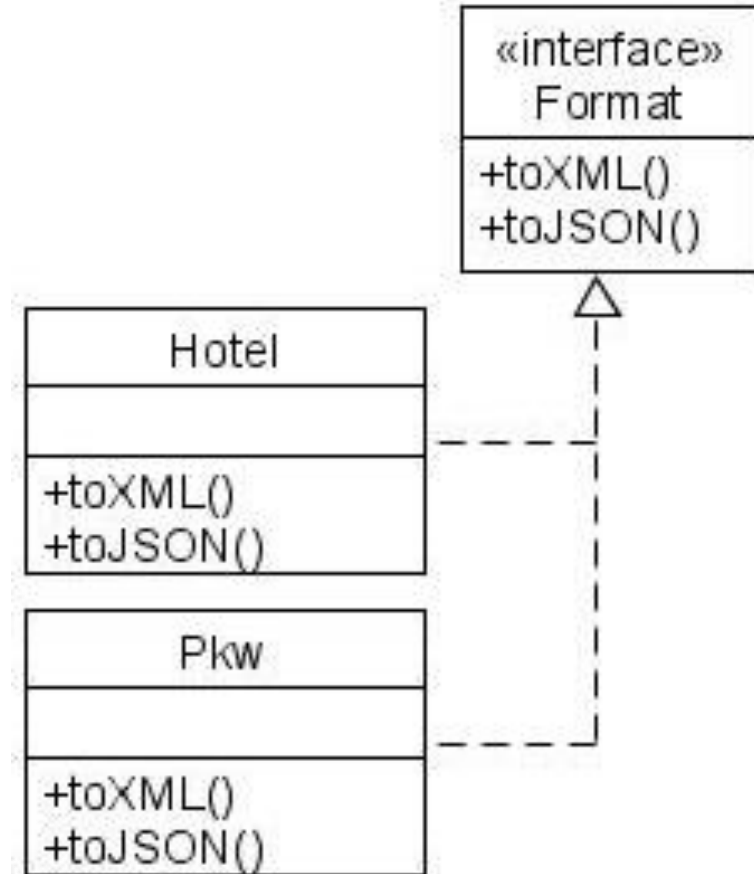
Visitor Pattern (4/5) - Nutzung

```
for(Basis b:bas) {
    System.out.println(b.accept(vis));
}
```



Visitor Pattern (5/5) - Diskussion

- Wesentlicher Vorteil: fachliche Funktionalität zu bestimmten Themen kompakt in konkreten Visitor-Realisierungen gebündelt (-> einfach Wart- und Erweiterbarkeit)
- Alternativ: Direkte Nutzung eines Interfaces, Berechnungen in jeder Klasse notwendig (weniger Klassen, schwerer wartbar)
- Alternative abhängig von Komplexität und Wahrscheinlichkeit einer Änderung wählbar
- Visitor ermöglicht Klassen zu ergänzen ohne deren Code anzufassen



Verantwortlichkeitsmuster – GRASP-Pattern

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

Video

- Expertenmuster
- Creator
- Low coupling
- High cohesion
- Don‘t talk to strangers
- Kunstgebilde
- Command Query Separation

GRASP (General Responsibility Assignment Software Patterns)
nach C. Larman

(Folien basierend auf Prof. T. Gervens)

Muster: Experte

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

Name: Expert(e)

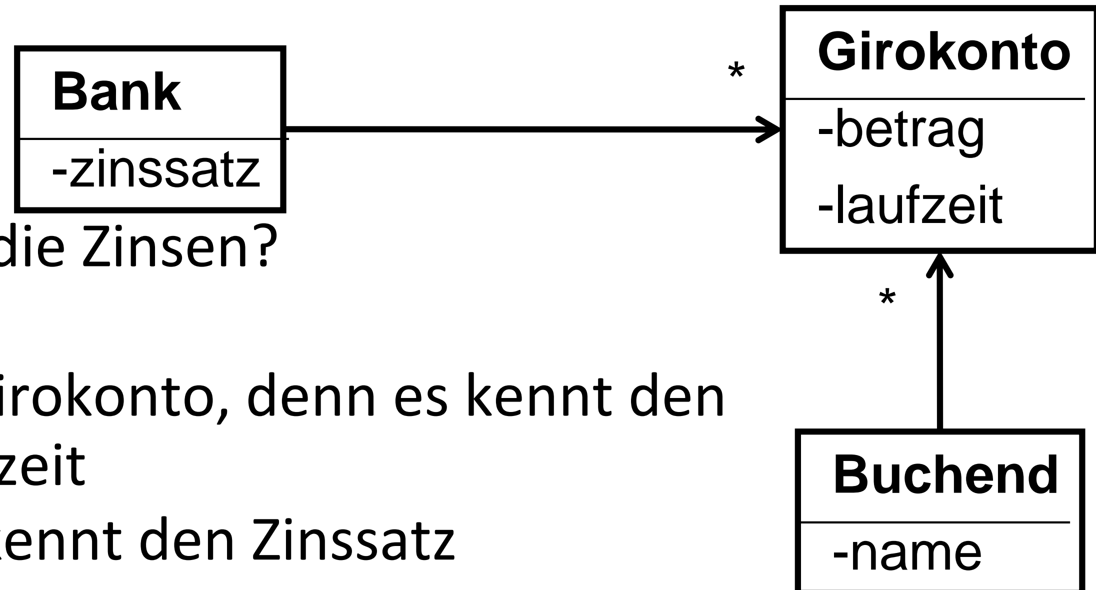
Regel:

Man übertrage eine gegebene Aufgabe bzw. eine Verantwortlichkeit auf diejenige Klasse, die das notwendige Wissen besitzt!

Hintergrund: Man hat:

- einerseits eine Vielzahl von Klassen (aus der Analyse oder vorherigen Gestaltungschritte)
- und andererseits eine Vielzahl zu vergebender Aufgaben und Verantwortlichkeiten

Beispiel: Expert (Fachwissen)



- Frage: Wer berechnet die Zinsen?
- Mögliche Antworten:
  - Objekt der Klasse Girokonto, denn es kennt den Betrag und die Laufzeit
  - die Bank, denn sie kennt den Zinssatz
- vorzuziehen: Die erste Möglichkeit, denn Girokonto besitzt mehr notwendiges Wissen und erhält Ergebnis
- Bemerkung:
  - Fachwissen ist teilweise über mehrere Klassen verteilt, man muss entscheiden, wer die größte Expertise ist
  - Alle Objekte können aktiv werden (anders als bei vielen realen Objekten)

Muster: Creator

Name: Creator

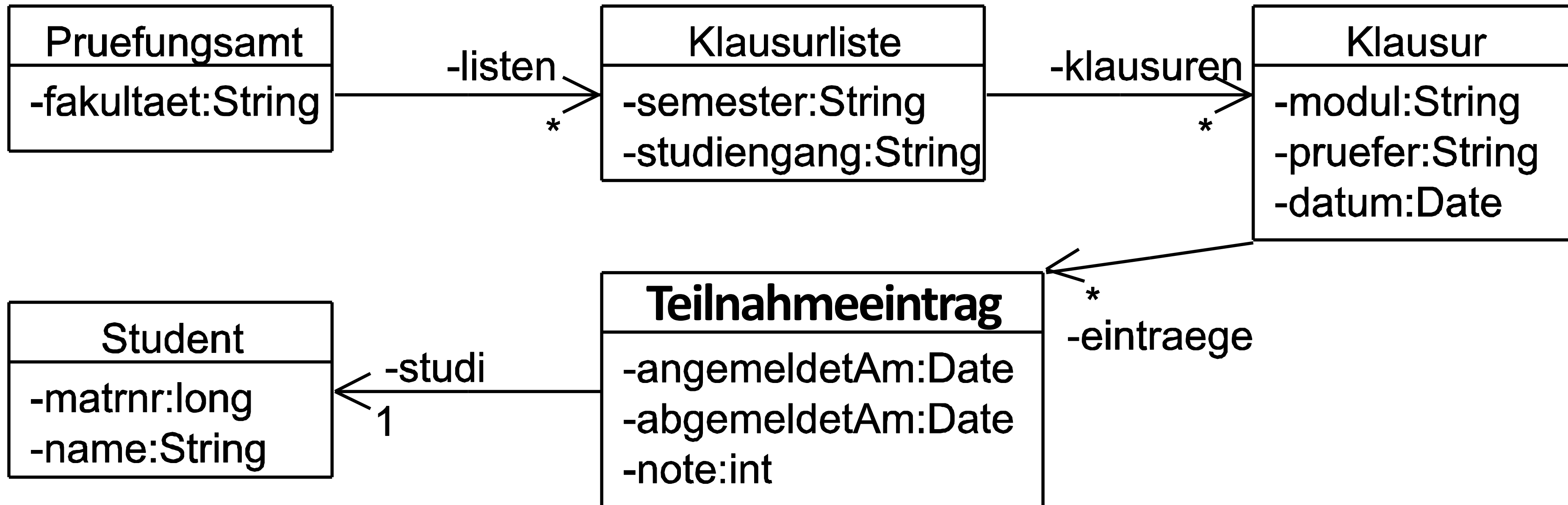
Regel: Gegeben sei eine Klasse A. Die Aufgabe, Objekte dieser Klasse zu erzeugen (Konstruktoraufrufe), soll an eine Klasse übergeben werden, die

- ein Aggregat von A-Objekten ist
- zu A-Objekten in enger Beziehung steht
- das notwendige Wissen (Initialisierungsdaten) besitzt, um A-Objekte zu erzeugen

Hintergrund:

- Die Objektwelt ist dynamisch, ständig entstehen neue Objekte. Objekterzeugung ist daher eine wichtige Aufgabe; die Zuständigkeit dafür sollte sorgfältig vergeben werden

Beispiel: Creator (1/2)

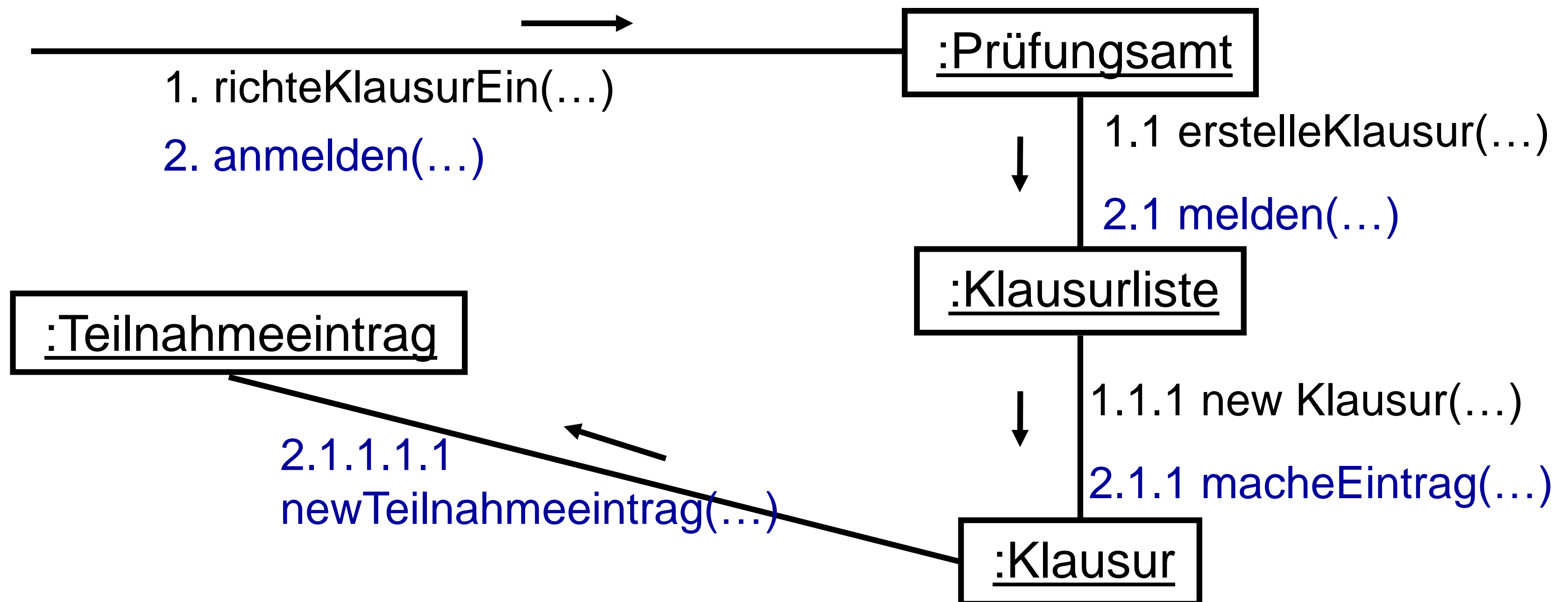


Wer legt wen an? Zum Beispiel:

Klausur : angelegt durch Klausurliste
Teilnahmeeintrag : angelegt durch Klausur

Beispiel: Creator (2/2)

Kommunikationsdiagramm: (ausdrucksstark wie einfaches Sequenzdiagramm)



Muster: Geringe Kopplung

Name: “geringe Verbindung” bzw. “low coupling”

Regel: Aufgaben unter den Klassen so verteilen, dass die Abhängigkeiten unter den Klassen möglichst gering sind!

Hintergrund: Klassen sollten möglichst isoliert sein, denn dadurch werden

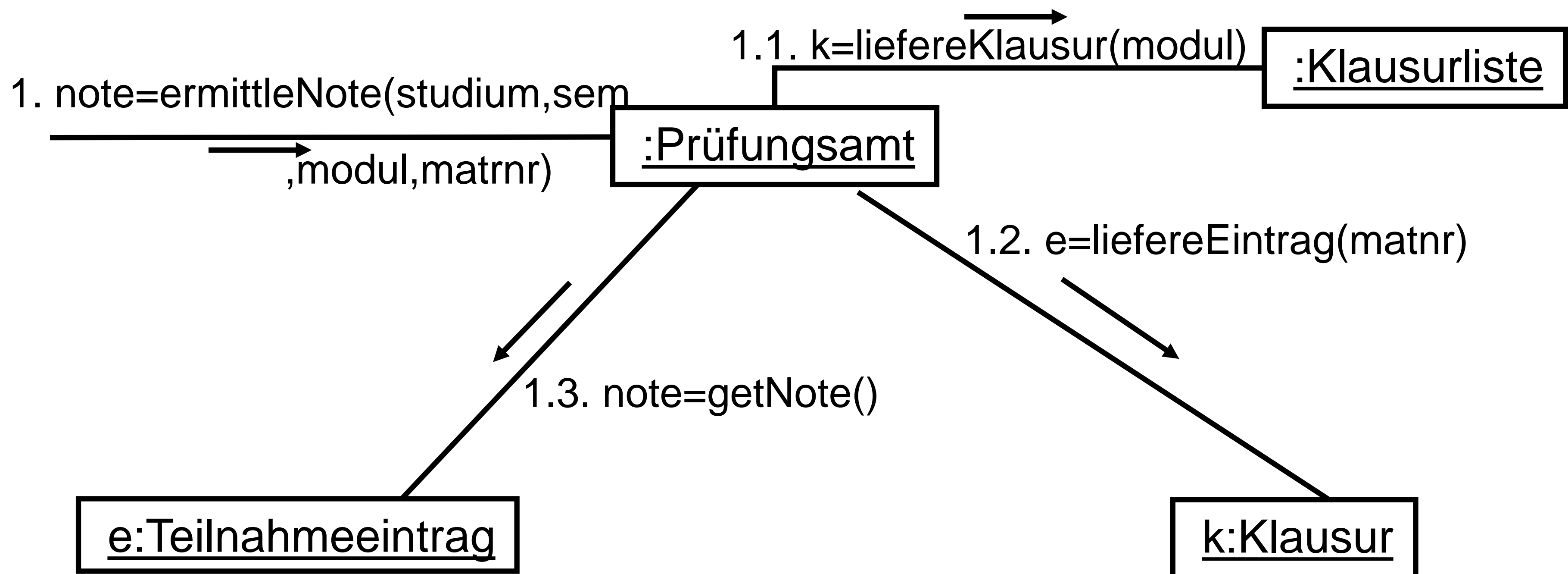
- Entwicklung (einschließlich Test)
- Verständnis
- Wiederverwendbarkeit

der Klassen erleichtert

Beispiel: Geringe Kopplung (1/2)

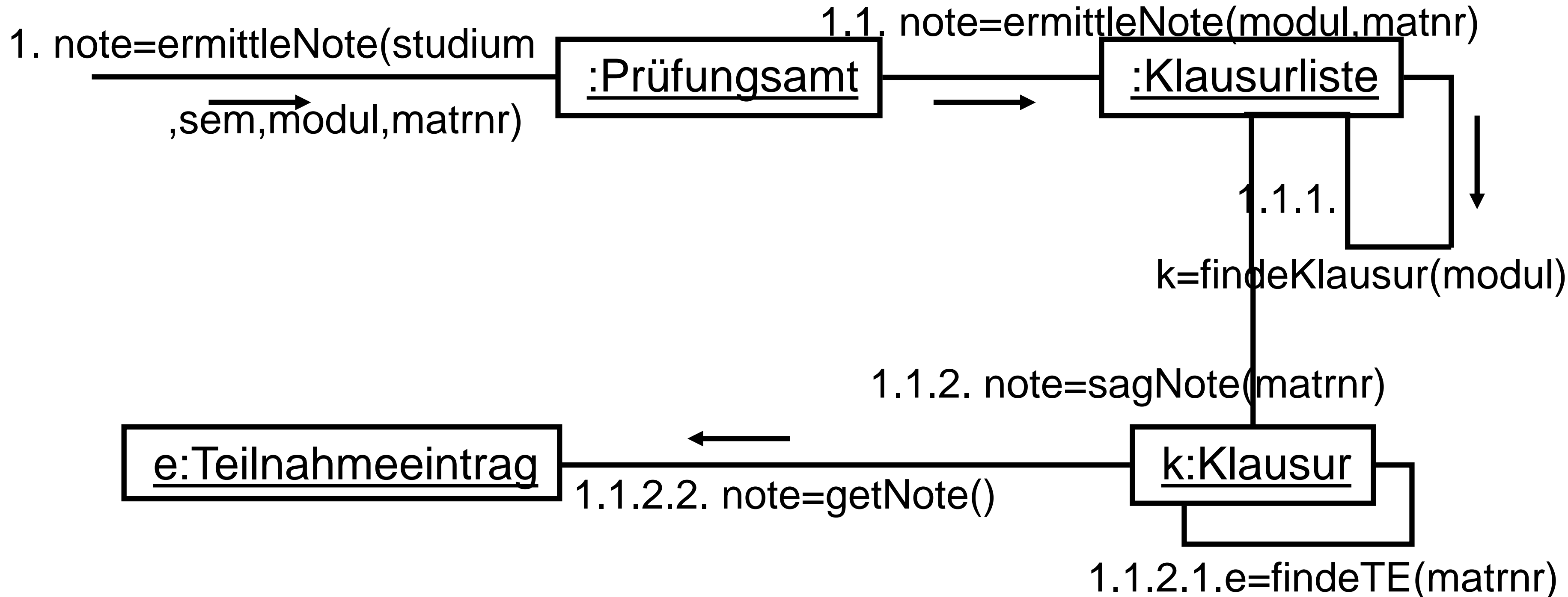
HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- Aufgabe: Es soll die Note eines Klausurteilnehmers ermittelt werden. Eine Lösung als Kommunikationsdiagramm könnte sein:



Schlecht!

Beispiel: Geringe Kopplung (2/2)



Besser:

- keine Abhängigkeit zwischen *Prüfungsamt* und Klausur
- *Prüfungsamt* benötigt kein Wissen über Organisation von *Klausur*
- entspricht auch dem “Experten-Muster”

Muster: hoher Zusammenhalt

Name: “hoher (funktionaler) Zusammenhalt” bzw. “high cohesion”

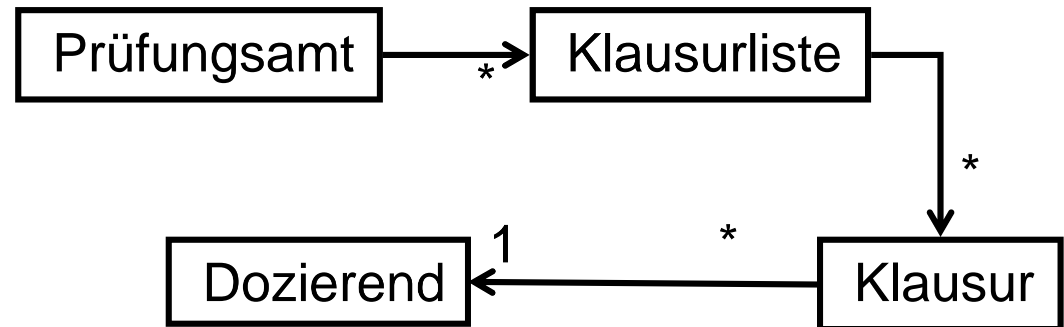
Regel: Die Verantwortungen, die einer Klasse übertragen werden, sollten

- ähnlich oder
- zueinander verwandt sein

Hintergrund: Klassen, die unterschiedlichste Aufgaben erfüllen, sind schwierig

- zu verstehen
- zu warten
- wiederzuverwenden

Beispiel: hoher Zusammenhalt



- Die Klasse Klausur enthält die Methoden
  - anzahlTeilnehmende()
  - notendurchschnitt()
  - standardabweichung()
  - getDozierend()

Klausur
anzahlTeilnehmende():int **notendurchschnitt():double** **Standardabweichung():double** **drucken():String** **getDozierend():String**

etc., aber z.B. nicht Methoden wie

- setDozierendname()
- gibMatrikelnummer(String name)

Muster: Don’t Talk to Strangers

Name: “Don’t talk to strangers“

Hintergrund: Ein Klient habe eine Assoziation zu einem (direkten) Objekt. Dieses wiederum habe eine Assoziation zu einem anderen (für den Klienten indirekten) Objekt.

Regel: Dann sollte das direkte Objekt die Zuständigkeit erhalten, mit dem indirekten Objekt zu kommunizieren (und nicht der Klient), so dass der Klient nichts über das indirekte Objekt wissen muss.

Konkretisierung: Don’t Talk to Strangers

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- Dieses Muster definiert Randbedingungen, zu welchen anderen Objekten Nachrichten geschickt oder nicht geschickt werden sollten.
- Erlaubte Nachrichten:
  - Zu dem *this* Objekt (oder *self*)
  - Einer Exemplarvariablen von *this*
  - Einem Objekt, das Parameter einer Methode ist
  - Einem Element einer Collection (Container) , welche Exemplarvariable von *this* ist
  - Einem Objekt, das innerhalb einer Methode erzeugt wurde
- Nicht erlaubt z.B.:
  - Ein Objekt soll niemals eine Nachricht zu einem Objekt senden, dessen Adresse es als Rückgabewert eines Methodenaufruf mit einem dritten Objekt erhalten hat

Muster: Reines Kunstgebilde

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

Name: “Reines Kunstgebilde” bzw. “pure fabrication”

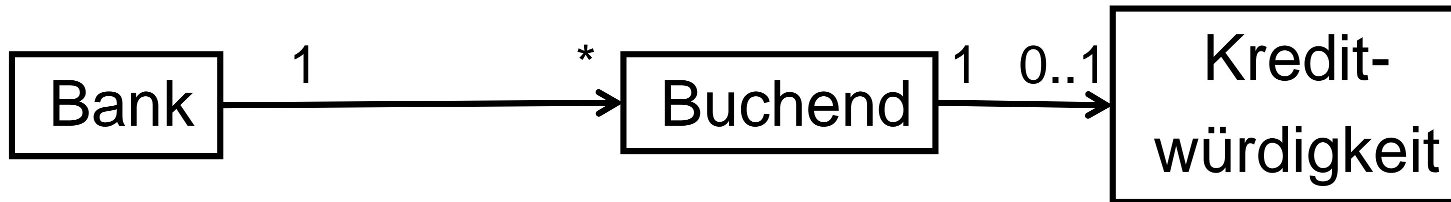
Regel: Falls man einer Klasse aufgrund

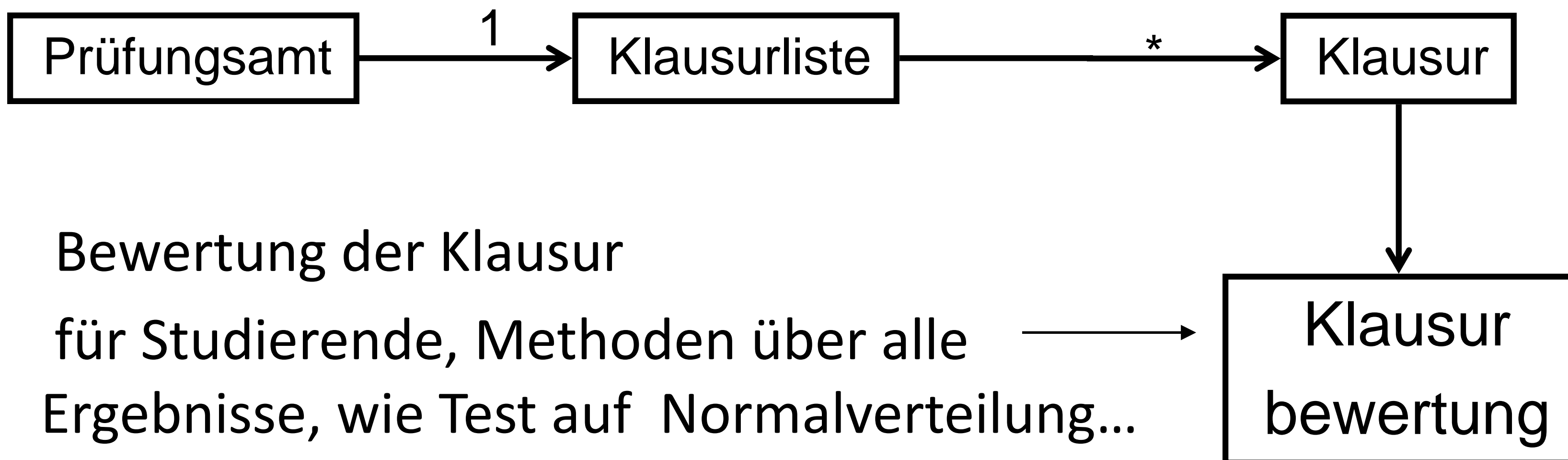
- natürlicher Gegebenheiten bzw.
- anderer logischer Gegebenheiten

bestimmte Aufgaben übertragen will und dadurch das Muster “hoher Zusammenhalt” verletzt wird, so sollte man einige Aufgaben in eine eigene Kunstklasse auslagern

- Hintergrund: Dieses dient zur Auflösung eines Konfliktes zwischen
  - natürlicher Modellierung und
  - “hohem Zusammenhalt”

Beispiel: Reines Kunstgebilde





Muster: Command-Query Separation

Name: "Ausführung-Abfrage Trennung" bzw. "Command-Query Separation (CQS)"

Regel: Die Methode einer Klasse soll eine der Funktionen

- Ausführen einer Aktion mit der Nebenwirkung, dass Objekt-/ Klassenvariablen verändert werden (möglichst vom Typ *void*)
- Ausführen einer Anfrage, um Daten ohne Nebenwirkung zurückzugeben

erfüllen, aber auf keinen Fall beides tun.

