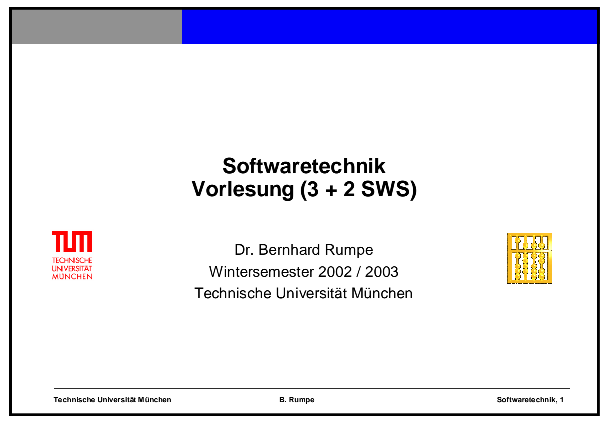
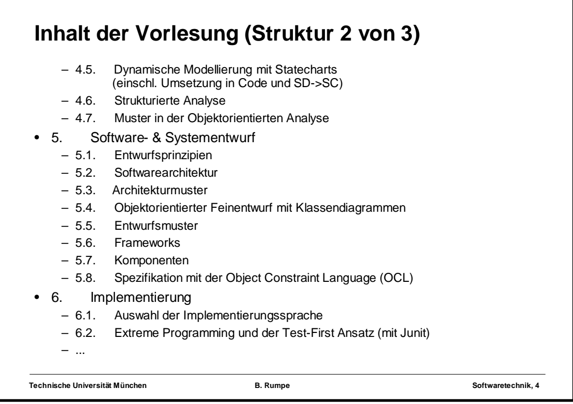


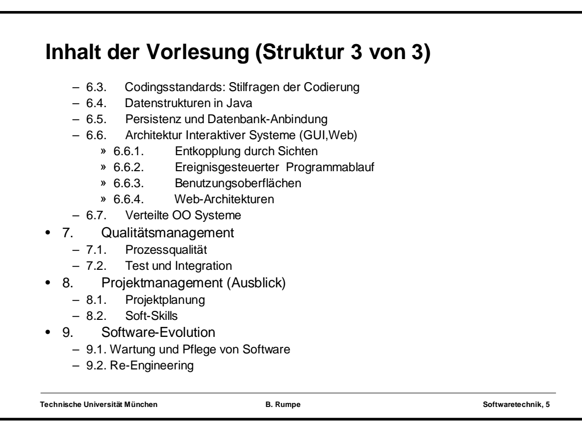
Inhalt der Vorlesung (Übersicht)

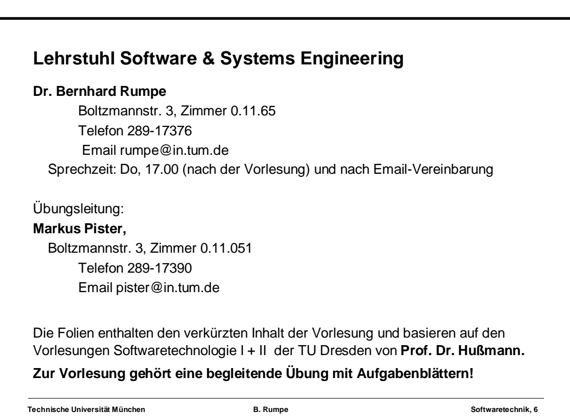
Kapitel	gehalten ab /	**Folie**
1. Einleitung & Überblick Softwaretechnik	16.10	**8**
2. Objektorientierung	24.10.	**77**
3. Anforderungsanalyse	30.10.	**96**
4. Systemanalyse & Systemmodellierung	13.11.	**158**
5. Software- & Systementwurf	4.12.	**344**
6. Implementierung	8.1.	**532**
7. Qualitätsmanagement	23.1	**730**
8. Projektmanagement (Ausblick)	29.1.	**764**
9. Software-Evolution	30.1.	**787**
Klausur	6.2.	

Inhalt der Vorlesung (Struktur 1 von 3)

- 1. Einleitung & Überblick Softwaretechnik
  - 1.2. Phasenmodelle
  - 1.3. Entwicklungsmethoden
  - 1.4. Modelle und Modellierungstechniken
- 2. Objektorientierung
  - 2.1. Die Idee
  - 2.2. CRC-Karten
- 3. Anforderungsanalyse
  - 3.1. Anforderungsermittlung
  - 3.2. Anforderungsmodellierung
  - 3.3. Prototyping
- 4. Systemanalyse & Systemmodellierung
  - 4.1. Systemanalyse
  - 4.2. Objektorientierte Analyse
  - 4.3. Statische Modellierung mit der UML (nach OMT)
  - 4.4. Modellierung von Szenarien
  - ...







Grundlegende Literatur

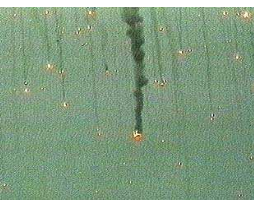
- B. Brügge, A. Dutoit: Object-Oriented Software Engineering. Prentice-Hall. 2000.
- H. Balzert: Lehrbuch der Software-Technik, Bd. 1, Spectrum Verlag (2te Auflage). 2000.
- Ian Sommerville: Software Engineering, 6th edition, Addison-Wesley 2001.
  - es gibt auch eine deutsche Übersetzung und einen zweiten Band
- Weitere Literatur bei den einzelnen Kapiteln
- Folien im PDF-Format und laufende Informationen zur Vorlesung und den Übungen:

http://www4.in.tum.de/~rumpe/se

Softwaretechnik WS 02/03

1
Einleitung & Überblick über das Gebiet der Softwaretechnik

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



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

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

Software-Katastrophe: Kein Einzelfall

- Finanzielle Katastrophen, Tendenz "steigend":
  - 1981: US Air Force Command&Control Software überschreitet Kostenvoranschlag fast um den Faktor 10 (***3,2 Mio US-$***)
  - 1987-1993: Integration der kalifornischen Systeme zur Führerschein- und KFZ-Registrierung abgebrochen: ***44 Mio US-$***
  - 1992: Integration des Reservierungssystems SABRE mit anderen Reservierungsystemen abgebrochen: ***165 Mio. US-$***
  - 1997: Entwicklung des Informationssystems SACSS für den Staat Kalifornien abgebrochen: ***300 Mio US-$***
- Terminkatastrophen:
  - 1994: Eröffnung des Denver International Airport um 9 Monate verzögert wegen Softwareproblemen im Gepäcktransport-System
- Technik-Katastrophen:
  - März 1999: Fehlstart einer Titan/Centaur-Rakete wegen falscher Software-Version
  - September 1999: Verlust der Sonde "Mars Climate Orbiter" wegen falscher Einheitenumrechnung
  - US-Unternehmen verloren im Jahr 2000 insgesamt 100 Milliarden US-$ wegen defekter Software.

Die permanente Softwarekrise ?

- ***1965:*** Der Begriff der ***Softwarekrise*** etabliert sich in Industrie und Wissenschaft.
  - Fehler in Computersystemen sind fast immer auf Softwarefehler zurückzuführen.
  - Software wird nicht termingerecht und/oder zu höheren Kosten als geschätzt fertiggestellt.
  - Software entspricht oft nicht den Anforderungen ihrer Benutzer.
- Studie von ***1979*** zu Softwareprojekten (USA):
  - 75% der Ergebnisse nie eingesetzt
  - 19% der Ergebnisse stark überarbeitet
  - 6% benutzbar.
- Studie von ***1994*** zu Software-Großprojekten (IBM Consulting):
  - 55% Kostenüberschreitung
  - 68% Terminüberschreitung
  - 88% Bedarf für starke Überarbeitung

Lehrstuhl Software & Systems Engineering / Prof. Broy

Dr. Bernhard Rumpe
Boltzmannstr. 3, Zimmer 0.11.65
Telefon 289-17376
Email rumpe@in.tum.de
Sprechzeit: Do, 17.00 (nach Vorlesung) und nach Email-Vereinbarung

Übungsleitung:

Markus Pister,
Boltzmannstr. 3, Zimmer 0.11.051
Telefon 289-17390
Email pister@in.tum.de

Ein chinesisches Sprichwort

Was man hört, vergißt man.

Was man sieht,
daran kann man sich erinnern.

Nur was man selbst tut,
kann man verstehen.

1.1 Softwaresysteme und Softwaretechnik

Literatur: **Sommerville 1.1**
Balzert Band 1, LE 1

Voraussetzungen, Übungen zur Vorlesung

- Voraussetzungen:
  - Vordiplom
  - generell gute Programmierkenntnisse (am besten Java)
    - » Wissen über Programmiersprachen wird hier nicht vermittelt !
- Übungsbetrieb:
  - Zentralübung am Dienstag, 8:30 - 10:00, Hörsaal MW1801
  - In den Übungen werden wesentliche Schritte eines kleinen Projekts geübt sowie Werkzeuge und Techniken diskutiert.
  - Übungsgruppe können Aufgaben gemeinsam bearbeiten.
  - Übungsbeginn: Dienstag, 22.10.

Softwaresysteme

software: computer programs, procedures, rules, and possibly associated documentation and data pertaining to the operation of a computer system.
(IEEE Standard Glossary of Software Engineering)

- ***Softwaresystem***:
  - Ein System (oder Teilsystem), dessen Komponenten aus Software bestehen.
- ***Software-Produkt***
  - Ein Produkt ist ein in sich abgeschlossenes, i.a. für einen Auftraggeber bestimmtes Ergebnis eines erfolgreich durchgeführten Projekts oder Herstellungsprozesses. Als Teilprodukt bezeichnen wir einen abgeschlossenen Teil eines Produkts
  - SW-Produkt: Produkt, das aus Software besteht.

Literatur

- B. Brügge, A. Dutoit: Object-Oriented Software Engineering. Prentice-Hall. 2000.
- H. Balzert: Lehrbuch der Software-Technik, Bd. 1, Spectrum Verlag (2te Auflage). 2000.
- Ian Sommerville: Software Engineering, 6th edition, Addison-Wesley 2001.
  - es gibt auch eine deutsche Übersetzung und einen zweiten Band

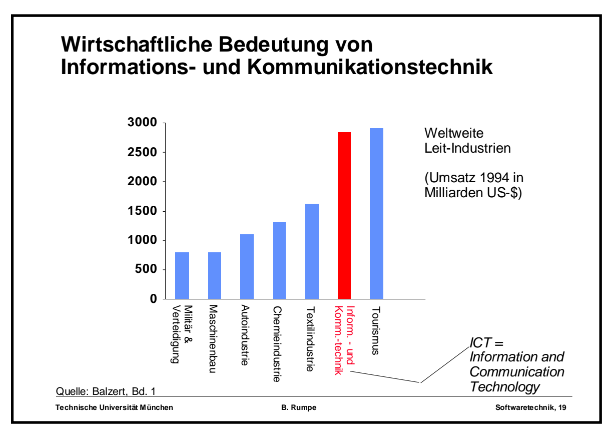
- Weitere Literatur bei den einzelnen Kapiteln
- Folien im PDF-Format und laufende Informationen zur Vorlesung und den Übungen:

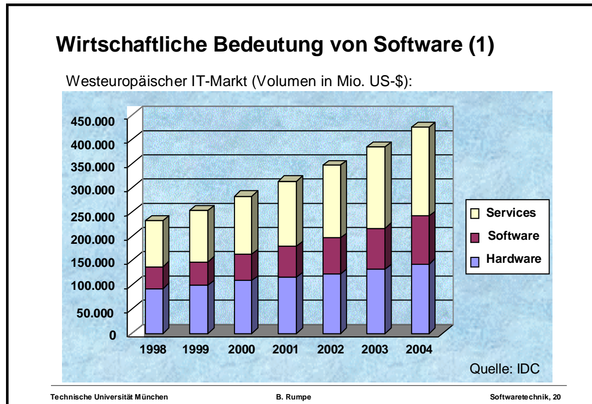
http://www4.in.tum.de/~rumpe/se

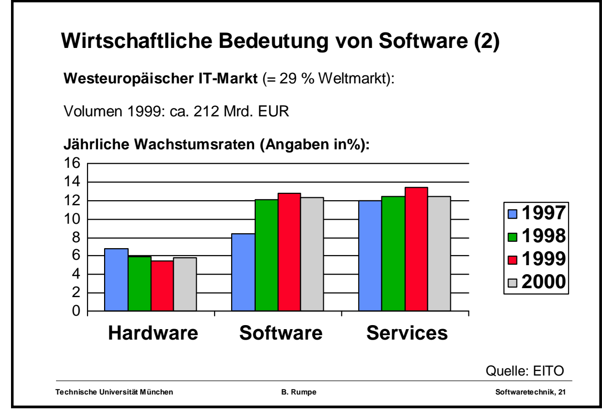
- Die Folien enthalten den verkürzten Inhalt der Vorlesung
- und basieren teilweise auf den Vorlesungen Softwaretechnologie I + II der TU Dresden von Prof. Hußmann

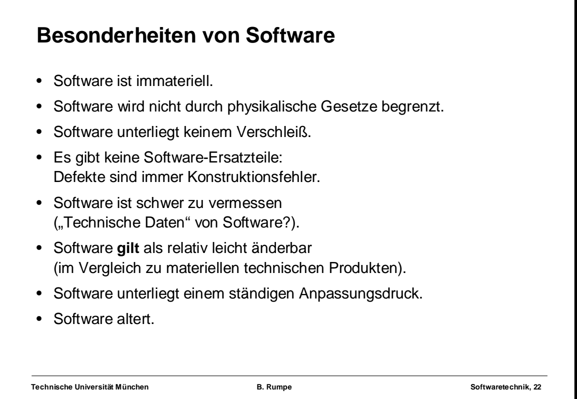
Klassifikation von Software

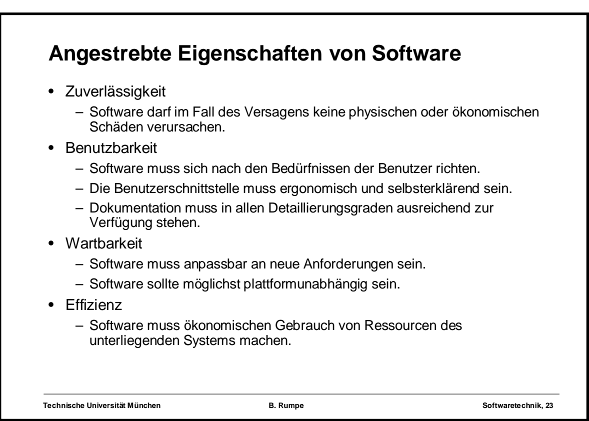
- Generisches Produkt oder Einzelanfertigung (vereinbartes Produkt)?
- Systemsoftware (Betriebssystem, Compiler, Editor, ...) oder Anwendungssoftware (application software)?
- Produktintegriert *(embedded)* oder für reine Computersysteme?
- Realzeitanforderungen oder flexiblere Zeitanforderungen?
- Datenintensiv oder berechnungsintensiv?
- Monolithisch oder verteilt?
- Standalone oder mit anderen Anwendungen integriert?

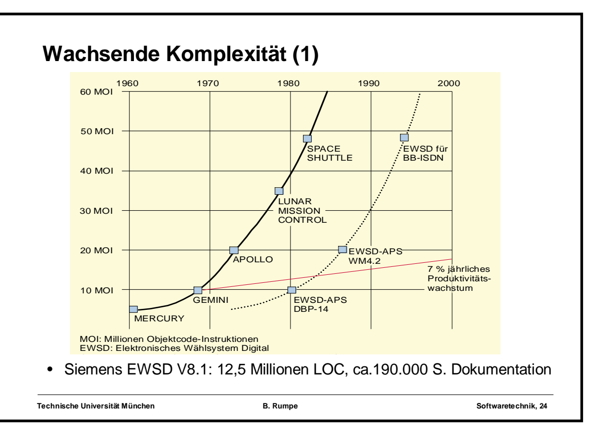

Wirtschaftliche Bedeutung von
Informations- und Kommunikationstechnik
3000
2500
2000
1500
1000
500
0
Militär & Verteidigung
Maschinenbau
Autoindustrie
Chemieindustrie
Textilindustrie
Inform. - und Komm.-technik
Tourismus
Weltweite
Leit-Industrien
(Umsatz 1994 in
Milliarden US-$)
ICT =
Information and
Communication
Technology
Quelle: Balzert, Bd. 1
Technische Universität München
B. Rumpe
Softwaretechnik, 19


Wirtschaftliche Bedeutung von Software (1)
Westeuropäischer IT-Markt (Volumen in Mio. US-$):
450.000
400.000
350.000
300.000
250.000
200.000
150.000
100.000
50.000
0
1998
1999
2000
2001
2002
2003
2004
Services
Software
Hardware
Quelle: IDC
Technische Universität München
B. Rumpe
Softwaretechnik, 20


Wirtschaftliche Bedeutung von Software (2)
Westeuropäischer IT-Markt (= 29 % Weltmarkt):
Volumen 1999: ca. 212 Mrd. EUR
Jährliche Wachstumsraten (Angaben in%):
16
14
12
10
8
6
4
2
0
Hardware
Software
Services
1997
1998
1999
2000
Quelle: EITO
Technische Universität München
B. Rumpe
Softwaretechnik, 21


Besonderheiten von Software
Software ist immateriell.
Software wird nicht durch physikalische Gesetze begrenzt.
Software unterliegt keinem Verschleiß.
Es gibt keine Software-Ersatzteile:
Defekte sind immer Konstruktionsfehler.
Software ist schwer zu vermessen
(„Technische Daten“ von Software?).
Software gilt als relativ leicht änderbar
(im Vergleich zu materiellen technischen Produkten).
Software unterliegt einem ständigen Anpassungsdruck.
Software altert.
Technische Universität München
B. Rumpe
Softwaretechnik, 22


Angestrebte Eigenschaften von Software
Zuverlässigkeit
Software darf im Fall des Versagens keine physischen oder ökonomischen Schäden verursachen.
Benutzbarkeit
Software muss sich nach den Bedürfnissen der Benutzer richten.
Die Benutzerschnittstelle muss ergonomisch und selbsterklärend sein.
Dokumentation muss in allen Detaillierungsgraden ausreichend zur Verfügung stehen.
Wartbarkeit
Software muss anpassbar an neue Anforderungen sein.
Software sollte möglichst plattformunabhängig sein.
Effizienz
Software muss ökonomischen Gebrauch von Ressourcen des unterliegenden Systems machen.
Technische Universität München
B. Rumpe
Softwaretechnik, 23


Wachsende Komplexität (1)
1960
1970
1980
1990
2000
60 MOI
50 MOI
40 MOI
30 MOI
20 MOI
10 MOI
SPACE SHUTTLE
EWSD für BB-ISDN
LUNAR MISSION CONTROL
APOLLO
EWSD-APS WM4.2
7 % jährliches Produktivitäts-wachstum
GEMINI
EWSD-APS DBP-14
MERCURY
MOI: Millionen Objektcode-Instruktionen
EWSD: Elektronisches Wählsystem Digital
Siemens EWSD V8.1: 12,5 Millionen LOC, ca.190.000 S. Dokumentation
Technische Universität München
B. Rumpe
Softwaretechnik, 24

Wachsende Komplexität (2)

- Enterprise-Resource-Planning Software R/3® von SAP

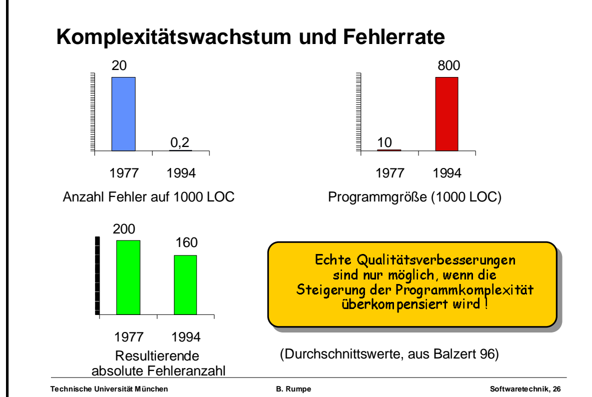
Jahr	Lines of Code	Anzahl Funktionsbausteine
1994	7 Millionen	14 000
1997 (Rel. 3.1)	30 Millionen	200 000
1999 (Rel. 4.5)	50 Millionen	400 000

- Weitere Zahlen zu R/3 Release 4:
  - 11 000 externe Tabellen (Datenbank)
  - 500 000 Tabellenfelder (extern und intern)
- Gesamtumfang der verwendeten Software (Anfang 2000):
  - Chase Manhattan Bank: 200 Mio. Code-Zeilen
  - Citicorp Bank: 400 Mio. Code-Zeilen
  - AT&T: 500 Mio. Code-Zeilen
  - General Motors: 2 Mrd. Code-Zeilen

Warum ist Software so schwer zu entwickeln?