„Das Stellen einer Frage sollte nicht die Antwort beeinflussen."

Hintergrund: Die Schnittstelle einer Klasse sollte möglichst übersichtlich und verständlich sein. Insbesondere muss transparent sein, wie Objekt- und Klassenvariablen verändert werden.

Beispiel: Command-Query Separation

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

Beispiel 1: Inkrement

```
private int x;
public int nextX() {
  this.x = this.x + 1;
  return this.x;
 }    //Schlecht!
```

```
private int x;
public int getX() {
  return this.x; }
public void incrementX() {
  this.x = this.x + 1; } //gut!
```

Beispiel 2: Monopoly Würfel

```
private int wert;
public int werfen() {
  this.wert=(int)(Math
         .random()*6) + 1;
  return this.wert;
  }   //Schlecht!
```

```
private int wert;
public void werfen() {
   this.wert=(int)(Math.random()*6)
         + 1;
  }
public int getWert(){
  return this.wert;   }       //gut!
```

Method Chaining (1/3)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- immer sinnvolle Rückgabe nutzen; wenn wählbar wird statt void Objekt selbst zurück gegeben (this)
- Variante: Rückgabe eines Objekts gleichen Typs; nutzt z. B. Referenzen des Ursprungsobjektes
- verstößt klar gegen Command-Query Separation
- Beispiel Integer-Menge

```
public class Main {
  public static void main(String[] args) {
    IntMenge tmp = new IntMenge();
    tmp = tmp.hinzu(1, 21, 11, 41, 31, 1)
             .kleinerAls(41)
             .groesserAls(11);
    System.out.println(tmp);
  }
}
```

```
[21, 31]
```

Method Chaining (2/3)

```
public class IntMenge {
    private Set<Integer> menge = new HashSet<>();

    public IntMenge(){ }

    public IntMenge hinzu(int... wert){
        for(int w:wert){
            this.menge.add(w);
        }
        return this; // hier sieht man Chaining
    }

    public IntMenge kleinerAls(int grenze){
        IntMenge ergebnis = new IntMenge();
        for(int w:this.menge){
            if(w < grenze){
                ergebnis.hinzu(w);
            }
        }
        return ergebnis;
    }
```

Method Chaining (3/3)

```
public IntMenge groesserAls(int grenze){
        IntMenge ergebnis = new IntMenge();
        for(int w: this.menge){
                if(w > grenze){
                        ergebnis.hinzu(w);
                }
        }
        return ergebnis;
}

public boolean beinhaltet(int wert){
        return this.menge.contains(wert);
}

@Override
public String toString(){
        return this.menge.toString();
}
}
```

Beispiel: Hilfsklasse Objekterzeugung (1/4)

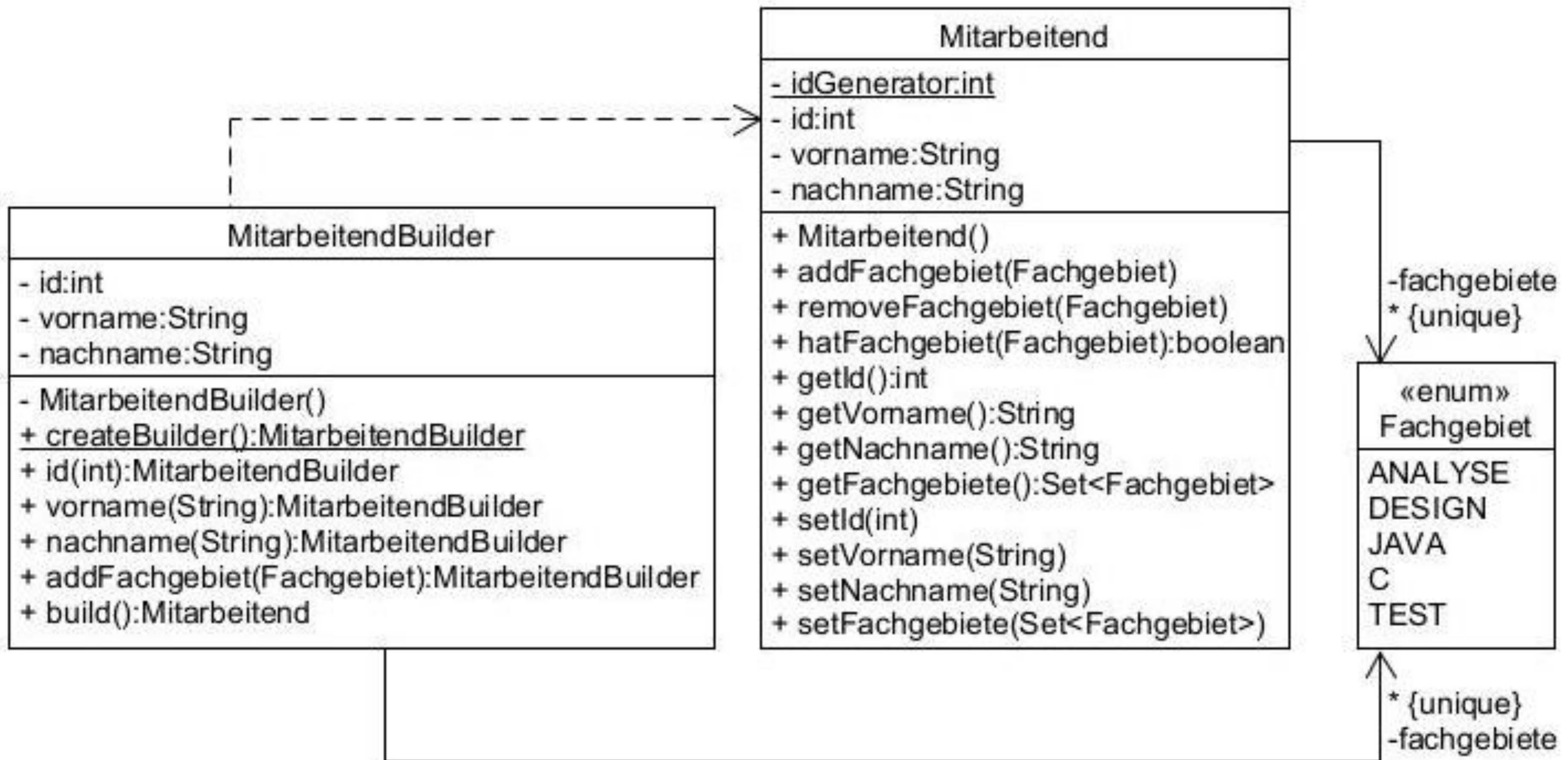
HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- Beispielnutzung

```
Mitarbeitend tmp = MitarbeitendBuilder
                    .createBuilder()
                    .vorname("Murat")
                    .nachname("Meier")
                    .addFachgebiet(Fachgebiet.C)
                    .addFachgebiet(Fachgebiet.JAVA)
                    .build();
```

- generell zur Erzeugung von Objekten nutzbar
- durch Fluent-Programming (Method Chaining) besser lesbar
- Methoden einfach ergänzbar

Beispiel: Hilfsklasse Objekterzeugung (2/4)



Beispiel: Hilfsklasse Objekterzeugung (3/4)

```
public class MitarbeitendBuilder {

  private int id;
  private String vorname = "Eva"; //Default-Wert
  private String nachname = "Mustermann";
  private Set<Fachgebiet> fachgebiete = new HashSet<>();

  public MitarbeitendBuilder() {}

  public static MitarbeitendBuilder createBuilder(){
    return new MitarbeitendBuilder();
  }

  public MitarbeitendBuilder vorname(String vorname) {
    this.vorname = vorname;
    return this;
  }

  public MitarbeitendBuilder nachname(String nachname) {
    this.nachname = nachname;
    return this;
  }
```

Beispiel: Hilfsklasse Objekterzeugung (4/4)

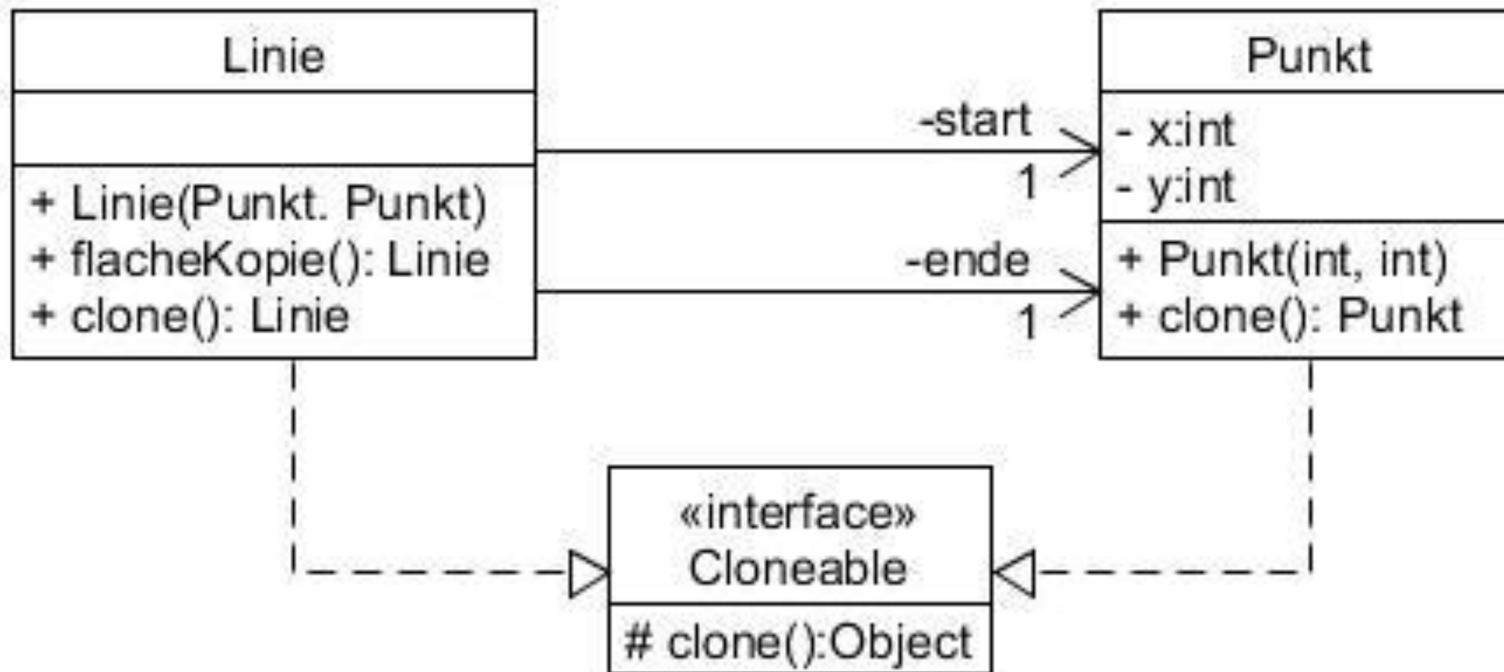
HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

```
    public MitarbeitendBuilder id(int id){
      this.id = id;
      return this;
    }

    public MitarbeitendBuilder addFachgebiet(Fachgebiet f){
      this.fachgebiete.add(f);
      return this;
    }

    public Mitarbeitend build() {
      Mitarbeitend erg = new Mitarbeitend();
      erg.setId(this.id);
      erg.setVorname(this.vorname);
      erg.setNachname(this.nachname);
      erg.setFachgebiete(this.fachgebiete);
      return erg;
    }
}
```

Erinnerung: clone(), Erzeugung echter Kopien (1/4)

Video

- Java arbeitete mit Referenzen, Default-Implementierung von clone() liefert nur flache Kopien
- Interface Cloneable implementieren und clone() überschreiben



- Erinnerung: Strings sind immutable (immer neues Objekt)

Erinnerung: clone(), Erzeugung echter Kopien (2/4)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- in Linie:

```
public Linie flacheKopie(){
  return new Linie(this.start, this.ende);
}
```

```
public static void main(String[] args) {
   Linie l1 = new Linie( new Punkt(1,2), new Punkt(3,4));
   System.out.println(l1);
   Linie l2 = l1.flacheKopie();
   System.out.println("l1 == l2 : " + (l1 == l2));
   l2.getStart().setX(42);
   System.out.println(l1);
   System.out.println(l2);
}
```

```
Linie{start=Punkt{x=1, y=2}, ende=Punkt{x=3, y=4}}
l1 == l2 : false
Linie{start=Punkt{x=42, y=2}, ende=Punkt{x=3, y=4}}
Linie{start=Punkt{x=42, y=2}, ende=Punkt{x=3, y=4}}
```

Erinnerung: clone(), Erzeugung echter Kopien (3/4)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- in Punkt:

```
public class Punkt implements Cloneable{ …

  @Override
  public Punkt clone() {  // darf Punkt statt Object stehen
    return new Punkt(this.x, this.y);
  }
```

- in Linie:

```
public class Linie implements Cloneable{ …

  @Override
  public Linie clone() {
    return new Linie(this.start.clone(), this.ende.clone());
  }
```

Erinnerung: clone(), Erzeugung echter Kopien (4/4)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

```
public static void main(String[] args) {
  Linie l1 = new Linie( new Punkt(1,2), new Punkt(3,4));
  System.out.println(l1);
  Linie l2 = l1.clone();
  System.out.println("l1 == l2 : " + (l1 == l2));
  l2.getStart().setX(42);
  System.out.println(l1);
  System.out.println(l2);
}
```

```
Linie{start=Punkt{x=1, y=2}, ende=Punkt{x=3, y=4}}
l1 == l2 : false
Linie{start=Punkt{x=1, y=2}, ende=Punkt{x=3, y=4}}
Linie{start=Punkt{x=42, y=2}, ende=Punkt{x=3, y=4}}
```

- Erinnerung an Praktikumsaufgabe: Ansatz funktioniert nur, wenn keine identischen Objektreferenzen mehrfach im zu clonenden Objekt enthalten

Kombination von Pattern: Beispiel Redux

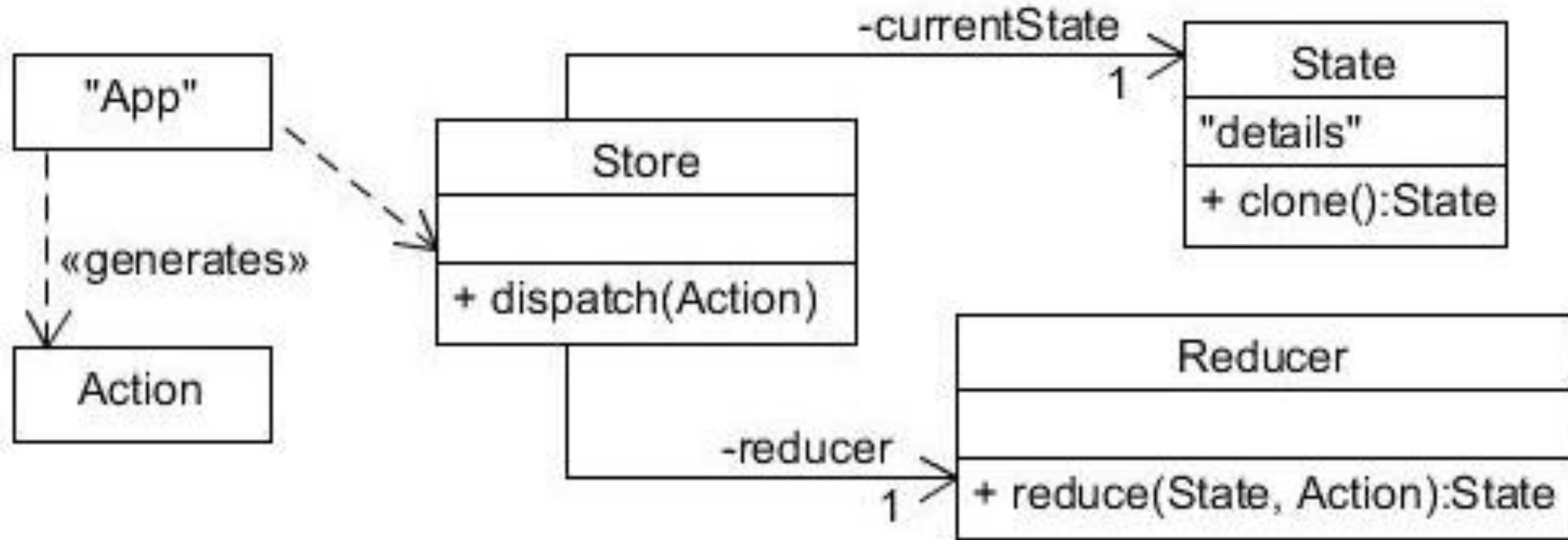
Skizze 0 | Skizze 1 | Skizze 2 | Skizze 3 | Skizze 4

- Objekte arbeiten typischerweise mit Referenzen (Zeigern) zur Verknüpfung von Objekten, das ist schnell, kann aber undurchsichtig werden
- Beispiel: Oberflächen, mit denen verschiedene Objekte der Geschäftsebene bearbeitet werden
- Ansatz: zentraler State, der alle relevanten Informationen hält
- Ansatz: Veränderung des States nur über zentralen Store
- Ansatz: es entstehen bei Aktionen immer neue State-Objekte

- ursprünglich für JavaScript entwickelt (basierend auf Flux)
- Ansatz auch Grundlage von Reactive Programming
- Folien motiviert durch: https://www.lestard.eu/2018/implement-your-own-redux-in-java/

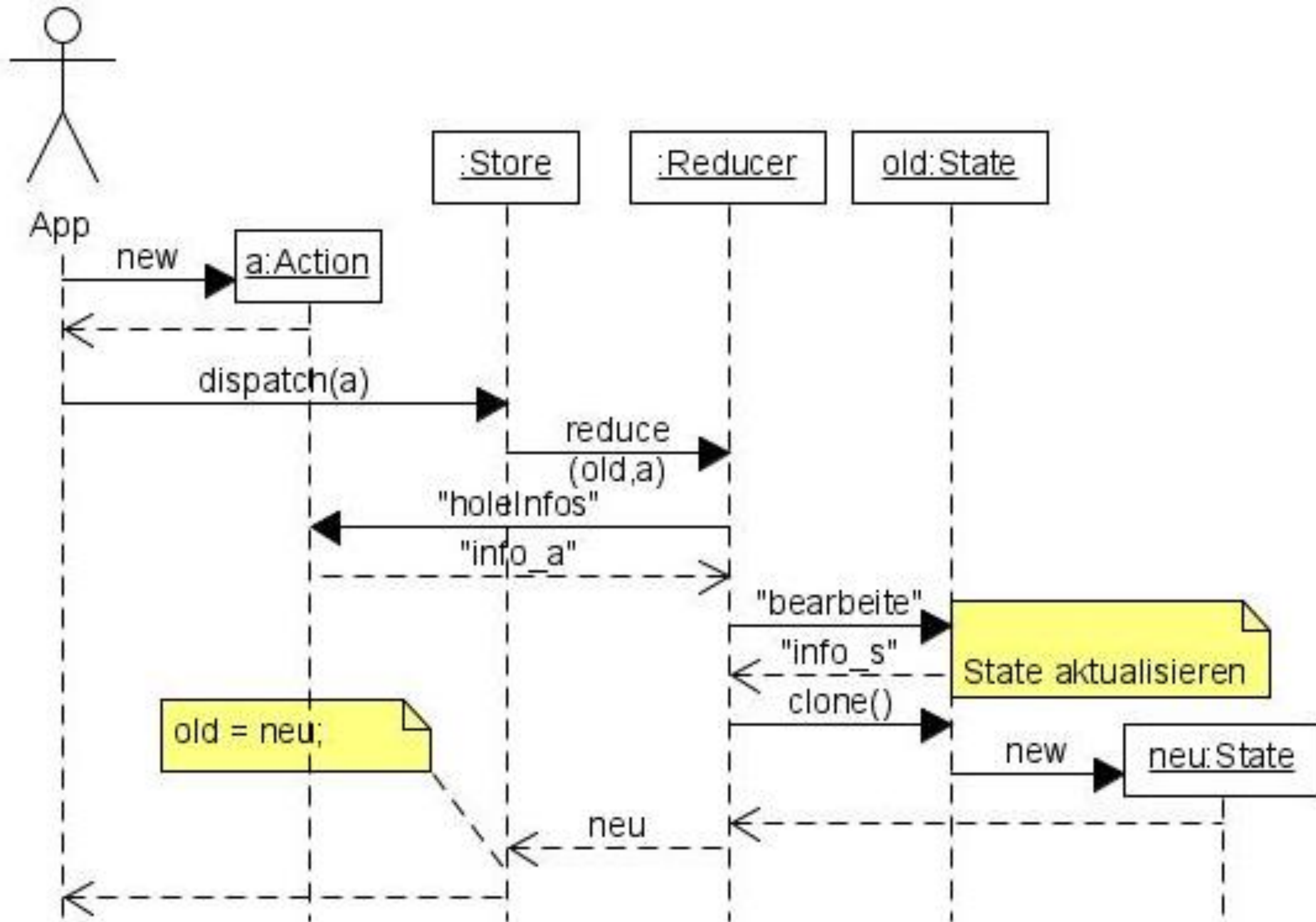
Redux – Konzept Version 0 (1/2)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES



- Nutzung („App“) erzeugt Action-Objekt a, beinhaltet, was gemacht werden soll
- Nutzung ruft dispatch beim Store mit Action a auf
- Store ruft Reducer mit noch aktuellem State currentState (old) und Action a auf
- Reducer berechnet neuen State neu aus currentState (old) und a
- Store: currentState = neu

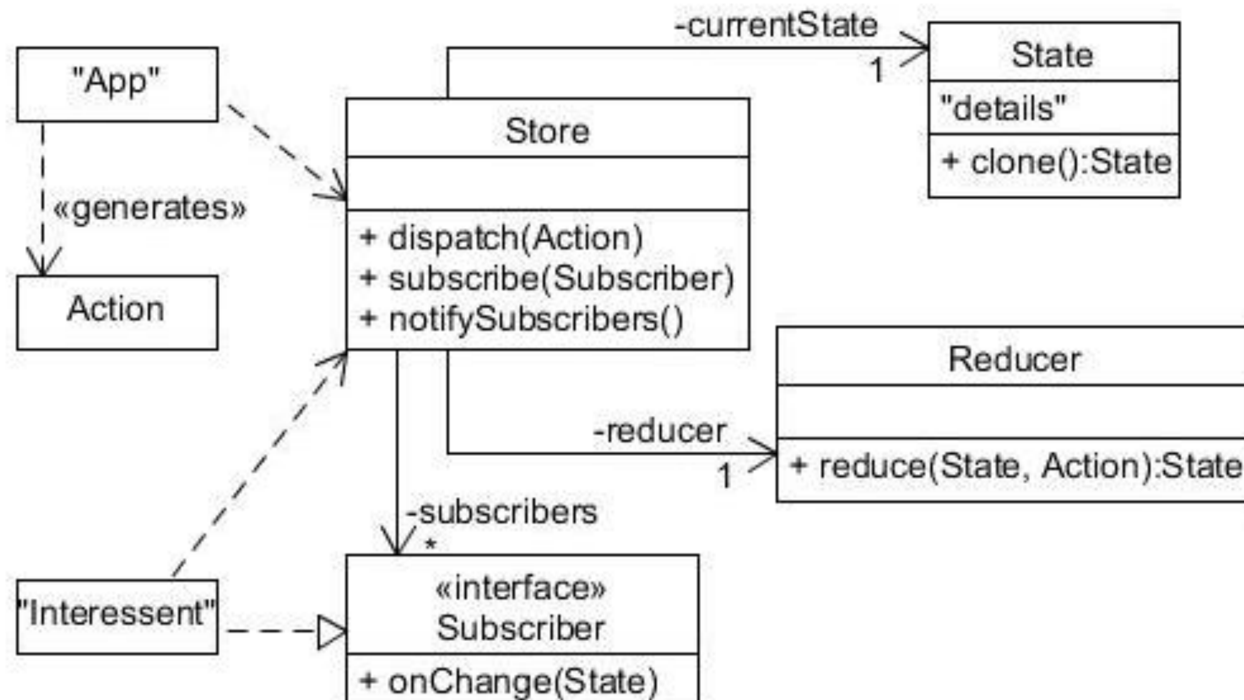
Redux – Konzept Version 0 (2/2)



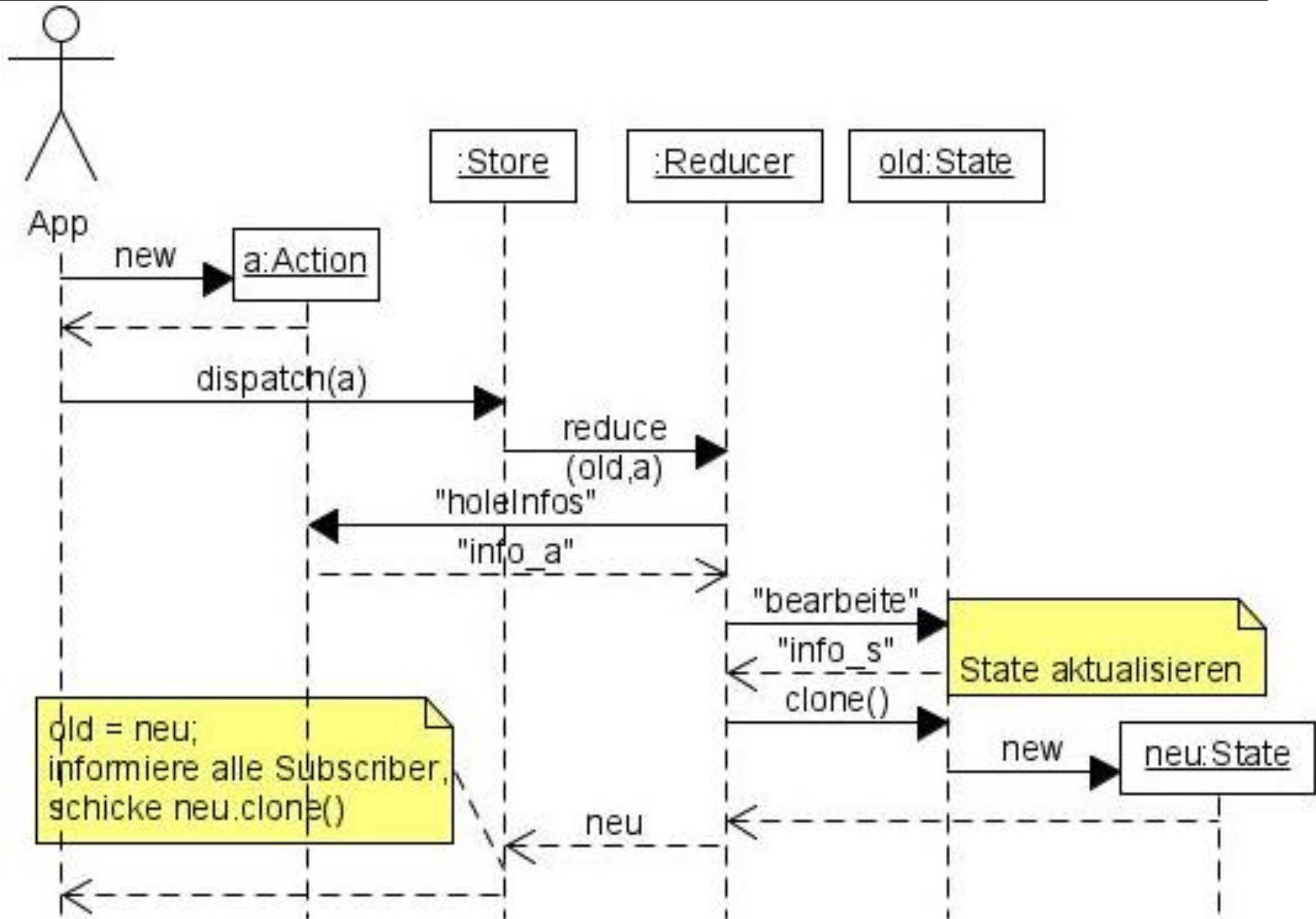
Redux – Konzept Version 1 (1/12)

Video

- offen, wie bekommen Interessierte, z. B. GUI-Komponenten Änderungen mit (Rückgabe neuen Zustands wäre denkbar)
- Lösung: Store bietet Observer-Observable-Lösung; d. h. Interessierte an (ggfls. bestimmten) Zustandsänderungen können sich anmelden (hier fasst Store konkreten und abstrakten Observable zusammen)



Redux – Konzept Version 1 (2/12)



Redux – Konzept Version 1 (3/12)

- Beispiel: Bearbeitung einer Taskliste
- „App“ und „Interessent“ sind Konsole (TextIO)
- Fachklassen sehen wie folgt aus: (entspricht „details“)

- Businessklassen befinden sich im (oder „hinter“ dem) State
- State kann auch als Model angesehen werden

Redux – Konzept Version 1 (4/12)

```
public class Action {
  // generell sollte auf String-Parameter, die auch andere
  // Werte kodieren sollen, aus Typsicherheitsgruenden
  // verzichtet werden
  private List<String> parameter;

  public Action(List<String> parameter) {
    this.parameter = parameter;
  }

  public Action(String... par1){
    this(Arrays.asList(par1));
  }

  public List<String> getParameter() {
    return this.parameter;
  }
}
```

Redux – Konzept Version 1 (5/12) – App (1/2)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

```
public class TextIO {

  private Store store = new Store(new State(), new Reducer());

  public TextIO() {
    this.store.subscribe(new Subscriber(){
      @Override
      public void onChange(State s){
        System.out.println(s.getTaskList());
      } });
  }

  public void dialog() {
    int eingabe = -1;
    while (eingabe != 0) {
      System.out.print(""
               + "(0) beenden\n"
               + "(1) Task hinzu\n"
      );
      eingabe = Eingabe.leseInt();
      // naechste Folie
```

Redux – Konzept Version 1 (6/12) – App (2/2)

```
      switch (eingabe) {
        case 1: {
          this.newTask();
          break;
        }
      }
    }
  }

  private void newTask() {
    System.out.print("neue Aufgabe: ");
    String text = Eingabe.leseString();
    System.out.print("Bearbeitende Person: ");
    String responsible = Eingabe.leseString();
    Action action = new Action(text, responsible);
    this.store.dispatch(action);
  }
}
```

Redux – Konzept Version 1 (7/12)

```
public class Store {

  private State currentState;
  private Reducer reducer;
  private List<Subscriber> subscribers = new ArrayList<>();

  public Store(State initialState, Reducer reducer) {
    this.currentState = initialState;
    this.reducer = reducer;
  }

  public State getState() {
    return this.currentState;
  }

  public void dispatch(Action action) {
    this.currentState = this.reducer.reduce(this.currentState
                                            , action);
    this.notifySubscribers();
  }
```

Redux – Konzept Version 1 (8/12)

```
  private void notifySubscribers() {
    for (Subscriber s: this.subscribers){
      s.onChange(this.currentState.clone());
    }
  }

  public void subscribe(Subscriber subscriber) {
    this.subscribers.add(subscriber);
    subscriber.onChange(this.currentState.clone());
  }
}
```

```
public interface Subscriber {
  void onChange(State state);
}
```

Redux – Konzept Version 1 (9/12)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

```
public class State implements Cloneable{

  private TaskList taskList;

  public State(){
    this.taskList = new TaskList();
  }

  private State(TaskList taskList) {
    this.taskList = taskList;
  }

  public TaskList getTaskList() {
    return this.taskList;
  }
```

Redux – Konzept Version 1 (10/12)

```
  public void add(String text, String responsible){
    this.taskList.add(text, responsible);
  }

  @Override
  public State clone() {// nur fuer interne Tests, sonst clone()
    State result = new State(this.taskList.clone());
    return result;
  }
}
```

Redux – Konzept Version 1 (11/12)

```
public class Reducer {

  public State reduce(State state, Action action) {
    if(action.getParameter().size() < 2){
      throw new IllegalArgumentException(
        "Hinzufuegen benoetigt zwei Parameter");
    }
    state.add(action.getParameter().get(0),
              action.getParameter().get(1));
    return state.clone();
  }
}
```

Redux – Konzept Version 1 (12/12)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

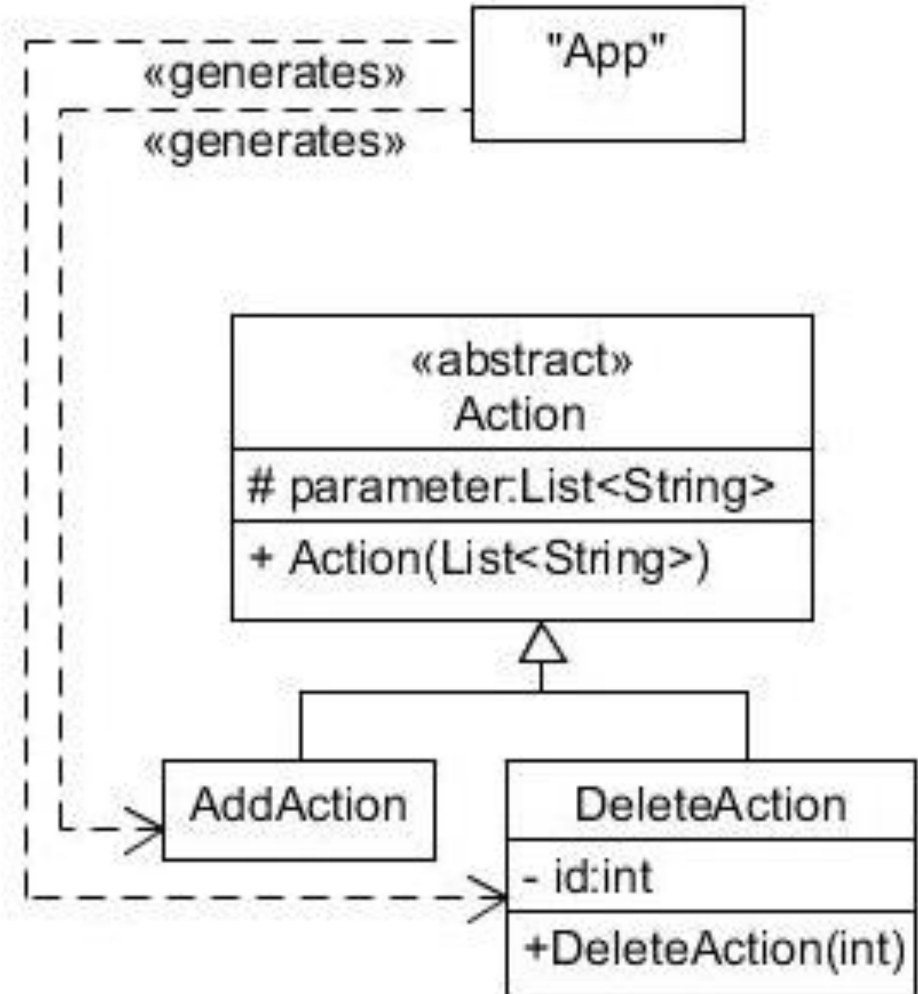
```
(0) beenden
(1) Task hinzu
1
neue Aufgabe: Redux lernen
Bearbeitende Person: ich
Task{id=1, text=Redux lernen, responsible=ich, finished=false}

(0) beenden
(1) Task hinzu
1
neue Aufgabe: Redux coden
Bearbeitende Person: mein Kumpel
Task{id=1, text=Redux lernen, responsible=ich, finished=false}
Task{id=4, text=Redux coden, responsible=mein Kumpel,
  finished=false}
```

Redux – Konzept Version 2 (1/11)

Video

Flexibilisierung des Ansatzes

- mehr Actions: Beispiel Delete-Operation
- ursprüngliche Action wird zur AddAction
- Interface oder abstrakte Klasse für Gemeinsamkeit
- offen: sinnvoller Umgang mit Parametern (Strings immer nutzbar, fast immer schwach)
- hier: individuelle Parameter



Redux – Konzept Version 2 (2/11)


```
public class DeleteAction extends Action{
  private int deleteId;

  public DeleteAction(int id){
    this.deleteId = id;
  }

  public int getDeleteId() {
    return deleteId;
  }

  @Override
  public String toString() {
    return "DeleteAction{" + "deleteId=" + deleteId + '}';
  }
}
```

Redux – Konzept Version 2 (3/11)

- in TextIO

```
…
        case 2: {
            this.deleteTask();
            break;
        }
…

 private void deleteTask() {
    System.out.print("welche Id: ");
    int id = Eingabe.leseInt();
    Action action = new DeleteAction(id);
    this.store.dispatch(action);
  }
```

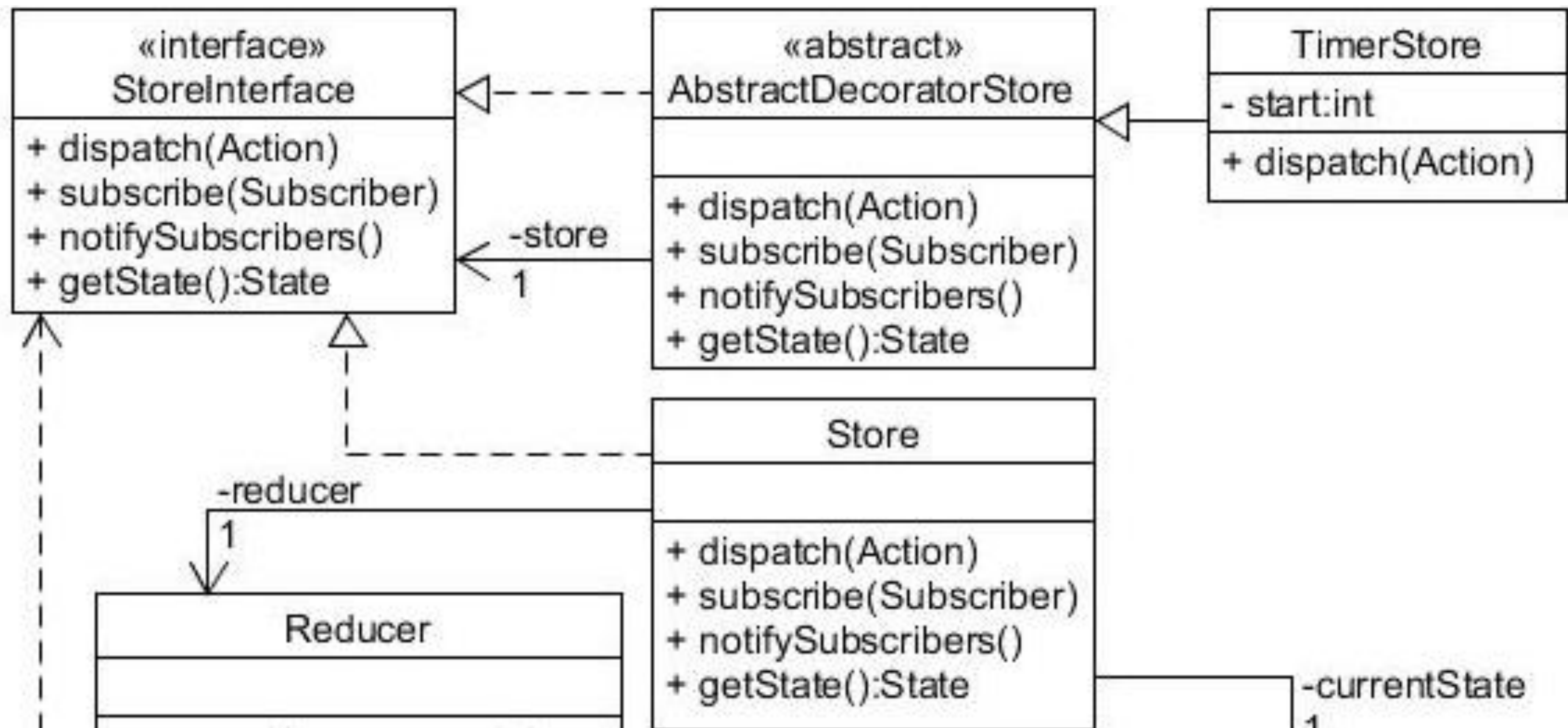
Redux – Konzept Version 2 (4/11)


```
public class Reducer {
  public State reduce(State state, Action action) {
    this.reduceIntern(state, action);
    return state.clone();
  }

  private void reduceIntern(State state, Action action) {
    if (action instanceof AddAction) {
      state.add(action.getParameter().get(0),
                action.getParameter().get(1));
      return; // Alternative  für jede Action
    }
    if (action instanceof DeleteAction) {
      state.delete(((DeleteAction) action).getDeleteId());
      return; // State bekommt delete(int)-Methode
    }

    throw new IllegalArgumentException(
              "Action " + action + " nicht unterstuetzt");
  }
}
```

Redux – Konzept Version 2 (5/11)

Flexibilisierung des Ansatzes

- Store-Varianten, die zusätzliche Aufgaben übernehmen
- Beispiel: Messe Zeit der Methodenausführung
- Ansatz: Decorator-Pattern, so Store-Varianten verknüpfbar



Redux – Konzept Version 2 (6/11)


```
public interface StoreInterface {

  public void dispatch(Action action);
  public State getState(); // !!!! (1)
  public void subscribe(Subscriber subscriber);
  public void notifySubscribers();

}
```

- ursprüngliche Store-Klasse bleibt erhalten, realisiert Interface
- (1) getState()-Methode darf nur zum Testen genutzt werden, nur weil jemand Store kennt, ist der Aufruf noch lange nicht erlaubt
- (1) falls getState() für alle nutzbar sein soll, muss der State bei Rückgabe gecloned werden [macht Testen schwieriger]

Redux – Konzept Version 2 (7/11)

```
public abstract class AbstractDecoratorStore
                                implements StoreInterface {

  protected StoreInterface store;

  public AbstractDecoratorStore(StoreInterface store) {
    this.store = Objects.requireNonNull(store);
  }

  @Override
  public State getState() {  // nur fuer Testzwecke
    return this.store.getState();
  }
```

Redux – Konzept Version 2 (8/11)

```
  @Override
  public void dispatch(Action action) {
    this.store.dispatch(action);
  }

  @Override
  public void notifySubscribers() {
    this.store.notifySubscribers();
  }

  @Override
  public void subscribe(Subscriber subscriber) {
    this.store.subscribe(subscriber);
  }
}
```

Redux – Konzept Version 2 (9/11)


```
public class TimerStore extends AbstractDecoratorStore {

  private long start;

  public TimerStore(StoreInterface store) {
    super(store);
  }

  @Override
  public void dispatch(Action action) {
    start = System.nanoTime();
    super.store.dispatch(action);
    System.out.println("Dauer von " + action
            + ": " + (System.nanoTime() - start));
  }
}
```

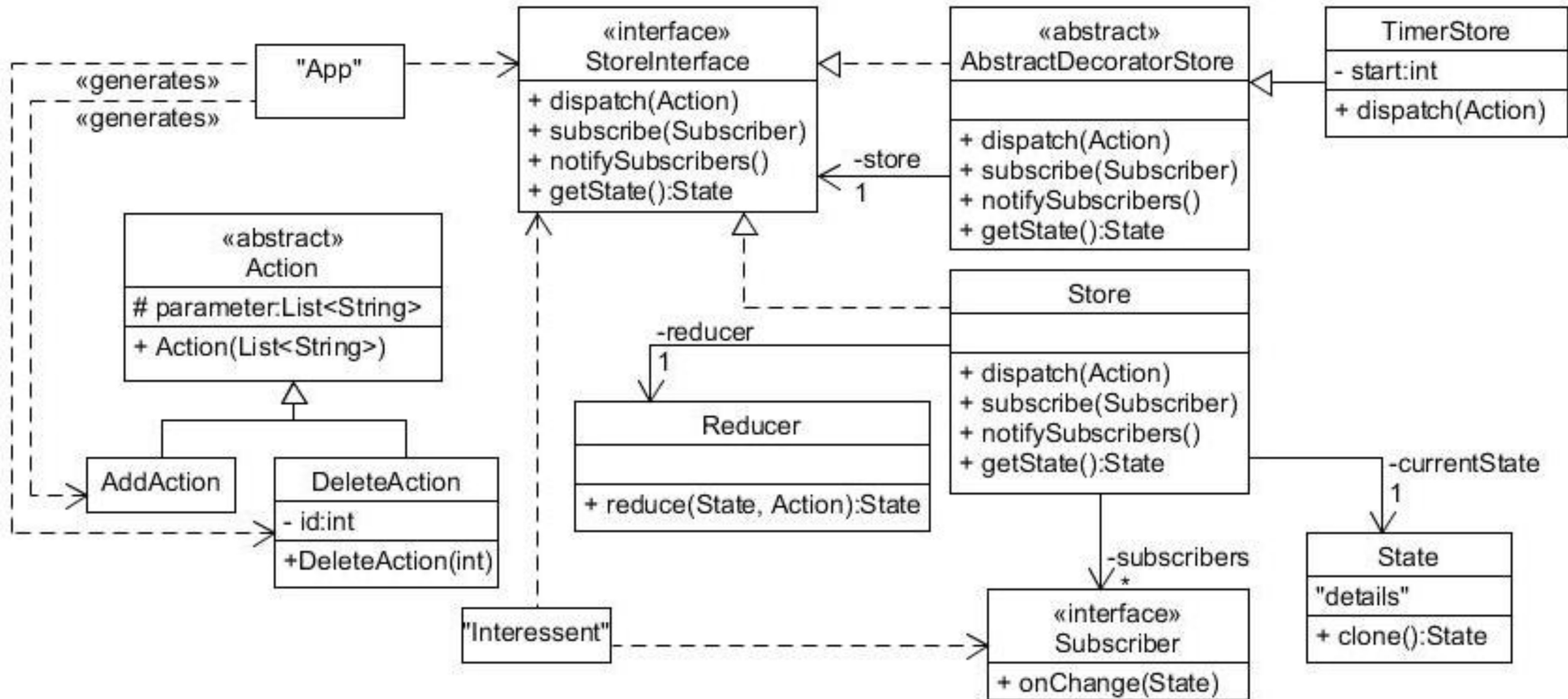
Redux – Konzept Version 2 (10/11)

- Nutzung in TextIO

```
private StoreInterface store = new TimerStore(
              new Store(new State(), new Reducer())
);
```

```
(0) beenden
(1) Task hinzu
(2) Task loeschen
1
neue Aufgabe: Flexibilisieren
Bearbeitende Person: ich
Task{id=1, text=Flexibilisieren, responsible=ich, finished=false]

Dauer von AddAction{parameter=[Flexibilisieren, ich]}: 8505520
(0) beenden
(1) Task hinzu
(2) Task loeschen
```

Redux – Konzept Version 2 (11/11)

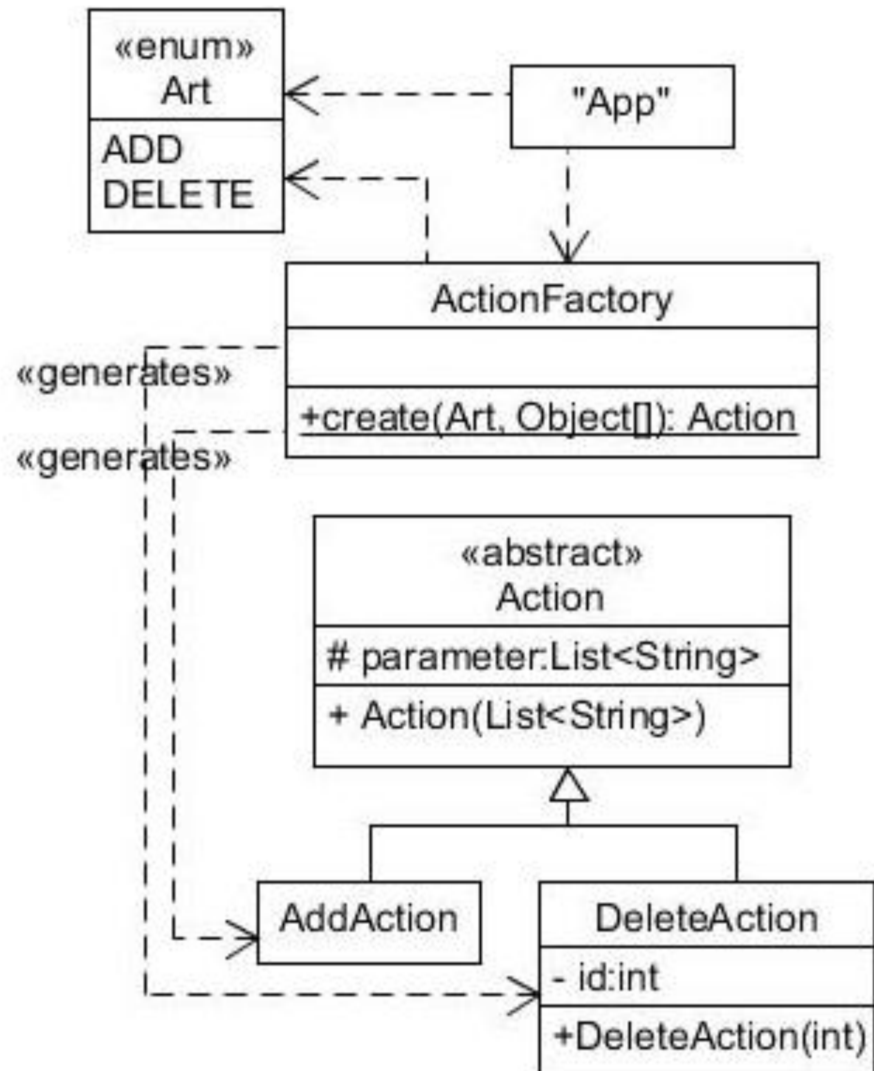


Redux – Konzept Version 3 (1/4)