- 0,1%-Defektniveau bedeutet:
  - » pro Jahr:
    - 20.000 fehlerhafte Medikamente
    - 300 versagende Herzschrittmacher
  - » pro Woche:
    - 500 Fehler bei medizinischen Operationen
  - » pro Tag:
    - 16.000 verlorene Briefe in der Post
    - 18 Flugzeugabstürze
  - » pro Stunde:
    - 22.000 Schecks nicht korrekt gebucht
- Auch in Zukunft:
  - » Massive QM-Anstrengungen notwendig.

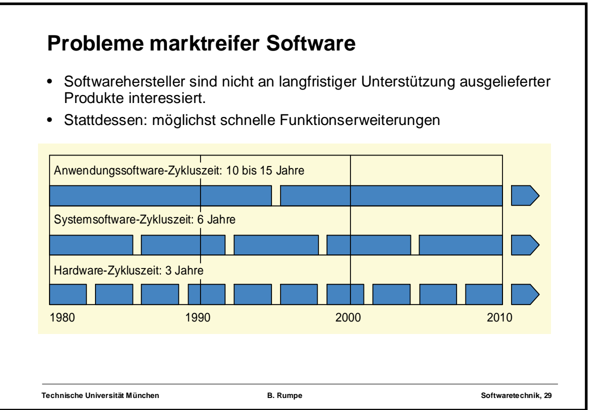
Komplexitätswachstum und Fehlerrate



(Durchschnittswerte, aus Balzert 96)

Probleme marktreifer Software

- Softwarehersteller sind nicht an langfristiger Unterstützung ausgelieferter Produkte interessiert.
- Stattdessen: möglichst schnelle Funktionserweiterungen



Zunehmende Qualitätsanforderungen

- Gefundene Defekte in 1000 Zeilen Quellcode (M. Cusumano, MIT 1990):
  - 1977: 7 - 20 Defekte
  - 1994: 0,05 - 0,2 Defekte
- Steigerung des Qualitätsniveaus um den Faktor 100 in 13 Jahren.
- Aber: **Komplexitätssteigerung** muss **kompensiert** werden !
  - Komplexitätssteigerung ca. Faktor 10 in 5 Jahren
- Zunehmende „Altlasten":
  - Anwendungssoftware wird oft 20 Jahre und länger eingesetzt.
  - In manchen Betrieben sind 60-70% der Softwarekosten für Anpassung von Altsoftware !

Softwaretechnik/Softwaretechnologie

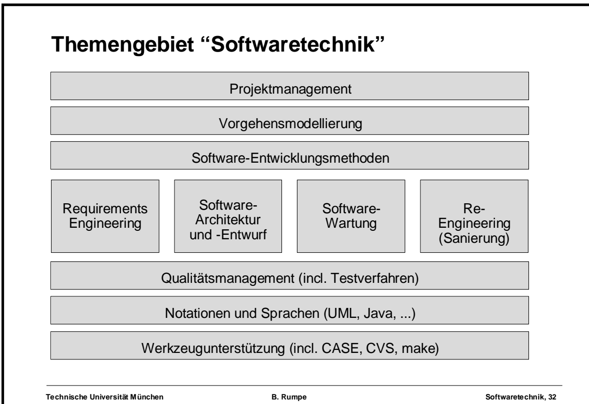
> software engineering: The establishment and use of sound engineering principles in order to obtain economically software that is reliable and runs on real machines.
> (F.L. Bauer, NATO-Konferenz Software-Engineering 1968)

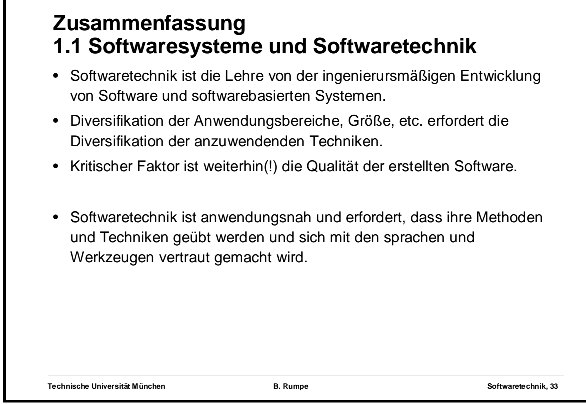
> software engineering: The practical application of scientific knowledge in the design and construction of computer programs and the associated documentation required to develop, operate, and maintain them
> (Berry Boehm 76)

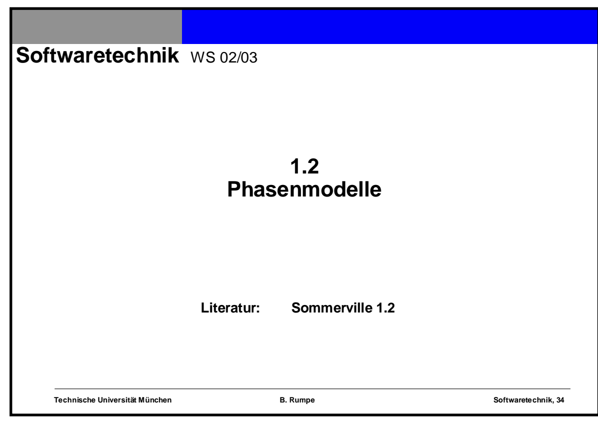
- Prinzipien einer Ingenieursdisziplin: Bereitstellung und systematische Verwendung von Methoden,Verfahren und Werkzeugen zur Lösung von Problemstellungen
- **Softwaretechnik, Softwaretechnologie**:
  - Meist als deutsche Synonyme für *software engineering* gebraucht.
- „Technologie": Lehre von der Herstellung technischer Produkte

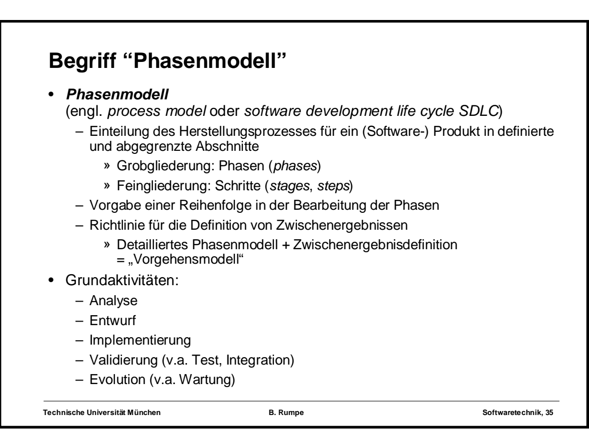
Aufgabenstellung der Softwaretechnik

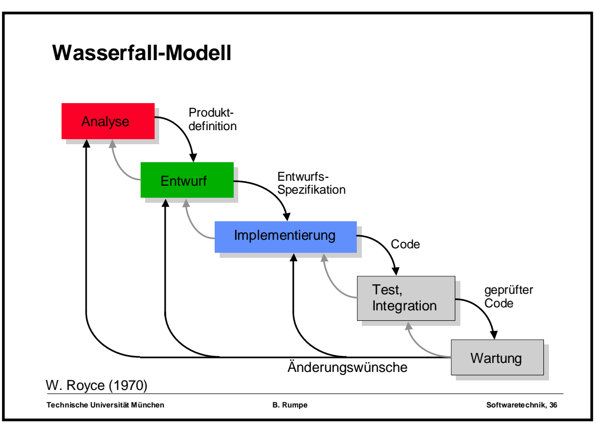
- Softwareentwicklung ist viel mehr als Programmieren.
- Dazu gehören:
  - Management großer und extrem komplexer Projekte
  - Präzise Erfassung und Erreichung von Kunden- und Marktanforderungen
  - Effizienzsteigerung in der Softwareentwicklung
  - Sicherstellung eines hohen Qualitätsniveaus
  - Berücksichtigung von Wartbarkeit und existierenden Altsystemen
- Es gehören aber auch dazu:
  - Guter Programmierstil
  - Effizienter Einsatz leistungsfähiger Programmiersprachen
  - Entwicklungswerkzeuge
  - Prinzipien wie
    Abstraktion, Strukturierung, Hierarchisierung und Modularisierung

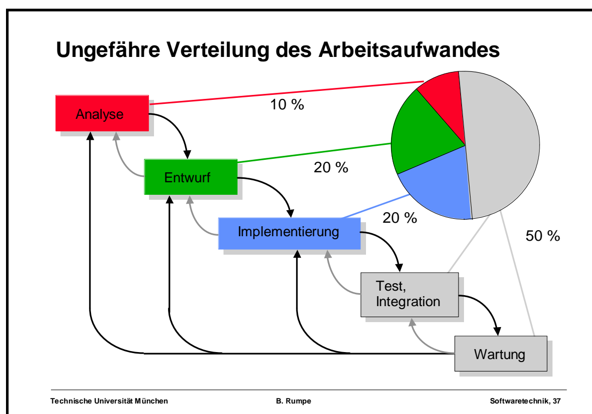


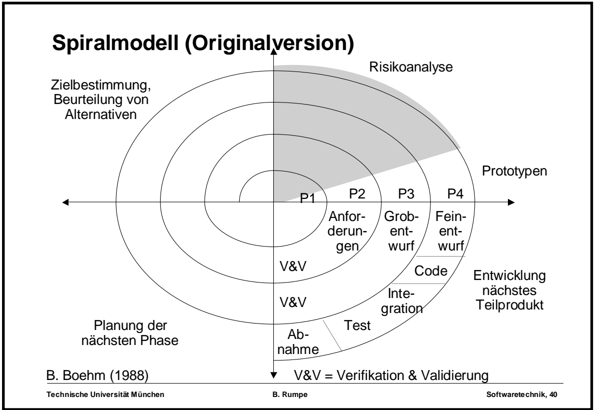


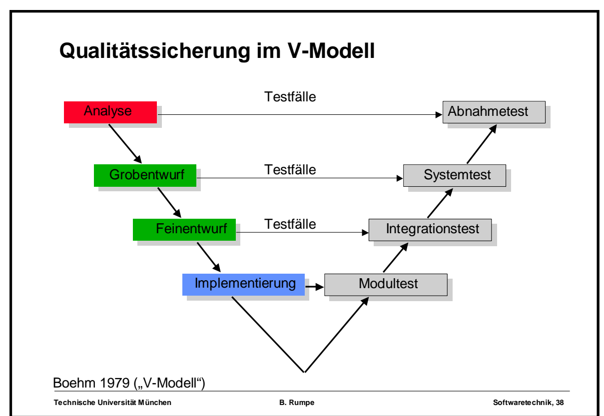


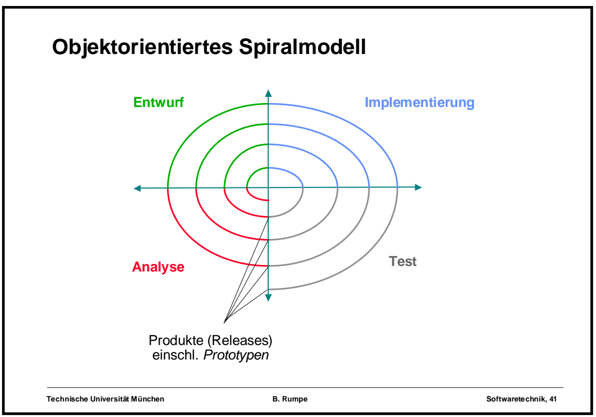


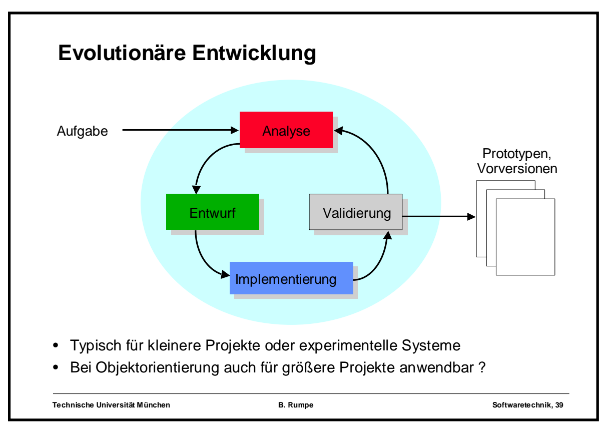


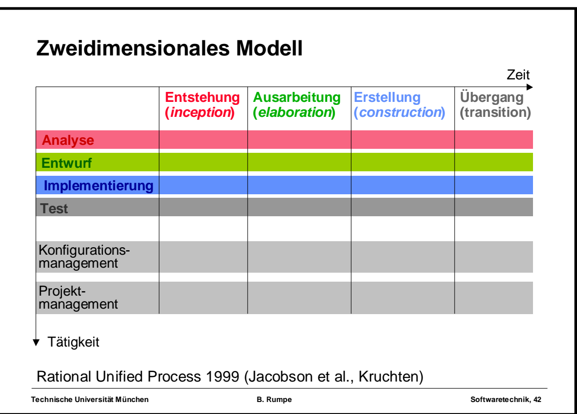

Ungefähre Verteilung des Arbeitsaufwandes
Analyse
10 %
Entwurf
20 %
Implementierung
20 %
50 %
Test, Integration
Wartung
Technische Universität München
B. Rumpe
Softwaretechnik, 37

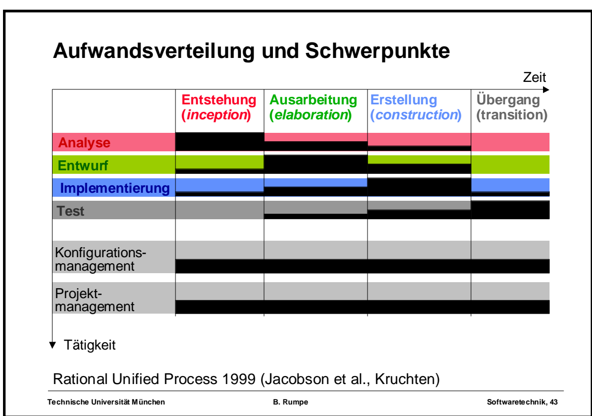

Spiralmodell (Originalversion)
Risikoanalyse
Zielbestimmung, Beurteilung von Alternativen
Prototypen
P1
P2
P3
P4
Anfor-derun-gen
Grob-ent-wurf
Fein-ent-wurf
Code
V&V
V&V
Inte-gration
Test
Ab-nahme
Entwicklung nächstes Teilprodukt
Planung der nächsten Phase
B. Boehm (1988)
V&V = Verifikation & Validierung
Technische Universität München
B. Rumpe
Softwaretechnik, 40

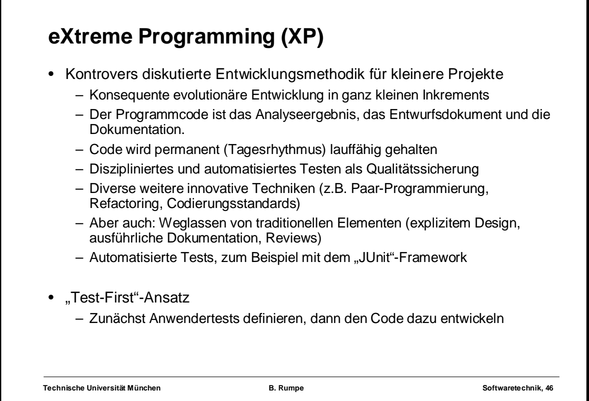

Qualitätssicherung im V-Modell
Analyse
Testfälle
Abnahmetest
Grobentwurf
Testfälle
Systemtest
Feinentwurf
Testfälle
Integrationstest
Implementierung
Modultest
Boehm 1979 („V-Modell“)
Technische Universität München
B. Rumpe
Softwaretechnik, 38

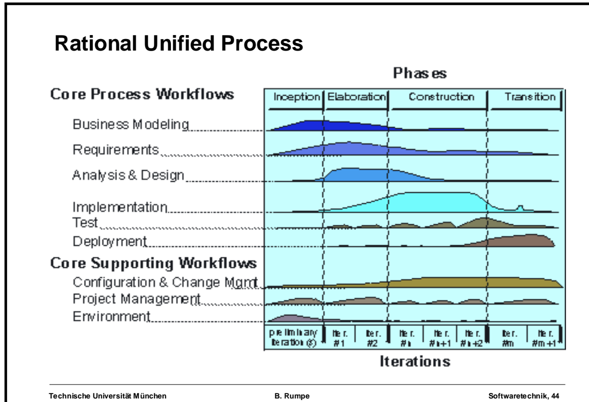

Objektorientiertes Spiralmodell
Entwurf
Implementierung
Analyse
Test
Produkte (Releases) einschl. Prototypen
Technische Universität München
B. Rumpe
Softwaretechnik, 41

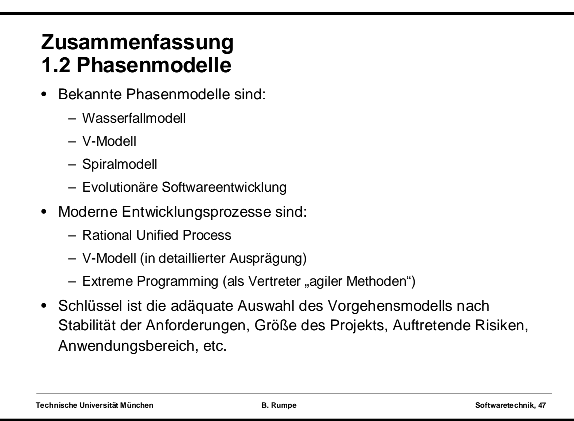

Evolutionäre Entwicklung
Aufgabe
Analyse
Entwurf
Validierung
Implementierung
Prototypen, Vorversionen
• Typisch für kleinere Projekte oder experimentelle Systeme
• Bei Objektorientierung auch für größere Projekte anwendbar ?
Technische Universität München
B. Rumpe
Softwaretechnik, 39

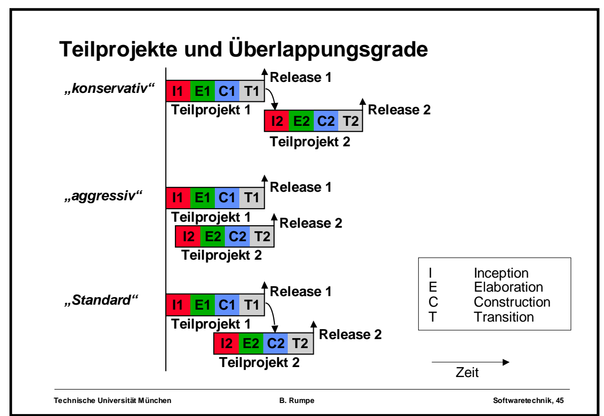

Zweidimensionales Modell
Zeit
Entstehung (inception)
Ausarbeitung (elaboration)
Erstellung (construction)
Übergang (transition)
Analyse
Entwurf
Implementierung
Test
Konfigurations-management
Projekt-management
Tätigkeit
Rational Unified Process 1999 (Jacobson et al., Kruchten)
Technische Universität München
B. Rumpe
Softwaretechnik, 42

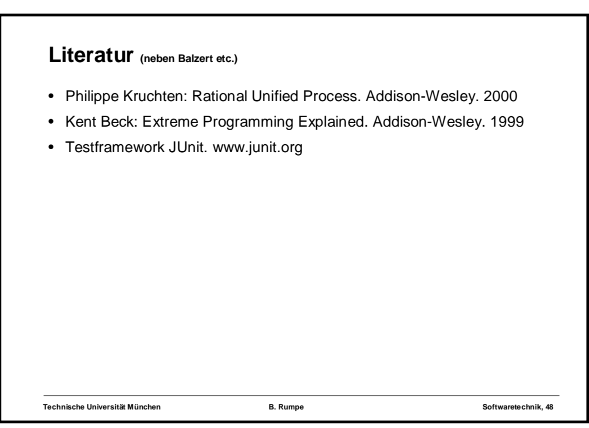

Aufwandsverteilung und Schwerpunkte
Zeit
Entstehung (inception)
Ausarbeitung (elaboration)
Erstellung (construction)
Übergang (transition)
Analyse
Entwurf
Implementierung
Test
Konfigurations-management
Projekt-management
Tätigkeit
Rational Unified Process 1999 (Jacobson et al., Kruchten)
Technische Universität München
B. Rumpe
Softwaretechnik, 43

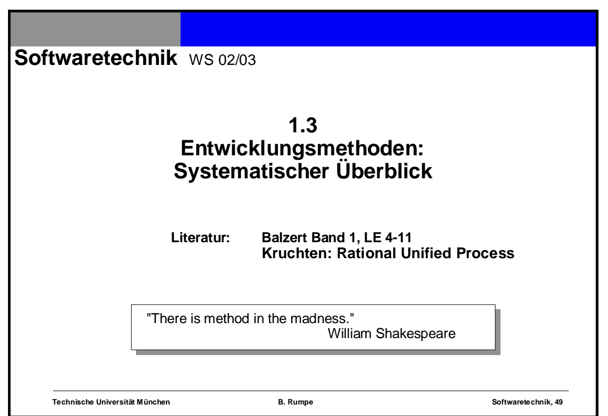

eXtreme Programming (XP)
Kontrovers diskutierte Entwicklungsmethodik für kleinere Projekte
Konsequente evolutionäre Entwicklung in ganz kleinen Inkrements
Der Programmcode ist das Analyseergebnis, das Entwurfsdokument und die Dokumentation.
Code wird permanent (Tagesrhythmus) lauffähig gehalten
Diszipliniertes und automatisiertes Testen als Qualitätssicherung
Diverse weitere innovative Techniken (z.B. Paar-Programmierung, Refactoring, Codierungsstandards)
Aber auch: Weglassen von traditionellen Elementen (explizitem Design, ausführliche Dokumentation, Reviews)
Automatisierte Tests, zum Beispiel mit dem „JUnit"-Framework
„Test-First"-Ansatz
Zunächst Anwendertests definieren, dann den Code dazu entwickeln
Technische Universität München
B. Rumpe
Softwaretechnik, 46