Video

Systematisierung: Erzeugung von Actions bündeln

- hier korrekte Form garantieren
- konkrete Factory
- Aufzählungswert pro Action-Art ist Möglichkeit, geht auch mit int-Parameter



Redux – Konzept Version 3 (2/4)

```
public class ActionFactory {

  // auch mehrere create-Methoden denkbar
  public static Action create(Art command, Object... value) {
    try {
      switch (command) {
        case ADD:
          List<String> tmp = new ArrayList<>();
          for(Object o:value){tmp.add(o.toString());}
          if (tmp.size() < 2) {
            throw new IllegalArgumentException(
                    "Hinzufuegen benoetigt zwei Parameter");
          }
          return new AddAction(tmp);
        case DELETE:
          if (value.length == 0) {
            throw new IllegalArgumentException(
                    "DELETE benoetigt Parameter");
          }
          return new DeleteAction((Integer) value[0]);
```

Redux – Konzept Version 3 (3/4)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

```
        default:
          throw new IllegalArgumentException("Action("
             + command + "," + Arrays.asList(value)
             + ") existiert nicht");
      }
    } catch (ClassCastException e) {
      throw new IllegalArgumentException("Action("
             + command + "," + Arrays.asList(value)
             + ") hat falschen Parametertyp :" + e);
    }
  }
}
```

Redux – Konzept Version 3 (4/4)

- in TextIO:

```
private void deleteTask() {
  System.out.print("welche Id: ");
  int id = Eingabe.leseInt();
  Action action = ActionFactory.create(Art.DELETE, id);
  this.store.dispatch(action);
}

private void newTask() {
  System.out.print("neue Aufgabe: ");
  String text = Eingabe.leseString();
  System.out.print("Bearbeitende Person: ");
  String responsible = Eingabe.leseString();
  Action action = ActionFactory.create(Art.ADD
      , text, responsible);
  this.store.dispatch(action);
}
```

Redux – Fazit

- für kleine Beispiele recht aufwändig
- sehr leicht erweiterbar, gibt feste Stellen an denen ergänzt wird
- Funktionalität aber auf einige Klassen verteilt; gefährlich, wenn man bei Änderungen eine vergisst
- clone() des State-Objekts kann viel Zeit kosten
  - State eher für Oberflächen-Daten als gesamte Daten
  - pragmatisch überlegen, ob clone() für alles benötigt wird
- gibt kein direktes Ergebnis für Aufrufer; ggfls. weiteres Publish-Subscribe für Antworten
- sehr gut für asynchrone Systeme (Action abschicken und weitermachen, anderer Thread erhält neue Zustände und wertet sie aus)

Beschreibung der Pattern

8.4

Name: Abstract Factory
Patterngruppe: Objekterzeugung
Kurzbeschreibung: Client kann mit einer AbstractFactory zu einer abstrakten Klasse passende Exemplare aus einem Satz konkreter Implementierungen für bestimmtes Produkt erzeugen, kennt den konkreten Typ des erzeugten Exemplars nicht
Kontext: viele verschiedene gleichartige, aber unterscheidbare Objekte sollen verwaltet werden
Problem: Klasse soll verschiedene Objekte bearbeiten, benötigt aber nur deren gemeinsame Eigenschaften
Lösung: Einführung von zwei abstrakten Klassen, die zum Einen Objekterzeugung, zum Anderen Objektzugriff erlauben, Client muss nur diese Klassen kennen

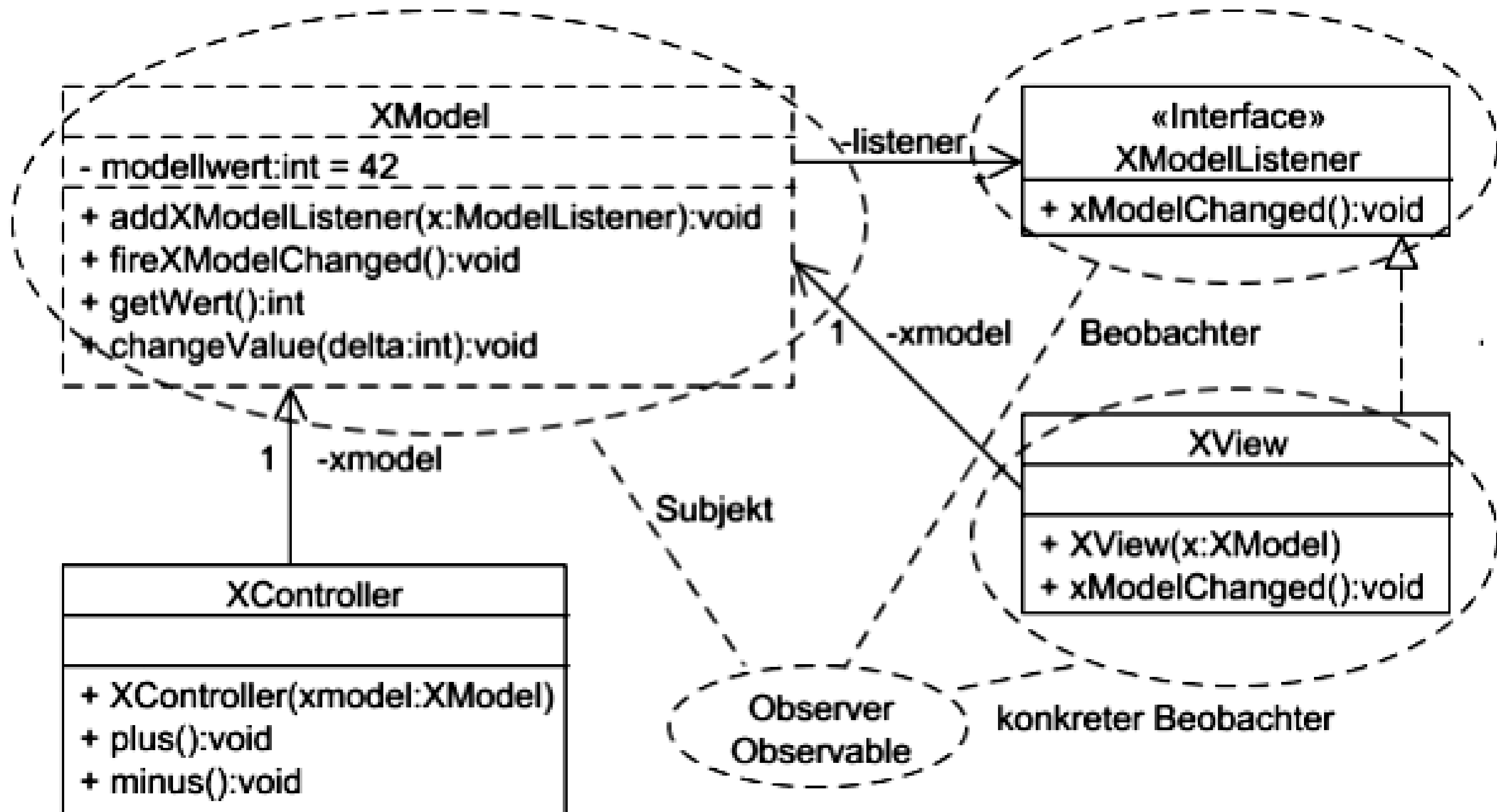
Einsatzgebiete: ...	**Varianten:** ...	**Struktur:** s.o.	**Beispiele:**

GoF-Pattern Übersicht (nicht auswendig lernen)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

Einsatzbereich		Aufgabenbereich: Erzeugung	Struktur	Verhalten
	Klasse	Factory	Adapter	Interpreter
				Template
	Objekt	Abstract Factory	Adapter	Command
		Builder	Bridge	Observer
		Prototype	Decorator	Visitor
		Singleton	Facade	Memento
			Composite	Strategy
			Proxy	Mediator
			Flyweight	State
				Chain of Responsibility

Pattern in der UML



Pattern-Name im gestrichelten Kreis, verbunden mit eingekreisten Klassen, verbunden mit Pattern und Benennung der Rollen

Kritische Betrachtung von Pattern

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- Pattern für Personen mit wenig Programmiererfahrung wenig geeignet, man muss erste Erfahrungen haben, um von Erfahrungen anderer Personen zu profitieren

- überlagernde Pattern schwer pflegbar, später in Implementierungen teilweise schwer erkennbar

- Pattern finden Einzug in Bibliotheken, Beispiel: Event-Handling in Java ist „Observer-Pattern“, und Architekturen, Beispiel: MQTT (auch Obs-Obs)

- Generell sind Pattern ein wichtiger Teilschritt zum ingenieurmäßigen SW-Engineering

- Gute Programmier-Aufgabe: Entwickeln Sie kleine Beispiele zu allen GoF-Pattern !!!

8.5

Video

- Functional Interfaces / Lambda Ausdrücke
- Optional
- Streams in Java
- Dependency Injection
- Services in Java Modulen
- Kombination aus Factories und Annotationen

Java 8 – Functional Interfaces (1/3)

- Ansatz: Funktionen als Parameter übergeben
- Vereinfachung für Interfaces, die genau eine Methode enthalten (auch SAM-Types für Single Abstract Method, selber mit @FunctionalInterface)

```
@FunctionalInterface  // Interface mit genau einer Methode
public interface Ausgabe {
  public void ausgeben(String s);
}
```

```
public class AusgabeImpl implements Ausgabe { // Standard
  @Override
  public void ausgeben(String s) {
    System.out.println("Impl: " + s);
  }
}
```

Java 8 – Functional Interfaces (2/3) – mit Lambda

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

```
public class Main {

  public static void main(String[] args) {
    Ausgabe impl = new AusgabeImpl();
    String text = "Text";
    impl.ausgeben(text);     // Impl: Text

    Ausgabe an2 = new Ausgabe(){
      @Override
      public void ausgeben(String s) {
        System.out.println("Ano: "+s);
      }
    };
    an2.ausgeben(text);      // Ano: Text

    Ausgabe an3 = s -> System.out.println("Lambda: "+s);
    an3.ausgeben(text);      // Lambda: Text
```

Java 8 – Functional Interfaces (3/3) – mit Lambda

```
    Ausgabe an4 = System.out::println;
    an4.ausgeben(text);       // Text

    Ausgabe an5 = s -> {
      System.out.println("Lambda: "+s);
      System.out.println("noch ne Zeile");
    };
    an5.ausgeben(text);       // Lambda: Text
                              // noch ne Zeile
  }
}
```

- Lambda-Ausdrücke beschreiben Funktionen

  `(Parameterliste) -> {Ausdruck bzw. Programmanweisungen}`

- Spezifikation: JSR 335: Lambda Expressions for the JavaTM Programming Language, https://jcp.org/en/jsr/detail?id=335

Optional (1/5)

Video

- Grundproblem der Programmierung sind undefinierte Referenzen, also NullPointerExceptions in Java
- immer wenn Objekt Ergebnis sein kann, muss programmierende Person damit rechnen einen Null-Wert zu erhalten
  - d. h. man muss immer darauf prüfen
  - oder Angebot (Schnittstelle) garantiert, dass es kein Null-Wert ist (kann man trauen?)
- bequeme Unart, wenn Ergebnis irgendwie nicht berechenbar, z. B. Parameter nicht ok, ist Ergebnis Null-Wert, als Abkürzung für „irgendwie ist der Aufruf gescheitert“
- Lösung: Ergebnis wird als Optional (generischer Typ) gekennzeichnet; Nutzung weiß damit, dass Ergebnis Null-Wert sein kann und muss reagieren

Optional (2/5) – Problem mit null (1/2)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

```
public class Einkaufsliste {
  private Map<String,Integer> produkte;

  public Einkaufsliste() {
    this.produkte = new HashMap<>();
  }

  public void hinzu(String prod, int anzahl) {
    this.produkte
        .put(prod, this.produkte.getOrDefault(prod, 0)
                   + anzahl);
  }

  public Integer anzahlVon(String prod) {
    return this.produkte.get(prod);
  }
...
```

Optional (3/5) – Problem mit null (2/2)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

```
public static void main0(String[] args) {
  Einkaufsliste ek = new Einkaufsliste();
  ek.hinzu("Bier", 3);
  ek.hinzu("Wasabi", 5);
  ek.hinzu("Bier", 6);
  System.out.println("ek: " + ek);
  System.out.println("Bier " + ek.anzahlVon("Bier"));
  System.out.println("Beer " + ek.anzahlVon("Beer"));
  int moreBeer = ek.anzahlVon("Beer") + 1;
}
```

```
ek: Einkaufsliste [produkte={Bier=9, Wasabi=5}]
Bier 9
Beer null
Exception in thread "main" java.lang.NullPointerException
```

Optional (4/5) – Problemlösung (1/2)

- Variante in Einkaufsliste

```
public Optional<Integer> anzahl(String prod) {
   return Optional.ofNullable(this.produkte.get(prod));

   // alternativ (zeigt weitere Optional.Erzeuger):
   //    if(this.anzahlVon(prod) == null) {
   //      return Optional.empty();
   //    }
   //    return Optional.of(this.anzahlVon(prod));
 }
```

- Optional in java.util

Optional (5/5) – Problemlösung (2/2)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

```
public static void main(String[] args) {
   Einkaufsliste ek = new Einkaufsliste();
   ek.hinzu("Bier", 3);
   ek.hinzu("Wasabi", 5);
   ek.hinzu("Bier", 6);
   System.out.println("ek: " + ek);
   System.out.println("Bier " + ek.anzahl("Bier").orElse(0));
   System.out.println("Beer " + ek.anzahl("Beer").orElse(0));
   int moreBeer = ek.anzahl("Beer").orElse(0) + 1;
   ek.anzahl("Bier")
     .ifPresent(b -> System.out.println(b + " mal da"));
   if (!ek.anzahl("Beer").isPresent()) {
     System.out.println("no beer");
   }
 }
```

```
ek: Einkaufsliste [produkte={Bier=9, Wasabi=5}]
Bier 9
Beer 0
9 mal da
no beer
```

Streams ab Java 8

Video

- Streams ab Java 8 sind gutes Beispiel zum Method Chaining (hier genauer Fluent Programming)
- allerdings werden Methoden pro Stream-Objekt abgearbeitet
- Sammlungen werden als Streams (Folgen) von Objektreferenzen angesehen
- Viele Stream-Methoden liefern wieder ein Stream-Objekt als Ergebnis
- Beispiele: Filtermethoden, Umwandlungen
- Streams kurzlebig, nur einmal nutzbar (dann wieder erstellbar)
- Hier nur kurzes Konzept (gibt weitere Methoden, zusätzliche Stream-Klassen, …)
- Hier auch weitere Nutzung von Lambda-Ausdrücken

Streams (1/14): POJO-Klasse (1/2)


```java
public class Studierend {
  private int matnr;
  private String name;

  public Studierend(){ // Default-Konstruktor
  }

  public Studierend(int matnr, String name) {
    this.matnr = matnr;
    this.name = name;
  }

  public int getMatnr() {
    return this.matnr;
  }
```

Streams (2/14): POJO-Klasse (2/2)

```
  public void setMatnr(int matnr) {
    this.matnr = matnr;
  }

  public String getName() {
    return this.name;
  }

  public void setName(String name) {
    this.name = name;
  }

  @Override
  public String toString(){
    return this.name + " (" + this.matnr +")";
  }
} // sinnvoll: equals und hashCode
```

Streams (3/14): Ausführungsrahmen


```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import java.util.stream.Stream;

import entity.Studierend;

public class Main {

private List<Studierend> studierende = new ArrayList<>();

public static void main(String[] args) {
  Main m = new Main();
  m.generate(10);
  m.show1(); // hier zu untersuchende Methode
}
```

Z. B.: http://www.angelikalanger.com/Articles/EffectiveJava/75.Java8.Fundamental-Stream-Operations/75.Java8.Fundamental-Stream-Operations.html

Streams (4/14): Erzeugung und einfache Nutzung

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

```
public void generate(int anzahl){
  for(int i = 0; i < anzahl; i = i + 2){
    this.studierende.add(new Studierend(i, "Ute"+i));
    this.studierende.add(new Studierend(i+1, "Udo"+i));
  }
}

public void show1(){
  this.studierende
      .forEach(s -> System.out.println(s));
}

// Hinweis: ist aequivalent zu
// this.studierende
//       .stream()
//       .forEach(s -> System.out.println(s));
```

```
Ute0 (0)
Udo0 (1)
Ute2 (2)
Udo2 (3)
Ute4 (4)
Udo4 (5)
Ute6 (6)
Udo6 (7)
Ute8 (8)
Udo8 (9)
```

Streams (5/14): Lambda - Beispiele


```
public void lambda(){
  this.studierende
      .forEach((Studierend s) -> {System.out.println(s);});
  // wenn Typen eindeutig, dann weglassen
  this.studierende.forEach((s) -> {System.out.println(s);});
  // nur ein Parameter, dann keine Klammern
  this.studierende.forEach( s -> {System.out.println(s);});
  // nur ein Ausdruck oder eine Zeile, dann keine Klammern
  this.studierende.forEach( s -> System.out.println(s));
  // wenn Objekt s Parameter der einzige aufgerufenen Methode
  this.studierende.forEach(System.out::println);
}
```

Streams (6/14): Möglichkeit zur Parallelisierung

```
public void show2(){ // Parallelisierung
  this.studierende
      .parallelStream()
      .forEach(s -> System.out.println(s));
}
```

```
// Hinweis: jede Collection in Stream
// verwandelbar, z. B.
// int[] arr= {9,7,3,1};
// Arrays.stream(arr)
//        .forEach(i -> System.out.println(i));
```

```
Ute6 (6)
Udo4 (5)
Udo6 (7)
Ute0 (0)
Ute2 (2)
Udo0 (1)
Udo2 (3)
Ute4 (4)
Ute8 (8)
Udo8 (9)
```

Streams (7/14): Filterung


```
public void show3(){
  this.studierende
      .stream()
      .filter(s -> s.getMatnr()% 3 == 0)
      .filter(s -> s.getMatnr()% 2 == 1)
      .forEach(s -> System.out.println(s));
}

// generell jede Boolesche Methode
// zum Filtern nutzbar
```

```
Udo2 (3)
Udo8 (9)
```

Streams (8/14): Filterung genauer (Einschub)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- Parameter in Stream-Methoden normale Objekte

```
public static Predicate<Integer> teiler(int val){
  return x -> x%val == 0;
}

public static void main(String[] args) {
  Stream<Integer> str = Stream.of(1,3,8,4,5,1,6);
  Predicate<Integer> pred1 = x -> x%2 == 0;
  str.filter(pred1)
     .forEach(System.out::println);
  System.out.println(pred1.test(42));
  // kein sinnvoller Zugriff auf str mehr moeglich
  str = Stream.of(1,3,8,4,5,1,6);
  str.filter(teiler(3))
     .forEach(System.out::println);
}
```

```
8
4
6
true
3
6
```

Streams (9/14): Abbildung / Umwandlung (map)

Video

```
public void show4(){
   this.studierende
       .stream()
       .filter(s -> s.getMatnr()% 3 == 0)
       .filter(s -> s.getMatnr()% 2 == 1)
       .map(s -> s.getMatnr() + ": " + s.getName())
       .forEach(s -> System.out.println(s));
 }
// Map-Ergebnis kann Objekt beliebigen Typs sein
// z. B. auch Object-Array
// man sieht auch mehrzeile Funktion mit Rückgabe
//    .map(s -> {
//        String[] erg = {s.getName(), ""+s.getMatnr()};
//        return erg;
//     })
```

```
3: Udo2
9: Udo8
```

Streams (10/14): Detailanalyse

```
public void show5(){
  this.studierende
      .stream()
      .peek(s -> System.out.println(s))
      .filter(s -> s.getMatnr()%3 == 0)
      .peek(s -> System.out.println(s))
      .filter(s -> s.getMatnr()%2 == 1)
      .peek(s -> System.out.println(s))
      .map(s -> s.getMatnr() + ": " + s.getName())
      .peek(s -> System.out.println(s))
      .forEach(s -> System.out.println("------"));
}
// logisch nacheinander abgearbeitet
// für Performance und Parallelität, startet
// Bearbeitung pro Objekt mit terminaler Methode
// peek sieht Objekt, konsumiert es nicht
// !! kein reines Method Chaining !!
```

```
Ute0 (0)
Ute0 (0)
Udo0 (1)
Ute2 (2)
Udo2 (3)
Udo2 (3)
Udo2 (3)
3: Udo2
------
Ute4 (4)
Udo4 (5)
Ute6 (6)
Ute6 (6)
Udo6 (7)
Ute8 (8)
Udo8 (9)
Udo8 (9)
Udo8 (9)
9: Udo8
------
```

Streams (11/14): Lazy Evaluation

```
public Set<Studierend> showLazy() {
  return this.studierende
        .stream()
        .filter(s -> {
          System.out.println(s);
          return s.getMatnr() > 2 && s.getMatnr() < 9;
        })
        .skip(2)
        .limit(3)
        .collect(Collectors.toSet());
}

... // in Main
System.out.println(m.showLazy());
```

```
Ute0 (0)
Udo0 (1)
Ute2 (2)
Udo2 (3)
Ute4 (4)
Udo4 (5)
Ute6 (6)
Udo6 (7)
[Udo4 (5), Udo6
(7), Ute6 (6)]
```

Streams (12/14): Zusammenfassung (reduce)

```
public void show6(){
  System.out.println(
    this.studierende
        .stream()
        .filter(s -> s.getMatnr()%3 == 0)
        .filter(s -> s.getMatnr()%2 == 1)
        .map(s -> s.getMatnr() + ": " + s.getName())
        .reduce("Studis:", (s1,s2) -> s1 + ", " + s2)
  );
}
```

```
Studis:, 3: Udo2, 9: Udo8
```

```
// reduce macht Schleife über alle Stream-Objekte
// s1: bisheriges Ergebnis (initial "Studis")
// s2: aktuelles Objekt aus dem Stream
```

Streams (13/14): Gruppierung

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

```
public void show7(){
  Map<Integer,List<Studierend>> aufgeteilt =
       this.studierende
           .stream()
           .collect(Collectors
                        .groupingBy(s -> s.getMatnr() % 3));
  aufgeteilt
       .forEach((k,v) -> System.out.println(k + ": " +v));
}
```

```
0: [Ute0 (0), Udo2 (3), Ute6 (6), Udo8 (9)]
1: [Udo0 (1), Ute4 (4), Udo6 (7)]
2: [Ute2 (2), Udo4 (5), Ute8 (8)]
```

```
// collect liefert Map mit Ergebniswert als key und Liste
// zugehöriger Objekte als value
```

Streams (14/14): viele weitere Möglichkeiten

```
public static void main(String[] args) {
  IntStream.range(1, 4).forEach(System.out::println);

  double d = IntStream.range(1, 4)
                      .average()
                      .orElse(42);
  System.out.println("d: " + d);

  double d2 = IntStream.range(1, 1)
                      .average()
                      .orElse(0);
  System.out.println("d2: " + d2);

  IntStream.range(1, 4)
    .mapToObj(p -> new int[]{p, p*p})
    .forEach( a -> System.out.println(a[0] + " " + a[1]));
}
```

```
1
2
3
d: 2.0
d2: 0.0
1 1
2 4
3 9
```

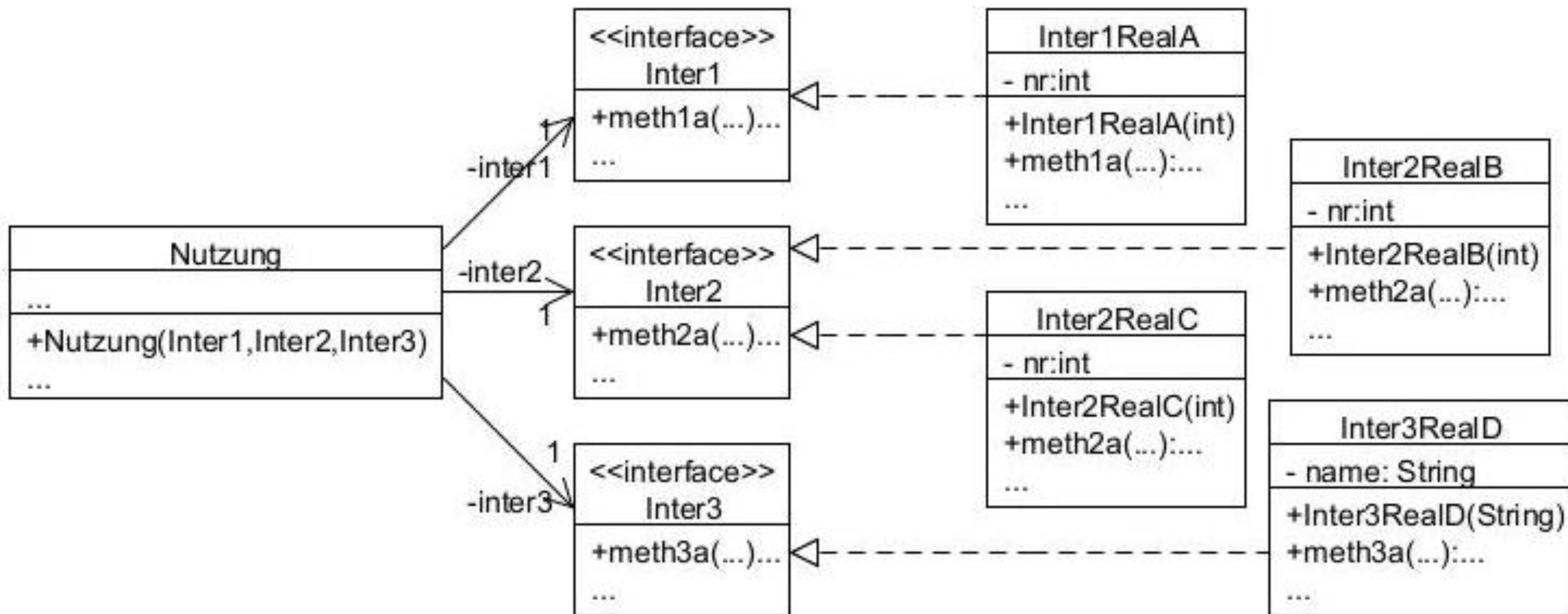
Dependency Injection

Video

woher kommen Objekte für Exemplarvariablen?

- Variante 1: Werte werden als Parameter übergeben, aus denen Objekte gebaut werden
- Variante 2: Objekte werden als Referenzen übergeben
  - Optimierung: Typen der Objektvariablen sind Interfaces; so konkrete Objekte leicht austauschbar
- Variante 2 heißt Dependency Injection mit get- und set-Methoden oder über Konstruktoren
- gutes Video: https://www.youtube.com/watch?v=IKD2-MAkXyQ
- Standard-Framework: CDI (Contexts and Dependency Injection, JSR-365, https://docs.jboss.org/cdi/spec/2.0/cdi-spec.html)

Dependency Injection - Beispiel




```
Nutzend nutzend = new Nutzend(new Inter1RealA(42)
                             , new Inter2RealC(43)
                             , new Inter3RealD("Hallo"));
```

CDI – Minibeispiel (1/4)

- Ein Klasse Nutzend, zentrales Objekt, wird in mehreren Klassen benötigt (soll hier Singleton sein; nur als Beispiel)

```
@Singleton // CDI-Anntotation
public class Nutzend {

  private int rechte = 42;
  private String name = "Douglas";

  public int getRechte() {return this.rechte;}
  public void setRechte(int rechte) {this.rechte = rechte;}
  public String getName() {return name;}
  public void setName(String name) {this.name = name;}

  @Override
  public String toString() {
    return "Nutzend [rechte="+rechte+", name=" +name+"]"; }
}
```

CDI – Minibeispiel (2/4)

```
public class ControllerA {
  @Inject
  private Nutzend nutzend;
  @PostConstruct
  public void initialize() {
      System.out.println("startA");
  }
  @PreDestroy
  public void cleanup() {
      System.out.println("endeA");
  }
  public void aendereRechte(int wert) {
      this.nutzend.setRechte(wert);
  }
  public Nutzend getNutzend() { return this.nutzend; }
}
```

CDI – Minibeispiel (3/4)

```
public class ControllerB {

  @Inject
  private Nutzend nutzend;

  @PostConstruct
  public void initialize() {
      System.out.println("startB");
  }

  @PreDestroy
  public void cleanup() {
      System.out.println("endeB");
  }

  public void aendereName(String wert) {
      this.nutzend.setName(wert);
  }
  public Nutzend getNutzend() { return this.nutzend; }
}
```

CDI – Minibeispiel (4/4)


```
public static void main(String[] args) {
  Weld weld = new Weld();
  try (WeldContainer wC = weld.initialize()) {
    ControllerA ca = wC.select(ControllerA.class).get();
    System.out.println("A: " + ca.getNutzend());
    ca.aendereRechte(41);
    ControllerB cb = wC.select(ControllerB.class).get();
    cb.aendereName("Dirk");
    System.out.println("B: " + cb.getNutzend());
  }
}
```

```
startA
A: Nutzend [rechte=42, name=Douglas]
startB
B: Nutzend [rechte=41, name=Dirk]
endeA
endeB
```

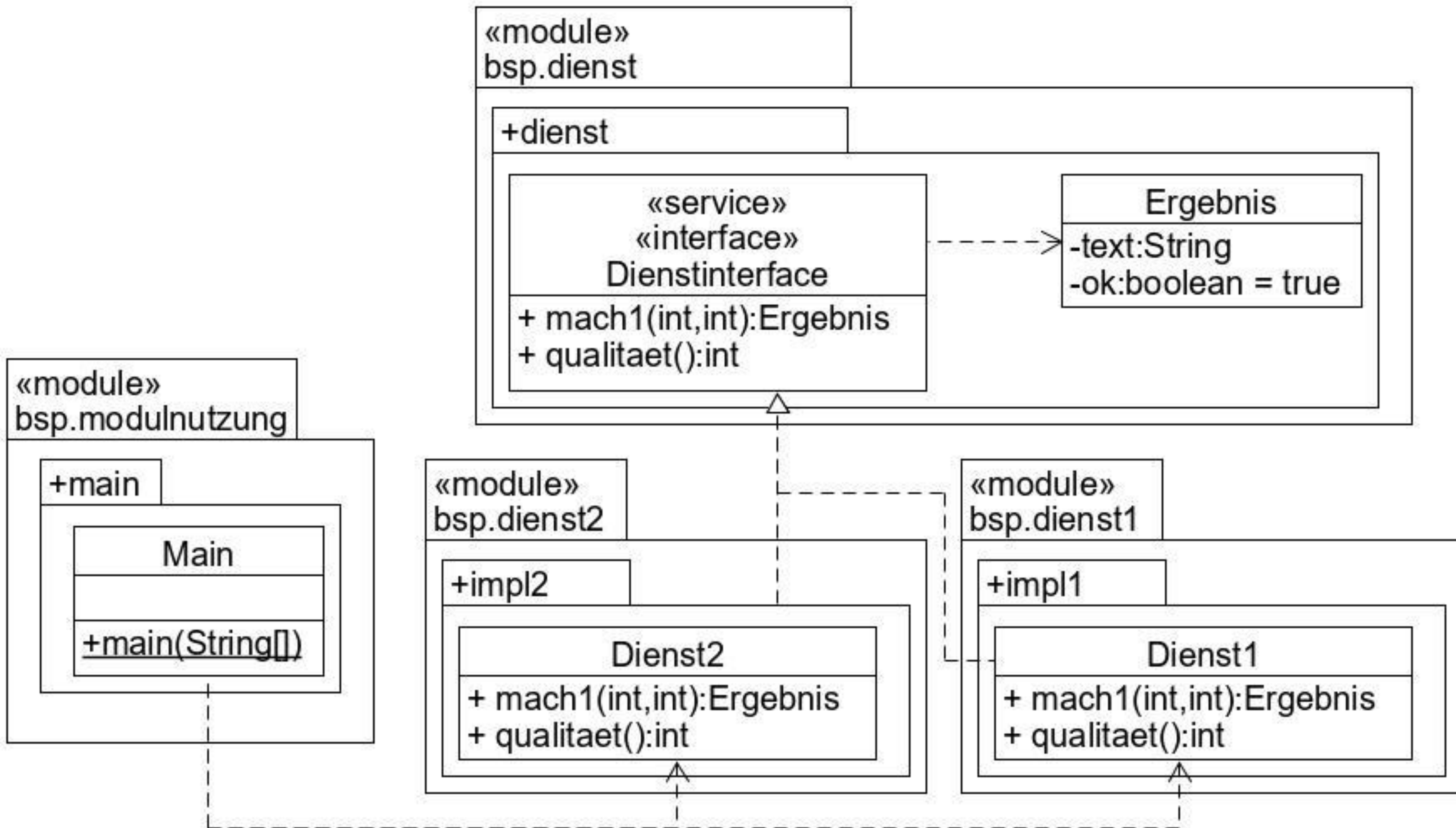
Java Module (1/5) – Services – Beispiel für Strategy

Video

- neben dem vorgestellten Modulansatz unterstützt das Java-Modulsystem Services (ursprüngliches Konzept ab Java 6)
- Service ist zunächst einfaches Interface (z. B. DienstInterface), zugehöriges Paket mit „**`exports`**“
- Service-Realisierer, z. B. Dienst1 realisieren Interface
  - benötigt „**`requires`**“ Modul mit Interface
  - hat parameterlosen Konstruktor
  - kennzeichnet die Dienstrealisierung
  - **`provides DienstInterface with Dienst1`**
- Dienstnutzungen müssen dies kennzeichnen
  - **`uses DienstInterfaces`**
  - JVM ermöglicht über alle vorhandenen Implementierungen zu iterieren und zu nutzen

Java Module (2/5) - Beispiel

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES



Java Module (3/5) – module-info.java Dateien

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

```
module bsp.dienst {
  exports dienst;
}

module bsp.dienst1 {
  requires transitive bsp.dienst;
  exports impl1;
  provides dienst.DienstInterface
      with impl1.Dienst1;
}

module bsp.dienst2 {
  requires transitive bsp.dienst;
  exports impl2;
  provides dienst.DienstInterface
      with impl2.Dienst2;
}
```

```
module bsp.modulnutzung {
  requires bsp.dienst1;
  requires bsp.dienst2;
  uses
   dienst.DienstInterface;
}
```

Java Module (4/5) - Nutzung

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

```
public class Main {
  public static void main(String[] args) {
    ServiceLoader<DienstInterface> sl
          = ServiceLoader.load(DienstInterface.class);
    for(DienstInterface di: sl) {
      Ergebnis erg = di.mach1(1, 42);
      System.out.println("Service: " + di.getClass()
          +" Q: " + di.qualitaet()
          + " erg: " + erg.getText());
    }
  }
}
```

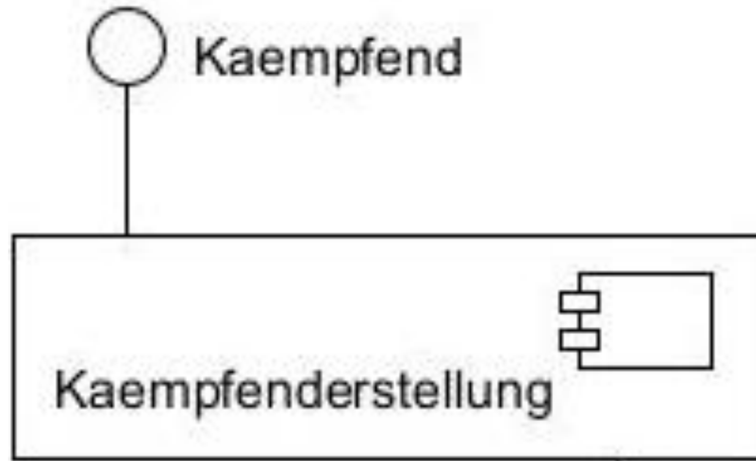
```
Beispielausgabe:
Service: class impl2.Dienst2 Q: 42 erg: 42
Service: class impl1.Dienst1 Q: 20 erg: 43
```

- es wird immer nur ein Service pro Ausführung erstellt (Singleton)
- es werden keine Objekte direkt erstellt (kein new; zwar erlaubt, verstößt aber gegen Konzept)

Java Module (5/5) – Variante Factory

```
module bsp.dienstfactory {
  requires transitive bsp.dienst;
  exports impl3;
  provides dienst.DienstInterface with impl3.DienstFactory;
}

public class DienstFactory {
  public static DienstInterface provider() {
      return new DienstInterface() {
            @Override
            public Ergebnis mach1(int arg0, int arg1) {
                  return new Ergebnis();
            }

            @Override
            public int qualitaet() { return 100; }
      };
  }
}
```

Nutzung von Komponenten




```
public class KaempfenderstellungFactory {
  public static Kaempfend erzeugen(typ:String, auswahl:int)
}
```

- Komponente: konfigurierbare, übersetzte Software, die klare Funktionalität anbietet
- Beispiel: Komponente bietet Kämpfer-Objekte an
- benötigt Klasse, die die Erzeugung ermöglicht

Validierungs-Framework (Bean Validation 1.1, JSR 349)


```
public abstract class Kaempfend {
    @Min(value=0, message = "Gesundheit nicht negativ")
    protected int gesundheit;

    @Min(value=3, message="minimale Staerke beachten")
    @Max(value= 15, message="maximale Staerke beachten")
    protected int staerke;

    @Min(value=5, message="minimales Geschick beachten")
    @Max(value= 20, message="maximales Geschick beachten")
    protected int geschick;

    // wie vorher
```

Beispielnutzung der Validierung

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

```
public static void main(String[] s) {
  AbstractKaempfendFactory kf = KaempfendArtFactory
            .kaempfendFactoryErstellen("basic");
  Kaempfend k = kf.kaempfendErstellen(2);
  k.setGesundheit(-1);
  k.setGeschick(22);
  ValidatorFactory factory = Validation
             .buildDefaultValidatorFactory();
  Validator validator = factory.getValidator();
  for (ConstraintViolation<Kaempfend> c :
                               validator.validate(k)) {
    System.out.println(" :: " + c.getMessage());
  }
}
```

```
 :: maximales Geschick beachten
 :: minimale Staerke beachten
 :: Gesundheit nicht negativ
```

Persistenz-Framework (JPA 2.1, JSR 338)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

```
@Entity
public abstract class Kaempfend {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    protected int knr;
    @Min(value = 0, message = "Gesundheit nicht negativ")
    protected int gesundheit;
    @Min(value = 3, message = "minimale Staerke beachten")
    @Max(value = 15, message = "maximale Staerke beachten")
    protected int staerke;
    @Min(value = 5, message = "minimales Geschick beachten")
    @Max(value = 20, message = "maximales Geschick beachten")

    // wie vorher, konkrete Klassen auch mit @Entity annotiert
```

Beispielnutzung der Persistenz


```
public static void main(String[] s) {
  AbstractKaempfendFactory kf = KaempfendArtFactory
                  .kaempfendFactoryErstellen("basic");
  Kaempfend k = kf.kaempfendErstellen(2);
  k.setGesundheit(100);
  k.setStaerke(7);
  k.setGeschick(9);
  EntityManagerFactory emf = Persistence
                  .createEntityManagerFactory("KaempfendPU");
  EntityManager em = emf.createEntityManager();
  em.getTransaction().begin();
  em.persist(k);
  em.getTransaction().commit();
  em.close();
  emf.close();
}
```

```
SELECT * FROM Kaempfend
```

KNR	DTYPE	GESCHICK	GESUNDHEIT	STAERKE
1	Xena	9	100	7

Video

7. Konkretisierungen im Feindesign

- 7.1 Zustandsdiagramme
- 7.2 Object Constraint Language

Verfeinerte Modellierung

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- Durch die verschiedenen Sichten der Systemarchitektur wird der Weg vom Anforderungsmodell zur Implementierung beschrieben

- Es bleiben offene Themen:
  - Wie bekomme ich ein gutes Klassendesign (nächstes Kapitel)?
  - Wie kann man das komplexe Verhalten von Objekten noch beschreiben (Klassendiagramme sind statisch, Sequenzdiagramme exemplarisch)?

    Antwort: Zustandsdiagramme
  - Wie kann man bei der Klassenmodellierung Randbedingungen formulieren, was in Klassendiagrammen (Bedingungen in geschweiften Klammern) nur bedingt möglich ist?

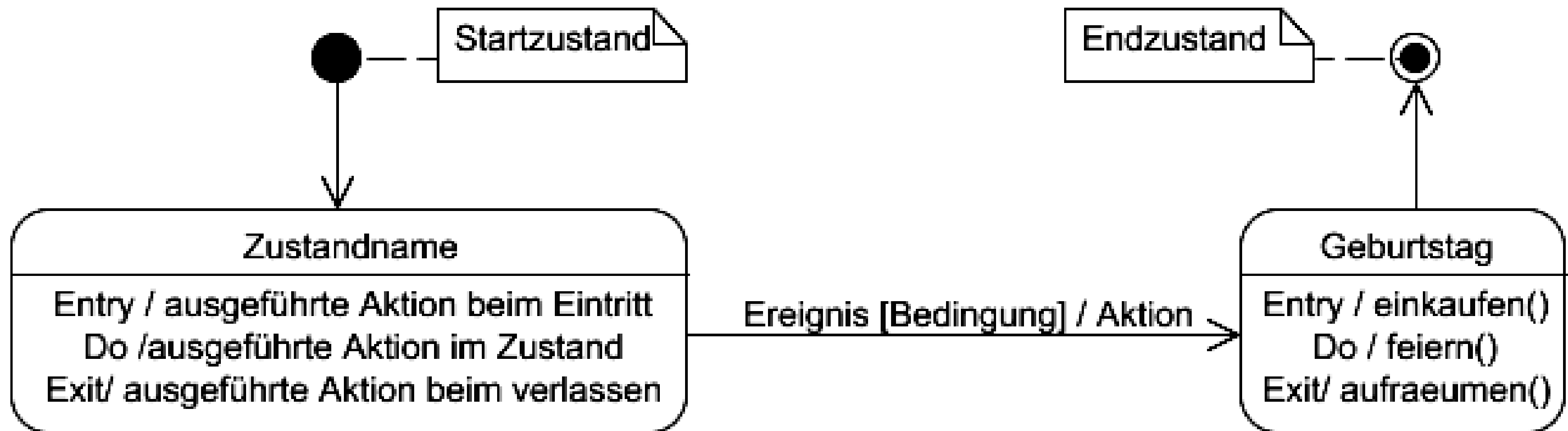
    Antwort: Object Constraint Language

Zustandsdiagramme

7.1

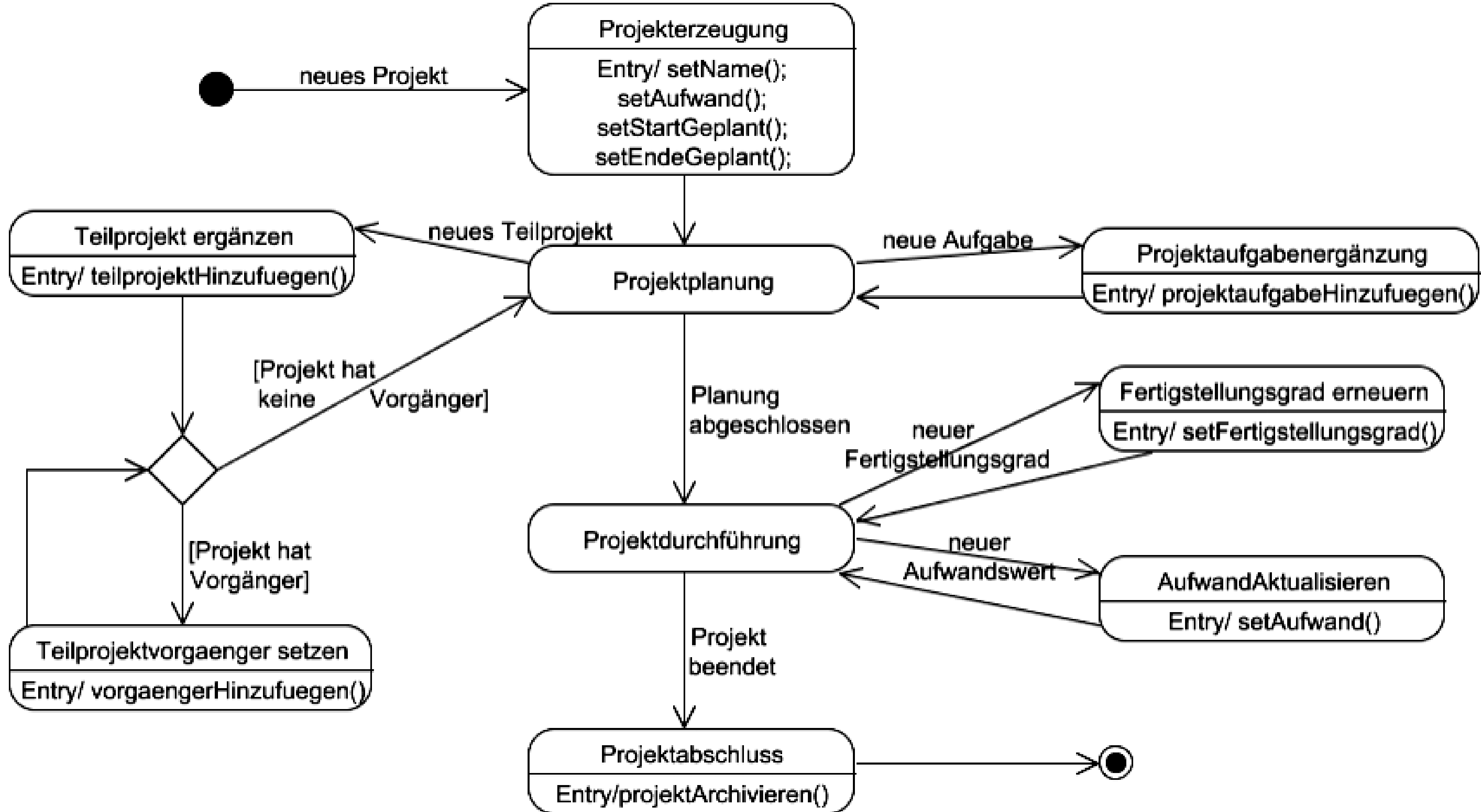
- generell wird der Zustand eines Objekts durch die Werte seiner Exemplar- und Klassenvariablen beschrieben
- Häufig wird der Begriff Zustand auch für eine spezielle Exemplarvariable genutzt, die z. B. über eine Enumeration realisierbar ist
- z. B. : Ampel: rot, rotgelb, gelb, grün
- z. B. : Projekt: vorbereitet, grob geplant, mitarbeitende Personen zugeordnet, verschoben, in Bearbeitung, in Endabnahme, in Gewährleistung, beendet
- Übergänge zwischen den Zuständen werden durch Ereignisse, zumeist Methodenaufrufe, veranlasst
- Übergänge lassen sich durch ein Zustandsdiagramm (ursprünglich Statechart nach D. Harel) spezifizieren
- Zustandsautomaten spielen auch in der theoretischen und technischen Informatik eine zentrale Rolle

Struktur von Zustandsdiagrammen



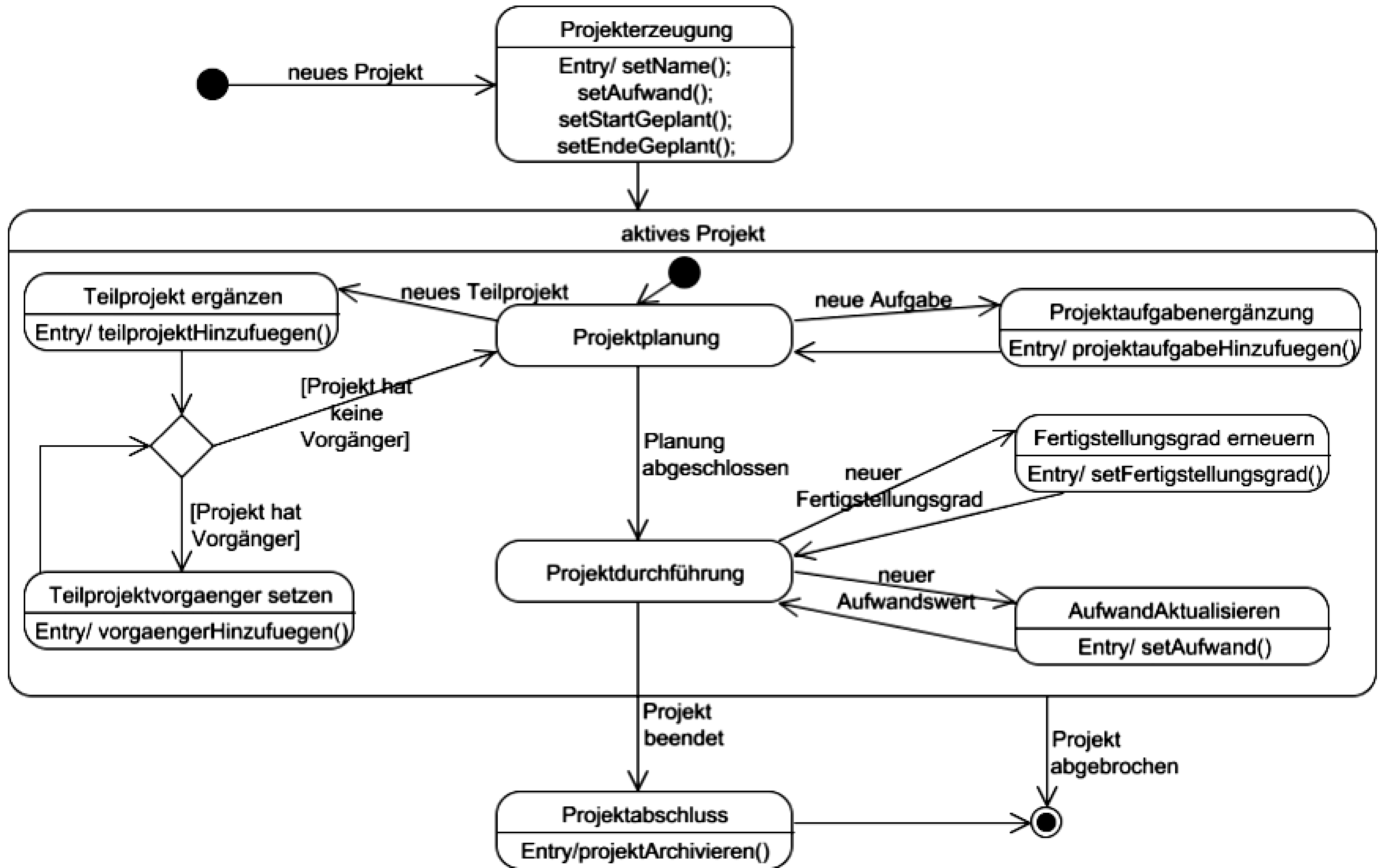
- Zustandsdiagramm gehört zu einem Objekt einer Klasse
- alle Angaben für Zustände und Transitionen sind optional
- Transition wird ausgeführt, wenn Ereignis eintritt und Bedingung erfüllt ist
- ohne Ereignis und Bedingung wird Transition dann ausgeführt, wenn Entry, Do, und Exit durchlaufen
- Einfacher Automat muss deterministisch sein