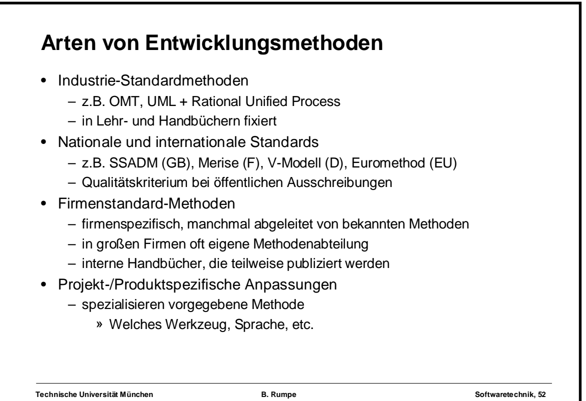

Rational Unified Process
Phases
Core Process Workflows
Inception
Elaboration
Construction
Transition
Business Modeling
Requirements
Analysis & Design
Implementation
Test
Deployment
Core Supporting Workflows
Configuration & Change Mgmt
Project Management
Environment
Iterations
Technische Universität München
B. Rumpe
Softwaretechnik, 44

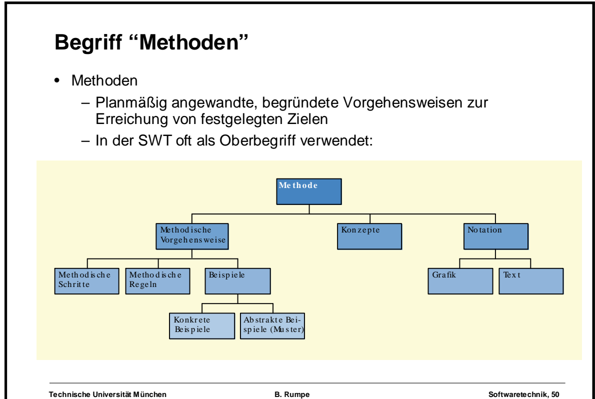

Zusammenfassung
1.2 Phasenmodelle
Bekannte Phasenmodelle sind:
Wasserfallmodell
V-Modell
Spiralmodell
Evolutionäre Softwareentwicklung
Moderne Entwicklungsprozesse sind:
Rational Unified Process
V-Modell (in detaillierter Ausprägung)
Extreme Programming (als Vertreter „agiler Methoden")
Schlüssel ist die adäquate Auswahl des Vorgehensmodells nach Stabilität der Anforderungen, Größe des Projekts, Auftretende Risiken, Anwendungsbereich, etc.
Technische Universität München
B. Rumpe
Softwaretechnik, 47

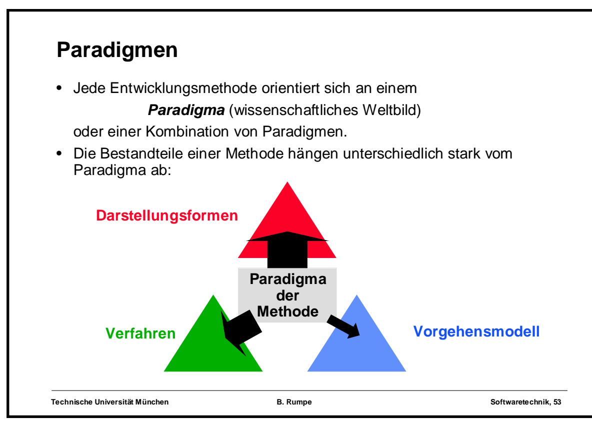

Teilprojekte und Überlappungsgrade
„konservativ"
„aggressiv"
„Standard"
I1 E1 C1 T1
I2 E2 C2 T2
Teilprojekt 1
Teilprojekt 2
Release 1
Release 2
I Inception
E Elaboration
C Construction
T Transition
Zeit
Technische Universität München
B. Rumpe
Softwaretechnik, 45

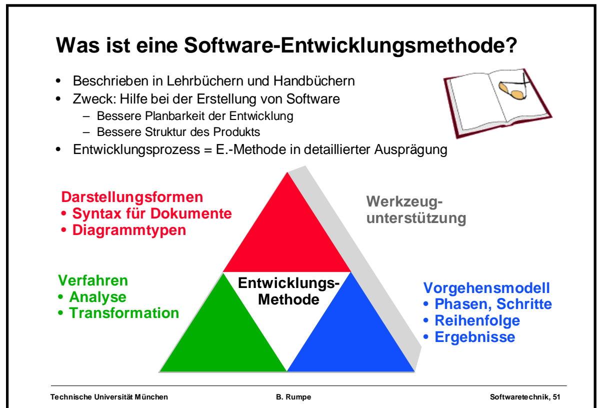

Literatur (neben Balzert etc.)
Philippe Kruchten: Rational Unified Process. Addison-Wesley. 2000
Kent Beck: Extreme Programming Explained. Addison-Wesley. 1999
Testframework JUnit. www.junit.org
Technische Universität München
B. Rumpe
Softwaretechnik, 48

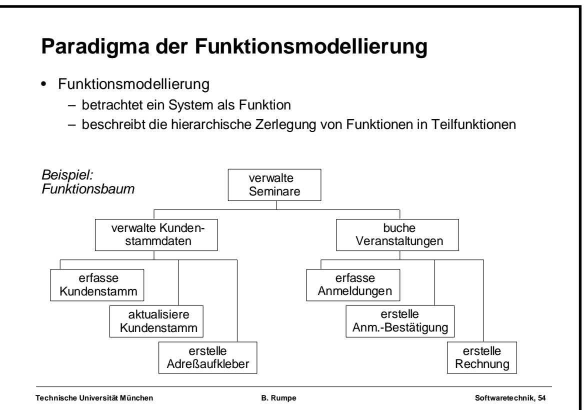

Softwaretechnik WS 02/03
1.3
Entwicklungsmethoden:
Systematischer Überblick
Literatur: Balzert Band 1, LE 4-11
Kruchten: Rational Unified Process
"There is method in the madness."
William Shakespeare
Technische Universität München
B. Rumpe
Softwaretechnik, 49

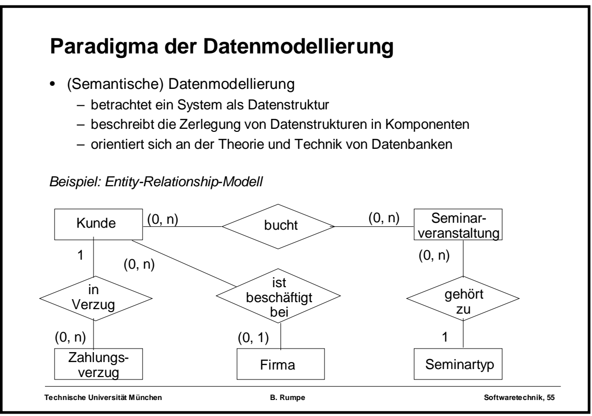

Arten von Entwicklungsmethoden
• Industrie-Standardmethoden
– z.B. OMT, UML + Rational Unified Process
– in Lehr- und Handbüchern fixiert
• Nationale und internationale Standards
– z.B. SSADM (GB), Merise (F), V-Modell (D), Euromethod (EU)
– Qualitätskriterium bei öffentlichen Ausschreibungen
• Firmenstandard-Methoden
– firmenspezifisch, manchmal abgeleitet von bekannten Methoden
– in großen Firmen oft eigene Methodenabteilung
– interne Handbücher, die teilweise publiziert werden
• Projekt-/Produktspezifische Anpassungen
– spezialisieren vorgegebene Methode
» Welches Werkzeug, Sprache, etc.
Technische Universität München
B. Rumpe
Softwaretechnik, 52


Begriff "Methoden"
• Methoden
– Planmäßig angewandte, begründete Vorgehensweisen zur Erreichung von festgelegten Zielen
– In der SWT oft als Oberbegriff verwendet:
Methode
Methodische Vorgehensweise
Konzepte
Notation
Methodische Schritte
Methodische Regeln
Beispiele
Grafik
Text
Konkrete Beispiele
Abstrakte Beispiele (Muster)
Technische Universität München
B. Rumpe
Softwaretechnik, 50

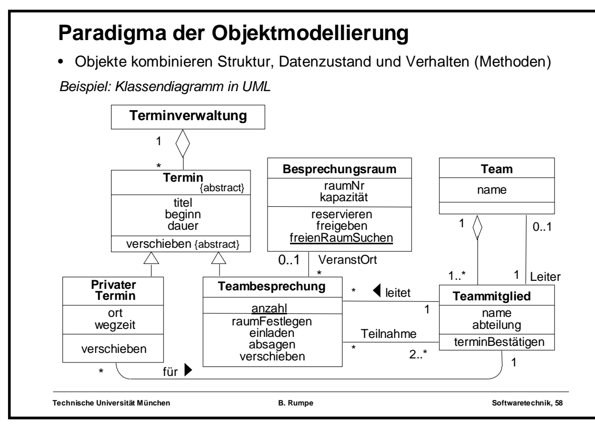

Paradigmen
• Jede Entwicklungsmethode orientiert sich an einem Paradigma (wissenschaftliches Weltbild) oder einer Kombination von Paradigmen.
• Die Bestandteile einer Methode hängen unterschiedlich stark vom Paradigma ab:
Darstellungsformen
Paradigma der Methode
Verfahren
Vorgehensmodell
Technische Universität München
B. Rumpe
Softwaretechnik, 53


Was ist eine Software-Entwicklungsmethode?
• Beschrieben in Lehrbüchern und Handbüchern
• Zweck: Hilfe bei der Erstellung von Software
– Bessere Planbarkeit der Entwicklung
– Bessere Struktur des Produkts
• Entwicklungsprozess = E.-Methode in detaillierter Ausprägung
Darstellungsformen
• Syntax für Dokumente
• Diagrammtypen
Werkzeug-unterstützung
Verfahren
• Analyse
• Transformation
Entwicklungs-Methode
Vorgehensmodell
• Phasen, Schritte
• Reihenfolge
• Ergebnisse
Technische Universität München
B. Rumpe
Softwaretechnik, 51

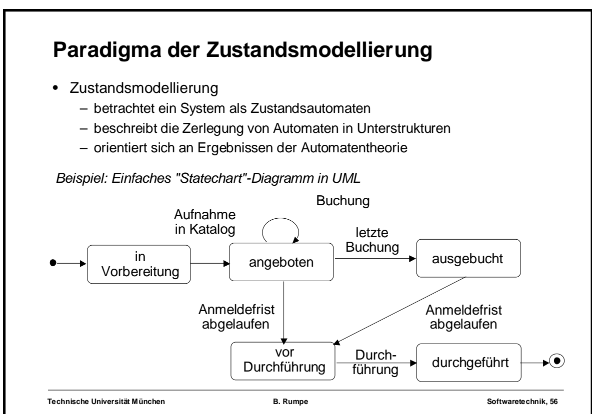

Paradigma der Funktionsmodellierung
• Funktionsmodellierung
– betrachtet ein System als Funktion
– beschreibt die hierarchische Zerlegung von Funktionen in Teilfunktionen
Beispiel: Funktionsbaum
verwalte Seminare
verwalte Kundenstammdaten
buche Veranstaltungen
erfasse Kundenstamm
aktualisiere Kundenstamm
erstelle Adreßaufkleber
erfasse Anmeldungen
erstelle Anm.-Bestätigung
erstelle Rechnung
Technische Universität München
B. Rumpe
Softwaretechnik, 54



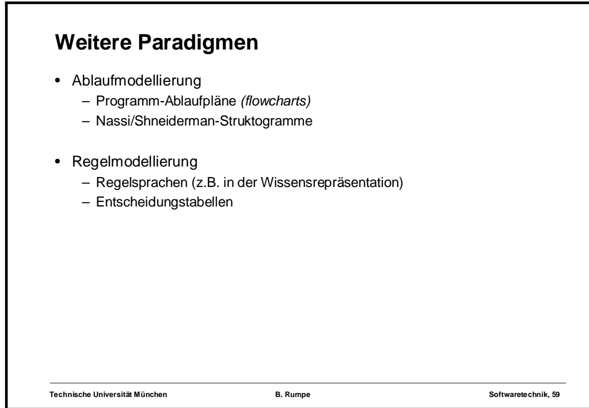
Paradigma der Datenmodellierung
• (Semantische) Datenmodellierung
– betrachtet ein System als Datenstruktur
– beschreibt die Zerlegung von Datenstrukturen in Komponenten
– orientiert sich an der Theorie und Technik von Datenbanken
Beispiel: Entity-Relationship-Modell
Kunde
bucht
Seminar-veranstaltung
in Verzug
ist beschäftigt bei
gehört zu
Zahlungs-verzug
Firma
Seminartyp
Technische Universität München
B. Rumpe
Softwaretechnik, 55



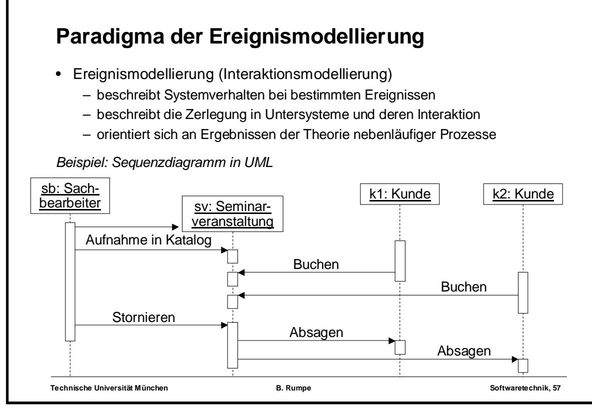
Paradigma der Objektmodellierung
• Objekte kombinieren Struktur, Datenzustand und Verhalten (Methoden)
Beispiel: Klassendiagramm in UML
Terminverwaltung
Termin {abstract}
titel
beginn
dauer
verschieben {abstract}
Besprechungsraum
raumNr
kapazität
reservieren
freigeben
freienRaumSuchen
Team
name
Privater Termin
ort
wegzeit
verschieben
Teambesprechung
anzahl
raumFestlegen
einladen
absagen
verschieben
Teammitglied
name
abteilung
terminBestätigen
VeranstOrt
leitet
Teilnahme
Leiter
für
Technische Universität München
B. Rumpe
Softwaretechnik, 58



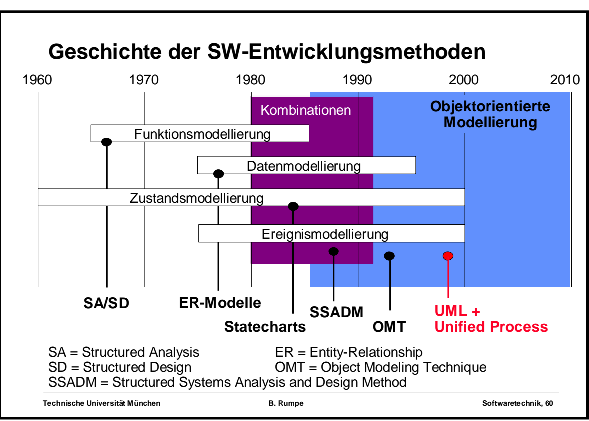
Paradigma der Zustandsmodellierung
• Zustandsmodellierung
– betrachtet ein System als Zustandsautomaten
– beschreibt die Zerlegung von Automaten in Unterstrukturen
– orientiert sich an Ergebnissen der Automatentheorie
Beispiel: Einfaches "Statechart"-Diagramm in UML
Buchung
Aufnahme in Katalog
letzte Buchung
in Vorbereitung
angeboten
ausgebucht
Anmeldefrist abgelaufen
Anmeldefrist abgelaufen
vor Durchführung
Durch-führung
durchgeführt
Technische Universität München
B. Rumpe
Softwaretechnik, 56



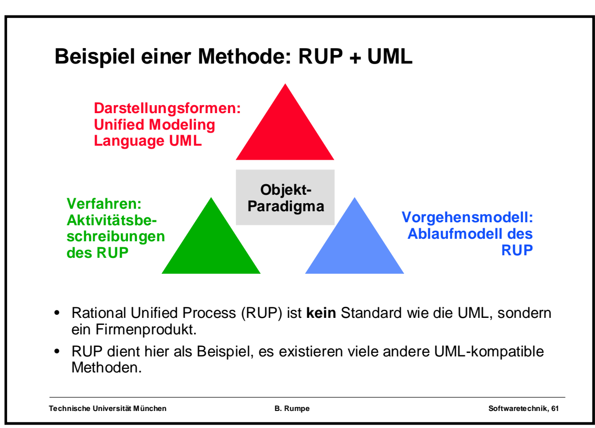
Weitere Paradigmen
• Ablaufmodellierung
– Programm-Ablaufpläne (flowcharts)
– Nassi/Shneiderman-Struktogramme
• Regelmodellierung
– Regelsprachen (z.B. in der Wissensrepräsentation)
– Entscheidungstabellen
Technische Universität München
B. Rumpe
Softwaretechnik, 59

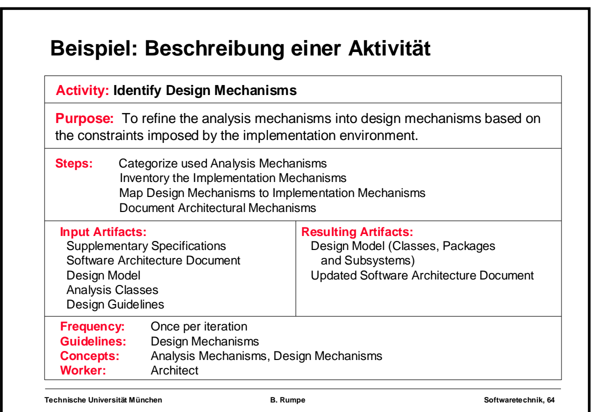


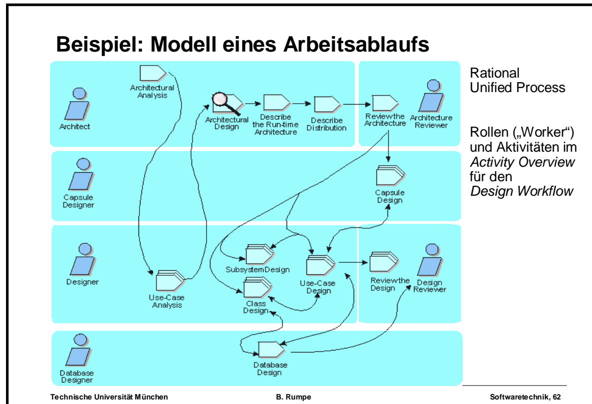
Paradigma der Ereignismodellierung
• Ereignismodellierung (Interaktionsmodellierung)
– beschreibt Systemverhalten bei bestimmten Ereignissen
– beschreibt die Zerlegung in Untersysteme und deren Interaktion
– orientiert sich an Ergebnissen der Theorie nebenläufiger Prozesse
Beispiel: Sequenzdiagramm in UML
sb: Sach-bearbeiter
sv: Seminar-veranstaltung
k1: Kunde
k2: Kunde
Aufnahme in Katalog
Buchen
Buchen
Stornieren
Absagen
Absagen
Technische Universität München
B. Rumpe
Softwaretechnik, 57

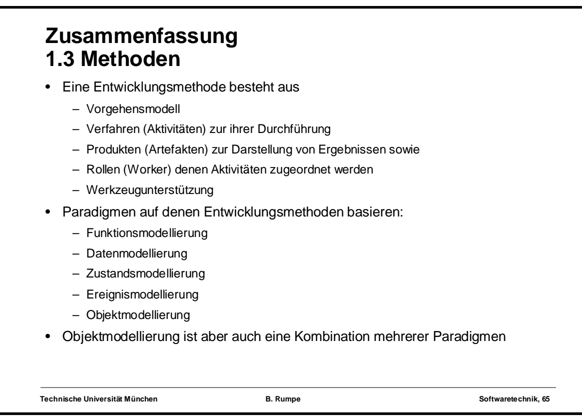


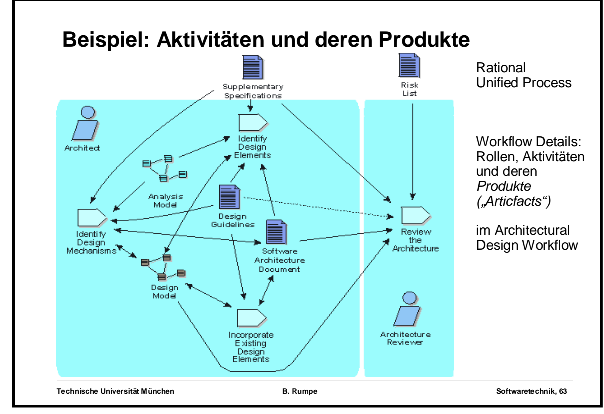
Geschichte der SW-Entwicklungsmethoden
1960
1970
1980
1990
2000
2010
Kombinationen
Objektorientierte Modellierung
Funktionsmodellierung
Datenmodellierung
Zustandsmodellierung
Ereignismodellierung
SA/SD
ER-Modelle
Statecharts
SSADM
OMT
UML + Unified Process
SA = Structured Analysis
SD = Structured Design
SSADM = Structured Systems Analysis and Design Method
ER = Entity-Relationship
OMT = Object Modeling Technique
Technische Universität München
B. Rumpe
Softwaretechnik, 60