Beispiel: Zustandsdiagramm eines Projekts



- man erkennt: nach Planung keine Planungsänderung

Hierarchische Zustände

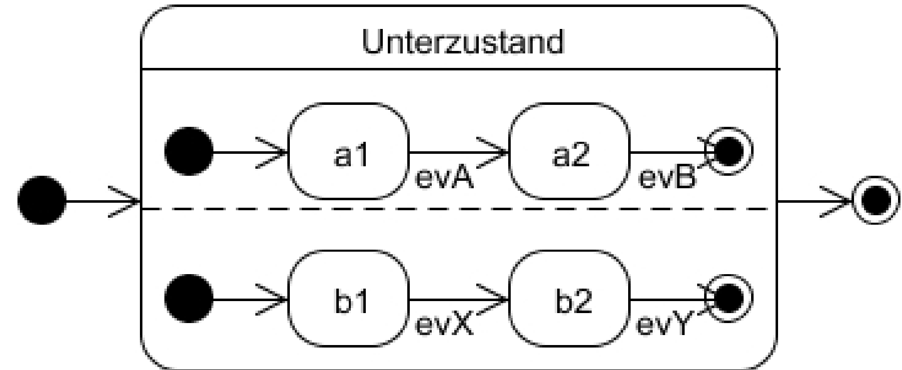


Parallele Unterzustände

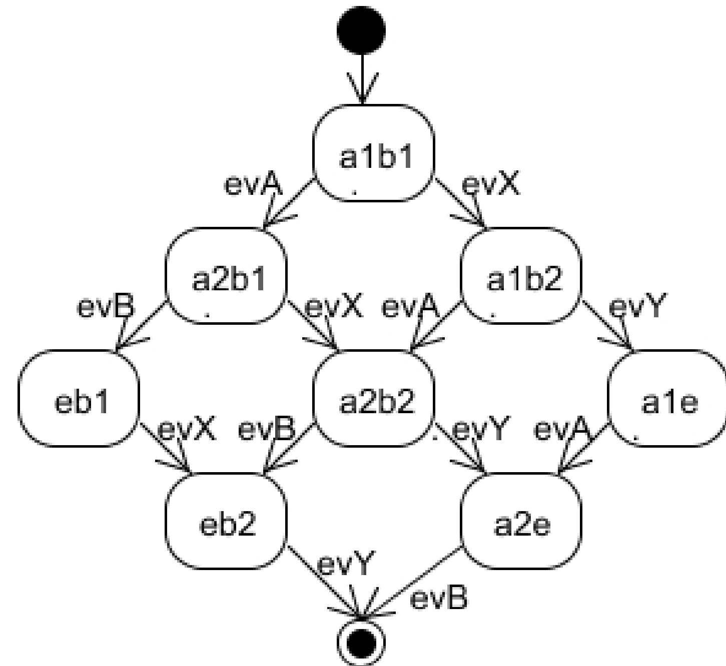
HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

Video

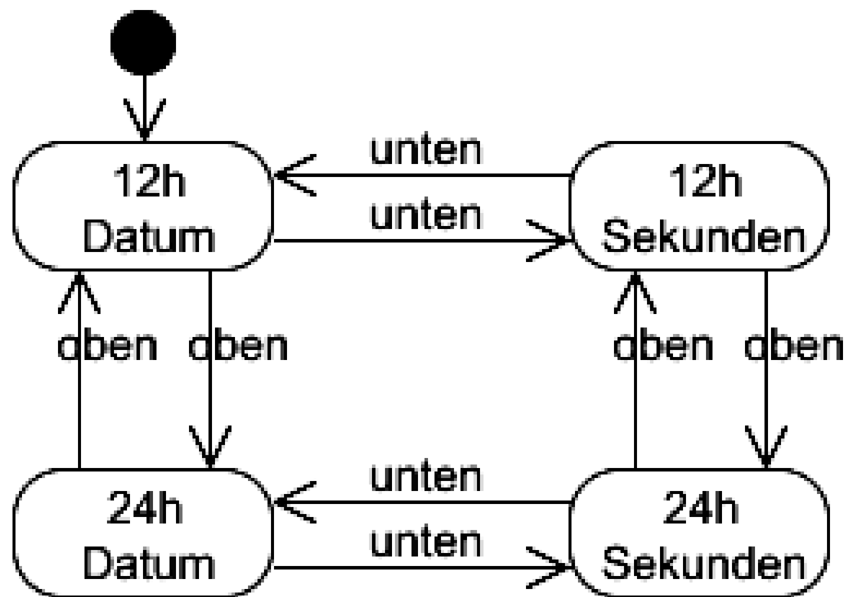
- unabhängige Teilzustände können in parallelen Zuständen bearbeitet werden

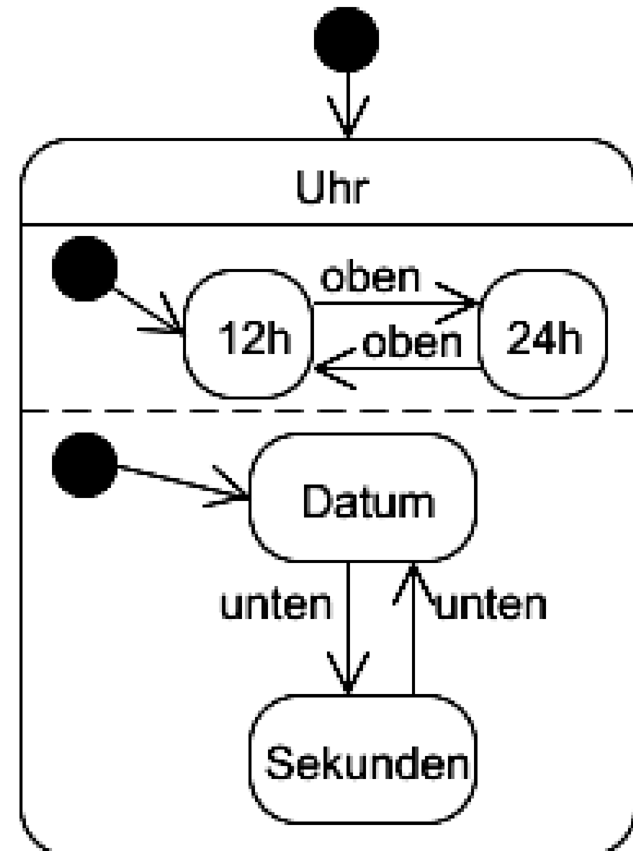


- ohne Parallelität müsste Kreuzprodukt der Zustände der parallelen Automaten betrachtet werden



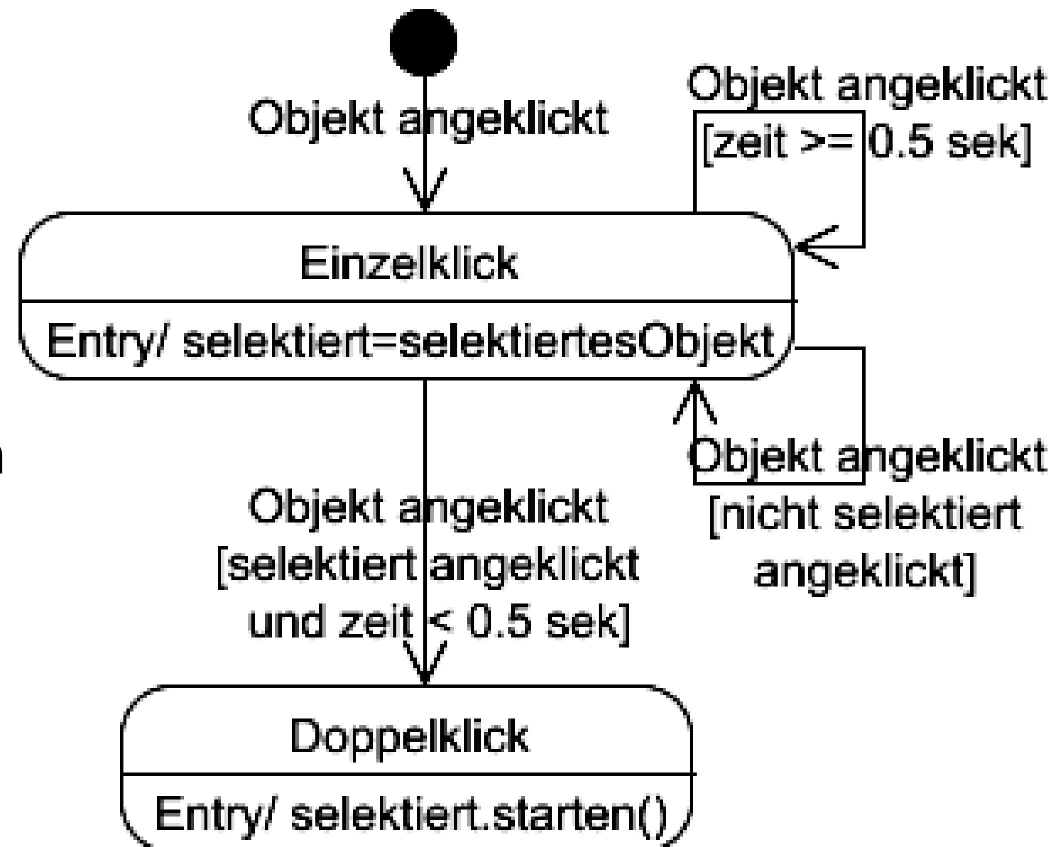
Beispiel: Uhr





Zustandsmodellierung und Realzeitsysteme

- in klassischen OO-Programmen gibt es meist wenige zentrale Klassen, für die sich eine Zustandsmodellierung anbietet
- In Systemen mit Zeit kann Zustandsmodellierung Zeitbedingungen beinhalten
- auch warte(5 sek)



Event [Condition] / Action

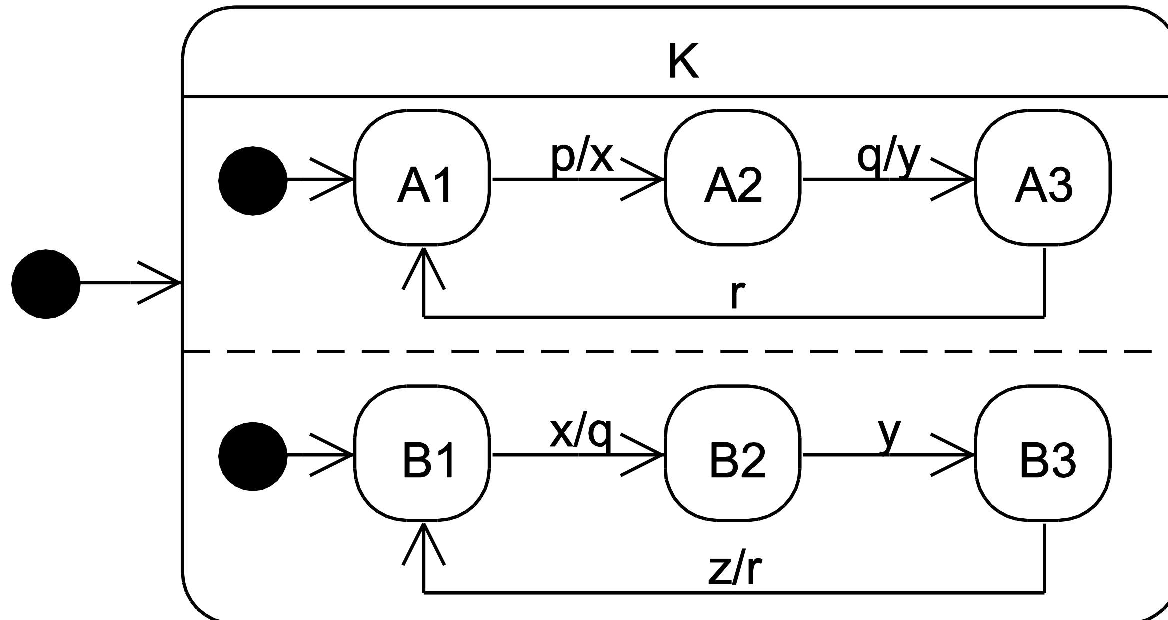
- Transitionsbeschriftung Ereignis[Bedingung]/Aktion
- Was ist Ereignis? Hängt von Applikation ab
  - Methodenaufruf
  - Ereignis im Programm (Variable wechselt Wert)
  - technische Systeme: Signale

typisches Beispiel: Steuersysteme

- erhalten Signale (->Ereignisse) von Sensoren wenn etwas passiert (z. B. ein-/ausgeschaltet)
- lesen Werte anderer Sensoren, Teilsysteme (-> Bedingung), die Entscheidungen beeinflussen
- senden Signale (-> Aktion) an andere Systeme

Microsteps und Macrosteps (1/2)

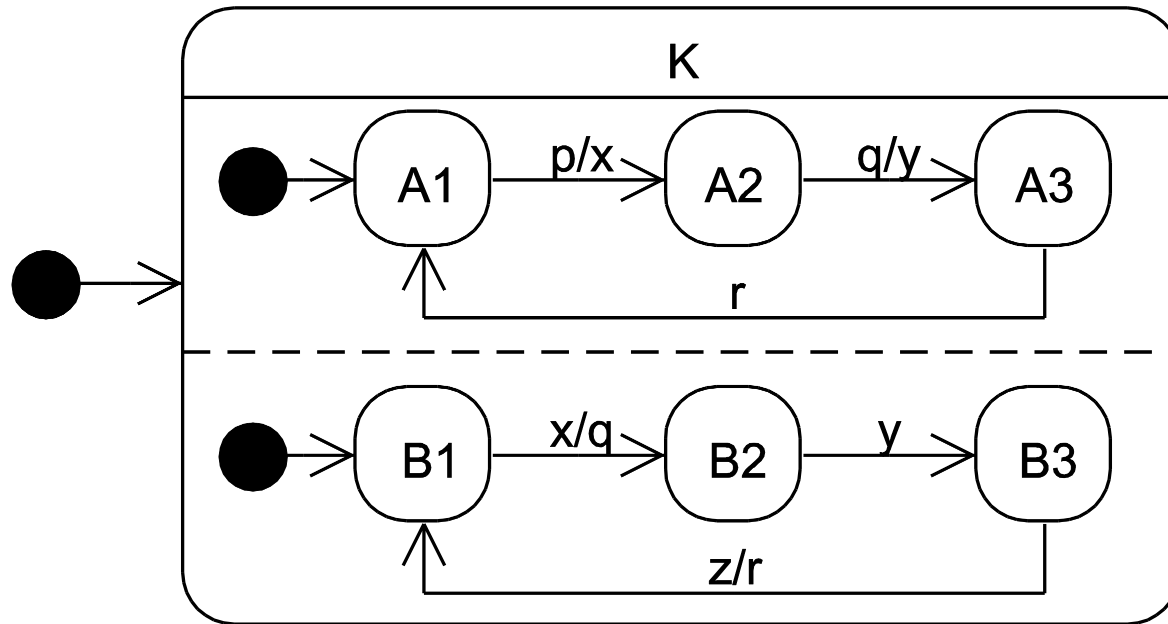
- Actions eines Teilautomaten können Events eines anderen Teilautomaten sein



- Microstep: einzelne Schritte betrachten

  Start -> K(A1,B1) –p-> K(A2,B1) –x-> K(A2,B2) –q-> K(A3,B2) –y-> K(A3,B3) –z-> K(A3,B1) –r-> K(A1,B1)

Microsteps und Macrosteps (2/2)



- Macrostep: nur Zustände nach vollständiger Bearbeitung betrachten (Ausnahme: Livelock)

  Start -> K(A1/B1) –p-> K(A3/B3) –z-> K(A1/B1)

- typischerweise nur an Macrosteps interessiert

Beispiel: Start-Stopp-Automatik (1/4)

Video

- zentrale Aufgabe: Start-Stopp-Automatik stellt den Motor immer dann selbstständig aus, wenn dieser nicht mehr benötigt wird (z. B. Halt an Ampel)
- Randbedingung: keine Abschaltung bis maximal 3 Grad und ab minimal 30 Grad
- Ablauf:
  - Zündschlüssel einstecken, Motorstartknopf drücken, dann startet Automatik
  - Motorein- und Abschaltung wird anhand der Kupplung erkannt
  - Automatik kann auch wieder gestoppt werden
- [Frage: was fehlt alles zur Realität]

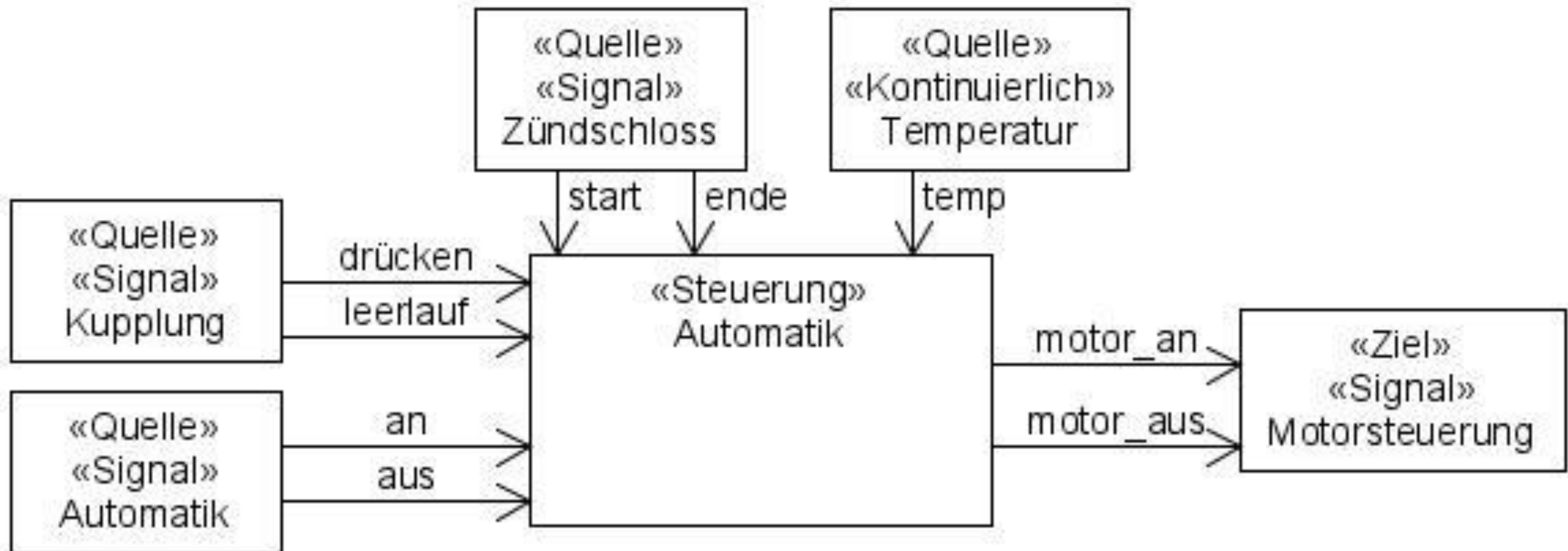
Beispiel: Start-Stopp-Automatik (2/4)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

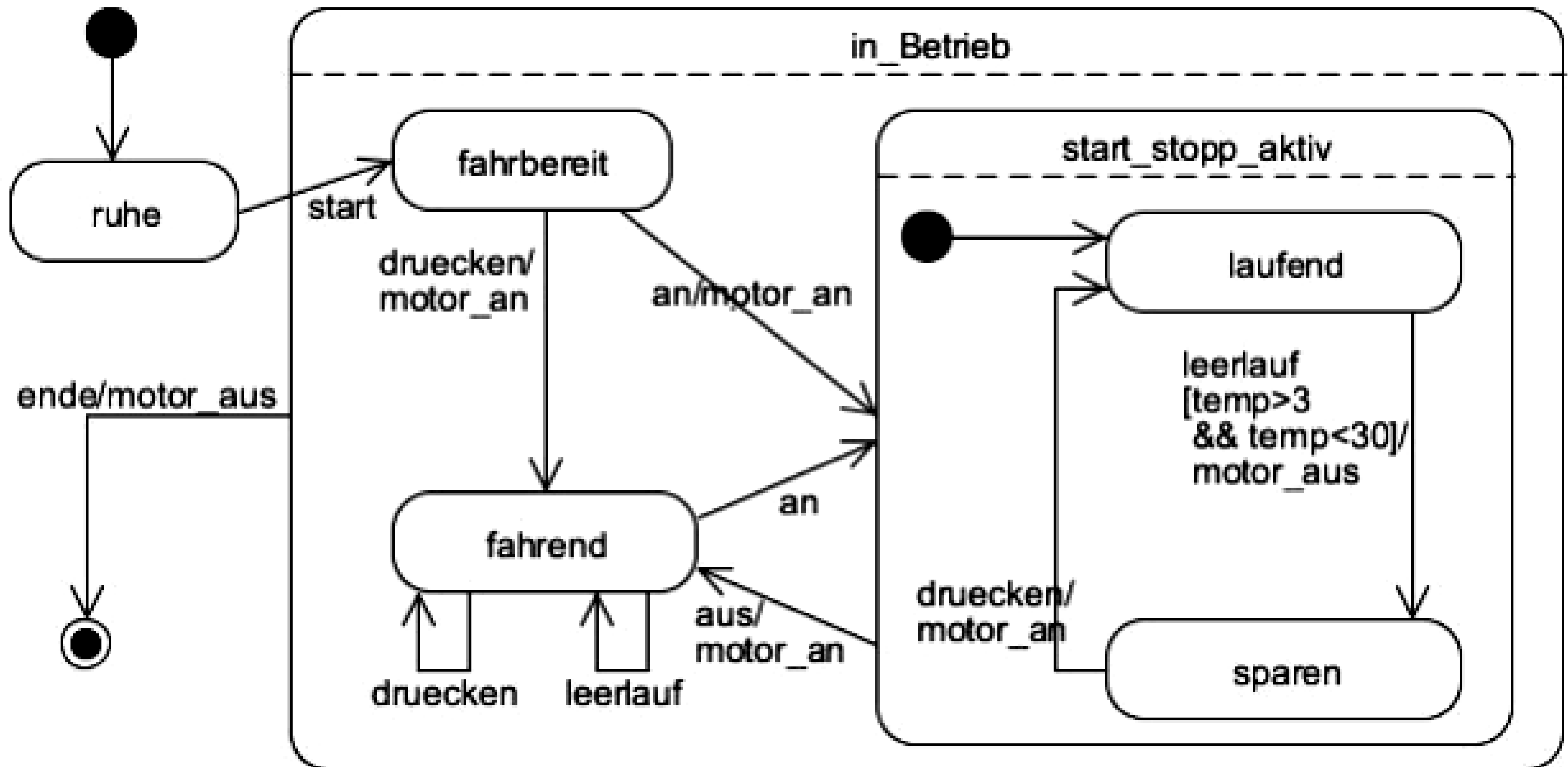
- Klärung, von welche Sensoren werden Signale empfangen:
  - Zündschloss: start und ende
  - Kupplung: leerlauf und druecken
  - Automatiksteuerung: an und aus
- Klärung, welchen Sensoren können abgefragt werden:
  - Temperaturwert temp in lokaler Variablen
- Klärung an welche Aktoren Signale geschickt werden
  - Motorsteuerung: motor_an und motor_aus

Beispiel: Start-Stopp-Automatik (3/4)

„Blockschaltbild“



Beispiel: Start-Stopp-Automatik (4/4)