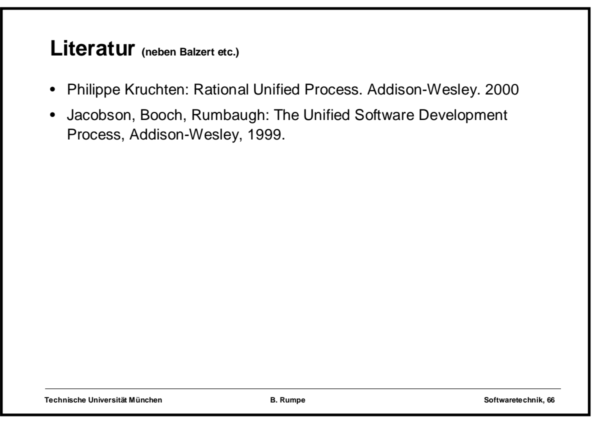

Beispiel einer Methode: RUP + UML
Darstellungsformen: Unified Modeling Language UML
Objekt-Paradigma
Verfahren: Aktivitätsbe-schreibungen des RUP
Vorgehensmodell: Ablaufmodell des RUP
• Rational Unified Process (RUP) ist kein Standard wie die UML, sondern ein Firmenprodukt.
• RUP dient hier als Beispiel, es existieren viele andere UML-kompatible Methoden.
Technische Universität München
B. Rumpe
Softwaretechnik, 61


Beispiel: Beschreibung einer Aktivität
Activity: Identify Design Mechanisms
Purpose: To refine the analysis mechanisms into design mechanisms based on the constraints imposed by the implementation environment.
Steps: Categorize used Analysis Mechanisms
Inventory the Implementation Mechanisms
Map Design Mechanisms to Implementation Mechanisms
Document Architectural Mechanisms
Input Artifacts: Supplementary Specifications
Software Architecture Document
Design Model
Analysis Classes
Design Guidelines
Resulting Artifacts: Design Model (Classes, Packages and Subsystems)
Updated Software Architecture Document
Frequency: Once per iteration
Guidelines: Design Mechanisms
Concepts: Analysis Mechanisms, Design Mechanisms
Worker: Architect
Technische Universität München
B. Rumpe
Softwaretechnik, 64


Beispiel: Modell eines Arbeitsablaufs
Architect
Architectural Analysis
Architectural Design
Describe the Run-time Architecture
Describe Distribution
Review the Architecture
Architecture Reviewer
Capsule Designer
Capsule Design
Designer
Use-Case Analysis
Subsystem Design
Use-Case Design
Class Design
Review the Design
Design Reviewer
Database Designer
Database Design
Rational Unified Process
Rollen („Worker") und Aktivitäten im Activity Overview für den Design Workflow
Technische Universität München
B. Rumpe
Softwaretechnik, 62


Zusammenfassung
1.3 Methoden
• Eine Entwicklungsmethode besteht aus
– Vorgehensmodell
– Verfahren (Aktivitäten) zur ihrer Durchführung
– Produkten (Artefakten) zur Darstellung von Ergebnissen sowie
– Rollen (Worker) denen Aktivitäten zugeordnet werden
– Werkzeugunterstützung
• Paradigmen auf denen Entwicklungsmethoden basieren:
– Funktionsmodellierung
– Datenmodellierung
– Zustandsmodellierung
– Ereignismodellierung
– Objektmodellierung
• Objektmodellierung ist aber auch eine Kombination mehrerer Paradigmen
Technische Universität München
B. Rumpe
Softwaretechnik, 65


Beispiel: Aktivitäten und deren Produkte
Supplementary Specifications
Risk List
Architect
Identify Design Elements
Analysis Model
Design Guidelines
Identify Design Mechanisms
Software Architecture Document
Review the Architecture
Design Model
Incorporate Existing Design Elements
Architecture Reviewer
Rational Unified Process
Workflow Details: Rollen, Aktivitäten und deren Produkte („Articfacts")
im Architectural Design Workflow
Technische Universität München
B. Rumpe
Softwaretechnik, 63


Literatur (neben Balzert etc.)
• Philippe Kruchten: Rational Unified Process. Addison-Wesley. 2000
• Jacobson, Booch, Rumbaugh: The Unified Software Development Process, Addison-Wesley, 1999.
Technische Universität München
B. Rumpe
Softwaretechnik, 66

Softwaretechnik WS 02/03

1.4
Modelle und Modellierungstechniken
Überblick

Begriff "Modell"

- Ein ***Modell*** ist seinem Wesen nach eine in Maßstab, Detailliertheit und/oder Funktionalität verkürzte beziehungsweise abstrahierte Darstellung des originalen Systems [Stachowiak 73]

- Ein ***Modell*** ist eine Abstraktion eines Systems mit der Zielsetzung das Nachdenken über ein System zu vereinfachen, indem irrelevante Details ausgelassen werden [Brügge 00].

- Beispiele für Modelle:
  - Schaltbild, Baupläne, Architekturmodell, Anatomie Gerippe, Modelleisenbahn, Drehbuch, Höhlenzeichnungen
  - Begriff „Modell " taucht zuerst im 13Jhr. beim Kirchenbau auf: ein Modell ist dort z.B. eine begehbare 1:10-Verkleinerung eines Doms

Modelle

- sind abstrahierend,
- haben einen Bezug zu einem Original und
- besitzen einen Zweck.

- Die **Abstraktion** erlaubt es die für den Zweck uninteressanten Details zu entfernen, um so bestimmte Eigenschaften des Modells besser studieren zu können.
- **Abbildungstreue**: Sind die (zu studierenden) Eigenschaften des Originals im Modell richtig wiedergegeben?
- **Deskriptives Modell**: das nach dem Original gebildet wird.
- **Präskriptives Modell**: zuerst das Modell, dann das Original (typisch für Software-Entwicklung)

Arten von Modellen - 1

klassifiziert nach Einsatzbereich:

- **Entwurfsmodell** beschreibt wesentliche Entwurfsentscheidungen, die Softwarestruktur, Klassenaufteilung etc.

- **Analysemodell,** etc.

- **Implementierung**: Auch der Code ist ein „Modell" des Systems

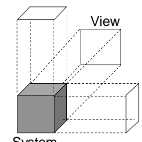
- **Vorgehensmodell** ist ein Modell des Projekts (der Vorgehensweise)

- **Produktmodell** ist eine Beschreibung der im Projekt entstehenden Artefakte und ihrer Zusammenhänge.

Arten von Modellen - 2

klassifiziert nach Sicht („View"):

- **Strukturmodell** beschreibt die Struktur des Systems oder eines Subsystems
  → UML: Klassen- und Objektdiagramme
- **Interaktionsmodell**: Nachrichtenaustausch oder Methodenaufrufe zwischen Systemkomponenten
  → UML: Sequenzdiagramme
- **Zustandsmodell** beschreibt den Zustandsraum einer Systemkomponente (Objekt) → UML: Statecharts



- **Testmodell, Rollenmodell, Anwendungsfall-Modell, Geschäftsmodell, Kollaborationsmodell**, etc.

- Eine „Sicht" ist eine auf bestimmte Aspekte reduzierte Darstellung des Systems. Orthogonale Sichten werden gemeinsam eingesetzt, um ein System zu beschreiben.

Beschreibungstechniken

- Zur Darstellung von Modellen ist eine **Notation** (**Beschreibungstechnik**) notwendig.

- Beispiele:
  - Text, Diagramm, Tabelle, Formeln, Programmtext
  - Strukturierter Text (z.B. eingeschränkte Satzform), Pseudocode, ...

- Für die Softwareentwicklung sind eine Vielzahl an Modellen erfunden und vorgeschlagen worden:
  - Petri-Netze, Statecharts, Ablaufdiagramme, Workflow-Diagramme, Algebraische Spezifikation, Logiken, E/R-Modell, etc.

  - In dieser Vorlesung: Im wesentlichen Konzentration auf die UML-Notationen

Bestandteile einer Notation

- Syntax
  - Konkretes Aussehen
  - Abstrakte Syntax
  - Kontextbedingungen zur Wohlgeformtheit
- Semantik
  - Bedeutungsbeschreibung (Semantik)
  - Interpretation in der realen Welt (bzw. der beschriebenen Domäne)
- Pragmatik (Methodik des Gebrauchs)
  - Analysetechniken (Typprüfung, Konsistenzchecks)
  - Simulationstechniken
  - Transformationstechniken (Z.B. Refactoring)
  - Generierungsmöglichkeiten

- Ziel ist eine „Theorie“ zur Bearbeitung der in einer Notation getroffenen Aussagen.
- Aber auch: Dokumentation und Kommunikation

Zusammenhänge zwischen Modellen

- Strukturelle Abhängigkeiten:
  - ist Teil/Erweiterung von
  - ist Ergänzung zu
  - importiert oder nutzt Elemente von
- kausale Abhängigkeiten
  - wird benötigt für/stützt sich auf
  - ist Vorversion von
  - ist Beispiel für
  - ist Prüfergebnis von
- semantische Beziehung
  - ist Übersetzung von/ist Source von
  - ist Spezifikation von/ist Implementierung von
  - ist Abstraktion von/ist Detaillierung von
  - ist durch Transformation entstanden aus

- Vielfältige Beziehungen zwischen Modellen müssen im Entwicklungssystem verwaltet bzw. berechnet werden können.

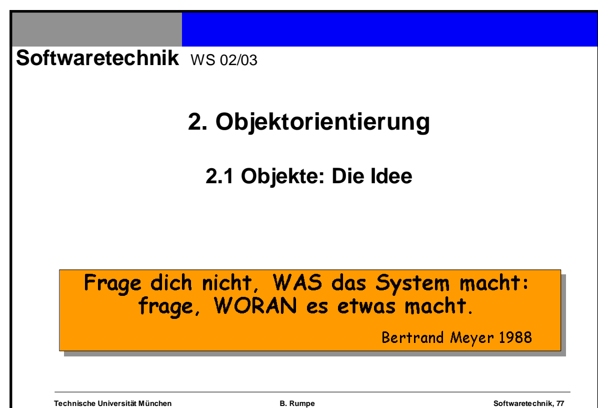
Warum UML?

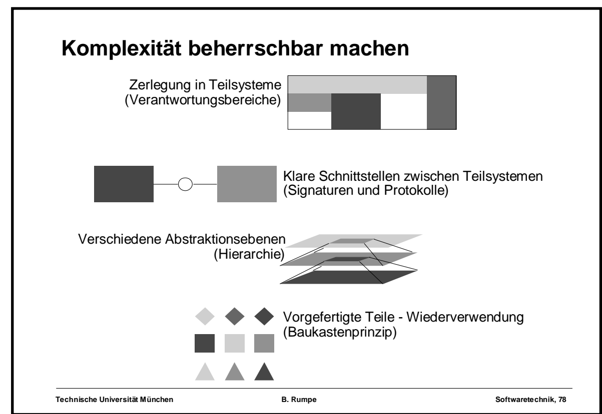
- Vorteile:
  - Objektorientierung ist zur Zeit das optimale Modellierungs-Paradigma
  - Kombination von Struktur-, Verhalten-, Interaktion-, und Verteilung
  - Für Analyse, Entwurf, Implementierung und Test einsetzbar.
  - Gute Werkzeugunterstützung für Editieren, Versionierung, Codegenerierung
  - Abstimmungsaufwand zur Erarbeitung einer gemeinsamen Kommunikationssprache ist begrenzt (aber nicht Null!)
  - Erweiterbarkeit der UML mit Stereotypen und Tags.
- Nachteile:
  - UML ist in vielen Facetten nicht präzise festgelegt.
  - Werkzeuge für Transformation, Analyse etc. fehlen noch.
  - UML ist keine „kleine Sprache“: Lernaufwand notwendig
  - Komponenten sind nicht adäquat darstellbar

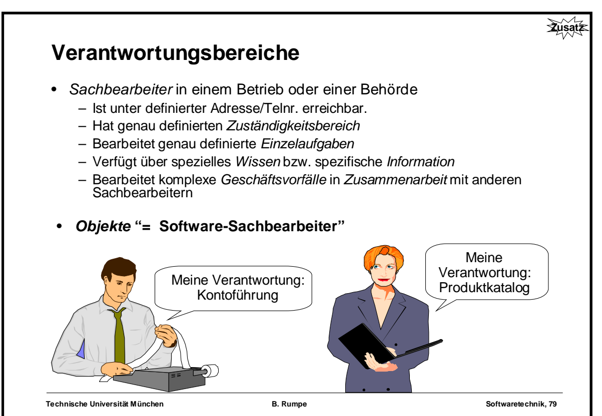
- Sprachen wie die UML werden erlernt durch Übung!

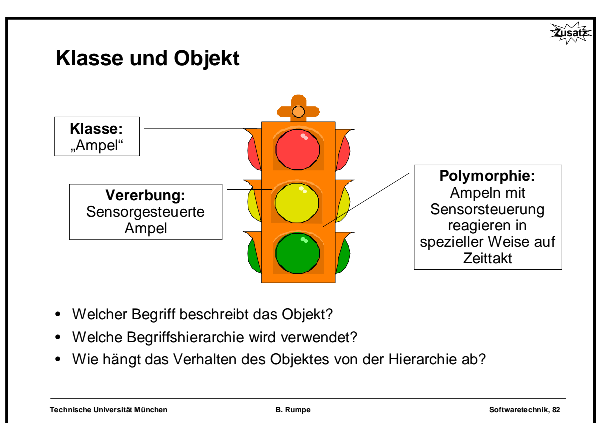
Zusammenfassung 1.4 Modelle

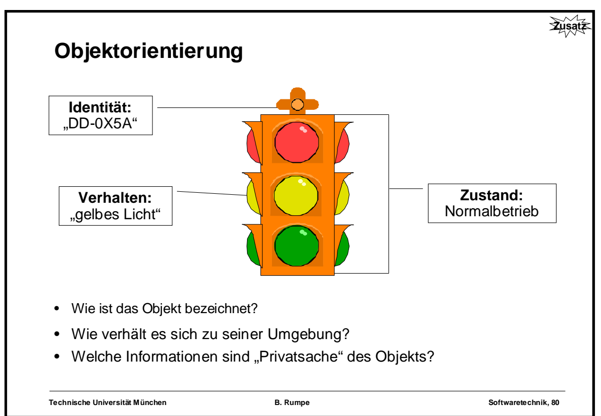
- Modelle sind abstrahierend, haben einen Bezug zu einem Original und besitzen einen Zweck.
- Ein Modell wird formuliert in einer Notation (Beschreibungstechnik) (textuell, tabellarisch oder graphisch) und besitzt Analyse-, Simulations-, Generierungsmechanismen, etc.
- UML bietet eine Kombination von neun Beschreibungstechniken für verschiedene „Sichten“.
- Eine „Sicht“ (view) ist eine auf bestimmte Aspekte (Verhalten, Struktur, Interaktionen, Verteilung) reduzierte Darstellung des Systems.

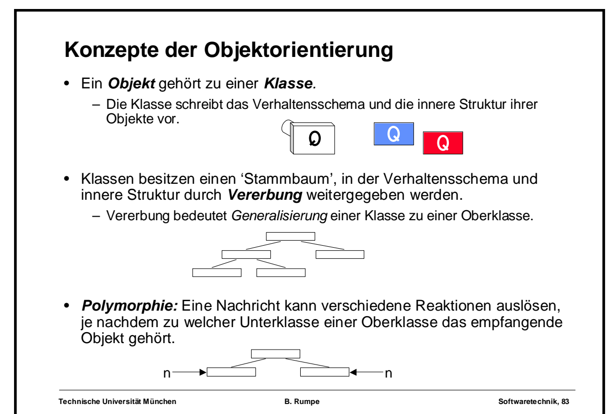


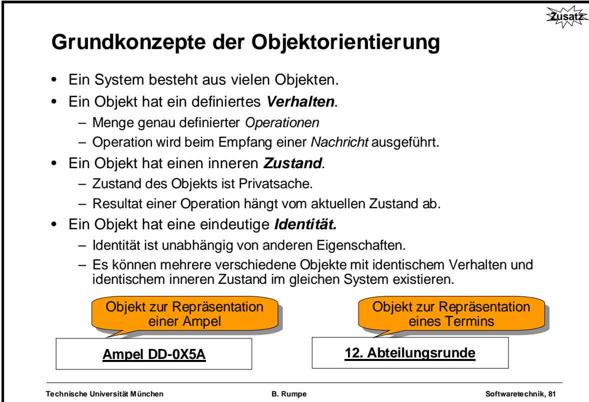


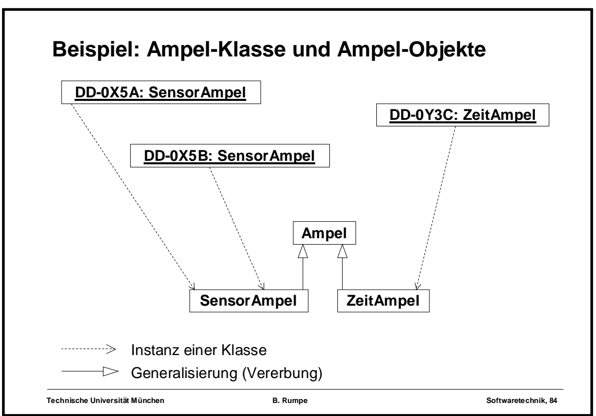

Zusatz
Verantwortungsbereiche
• Sachbearbeiter in einem Betrieb oder einer Behörde
– Ist unter definierter Adresse/Telnr. erreichbar.
– Hat genau definierten Zuständigkeitsbereich
– Bearbeitet genau definierte Einzelaufgaben
– Verfügt über spezielles Wissen bzw. spezifische Information
– Bearbeitet komplexe Geschäftsvorfälle in Zusammenarbeit mit anderen Sachbearbeitern
• Objekte "= Software-Sachbearbeiter"
Meine Verantwortung: Kontoführung
Meine Verantwortung: Produktkatalog
Technische Universität München
B. Rumpe
Softwaretechnik, 79

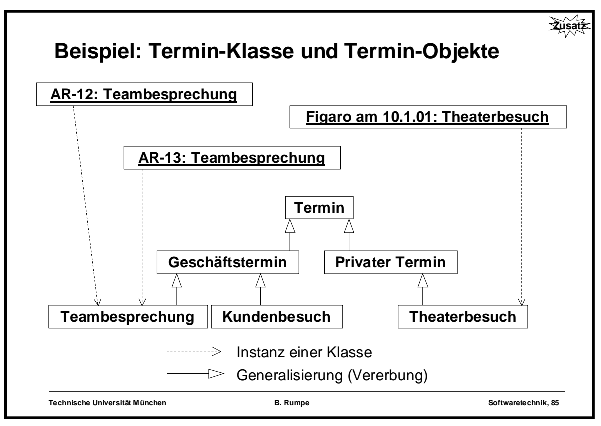

Zusatz
Klasse und Objekt
Klasse: „Ampel"
Vererbung: Sensorgesteuerte Ampel
Polymorphie: Ampeln mit Sensorsteuerung reagieren in spezieller Weise auf Zeittakt
• Welcher Begriff beschreibt das Objekt?
• Welche Begriffshierarchie wird verwendet?
• Wie hängt das Verhalten des Objektes von der Hierarchie ab?
Technische Universität München
B. Rumpe
Softwaretechnik, 82

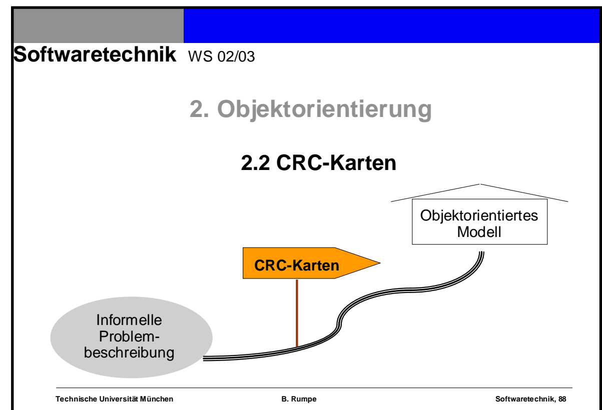

Zusatz
Objektorientierung
Identität: „DD-0X5A"
Verhalten: „gelbes Licht"
Zustand: Normalbetrieb
• Wie ist das Objekt bezeichnet?
• Wie verhält es sich zu seiner Umgebung?
• Welche Informationen sind „Privatsache" des Objekts?
Technische Universität München
B. Rumpe
Softwaretechnik, 80