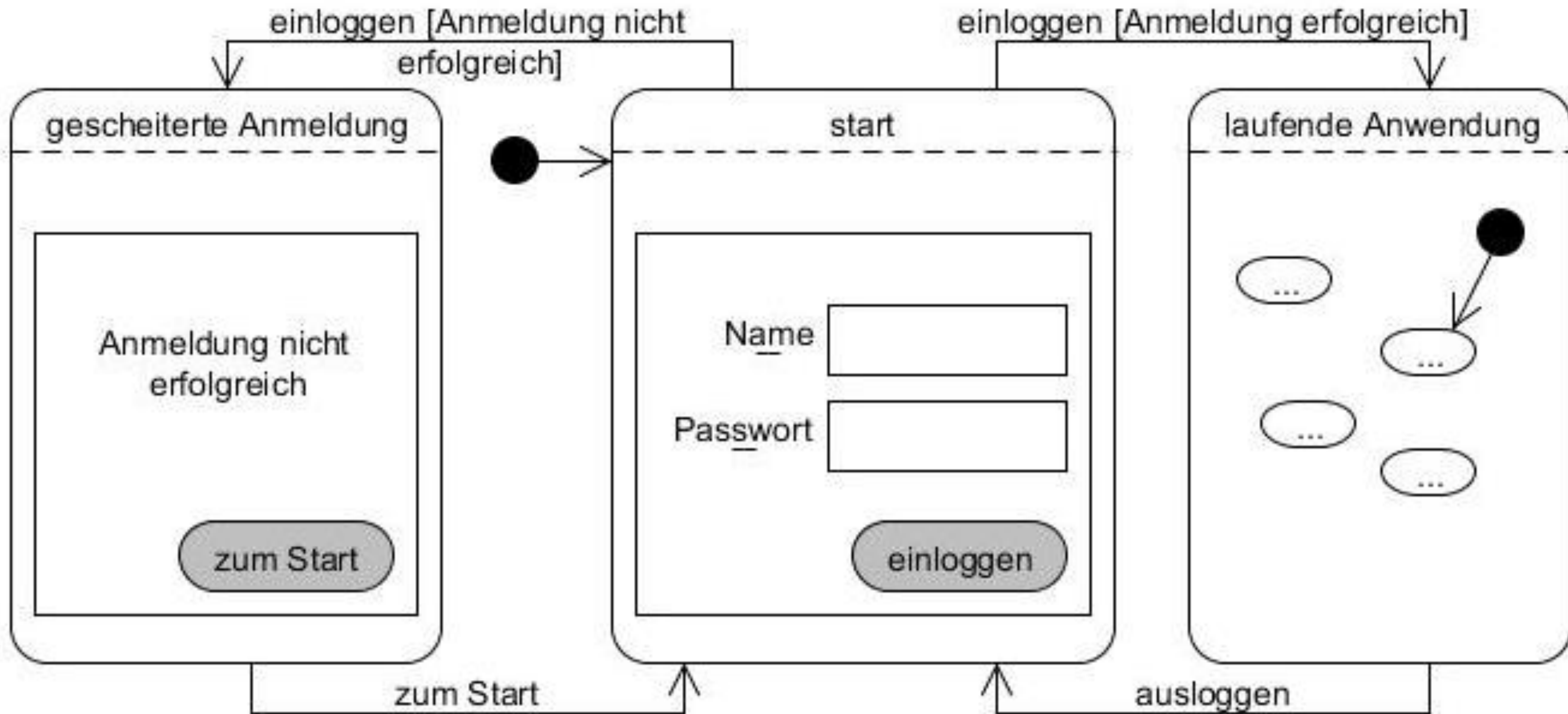


Umsetzung von Zustandsdiagrammen

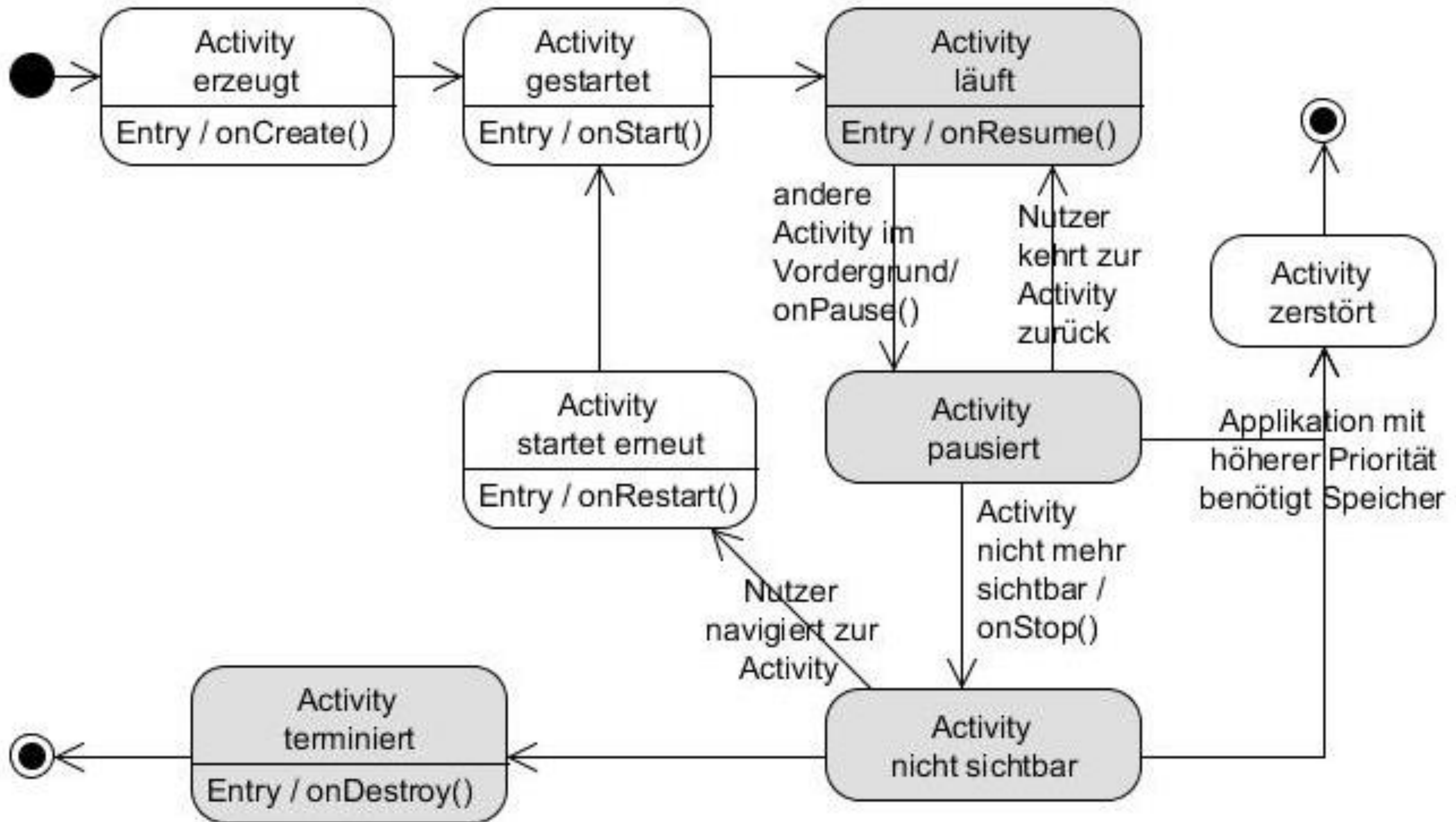
HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- Abhängig davon, wie formal die Zustände und Transitionen spezifiziert sind, kann aus Zustandsdiagrammen Programmcode erzeugt werden
- Typisch: Iteratives Vorgehen: informelle Beschreibungen werden schrittweise durch formalere ersetzt
- Ereignisse können für folgendes stehen
  - Methodenaufrufe
  - externe Ereignisse des GUI (-> Methodenaufruf)
  - Teilsituation, die bei der Abarbeitung einer Methode auftreten kann
- Automat wird zunächst zu komplexer Methode, die z. B. anhand der Zustände in Teilmethoden refaktoriert werden kann

GUI als Zustandsautomat



Android als Zustandsdiagramm

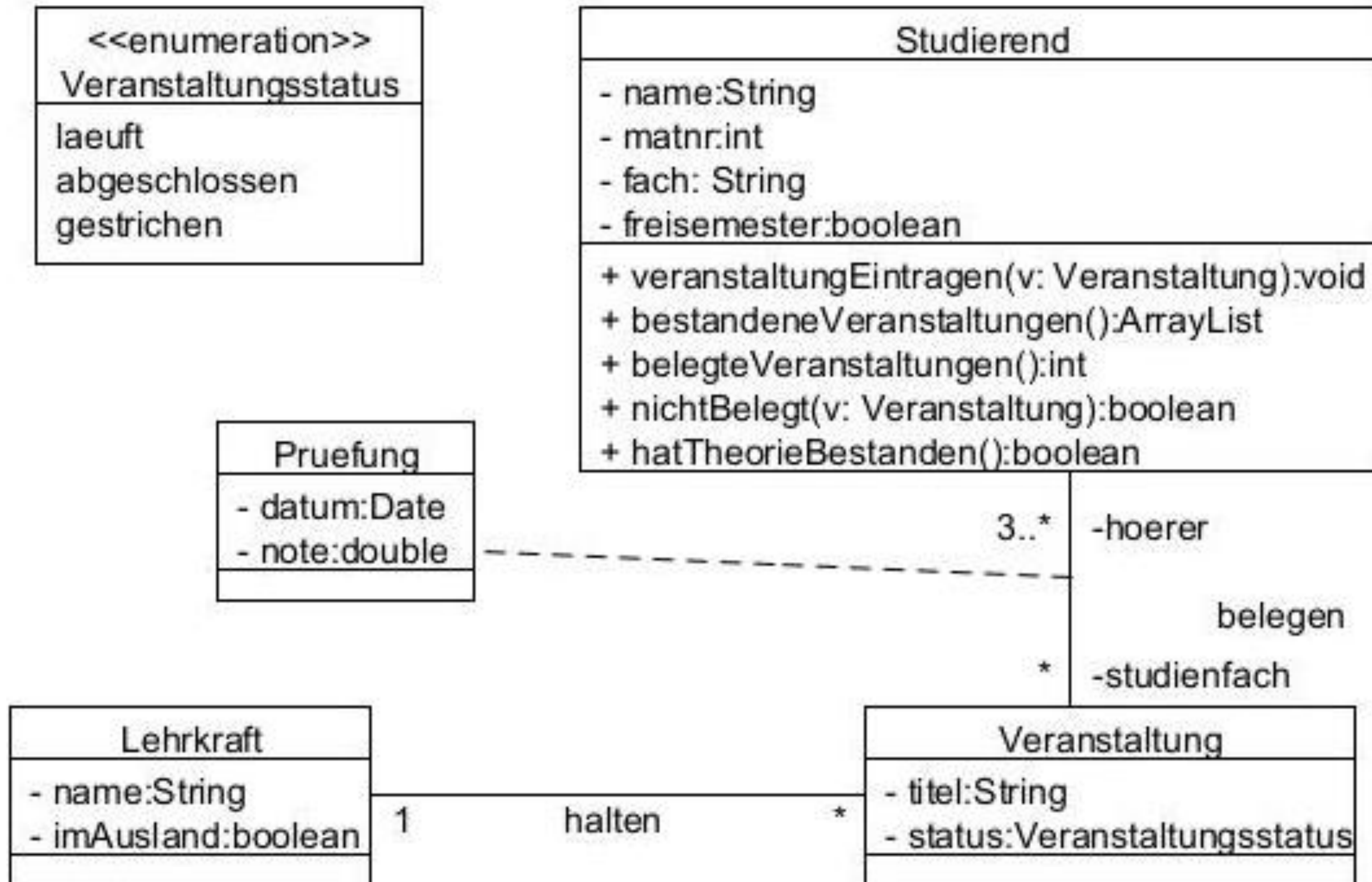


Klassendiagramm und versteckte Randbedingungen

7.2

Video

Welche Randbedingungen vermuten Sie?



Grundidee von Object Constraint Language (OCL)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- Rahmenbedingungen (Constraints) definieren, die von Objekten bzw. Objektmengen eingehalten werden können
- Constraints sind prüfbar
- möglichst einfach formulierbar (ursprünglich zur Formulierung von Geschäftsregeln für Versicherungsanwendungen, [Syntropy, IBM])
- Angepasst an Objektorientierung:
  - Zugriff auf Exemplarvariablen
  - Zugriff auf Methoden, die keine Objektveränderungen vornehmen
  - Vererbung wird beachtet
- typisiert, Collections wichtiger Typ

Einfache Bedingungen für Objekte (Invarianten)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- Die Matrikelnummer ist mindestens 10000

```
context Studierend inv hoheMatrikelnummern:
self.matnr >= 10000
```

- eine Variante:

```
context s:Studi inv:
s.matnr >= 10000
```

- **context** gibt eindeutig an, um welche Klasse es geht
- Strukturierung durch Nutzung der Paketstruktur

```
package com::meineFirma::meineSW
  context Studierend inv: ...
  context Studierend inv: ...
endpackage
```

Vor- und Nachbedingungen für Methoden

- Wenn Studierend-Objekt da, dann hört er Veranstaltungen

```
context Studierend::belegteVeranstaltungen():Integer
pre studiIstDa: self.freisemester = false
post hoertVeranstaltungen: result > 0
```

- Man kann auf Parameter der Methoden zugreifen
- **`result`** ist vordefiniert (vom Rückgabetyp)

- Erhöhung der Anzahl der belegten Veranstaltungen:

```
context Studierend::veranstaltungEintragen(v: Veranstaltung)
pre: nichtBelegt(v)
post: self.belegteVeranstaltungen()@pre
                  = self.belegteVeranstaltungen()-1
```

- **`self.belegteVeranstaltungen()@pre`**, für Ergebnis vor der Methodenausführung

Einschub: Basistypen und Operationen

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- Jeder OCL-Ausdruck hat einen Typ
- Verknüpfe Ausdrücke müssen vom Typ her passen
- Geringe Typanpassungen möglich

Typ	Beispielwerte
Boolean	true, false
Integer	1, -5, 42, 4242424242
Real	3.14, 42.42, -99.999
String	'Hallo Again', 'Heidi', ''

Typ	Beispieloperationen
Boolean	and,or,xor,not,implies,if then else endif
Integer	*, +, -, /, abs()
Real	*, +, -, /, floor()
String	concat(), size(), substring()

Zugriff auf Assoziationen

Video

- Zugriff auf verbundene Elemente möglich, Kardinalitäten beachten (einfach oder Menge)
- wenn Relation benannt, dann dieser Name

  sonst über Name der verbundenen Klasse (klein)

- Lehrkräfte laufender Veranstaltungen sind nicht im Ausland

```
context Veranstaltung inv:
self.status = Veranstaltungsstatus::laeuft
       implies
       not self.lehrkraft.imAusland
```

- Man sieht auch Zugriff auf eine Enumeration

Assoziationsklassen

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

- Ausgehend von Assoziationsklassen kann mit Punktnotation auf beteiligte Klassen (deren Objekte) mit deren Rollennamen zugegriffen werden

- Prüfungsnoten nur für abgeschlossene Veranstaltungen

```
context Pruefung inv:
 self.studienfach.status =
     Veranstaltungsstatus::abgeschlossen
  implies
     (self.note>=1.0 and self.note<=5.0)
```

Beispiele: Mengenoperationen (1/2)

- bei der Betrachtung zugehöriger Objekte ist das Ergebnis meist eine Collection von Objekten
- in OCL auch: Set, OrderedSet, Sequence, Bag
- auf Collections existieren verschiedene Methoden, genereller Aufruf

```
collection -> methode(<parameter>)
```

- Ergebnis kann wieder eine Collection oder ein Wert eines anderen Typs sein
- Studi macht höchstens 12 Veranstaltungen

```
context Studierend inv:
self.studienfach
       -> select (s | s.status =
                  Veranstaltungsstatus::laeuft)
       -> size() <= 12
```

Beispiele: Mengenoperationen (2/2)

- Korrektheit von hatTheorieBestanden

```
context Studierend::hatTheorieBestanden():Boolean
post: result = self.pruefung
   -> exists( p | p.note<=4.0
              and p.studienfach.titel='Theorie'))
```

- Korrektheit für bestandeneVeranstaltungen

```
context Studierend::
bestandeneVeranstaltungen():Collection
post: result=self.pruefung
        ->select( p | p.note<=4.0)
        ->iterate(p:Pruefung;
            erg:Collection=Collection{}|
               erg->including(p.studienfach))
```

Video

9. Implementierungsaspekte

nur Ideen, Inhalte der anderen Vorlesungen

Beispiel: Rahmenbedingungen für SW-Architektur

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

9.1

- Berücksichtigung von speziellen SW-Schnittstellen nicht objektorientiert entwickelter Systeme, z. B. von Application Programming Interfaces (API) fremder SW
- Berücksichtigung/Benutzung existierender Datenhaltungssysteme, z. B. Vorgabe des Datenbankmanagementsystems (DBMS)
- Berücksichtigung bestimmter Design-Prinzipien, z. B. Gesamtsteuerung mit Enterprise Java Beans (JEE) oder .NET für die Realisierung
- Alt-Software (z. B. in COBOL), so genannte Legacy-Systeme müssen eingebunden werden; Einsatz einer Middleware (z. B. Common Object Request Broker Architecture, CORBA)

Einfluss nichtfunktionaler Anforderungen

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

Beispiel: Sicherheit (Security)

- Alle Nachrichten müssen über den speziellen Krypto-Server laufen; dieser hat bestimmte Bandbreite (Bottle-neck); SW muss auf allen Seiten möglichst viel ohne Verbindung arbeiten können (Redundanz wird erlaubt)

Beispiel: Sicherheit (Safety)

- Berechnungen zur Steuerung müssen redundant auf drei Rechnern mit unterschiedlichen Verfahren durchgeführt werden

Beispiel: Performance

- Die rechenintensiven 3D-Berechnungen müssen sehr schnell sein; dies kann zum Einsatz von C mit langen komplexen Funktionen führen

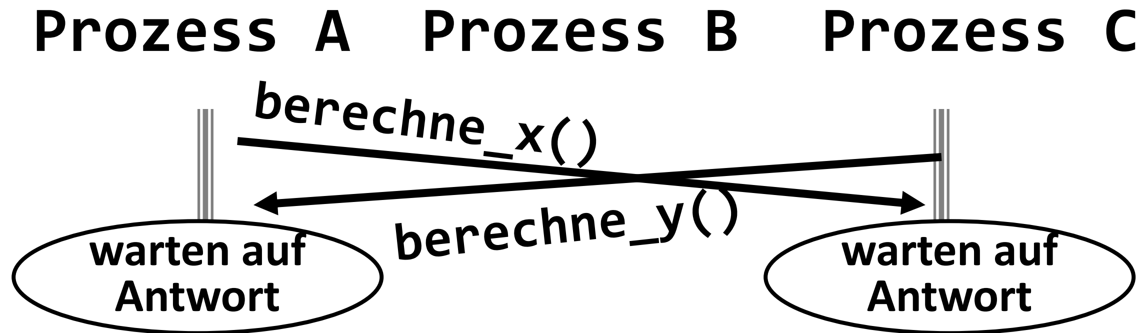
Rahmenbedingung: verteilte Systeme

9.2

- in der klassischen OO-Programmierung gibt es einen Programmablauf (Prozess) und man nutzt synchrone Aufrufe: Objekt O1 ruft Methode von Objekt O2 auf; O2 übernimmt die Programmausführung und antwortet dann O1
- bei verteilten Systemen laufen viele Prozesse parallel ab, die Informationen austauschen können
- synchroner Aufruf ist möglich, bedeutet aber, dass Verbindung aufgebaut werden muss und Sender bzw. Empfänger auf Bereitschaft warten müssen
- asynchroner Aufruf bedeutet, dass Sender Aufruf abschickt und danach weiterarbeitet; später prüft, ob ein Ergebnis vorliegt
- asynchrone Aufrufe sind schneller (nur abschicken); Prozesse sind aber schwer zu synchronisieren
- die Herausforderung effizienter verteilter Systeme hat nicht die eine Lösung und wird Sie Ihr Informatik-Leben-lang verfolgen

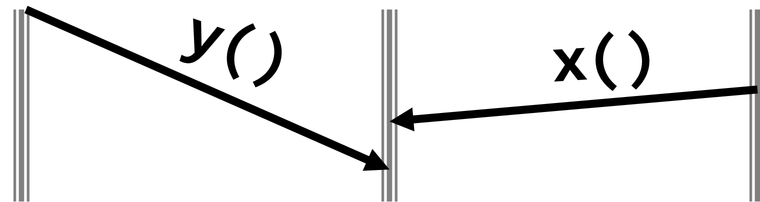
Probleme der Aufrufmechanismen

synchroner Aufruf

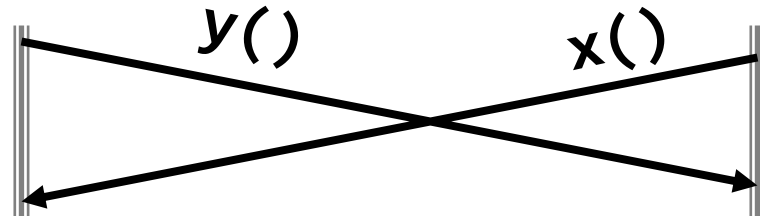


Problem: Deadlock

asynchroner Aufruf



Problem: B denkt, x hat vor y stattgefunden



Problem: C denkt, x hat vor y stattgefunden, A denkt, y hat vor x stattgefunden

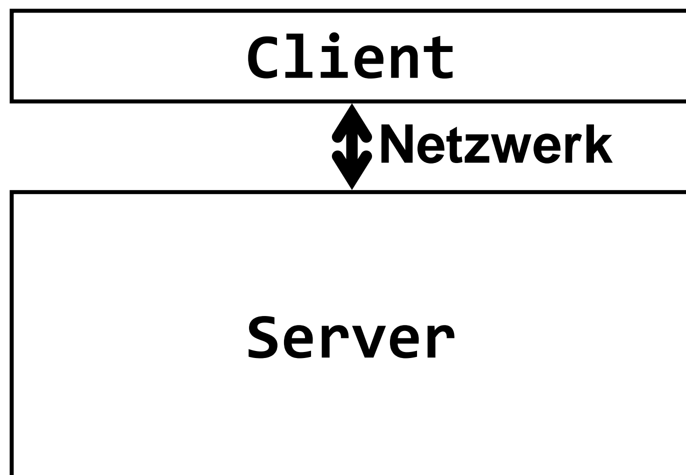
Typische Probleme verteilter Systeme

- Deadlocks: kein Prozess/Thread kann voran schreiten
- partielle Deadlocks: einige Prozesse im Deadlock, andere nicht
- Livelocks: System versucht, sich zyklisch zu synchronisieren, ohne dass das System voran schreitet
- (starke) Fairness : kommen Prozesse, die immer mal wieder darauf warten, in den kritischen Bereich zu kommen, auch dran
- (schwache) Fairness: kommen Prozesse, die immer darauf warten, in den kritischen Bereich zu kommen, auch dran

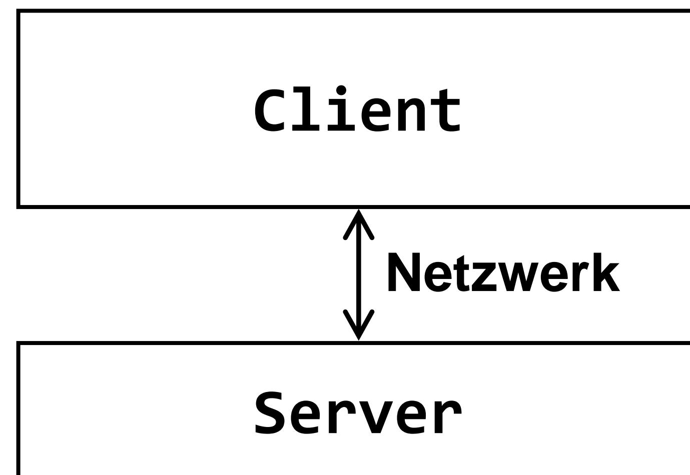
- **`synchronized()`** in Java (Methode wird garantiert ohne Parallelnutzung des aufgerufenen Objekts genutzt) hat starken negativen Einfluss auf die Laufzeit
- Erinnerung/Ausblick: Notwendige Transaktionssteuerung bei Datenbankmanagementsystemen

Beispiel: Varianten von Client-Server-Systemen

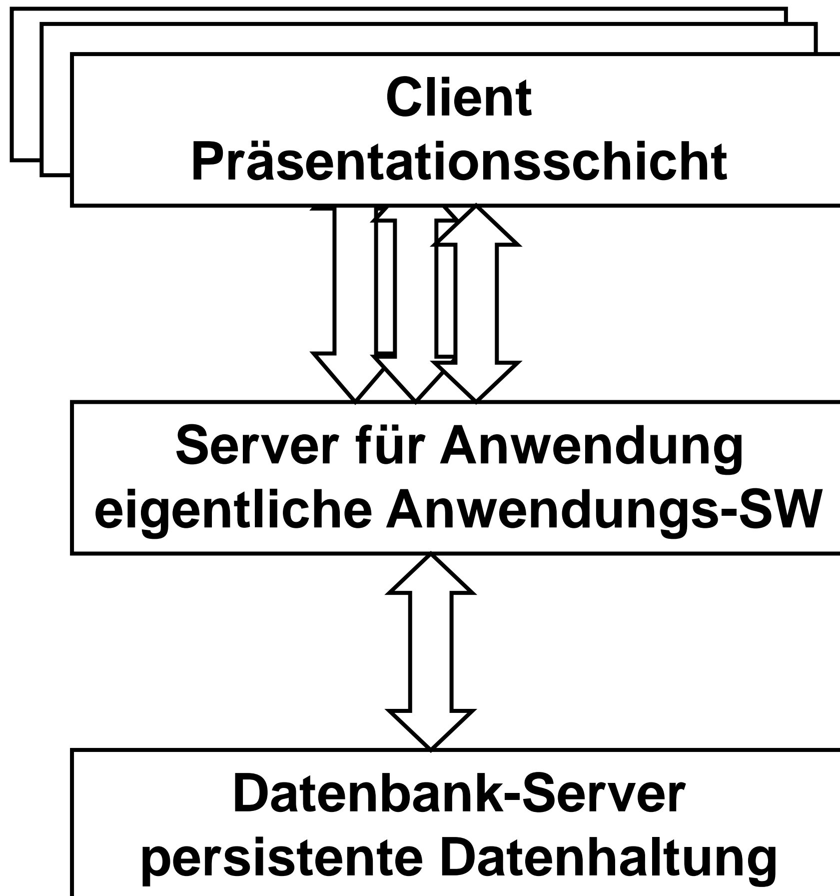
- Thin Client: Hier nur Datenannahme, Weiterleitung, Darstellung, keine komplexen Berechnungen
- Beispiele: Web-Browser, DB-Clients