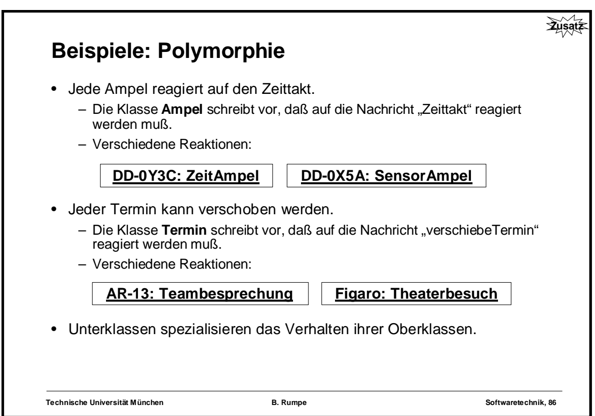

Konzepte der Objektorientierung
• Ein Objekt gehört zu einer Klasse.
– Die Klasse schreibt das Verhaltensschema und die innere Struktur ihrer Objekte vor.
• Klassen besitzen einen 'Stammbaum', in der Verhaltensschema und innere Struktur durch Vererbung weitergegeben werden.
– Vererbung bedeutet Generalisierung einer Klasse zu einer Oberklasse.
• Polymorphie: Eine Nachricht kann verschiedene Reaktionen auslösen, je nachdem zu welcher Unterklasse einer Oberklasse das empfangende Objekt gehört.
n
n
Technische Universität München
B. Rumpe
Softwaretechnik, 83

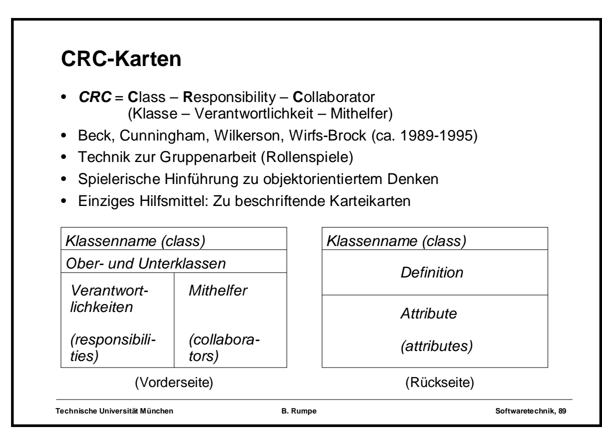

Zusatz
Grundkonzepte der Objektorientierung
• Ein System besteht aus vielen Objekten.
• Ein Objekt hat ein definiertes Verhalten.
– Menge genau definierter Operationen
– Operation wird beim Empfang einer Nachricht ausgeführt.
• Ein Objekt hat einen inneren Zustand.
– Zustand des Objekts ist Privatsache.
– Resultat einer Operation hängt vom aktuellen Zustand ab.
• Ein Objekt hat eine eindeutige Identität.
– Identität ist unabhängig von anderen Eigenschaften.
– Es können mehrere verschiedene Objekte mit identischem Verhalten und identischem inneren Zustand im gleichen System existieren.
Objekt zur Repräsentation einer Ampel
Objekt zur Repräsentation eines Termins
Ampel DD-0X5A
12. Abteilungsrunde
Technische Universität München
B. Rumpe
Softwaretechnik, 81

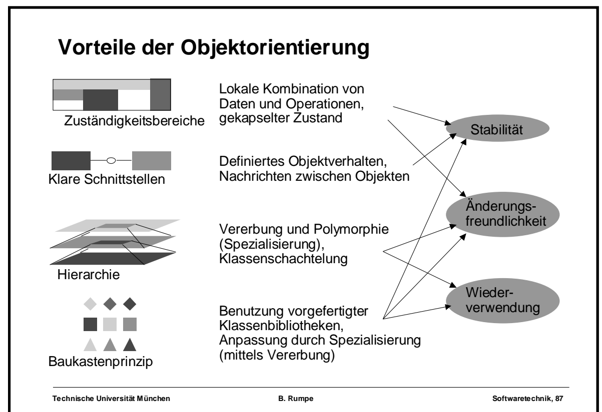

Beispiel: Ampel-Klasse und Ampel-Objekte
DD-0X5A: SensorAmpel
DD-0Y3C: ZeitAmpel
DD-0X5B: SensorAmpel
Ampel
SensorAmpel
ZeitAmpel
Instanz einer Klasse
Generalisierung (Vererbung)
Technische Universität München
B. Rumpe
Softwaretechnik, 84

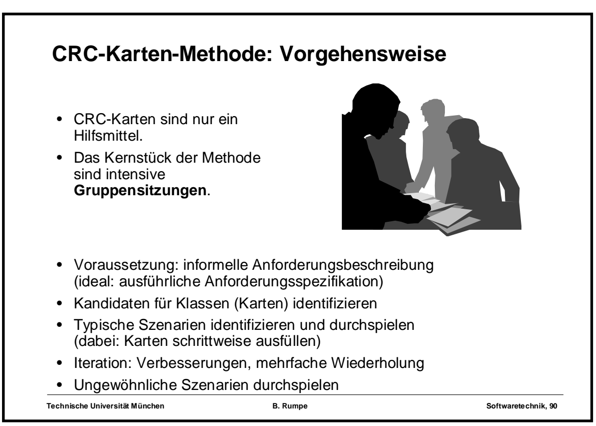

Zusatz
Beispiel: Termin-Klasse und Termin-Objekte
AR-12: Teambesprechung
Figaro am 10.1.01: Theaterbesuch
AR-13: Teambesprechung
Termin
Geschäftstermin
Privater Termin
Teambesprechung
Kundenbesuch
Theaterbesuch
Instanz einer Klasse
Generalisierung (Vererbung)
Technische Universität München
B. Rumpe
Softwaretechnik, 85


Softwaretechnik WS 02/03
2. Objektorientierung
2.2 CRC-Karten
Objektorientiertes Modell
CRC-Karten
Informelle Problem-beschreibung
Technische Universität München
B. Rumpe
Softwaretechnik, 88


Zusatz
Beispiele: Polymorphie
Jede Ampel reagiert auf den Zeittakt.
– Die Klasse Ampel schreibt vor, daß auf die Nachricht „Zeittakt“ reagiert werden muß.
– Verschiedene Reaktionen:
DD-0Y3C: ZeitAmpel
DD-0X5A: SensorAmpel
Jeder Termin kann verschoben werden.
– Die Klasse Termin schreibt vor, daß auf die Nachricht „verschiebeTermin“ reagiert werden muß.
– Verschiedene Reaktionen:
AR-13: Teambesprechung
Figaro: Theaterbesuch
Unterklassen spezialisieren das Verhalten ihrer Oberklassen.
Technische Universität München
B. Rumpe
Softwaretechnik, 86


CRC-Karten
CRC = Class – Responsibility – Collaborator
(Klasse – Verantwortlichkeit – Mithelfer)
Beck, Cunningham, Wilkerson, Wirfs-Brock (ca. 1989-1995)
Technik zur Gruppenarbeit (Rollenspiele)
Spielerische Hinführung zu objektorientiertem Denken
Einziges Hilfsmittel: Zu beschriftende Karteikarten
Klassenname (class)
Ober- und Unterklassen
Verantwort-lichkeiten
Mithelfer
(responsibili-ties)
(collabora-tors)
(Vorderseite)
Klassenname (class)
Definition
Attribute
(attributes)
(Rückseite)
Technische Universität München
B. Rumpe
Softwaretechnik, 89


Vorteile der Objektorientierung
Zuständigkeitsbereiche
Lokale Kombination von Daten und Operationen, gekapselter Zustand
Stabilität
Klare Schnittstellen
Definiertes Objektverhalten, Nachrichten zwischen Objekten
Änderungs-freundlichkeit
Hierarchie
Vererbung und Polymorphie (Spezialisierung), Klassenschachtelung
Wieder-verwendung
Baukastenprinzip
Benutzung vorgefertigter Klassenbibliotheken, Anpassung durch Spezialisierung (mittels Vererbung)
Technische Universität München
B. Rumpe
Softwaretechnik, 87


CRC-Karten-Methode: Vorgehensweise
CRC-Karten sind nur ein Hilfsmittel.
Das Kernstück der Methode sind intensive Gruppensitzungen.
Voraussetzung: informelle Anforderungsbeschreibung (ideal: ausführliche Anforderungsspezifikation)
Kandidaten für Klassen (Karten) identifizieren
Typische Szenarien identifizieren und durchspielen (dabei: Karten schrittweise ausfüllen)
Iteration: Verbesserungen, mehrfache Wiederholung
Ungewöhnliche Szenarien durchspielen
Technische Universität München
B. Rumpe
Softwaretechnik, 90

Klassen finden: Beispiel Terminverwaltung

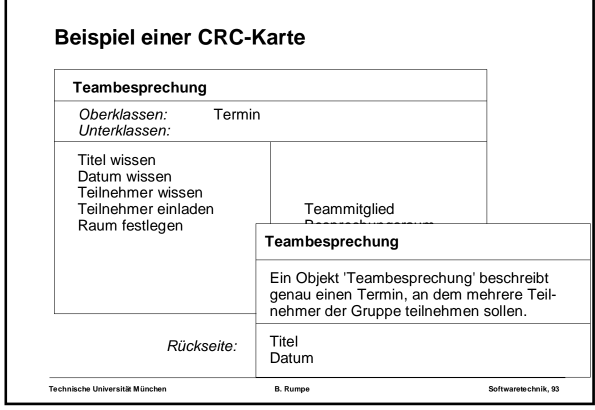
Problembeschreibung:

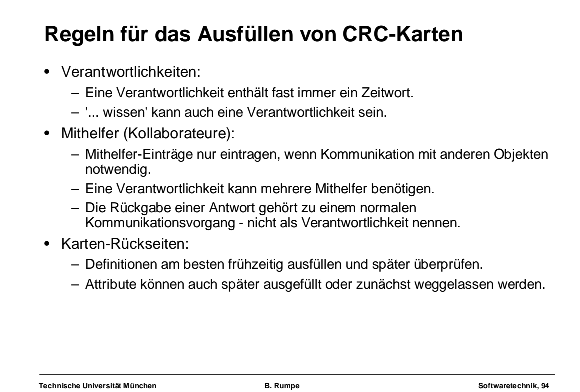
Es ist ein Terminverwaltungssystem für Arbeitsgruppen zu entwickeln. Das System soll alle geplanten Teambesprechungen (z.B. Projektbesprechungen) speichern und die Reservierung von Besprechungsräumen unterstützen. Das System soll automatisch Kollisionen mit bereits bekannten Terminen vermeiden. Deshalb soll auch die Eintragung privater Termine möglich sein.

Kandidaten für Klassen:

Gruppenspiel

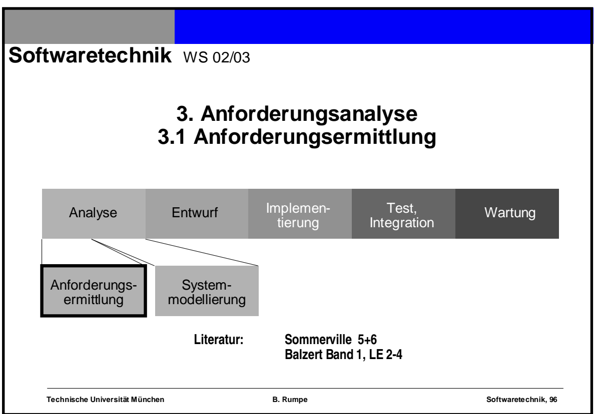
- Ideale Gruppengröße: 5 bis 6 aktive Teilnehmer/-innen
- Teilnehmer(-innen):
  - Fachspezialisten
  - Systemanalytiker
  - Systementwickler
  - Manager (?)
  - Moderator, 'Facilitator'
- Gruppendynamik:
  - CRC-Karten-Sitzungen können Teamgeist stärken
  - Vorhandene Gruppen-Probleme können aufbrechen
  - **Kein** Mittel zur Klärung und Lösung von Problemen im Team!





Zusammenfassung 2.2 Objektorientierung – CRC-Karten

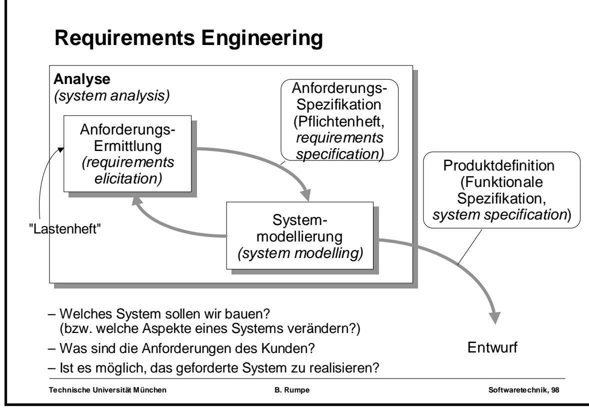
- CRC = Class-Responsibility-Collaboration
- Eine CRC-Karte beschreibt die Struktur, die zu erbringende Funktionalität und die Mithelfer einer Klasse.
- CRC ist ein Mittel zur
  - Erhebung von Anforderungen (wenn Anwender „mitspielen“) und zur
  - Entwicklung von objektorientierten Architekturen aus einer Menge von Anforderungen.
- CRC-Karten werden in Gruppensitzungen („Workshops“) entwickelt



Definitionen:

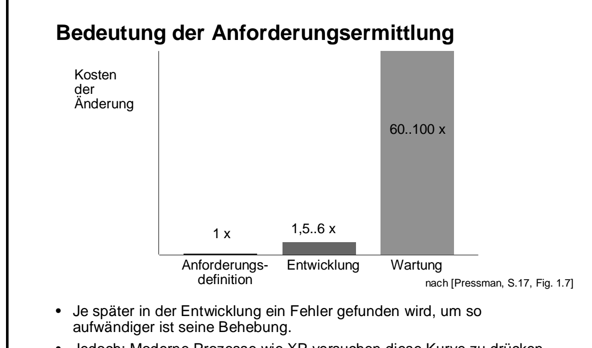
- ***Anforderungsermittlung*** (*requirements elicitation*):
  - Tätigkeit: Feststellung der Anforderungen an das geplante System in Zusammenarbeit mit Kunden und potentiellen Benutzern
  - Ergebnis: Anforderungsspezifikation (Pflichtenheft)
- ***Systemmodellierung*** (*system modelling*)
  - Tätigkeit: Detaillierte und strukturierte Beschreibung der Anforderungen in einer Form, die als Grundlage für den Systementwurf dienen kann.
  - Ergebnis: Funktionale Spezifikation (Produktdefinition)
- Oberbegriff für beide Tätigkeiten:
  Analyse (engl. *requirements analysis, system analysis*)
- ***requirements engineering*** ist ein Name für das Teilgebiet des Software-Engineering, das sich mit den frühen Phasen der Entwicklung befasst. (Es gibt keinen wirklich adäquaten deutschen Begriff für requirements engineering.)

Requirements Engineering



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

Bedeutung der Anforderungsermittlung



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

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

Probleme bei der Anforderungsermittlung

- Viele Beteiligte:
  - Kunden, Benutzer
  - Informatik-Spezialisten
  - Betriebswirtschaft-Spezialisten
  - Management, Marketing, ...
- Kunden/Benutzer wissen meist **nicht**, was sie wirklich wollen.
- Fachsprachen, nicht allgemein verständliche Begriffe
- Verschiedene Beteiligte verwenden inkonsistente Begriffe.
- Verschiedene Beteiligte haben widersprüchliche Ziele.
- Organisatorische Rahmenbedingungen unklar oder veränderlich
- Ständig veränderte Anforderungen, auch während Analyse und Entwurf des Systems.
- Neue Beteiligte während oder nach der Analyse

Funktionale/nicht-funktionale Anforderungen

- Funktionale Anforderung:
  - Beschreibung dessen, **was** das System tun soll
  - Bsp.: "Das System soll einen Mechanismus zur Identifizierung von Benutzern vorsehen."
- Nicht-funktionale Anforderung:
  - Einschränkende Bedingung, **wie** die funktionalen Anforderungen zu realisieren sind
  - Bsp.: "Der Identifizierungsvorgang muss in höchstens 5 Sekunden abgeschlossen sein."
- Mischfälle:
  - Bsp.: "Zur Identifizierung ist das Softwarepaket XY zu verwenden, von dem bekannt ist, dass es den Vorgang in höchstens 5 Sekunden bewältigt."
    -> Zerlegen in eine funktionale und nicht-funktionale Anforderungen

Typen von nicht-funktionalen Anforderungen

- Anforderungen an das Produkt:
  - Effizienzanforderungen: Zeitanforderungen, Speicheranforderungen
  - Zuverlässigkeitsanforderungen, „Quality of Service"
  - Ergonomische Anforderungen
  - Portabilitätsanforderungen
- Anforderungen an den Entwicklungsprozess:
  - Verwendung von Standards (V-Modell, UML, ...)
  - Anforderungen an die Implementierungstechnik
  - Anforderungen an die Auslieferungsform
- Externe Anforderungen:
  - Interoperabilitätsanforderungen
  - Juristische Anforderungen: Datensicherheit, Datenschutz
  - Ethische Anforderungen

Inhalte einer Anforderungsspezifikation

- Zielsetzung
- Allgemeine Beschreibung
  - Umgebung, generelle Funktion, Restriktionen, Benutzer
- Spezifische funktionale Anforderungen
  - möglichst quantitativ (z.B. Tabellenform)
  - eindeutig identifizierbar (Nummern)
- Spezifische nicht-funktionale Anforderungen
  - z.B. Antwortzeit, Speicherbedarf, HW/SW-Plattform
  - Entwicklungs- und Produkt-Standards
- Qualitäts-Zielbestimmung
- Zu erwartende Evolution des Systems
  - Grobe Identifikation von Versionen
- Formalia: **Abkürzungsverzeichnis, Glossar,** Index, Referenzen
  (sehr wirkungsvoll zur Konsistenzsicherung!)

Beispiele funktionaler Anforderungen

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

4.1 Kundenverwaltung

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

Beispiel: IEEE/ANSI Standard-Gliederung (1) „Software Requirements Specification“ IEEE SRS

1. Introduction
   1.1 Purpose
   1.2 Scope
   1.3 Definitions, Acronyms, and Abbreviations
   1.4 References
   1.5 Overview
2. Overall Description
   2.1 Product Perspective
   2.2 Product Functions
   2.3 User Characteristics
   2.4 General Constraints
   2.5 Assumptions and Dependencies
3. Specific Requirements

[ANSI/IEEE Std 830-1998]

Gliederung nach Versionen (Beispiel)

Nr.	Anforderung	V1 (6/02)	V2 (03)	V2+
	Funktionsgruppe 2			
2.1	Mehrbenutzerfähigkeit	nein	erwünscht	ja
2.2	max. Anzahl Benutzer	n/a	10	tbd
...	...			

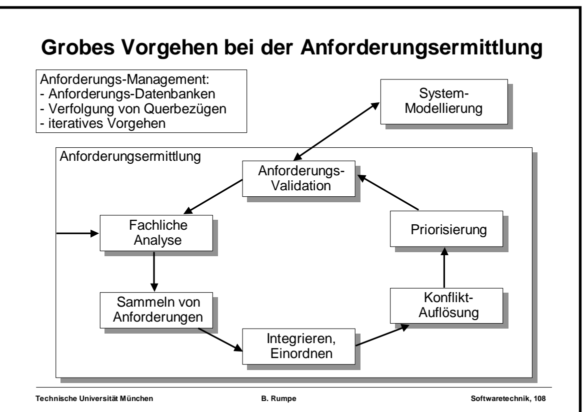
Jargon: n/a = not applicable
tbd = to be discussed
ffs = for further study
...

Beispiel: IEEE/ANSI Standard-Gliederung (2)

3. Specific Requirements
   3.1 External Interface Requirements
      3.1.1 User Interfaces
      3.1.2 Hardware Interfaces
      3.1.3 Software Interfaces
      3.1.4 Communication Interfaces
   3.2 Classes/Objects
      3.2.i Class/Object i
      3.2.i.1 Attributes
         3.2.i.1.j Attribute i.j
      3.2.i.2 Functions, Services, Methods
         3.2.i.2.k Method i.k
   3.3 Performance Requirements
   3.4 Design Constraints
   3.5 Attributes
   3.6 Other Requirements

Neben der hier gezeigten objektorientierten Variante gibt es eine Variante, die Funktionen als Spezifikationseinheiten nutzt

Grobes Vorgehen bei der Anforderungsermittlung



Analysephase als Kommunikationsleistung

- Die Wirklichkeit der Anforderungsermittlung:
  - Kreative, analytische und ***kommunikative*** Leistung
  - Hohe Bedeutung ***sozialer Kompetenz*** (offene, freundschaftliche, vertrauensfördernde Zusammenarbeit)
  - Weitere Kompetenzen: ***Urteilsfähigkeit, Kreativität, Erfahrung***
- Hilfsmittel des Systemanalytikers:
  - Organisationsprinzipien und -werkzeuge für Informationen
  - Sammlung von bewährten Techniken *(best practices, z.B. CRC-Karten)*
  - Systemmodellierung
- Modellierung, Methoden sind für die Analyse manchmal überbetont!
- Im folgenden zwei praxisrelevante Hilfsmittel (Beispiele):
  - Viewpoint-Analyse
  - Unternehmens- und Geschäftsprozess-Modellierung

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

Viewpoint-Analyse

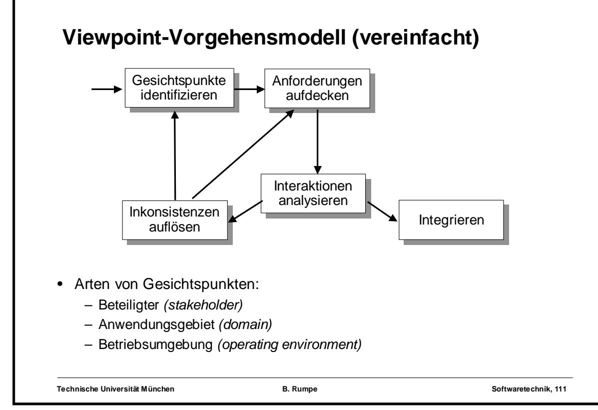
Ziel: Vollständiges Auffinden aller Anforderungen an eine komplexe Entwicklungsaufgabe

Zu untersuchendes Problem

Mittel: Untersuchung aus verschiedenen Gesichtspunkten (*viewpoints*)

Quelle: PREview-Methode nach Sommerville/Sawyer 1997

Viewpoint-Vorgehensmodell (vereinfacht)



- Arten von Gesichtspunkten:
  - Beteiligter *(stakeholder)*
  - Anwendungsgebiet *(domain)*
  - Betriebsumgebung *(operating environment)*

Ein Gesichtspunkt *(viewpoint)*

- **Gesichtspunkt-Name:** Dozent
- **Teilbereiche:** Veranstaltungsverwaltung, Buchungsverwaltung
- **Globale Belange:** Vollständigkeit, Antwortzeit
- **Quellen:** Interview vom ...
- **Anforderungen:**
  - D1: Jeder Dozent muss über alle geplanten und durchgeführten Veranstaltungen seines Themengebiets informiert werden.
  - D2: Terminänderungen für Veranstaltungen sind nicht ohne Zustimmung des Dozenten möglich.
  - D3: Die Endabrechnung für eine Veranstaltung muss nach spätestens 7 Werktagen abgeschlossen sein.
  - ...
- **Änderungsgeschichte:**

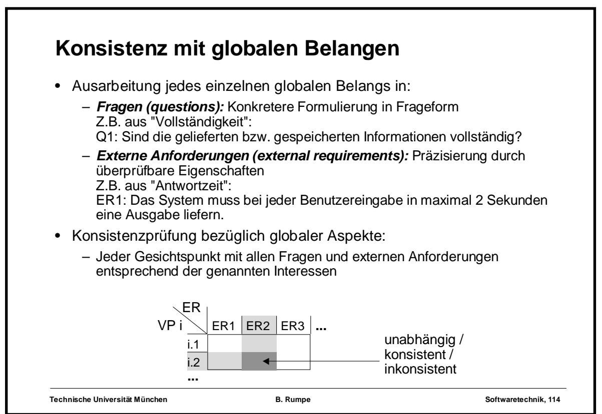
Globale Belange und Teilbereiche

- ***Globale Belange (concerns):***
  - Besonders wichtige Anforderungen
  - Entscheiden über die Brauchbarkeit und den Erfolg des Produkts
  - Beispiele:

    Zuverlässigkeit, Vertraulichkeit, Datenkorrektheit, Datenvollständigkeit, Reaktionszeit, Anschaffungs- und Betriebskosten, Wartbarkeit
- ***Teilbereich (focus):***
  - Beschreibung des Teilsystems oder Teilaspekts, für den ein Gesichtspunkt Aussagen macht
  - Entweder gemäß vorgegebener Grobstruktur (Stichwortkatalog) oder individueller Text

Konsistenz mit globalen Belangen

- Ausarbeitung jedes einzelnen globalen Belangs in:
  - ***Fragen (questions):*** Konkretere Formulierung in Frageform
    Z.B. aus "Vollständigkeit":
    Q1: Sind die gelieferten bzw. gespeicherten Informationen vollständig?
  - ***Externe Anforderungen (external requirements):*** Präzisierung durch überprüfbare Eigenschaften
    Z.B. aus "Antwortzeit":
    ER1: Das System muss bei jeder Benutzereingabe in maximal 2 Sekunden eine Ausgabe liefern.
- Konsistenzprüfung bezüglich globaler Aspekte:
  - Jeder Gesichtspunkt mit allen Fragen und externen Anforderungen entsprechend der genannten Interessen



Konsistenz zwischen Gesichtspunkten

- Grundsätzlich muss jeder Gesichtspunkt mit jedem anderen verglichen werden!
- 1. Schritt: Überlappung der "Teilbereiche" prüfen

VP i
Teilbereiche: xxx

→ Über-lappung? ←

VP j
Teilbereiche: yyy

- 2. Schritt: Falls Überlappung, dann Detailüberprüfung

VP i \ VP j	j.1	j.2	j.3	...
i.1				
i.2				
i.3				
...				

unabhängig / konsistent / inkonsistent

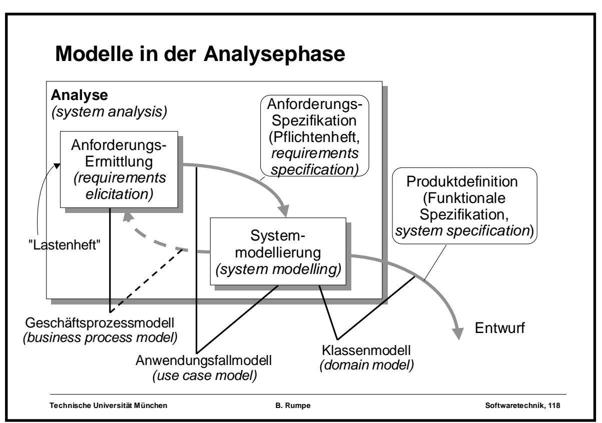
Viewpoint-Analyse: Zusammenfassung

- Prinzipien:
  - Separierte Berücksichtigung von Einzelaspekten
    - » "divide et impera" *(divide and conquer)*
  - Trennung von Informationssammlung ("Brainstorming") und systematischer Analyse
  - Inkaufnahme von temporären Inkonsistenzen zum Auffinden problematischer Themen
- Vorteile:
  - Hilfreich zum systematischen Finden aller Anforderungen
  - Ideen übertragbar auch auf andere Darstellungsweisen
- Nachteile:
  - Kombinatorische Explosion bei der Konsistenzprüfung
  - Werkzeugunterstützung mangelhaft

Weitere Aspekte der Anforderungsermittlung

- Psychologische und soziale Randbedingungen:
  - z.B.: Mitarbeiter fürchten durch den Rationalisierungseffekt des neuen Systems um ihre Arbeitsplätze (oder deren Attraktivität) und kooperieren deshalb schlecht.
- Paradox:
  - Unbefriedigend realisierte Funktion wird, weil kaum genutzt, als unnötige Funktion mißverstanden.
  - Schlechte IST-Realisierung führt zu Unrecht zum Wegfall in der SOLL-Beschreibung.
- Abstraktionsdilemma:
  - Höhere Abstraktionsebene führt zu besser anpassbaren Systemen, also zukunftssicheren Systemen.
  - Niedrigere Abstraktionsebene führt zu besserer Anwenderbeteiligung, also besser benutzbaren Systemen.

Modelle in der Analysephase



Geschäftsprozess *(business process)*

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

- Geschäftsprozessmodellierung ist eine typische Domäne der Wirtschaftsinformatik:
  - Beschreibung von Geschäftsprozessen
  - Analyse und Optimierung von Geschäftsprozessen
  - Ableitung einer flexiblen Infrastruktur für die effiziente Durchführung von Geschäftsprozessen
- Geschäftsprozessmodellierung in der Systemanalyse:
  - Zeitlich vor der Erstellung der Anforderungsspezifikation
  - Analyse des "IST-Zustands" hilft bei Beurteilung der Ziele und des "SOLL"
  - *Grobe* Geschäftsprozessmodelle: Hilfreich beim Finden von Beteiligten und globalen Belangen
  - *Detaillierte* Geschäftsprozessmodelle: Hilfreich beim Finden von funktionalen Anforderungen und Anwendungsfällen

Grobes Geschäftsprozessmodell

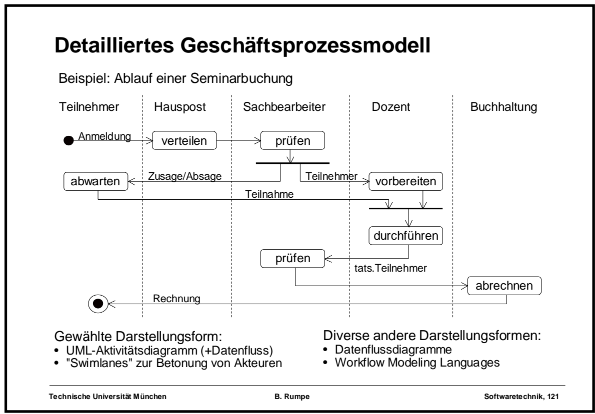
- Strategisch orientiert
- Ziel: Gesamtverständnis des Unternehmens und der Aufgabe
- Beispiel (Notation an ARIS angelehnt):

Markt-forschung → Seminar-entwicklung → Programm-planung → Durch-führung

Seminar-entwicklung → Interne Schulung → Vertrieb

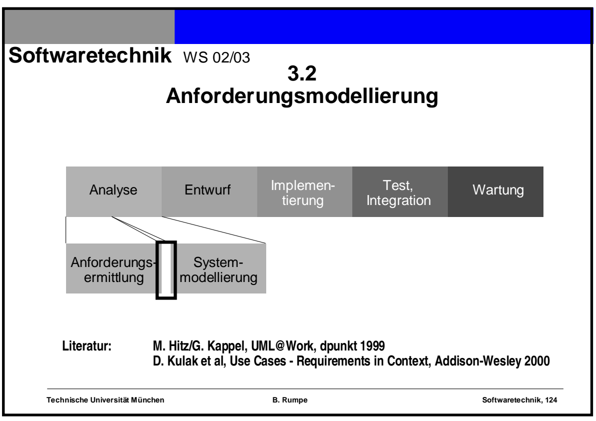
Detailliertes Geschäftsprozessmodell

Beispiel: Ablauf einer Seminarbuchung



Gewählte Darstellungsform:
- UML-Aktivitätsdiagramm (+Datenfluss)
- "Swimlanes" zur Betonung von Akteuren

Diverse andere Darstellungsformen:
- Datenflussdiagramme
- Workflow Modeling Languages



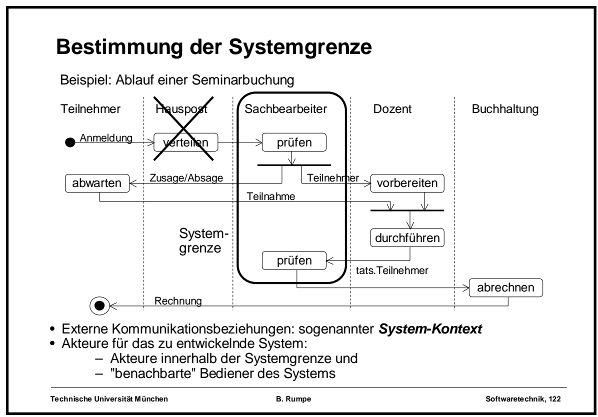
Softwaretechnik WS 02/03

3.2 Anforderungsmodellierung

Literatur: **M. Hitz/G. Kappel, UML@Work, dpunkt 1999**
D. Kulak et al, Use Cases - Requirements in Context, Addison-Wesley 2000

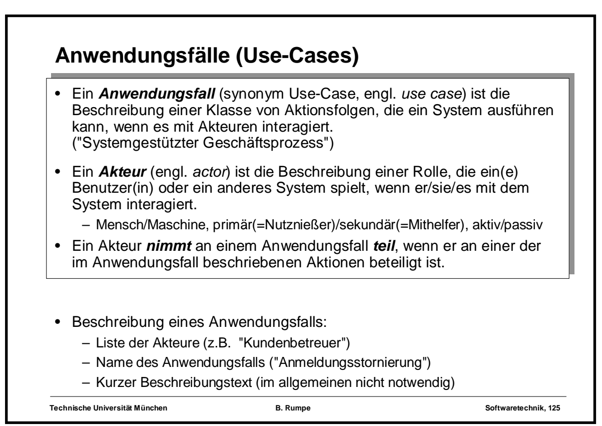
Bestimmung der Systemgrenze

Beispiel: Ablauf einer Seminarbuchung



- Externe Kommunikationsbeziehungen: sogenannter ***System-Kontext***
- Akteure für das zu entwickelnde System:
  - Akteure innerhalb der Systemgrenze und
  - "benachbarte" Bediener des Systems

Anwendungsfälle (Use-Cases)



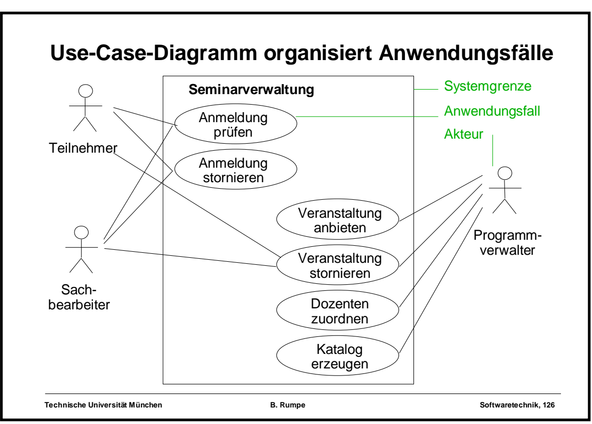
- Ein ***Anwendungsfall*** (synonym Use-Case, engl. *use case*) ist die Beschreibung einer Klasse von Aktionsfolgen, die ein System ausführen kann, wenn es mit Akteuren interagiert. ("Systemgestützter Geschäftsprozess")
- Ein ***Akteur*** (engl. *actor*) ist die Beschreibung einer Rolle, die ein(e) Benutzer(in)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

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

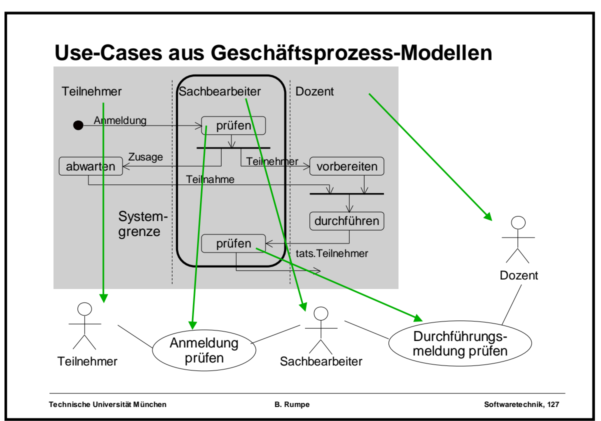
Zusammenfassung 3.1 Anforderungsermittlung

- Fehler in der Anforderungsermittlung sind später teuer zu beheben.
- Funktionale und nicht-funktionale Anforderungen trennen.
- Glossar ist ein einfaches und wesentliches Hilfsmittel zur Konsistenzsicherung.
- Viewpoint-Analyse zerlegt die Aufgabe der Anforderungsermittlung nach Gesichtspunkten, um so eine bessere Vollständigkeit und Konsistenz zu erreichen.
- Modelle in der Anforderungsermittlung:
  - Aktivitätsdiagramme für die Geschäftsprozesse
  - sowie die nachfolgend diskutierten Use Cases und Klassendiagramme

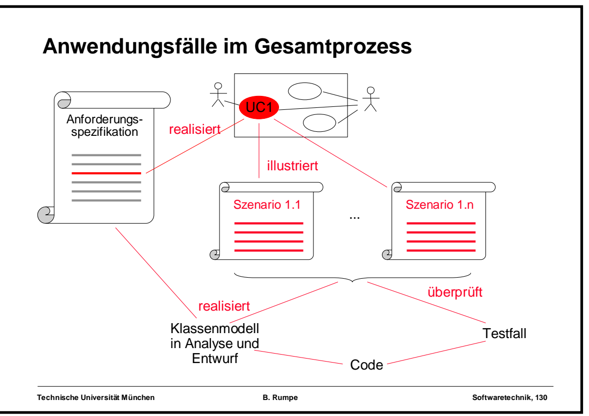
Use-Case-Diagramm organisiert Anwendungsfälle



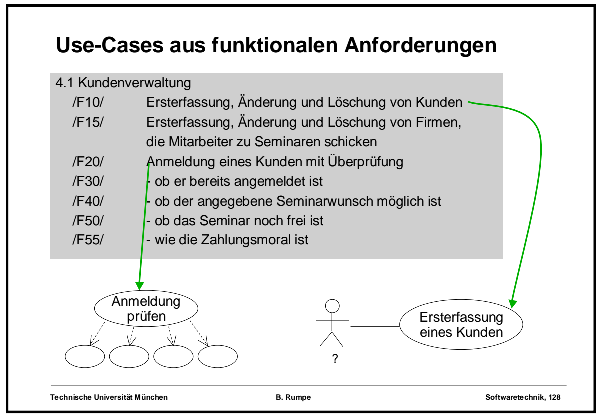
Use-Cases aus Geschäftsprozess-Modellen



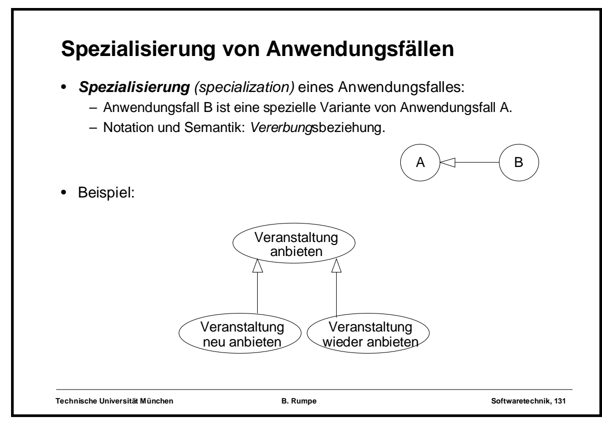
Anwendungsfälle im Gesamtprozess



Use-Cases aus funktionalen Anforderungen



Spezialisierung von Anwendungsfällen



Szenarien

- Ein ***Szenario*** ist eine konkrete Folge von Aktionen, die das Systemverhalten illustriert. Ein Szenario kann einen Anwendungsfall oder auch Ausschnitte systeminternen Verhaltens illustrieren.

- Für jeden Anwendungsfall sollte mindestens ein Szenario definiert werden.
- Oft sind mehrere Szenarien sinnvoll (z.B. Normalfall und Ausnahmefälle).

- **Beispiel** eines Szenarios für "Anmeldungsstornierung":
  – Kunde Maier, Kundennummer 32, sagt telefonisch für das Seminar "UML 1", Veranstaltungsnummer 124, ab.
  – Die Teilnahme von Maier an Veranstaltung Nr. 124 wird gelöscht.
  – Es wird überprüft, ob eine Stornierung der gesamten Veranstaltungen nötig ist.
  – Ggf. werden geleistete Anzahlungen rückerstattet.