- Fat Client: Client führt eigene komplexe Berechnungen aus; nutzt Server nur zur Verwaltung zentraler Informationen und zum Nachrichtenaustausch
- Beispiel: vernetzbare Stand-alone-Spiele (Autorennen)



Beispiel: 3-Tier-Architektur



Verteilung:

- Nur Darstellung (GUI) beim Client
- eigener Server für Anwendung
- eigene Datenspeicherung

Vorteile:

- benötigte DB-Verbindungen können angepasst werden (Kosten)
- Datenbank nicht direkt für Client zugreifbar (Sicherheit)
- Änderungen einer Schicht müssen andere Schichten nicht beeinflussen

Nutzung von Programmbibliotheken

9.4

- Programmbibliotheken stellen Standardlösungen für häufig wiederkehrende Probleme dar
- typische Nutzung: entwickelnde Person erzeugt und ruft Objekte (Klassen) der Bibliothek auf
- Bibliotheken sind geprüft, (hoffentlich) für Laufzeiten optimiert
- Dokumentation von Bibliotheken wichtig zum effizienten Einsatz (was rufe ich wann auf)
- Je größer der Verbreitungsgrad, desto einfacher die Weiterverwendung von Ergebnissen (großer Vorteil der Java-Klassenbibliothek)
- Grundregel für erfahrene entwickelnde Personen: Erfinde das Rad niemals zweimal, weiß aber, wo viele Blaupausen für viele verschiedene Räder sind
- Grundregel für mit Informatik-Beginnende: Lerne zu verstehen, wie man das erste Rad baut, baue das erste Rad und lerne warum man wie die Blaupause variieren kann

Idee von Komponenten

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

9.5

- Komponenten sind komplexe in sich abgeschlossene „binäre“ SW-Bausteine, die größere Aufgaben übernehmen können
- Ansatz: SW statt aus kleinen Programmzeilen aus großen Komponenten (+ Klebe-SW) zusammen bauen
- Komponenten werden konfiguriert, dazu gibt es get-/set-Methoden (Schnittstelle) oder/und Konfigurationsdateien
- Beispiel Swing-Klassen, wie JButton haben (u. a.) Komponenteneigenschaft; man kann u. a. einstellen:
  - Farben (Hintergrund, Vordergrund)
  - Schrifttypen
  - Form der Ecken
  - dargestelltes Bild
- Komponenten sind themenorientiert und können unterschiedliche Aufgaben erfüllen (z. B. Daten filtern, Werte überwachen)

Idee der Framework-Technologie

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

9.6

- statt vollständiger SW werden Rahmen programmiert, die um Methodenimplementierungen ergänzt werden müssen
- Frameworks (Rahmenwerke) können die Steuerung gleichartiger Aufgaben übernehmen
- typische Nutzung: entwickelnde Person instanziiert Framework-Komponenten, d. h. übergibt seine Objekte zur Bearbeitung durch das Framework; typischer Arbeitsschritt: Framework steuert, d. h. ruft Methode der entwickelnden Person auf
- eventuelles Problem: schwieriger Wechsel zu anderem Framework oder bei Ablösung des Frameworks

Ziele komplexere Framework-Ansätze

neben Spezialaufgaben werden hauptsächlich folgende Aufgaben gelöst

- sorgenfreies Lesen und Speichern von Objekten in Datenbanken (Persistenz)
- sorgenfreie konsistente Verteilung von Informationen (Prozesskommunikation)
- sorgenfreie Steuerung verteilter Abläufe mit Überwachung von Transaktionen

- Beispiele sind Jakarta Enterprise Edition, Microsoft Dot-Net-Technologie, Spring, Hibernate, viel im Bereich AJAX

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

9.7

Typische Java-Möglichkeiten

- Anschluss an klassische relationale DB über JDBC (typisch bei Anbindung an existierende DB)
- Nahtlose Integration der Datenhaltung in die Entwicklung (Ansatz: statt Objekt zu erzeugen Methode holeObjekt(), später sichere Objekt), typisch für Hibernate (häufig genutzt, bei kleinen Spezialanwendungen, z. B. Handy, Organizer)
- relativ nahtlose Integration durch zusätzliche Software, die objekt-relationales Mapping übernimmt
- Nutzung eines Frameworks, das Persistenz und Transaktionssteuerung übernimmt, Enterprise Java Beans

Beispiel: JavaBeans (kleiner Ausschnitt)

- Java unterstützt Reflektion, damit kann ein Objekt nach seiner Klasse, seinen Exemplarvariablen und Exemplarmethoden befragt werden
- Hält man sich an folgende einfache Regel für eine Klasse
  - sie implementiert Serializable (geht nur, wenn alle verwendeten Typen Serializable)
  - für alle Exemplarvariablen gibt es die Standard get- und set-Methoden
  - es gibt einen leeren Default-Konstruktor

  dann sind einige Framework-Ansätze nutzbar
  - Objekte speichern und lesen in XML
  - Nutzung als JavaBeans (sinnvoll weitere Standardmethoden)
  - Objekte speichern in einer Datenbank mit JPA, als Entity
  - Objekte im Binärformat lesen und schreiben (reicht Serializable)

XMLEncoder und XMLDecoder (Ausschnitt)

HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

```
private void speichern(String datei){
  try (XMLEncoder out= new XMLEncoder(
           new BufferedOutputStream(new FileOutputStream(datei)))){
    out.writeObject(table.getModel());
  } catch (FileNotFoundException e) {} //wegschauen
}

private void laden(String datei){
try ( XMLDecoder in= new XMLDecoder(
           new BufferedInputStream(new FileInputStream(datei)))){
    table.setModel((DefaultTableModel)in.readObject());
    } catch (FileNotFoundException e) {} //wegschauen
}
```

Refactoring

9.10

- Komplexe Methoden sollen grundsätzlich vermieden werden
- Lösungsansatz: Refactoring, d. h. ein Programmblock wird in einer Methode mit selbsterklärendem Namen ausgegliedert
- Wann ist Ausgliederung möglich?
  - Im Block darf nur eine lokale Variable auf der linken Seite einer Zuweisung stehen
- Wie funktioniert Refactoring?
  - Bestimme alle lokalen Variablen, die im Block lesend genutzt werden; diese werden zu Parametern
  - Falls eine lokale Variable links in einer Zuweisung vorkommt, bestimmt sie den Rückgabetypen (sonst void)
- Exemplarvariablen spielen keine Rolle, da auf sie in allen Methoden der Klasse zugegriffen werden darf
- Probleme bei mehr als einer zu verändernden lokalen Variablen oder bei enthaltenen Rücksprüngen (aufwändig regelbar)

Refactoring – Positives Beispiel

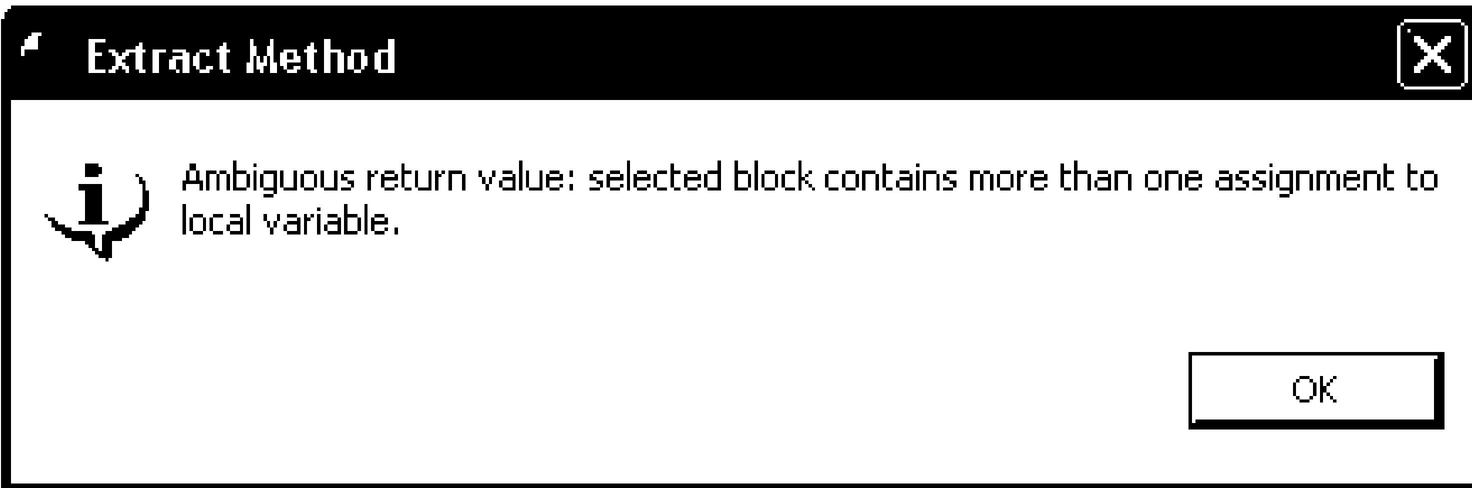
HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

```java
public int ref(int x, int y, int z){
  int a = 0;
  if(x > 0){
    a = x;
    x++;
    --y;
    a = a + y + z;
  }
  return a;
}
```

```java
public int ref(int x, int y, int z){
  int a = 0;
  if(x > 0){
    a = this.mach(x, y, z);
  }
  return a;
}

private int mach(int x, int y, int z){
  int a;
  a = x;
  x++;
  --y;
  a = a + y + z;
  return a;
}
```

Refactoring – nicht einfaches Beispiel

```
public int ref2(int x){
  int a = 0;
  int b = 0;
  int c = 0;
  if(x > 0){
    a = x;
    b = x;
    c = x;
  }
  return a + b + c;
}
```



Refactoring – (nicht) einfaches Beispiel in C++

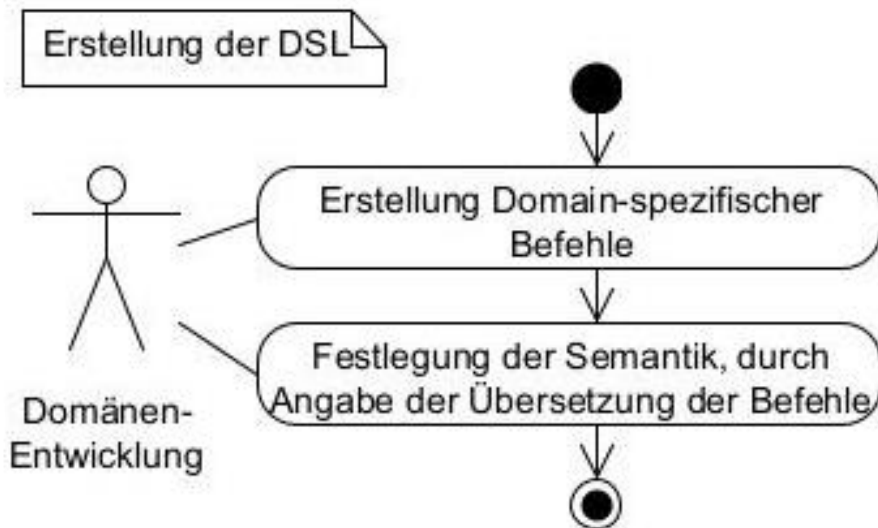

```
int Rechnung::ref2(int x){
  int a = 0;
  int b = 0;
  int c = 0;
  if (x > 0) {
    abcAnpassen(a, b, c, x);
  }
  return a + b + c;
}

void Rechnung::abcAnpassen(int& a, int& b, int& c, int x){
  a = x;
  b = x;
  c = x;
}
```

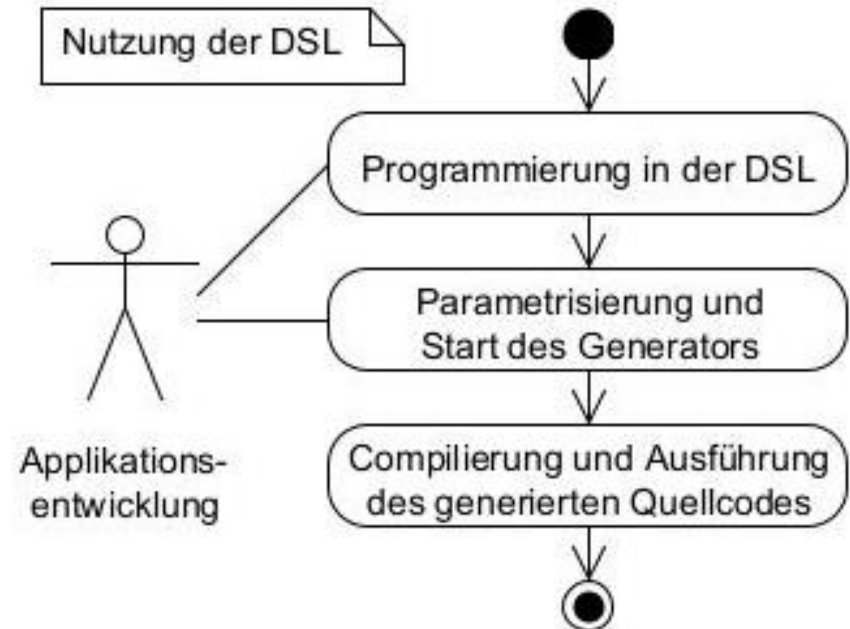
Domain Specific Languages (DSL)

9.8

- Problem: General Purpose Sprachen sind sehr mächtig, aber für spezifische Entwicklungsbereiche geht sehr viel Energie in für den Bereich gleichartige Programmierung
- Spezielle Entwicklungssprache für individuellen Bereich, spezielle komplexe Hochsprachelemente anbietet
- Neue Sprache z. B. mit XML (Syntax mit XML-Schema) darstellbar; Umwandlung in Programm mit Übersetzung (z. B. XSLT) ; hilfreich ist Visualisierungsmöglichkeit der DSL
- Hinweis: UML (evtl. mit konkreter Ausprägung) kann mit MDA-Transformationen auch als spezieller DSL-Ansatz angesehen werden

DSL Prozesse





Model Driven Architecture

9.9

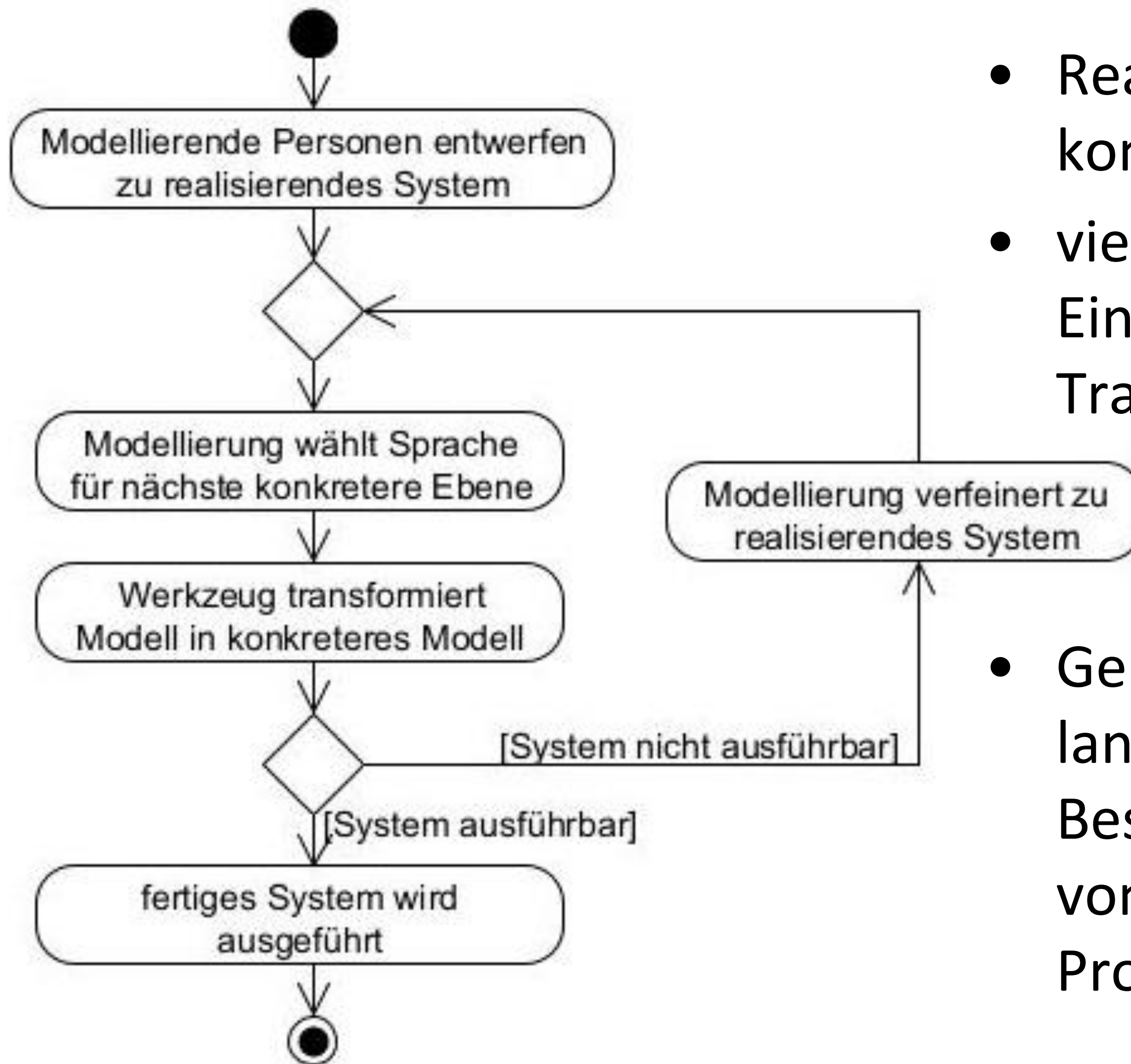
- Ansatz: Häufig benötigt man die gleichen Ideen (z. B. Sortierverfahren) in sehr unterschiedlichen Sprachen; warum nicht in einer Sprache modellieren und dann in andere Sprachen transformieren?
- Da Sprachen extrem unterschiedlich, soll Modellumwandlung schrittweise passieren
- Zur Modellbeschreibung wird eigene Sprache mit eigener Semantik benötigt (Metamodell und Metametamodell)
- Ansatz: Umwandlung des CIM mit Transformationsregeln in ein PIM und dann ein PSM

  CIM: Computer Independent Model

  PIM: Platform Independent Model

  PSM: Platform Specific Model
- z. B. UML-Modell, dann Realisierungssprache wählen, dann HW-Plattform mit Kommunikationsprotokollen wählen (zwei parametrisierte Transformationen)

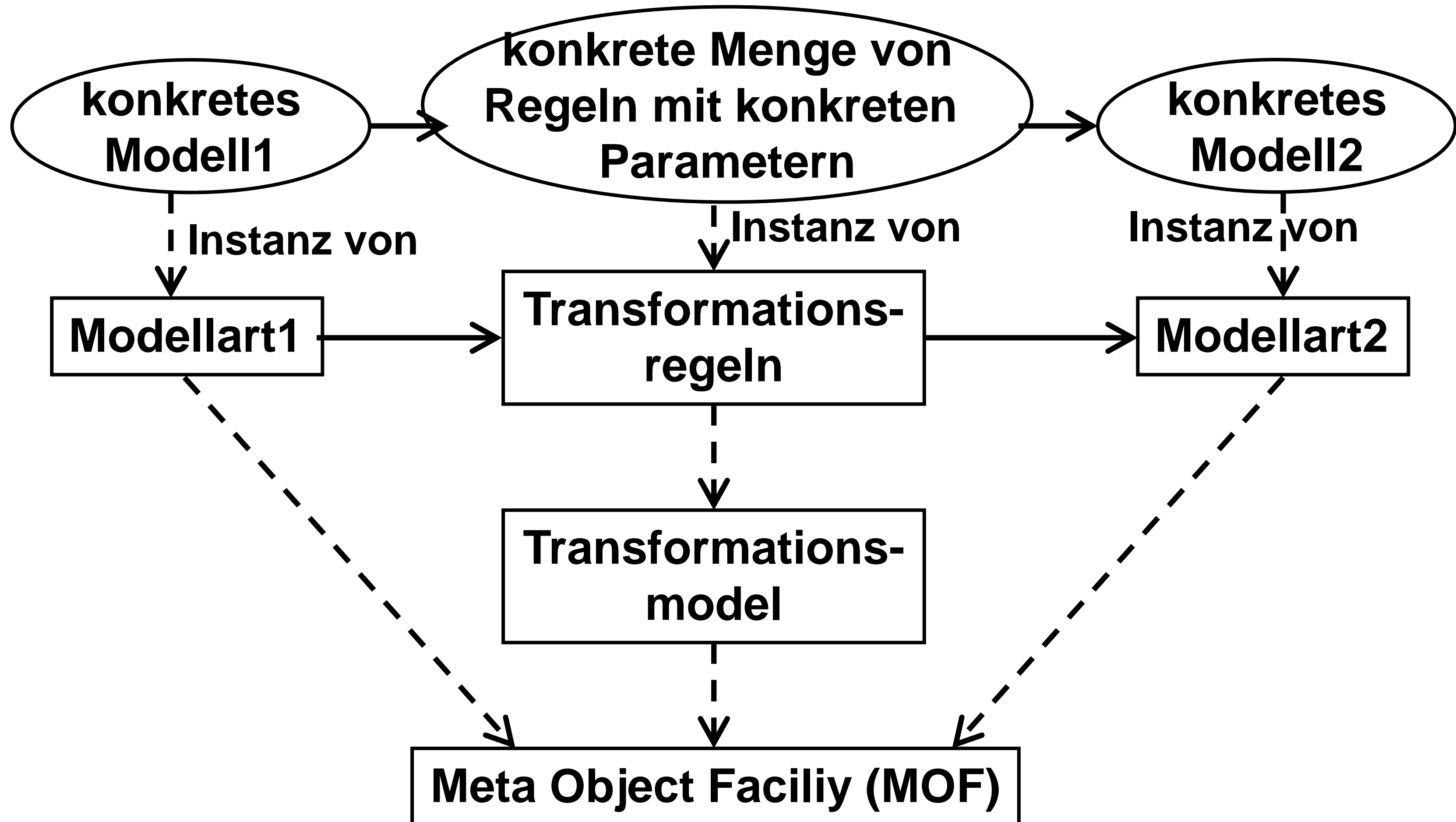
Prozess der MDA (Theorie)



- Realität: häufig nur eine konkrete Ebene
- viele manuelle Einstellungen für die Transformation

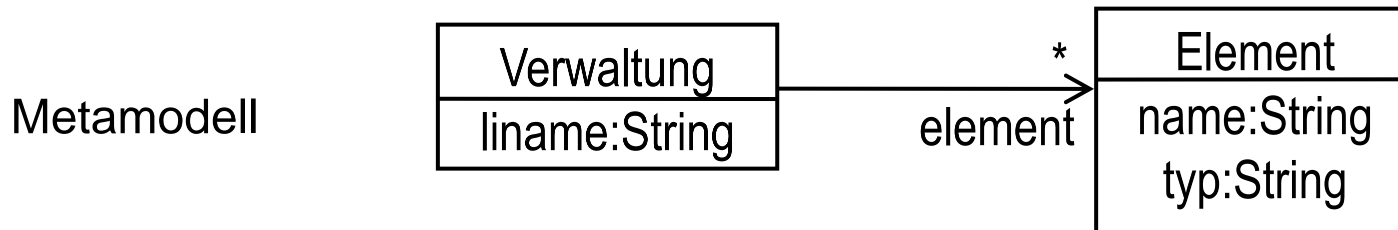
- Generieren gibt es schon lange (YACC, Dateien zur Beschreibung von Fenstern, von UML zum Programmskelett)

Formaler Hintergrund der MDA



Model Driven Software Development

Metamodell



Codegenerator

```
public class {Verwaltung.liname} {
  <foreach Element e:Verwaltung.element>
    private List<{e.typ}> {e.name};
}
```

Modell

Verwaltung
- liname=„Hauptliste“
- Element
  - name=„bestellende“
  - typ=„Bestellend“
- Element
  - name=„produkte“
  - typ=„Produkt“

generierter Code

```
public class Hauptliste {
    private List<Bestellend> bestellende;
    private List<Produkt> produkte;
}
```

z. B. https://projects.eclipse.org/projects/modeling.emf.mwe