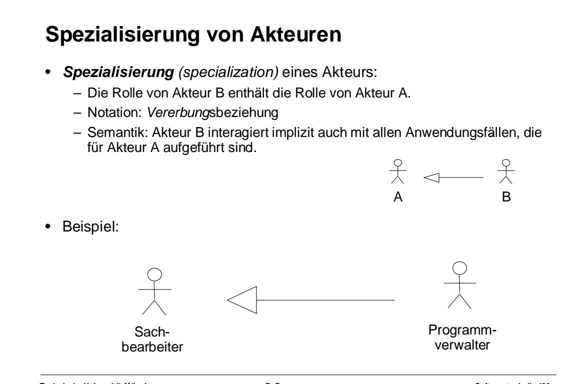
Abstrakte Anwendungsfälle

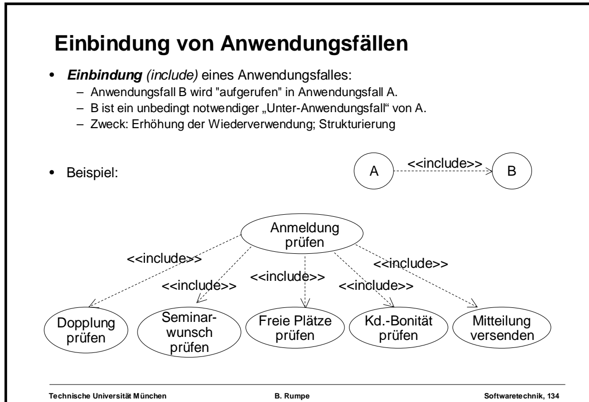
- ***Abstrakter*** Anwendungsfall *(abstract use case)*:
  – Ein Anwendungsfall, der nicht eigenständig durch Aktionsfolgen realisiert wird, sondern nur Gemeinsamkeiten anderer Anwendungsfälle festhält.
  – Oft beziehen sich abstrakte Anwendungsfälle auf abstrakte Klassen.
- Spezialisierung von Anwendungsfällen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

{abstract}
Veranstaltung anbieten

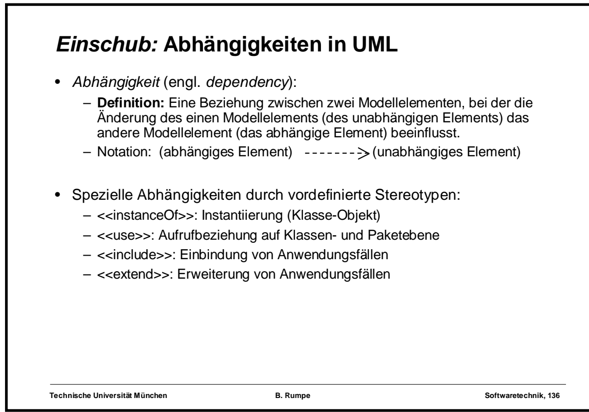
Veranstaltung neu anbieten

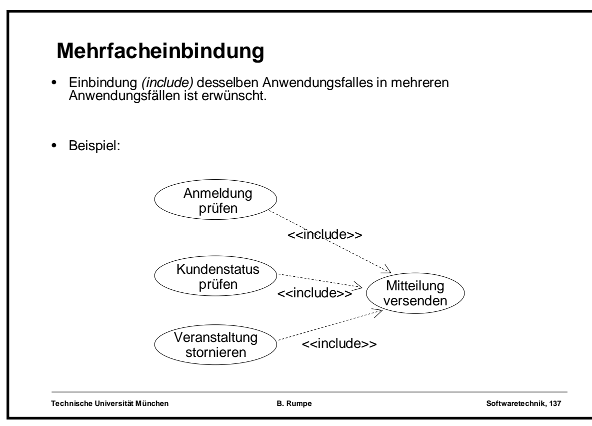
Veranstaltung wieder anbieten

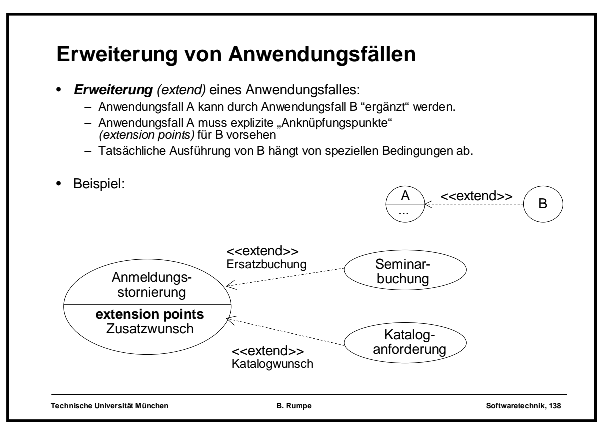

Spezialisierung von Akteuren
• Spezialisierung (specialization) eines Akteurs:
– Die Rolle von Akteur B enthält die Rolle von Akteur A.
– Notation: Vererbungsbeziehung
– Semantik: Akteur B interagiert implizit auch mit allen Anwendungsfällen, die für Akteur A aufgeführt sind.
A
B
• Beispiel:
Sach-
bearbeiter
Programm-
verwalter
Technische Universität München
B. Rumpe
Softwaretechnik, 133

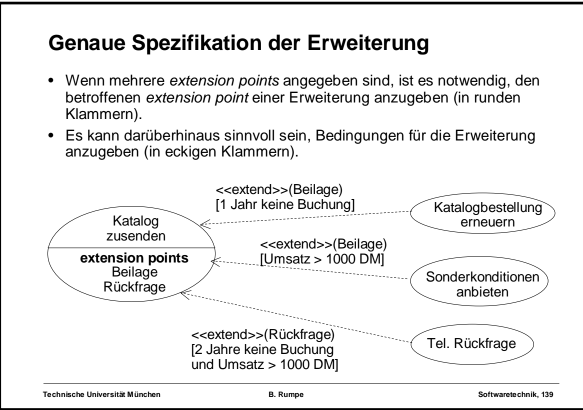

Einbindung von Anwendungsfällen
• Einbindung (include) eines Anwendungsfalles:
– Anwendungsfall B wird "aufgerufen" in Anwendungsfall A.
– B ist ein unbedingt notwendiger „Unter-Anwendungsfall" von A.
– Zweck: Erhöhung der Wiederverwendung; Strukturierung
• Beispiel:
A
<<include>>
B
Anmeldung
prüfen
<<include>>
<<include>>
<<include>>
<<include>>
<<include>>
Dopplung
prüfen
Seminar-
wunsch
prüfen
Freie Plätze
prüfen
Kd.-Bonität
prüfen
Mitteilung
versenden
Technische Universität München
B. Rumpe
Softwaretechnik, 134

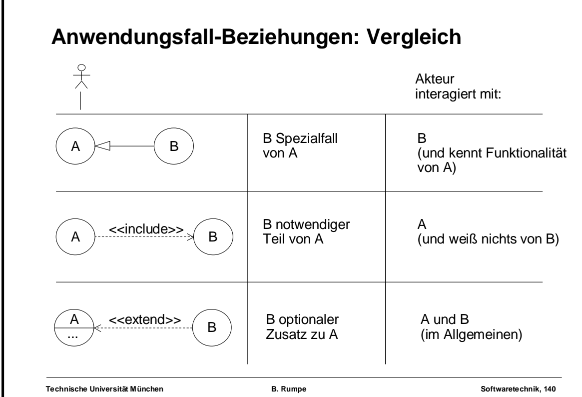
Einschub: Stereotypen in UML
• UML ist als erweiterbare und anpassbare Sprache angelegt.
– Spezialisierung von UML durch „Profile" für Real-Time, Frameworks, etc.
• Einer der Spezialisierungsmechanismen von UML:
"Stereotypen" (stereotypes)
» Markierungsmechanismus für neue Modellelemente, die Spezialfall eines bestehenden Modellelements sind
» Stereotypen wirken auf Darstellung, Bedeutung oder bei der Anwendung (Codegenerierung, etc.)
– Beispiel: „Actor" ist eine Spezialform einer „Klasse".
– Notation: Stereotyp-Notation oder Icon-Notation (hier für „Actor"):
<<actor>>
Teilnehmer
Teilnehmer
Technische Universität München
B. Rumpe
Softwaretechnik, 135

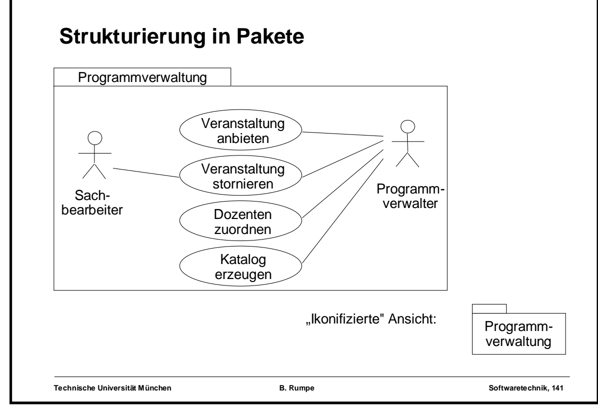

Einschub: Abhängigkeiten in UML
• Abhängigkeit (engl. dependency):
– Definition: Eine Beziehung zwischen zwei Modellelementen, bei der die Änderung des einen Modellelements (des unabhängigen Elements) das andere Modellelement (das abhängige Element) beeinflusst.
– Notation: (abhängiges Element) - - - - - - -> (unabhängiges Element)
• Spezielle Abhängigkeiten durch vordefinierte Stereotypen:
– <<instanceOf>>: Instantiierung (Klasse-Objekt)
– <<use>>: Aufrufbeziehung auf Klassen- und Paketebene
– <<include>>: Einbindung von Anwendungsfällen
– <<extend>>: Erweiterung von Anwendungsfällen
Technische Universität München
B. Rumpe
Softwaretechnik, 136


Mehrfacheinbindung
• Einbindung (include) desselben Anwendungsfalles in mehreren Anwendungsfällen ist erwünscht.
• Beispiel:
Anmeldung
prüfen
<<include>>
Kundenstatus
prüfen
<<include>>
Mitteilung
versenden
Veranstaltung
stornieren
<<include>>
Technische Universität München
B. Rumpe
Softwaretechnik, 137


Erweiterung von Anwendungsfällen
• Erweiterung (extend) eines Anwendungsfalles:
– Anwendungsfall A kann durch Anwendungsfall B "ergänzt" werden.
– Anwendungsfall A muss explizite „Anknüpfungspunkte" (extension points) für B vorsehen
– Tatsächliche Ausführung von B hängt von speziellen Bedingungen ab.
• Beispiel:
A
...
<<extend>>
B
<<extend>>
Ersatzbuchung
Anmeldungs-
stornierung
extension points
Zusatzwunsch
Seminar-
buchung
Katalog-
anforderung
<<extend>>
Katalogwunsch
Technische Universität München
B. Rumpe
Softwaretechnik, 138



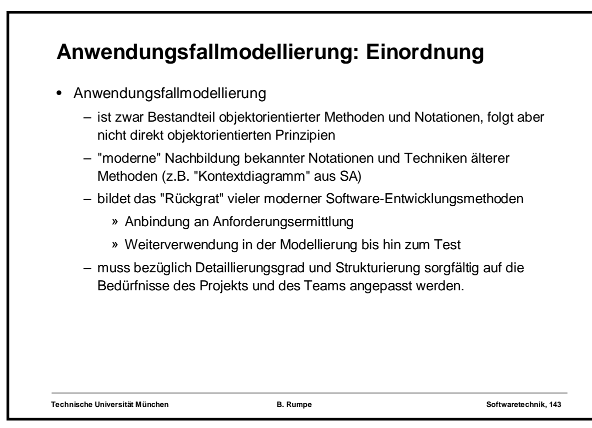


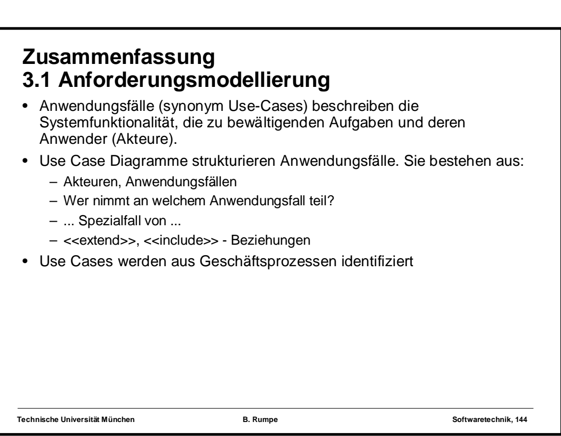


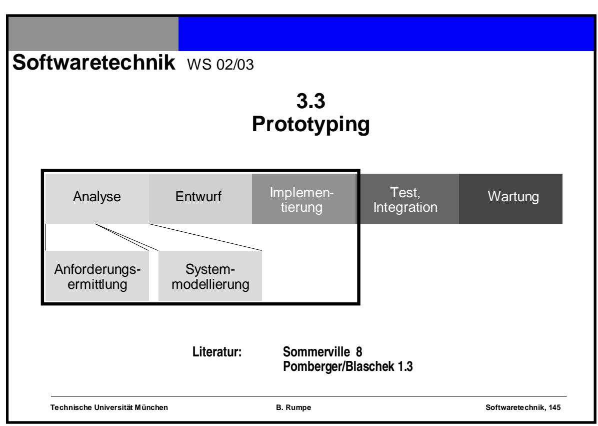
Anwendungsfälle: Detaillierungsgrad

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

Technische Universität München B. Rumpe Softwaretechnik, 142





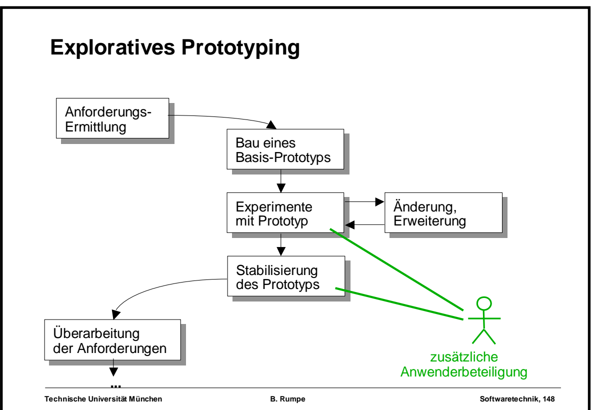


Prototyping und Prototypen

- Prototyp in klassischen Ingenieurdisziplinen:
  - Erstes Exemplar, Modell für ein neues Produkt
  - Voll funktionsfähig
  - Noch nicht in Serie fertigbar
  - Lange Entwicklungszeit, teuer
- Prototyp in der Software-Entwicklung:
  - Problemlos in Serie fertigbar !
  - Erstes Exemplar, Modell für ein neues Produkt
  - Nicht voll funktionsfähig
  - kurze Entwicklungszeit, billig
  - "Rapid Prototyping"
  - Zweck:
    - » Frühe Einbindung von Anwendern in die Entwicklung
    - » Rechtzeitige Abklärung der Realisierbarkeit
    - » Deshalb: Einsatz bereits in der Analysephase sinnvoll

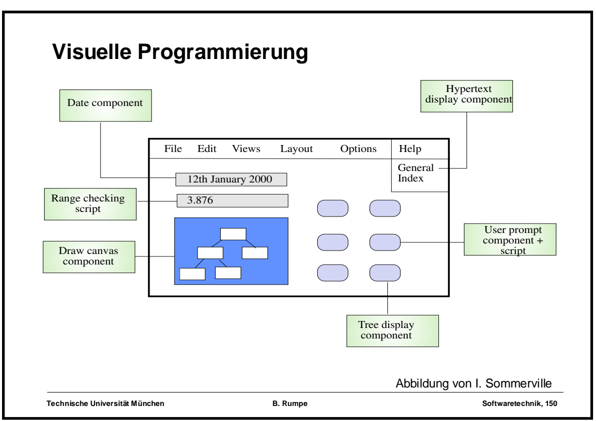
Klassifikation von Prototyping

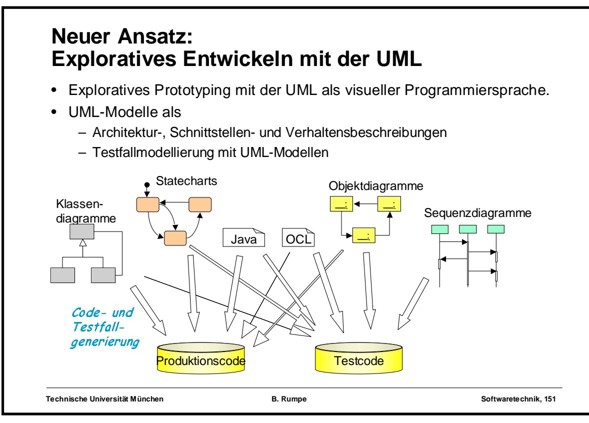
	Weiterverwendung des Prototyps	Phase im Phasenmodell (vorwiegend)	Zielgruppe
explorativ	wegwerfen	Analyse	Anwender, Systemanalytiker
experimentell	wegwerfen	(Analyse,) Entwurf, Implementierung	Entwickler
evolutionär	ausbauen *(inkrementelle Entwicklung)*	(sinnvoll nur bei evolutionärer Entwicklung)	Entwickler,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



Prototyping von Benutzungsschnittstellen

- Benutzungsschnittstelle (User Interface)
  - meist (aber nicht immer) graphisch (Fenster, Menüs)
  - entscheidend für den Anwender
  - Exploratives Prototyping erhöht Akzeptanz
- Werkzeuge:
  - Editoren
  - Generatoren
  - Bausteinbibliotheken
- Explorativer Prototyp:
  - Realistische Oberfläche
  - Funktionalität fehlt oder extrem eingeschränkt
  - Eventuell Simulation von Funktionalität, d.h. Illustration von Anwendungsfällen



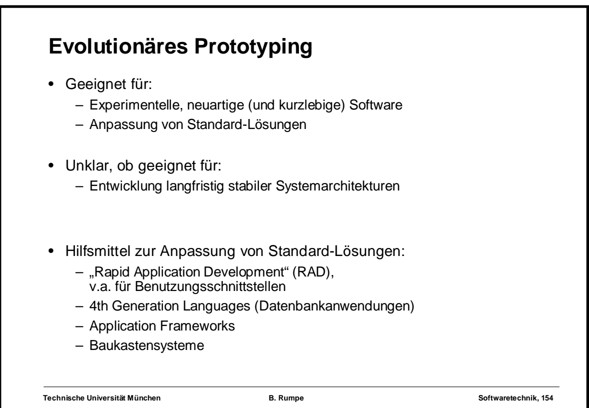


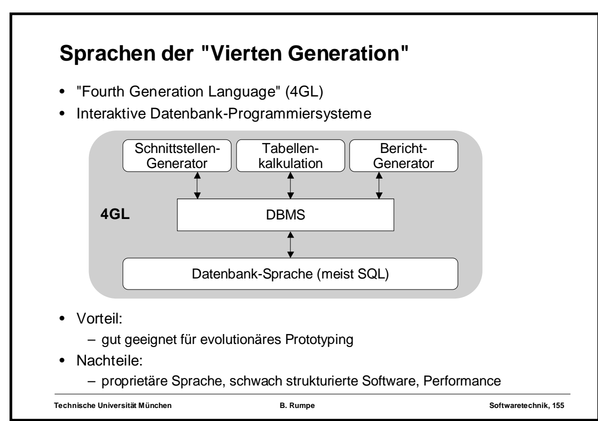
Prototyping durch Integration

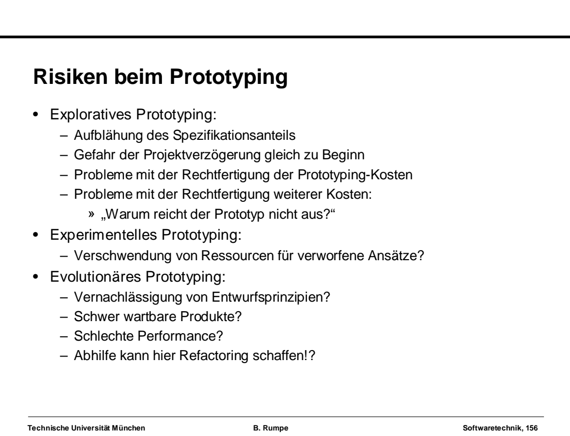
- Erstellung eines Prototyps aus vorhandenen eigenständigen Programmen:
  - Anwendungsprogramme
    z.B. Büro-Anwendungen, Web-Browser und -Server
  - Werkzeuge des Betriebssystems
    z.B. UNIX-Kommandos (awk, sed, ...)
- Integrationstechniken:
  - Anwendungsverbindung (z.B. OLE/DCOM)
  - Skriptsprachen (z.B. TCL/TK, Perl, Python)
  - Informationsablage in Textdateien
- Besonders geeignet zur Beurteilung von Arbeitsabläufen, auch des verteilten Arbeitens

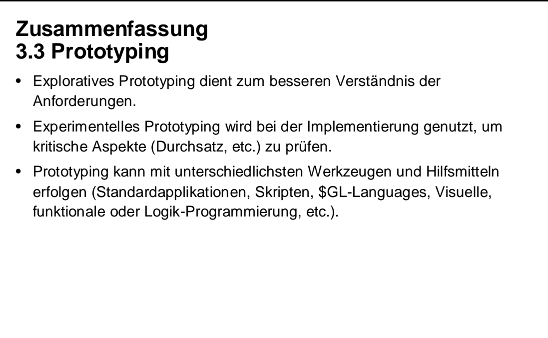
Prototyping durch Simulation

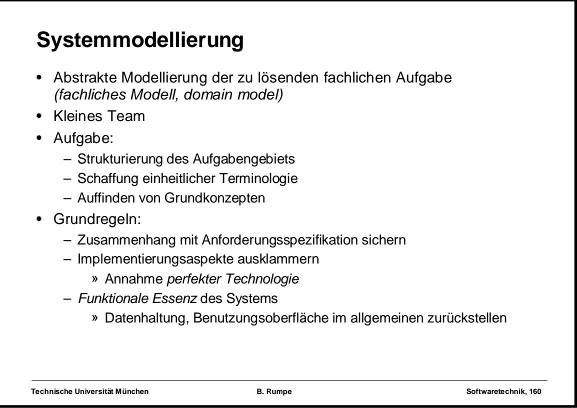
- Pure Funktionalität, Benutzerinteraktion nur simuliert
- Varianten:
  - detaillierte Funktionssimulation (einzelne Szenarien)
  - Simulation großer Benutzerzahlen oder Datenmengen (Abstraktion der Funktionalität)
- Programmierwerkzeuge:
  - "Animation" von abstrakten Modellen, z.B. der UML
  - Spezielle Simulations-Frameworks und -Applikationen
    » diskrete Simulation, statistische Modelle
  - Sprachen hohen Abstraktionsgrades (z.B. Prolog, Haskell)
- Besonders geeignet für:
  - Illustration und Qualitätsüberprüfung von Modellen
  - Überprüfung der "Skalierbarkeit" von Konzepten
  - Performance-Vorhersagen

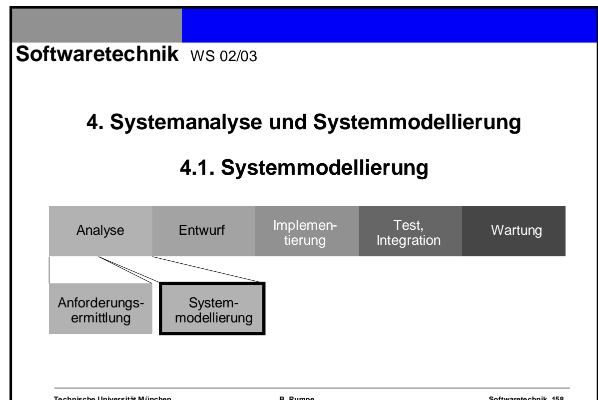


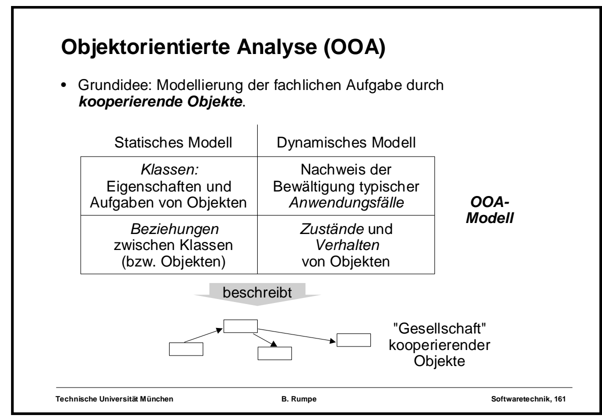


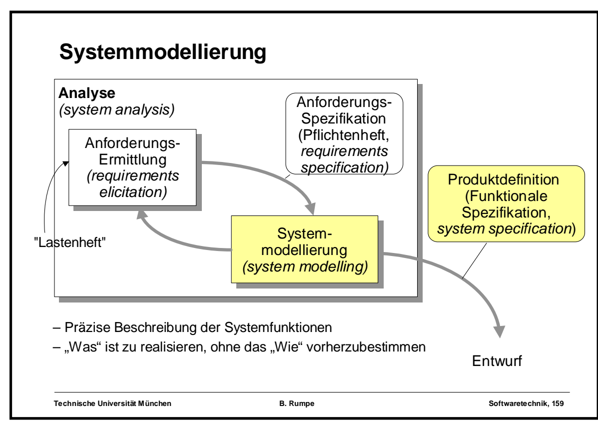


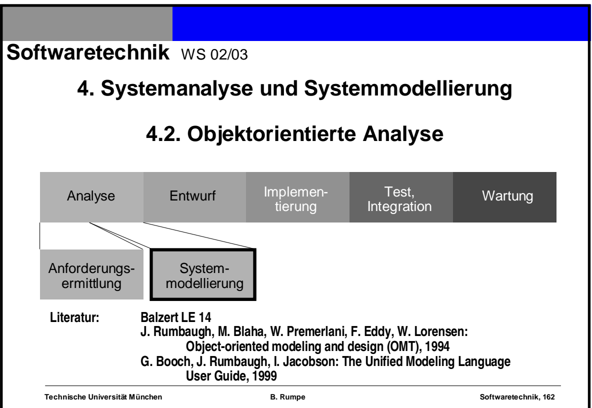

Zusammenfassung
3.3 Prototyping
Exploratives Prototyping dient zum besseren Verständnis der Anforderungen.
Experimentelles Prototyping wird bei der Implementierung genutzt, um kritische Aspekte (Durchsatz, etc.) zu prüfen.
Prototyping kann mit unterschiedlichsten Werkzeugen und Hilfsmitteln erfolgen (Standardapplikationen, Skripten, $GL-Languages, Visuelle, funktionale oder Logik-Programmierung, etc.).
Technische Universität München
B. Rumpe
Softwaretechnik, 157

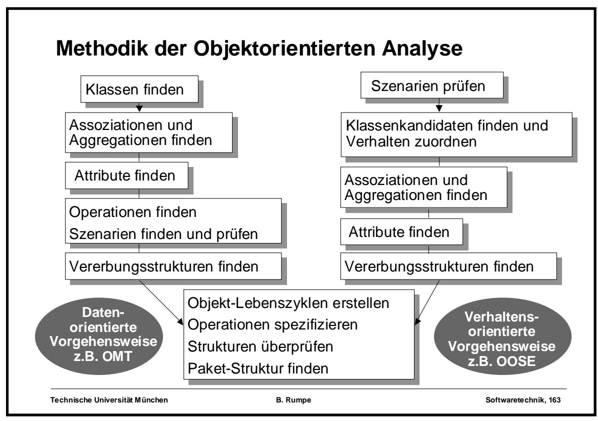

Systemmodellierung
Abstrakte Modellierung der zu lösenden fachlichen Aufgabe (fachliches Modell, domain model)
Kleines Team
Aufgabe:
– Strukturierung des Aufgabengebiets
– Schaffung einheitlicher Terminologie
– Auffinden von Grundkonzepten
Grundregeln:
– Zusammenhang mit Anforderungsspezifikation sichern
– Implementierungsaspekte ausklammern
» Annahme perfekter Technologie
– Funktionale Essenz des Systems
» Datenhaltung, Benutzungsoberfläche im allgemeinen zurückstellen
Technische Universität München
B. Rumpe
Softwaretechnik, 160


Softwaretechnik WS 02/03
4. Systemanalyse und Systemmodellierung
4.1. Systemmodellierung
Analyse
Entwurf
Implementierung
Test, Integration
Wartung
Anforderungs-ermittlung
System-modellierung
Technische Universität München
B. Rumpe
Softwaretechnik, 158


Objektorientierte Analyse (OOA)
Grundidee: Modellierung der fachlichen Aufgabe durch kooperierende Objekte.
Statisches Modell
Dynamisches Modell
Klassen: Eigenschaften und Aufgaben von Objekten
Nachweis der Bewältigung typischer Anwendungsfälle
Beziehungen zwischen Klassen (bzw. Objekten)
Zustände und Verhalten von Objekten
OOA-Modell
beschreibt
"Gesellschaft" kooperierender Objekte
Technische Universität München
B. Rumpe
Softwaretechnik, 161


Systemmodellierung
Analyse (system analysis)
Anforderungs-Ermittlung (requirements elicitation)
Anforderungs-Spezifikation (Pflichtenheft, requirements specification)
System-modellierung (system modelling)
Produktdefinition (Funktionale Spezifikation, system specification)
"Lastenheft"
Entwurf
– Präzise Beschreibung der Systemfunktionen
– „Was" ist zu realisieren, ohne das „Wie" vorherzubestimmen
Technische Universität München
B. Rumpe
Softwaretechnik, 159


Softwaretechnik WS 02/03
4. Systemanalyse und Systemmodellierung
4.2. Objektorientierte Analyse
Analyse
Entwurf
Implementierung
Test, Integration
Wartung
Anforderungs-ermittlung
System-modellierung
Literatur: Balzert LE 14
J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen: Object-oriented modeling and design (OMT), 1994
G. Booch, J. Rumbaugh, I. Jacobson: The Unified Modeling Language User Guide, 1999
Technische Universität München
B. Rumpe
Softwaretechnik, 162

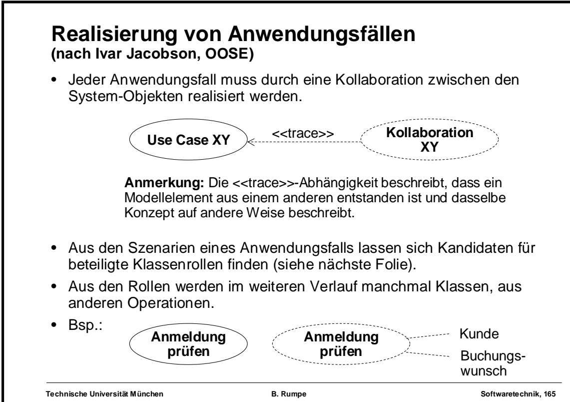


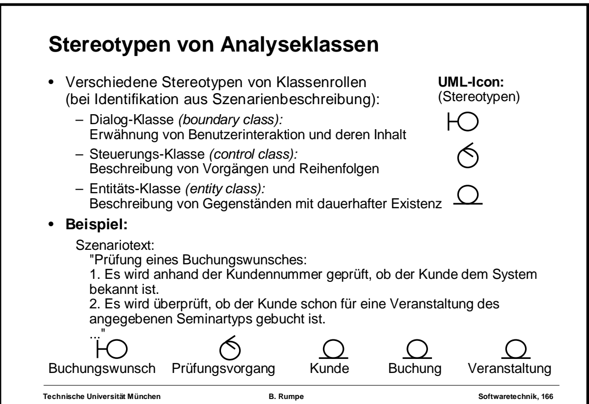
Von Anwendungsfällen zum Analysemodell

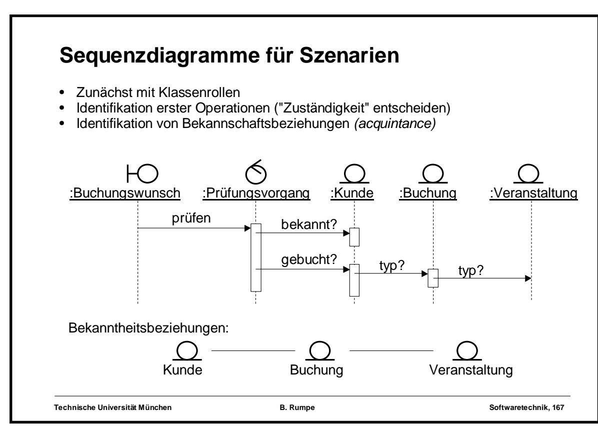
- **Anwendungsfall:** Beschreibung einer Klasse von Aktionsfolgen, die ein System ausführen kann, wenn es mit externen Akteuren interagiert.
  - Szenarien liefern *beispielhafte* Beschreibung des Verhaltens
  - Gesamtsystem als Einheit gesehen ("von außen")
- **Analysemodell** (Klassendiagramm): Modellierung des Systemverhaltens auf abstraktem Niveau.
  - Allgemeine *schematische* Beschreibung des Verhaltens
  - Interne Struktur des Systems ("von innen")
- **Kollaboration** *(collaboration):* Beschreibung des Austauschs von Nachrichten zwischen Objektinstanzen zur Erreichung eines bestimmten Ziels.
  - "von innen" beschrieben: Sequenzdiagramm, Kollaborationsdiagramm
  - "von außen" beschrieben: Kollaborationsmuster, Rollen

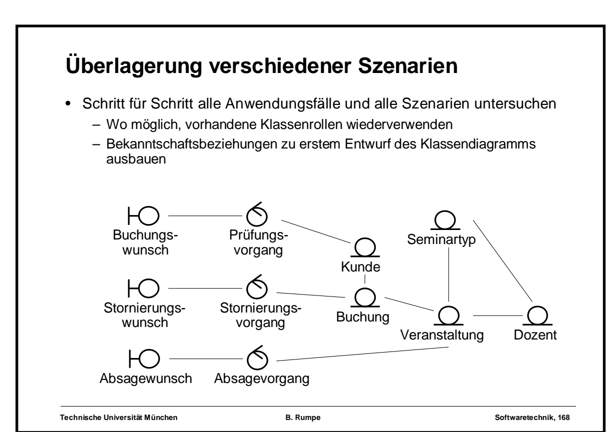
Muster — Rolle 1, Rolle 2

Technische Universität München B. Rumpe Softwaretechnik, 164



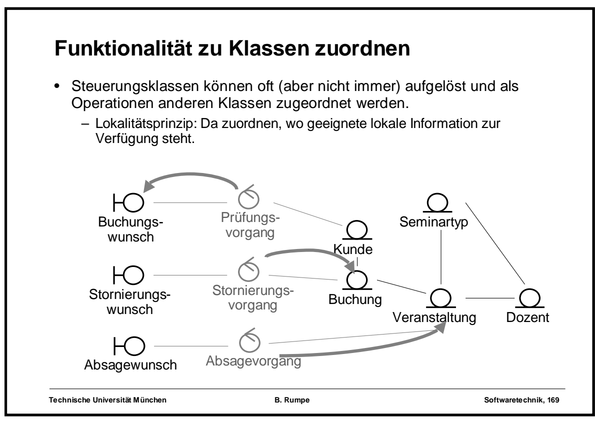






Funktionalität zu Klassen zuordnen

- Steuerungsklassen können oft (aber nicht immer) aufgelöst und als Operationen anderen Klassen zugeordnet werden.
  - Lokalitätsprinzip: Da zuordnen, wo geeignete lokale Information zur Verfügung steht.



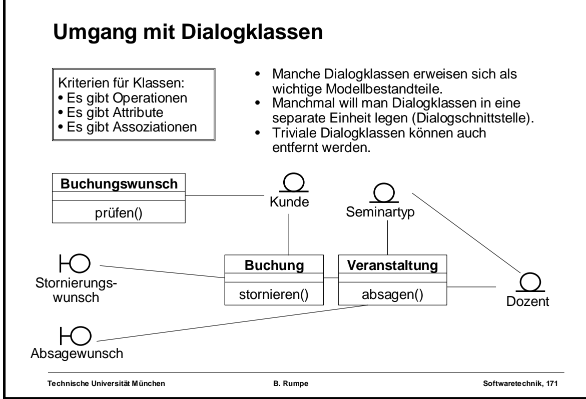
„Objektifizierung“ von Funktionalität

- Manchmal ist es sinnvoll, Steuerungsklassen in der Implementierung beizubehalten
  - Vermeidung zu großer Implementierungsklassen
  - Austauschbarkeit von alternativen Vorgängen durch eine Vererbungshierarchie von Steuervorgängen
  - Speicherbarkeit von Zwischenzuständen lang andauernder Vorgänge
  - Wiederverwendbarkeit derselben Vorgangsklasse für unterschiedliche Implementierungen
- Objektifizierung von Funktionalität bringt
  - Flexibilität in der Implementierung
  - Entkopplung
- Aber: Organisatorischer Zusatzaufwand und sollte daher nur eingesetzt werden, wenn sinnvoll begründbar!
- Entwurfsmuster: Strategy

Umgang mit Dialogklassen

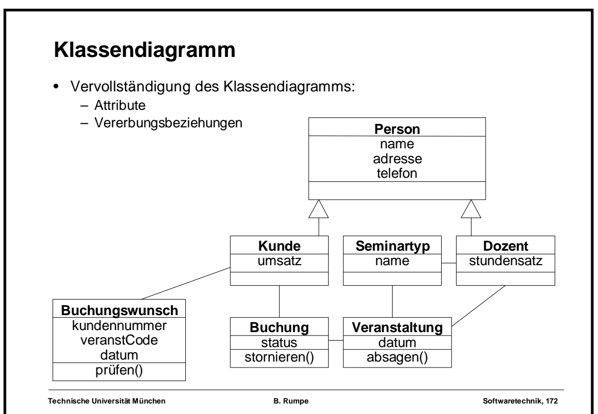
Kriterien für Klassen:
• Es gibt Operationen
• Es gibt Attribute
• Es gibt Assoziationen

- Manche Dialogklassen erweisen sich als wichtige Modellbestandteile.
- Manchmal will man Dialogklassen in eine separate Einheit legen (Dialogschnittstelle).
- Triviale Dialogklassen können auch entfernt werden.



Klassendiagramm

- Vervollständigung des Klassendiagramms:
  - Attribute
  - Vererbungsbeziehungen

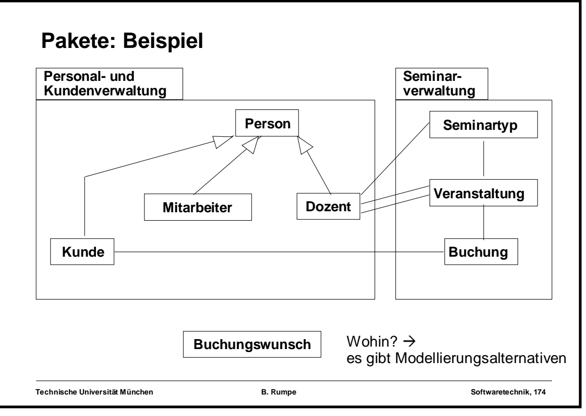


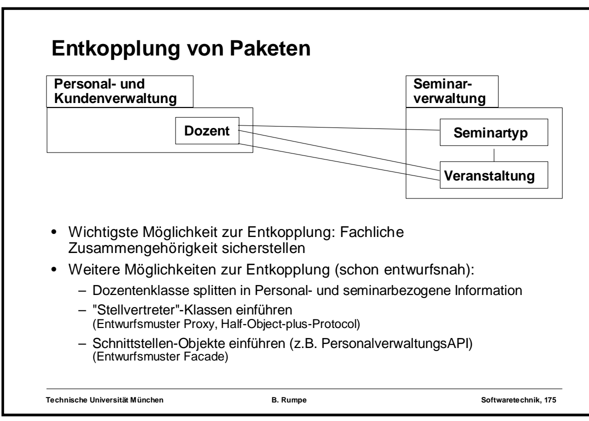
Pakete finden

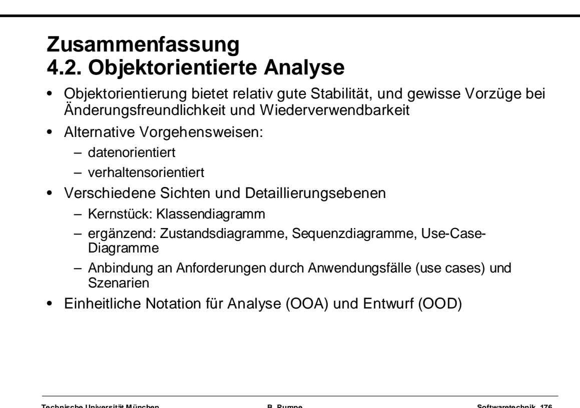
- Ein Paket ist eine Gruppe von Klassen
  - UML: "Subsystem" als spezielles Paket (Architektureinheit)
- Ein Paket:
  - ist für sich allein verständlich
  - hat eine wohldefinierte Schnittstelle zur Umgebung
  - ermöglicht Betrachtung des Systems aus einer abstrakteren Sicht
- Ziel: Starke Bindung innerhalb des Pakets
  - Einheitlicher Themenbereich
  - Aggregation und Vererbung soweit wie möglich nur innerhalb des Pakets
- Ziel: Schwache Kopplung zwischen Paketen
  - Möglichst wenig Assoziationen über Paketgrenzen hinweg
- Faustregeln für ein sinnvolles Paket:
  - 10-15 Klassen
  - 1 DIN A4 Seite für ein Diagramm

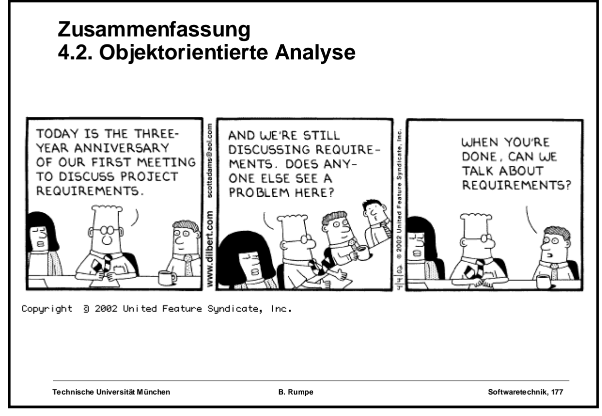
UML:

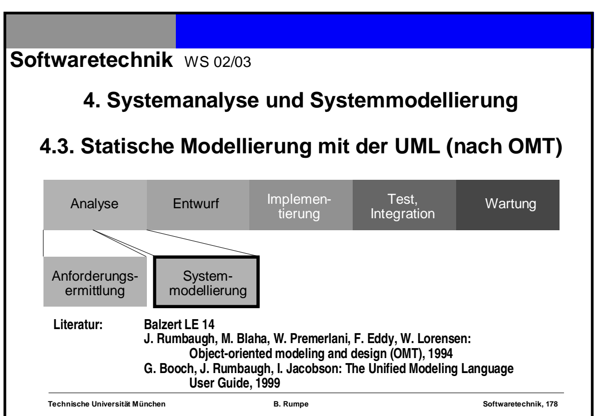
Pakete: Beispiel

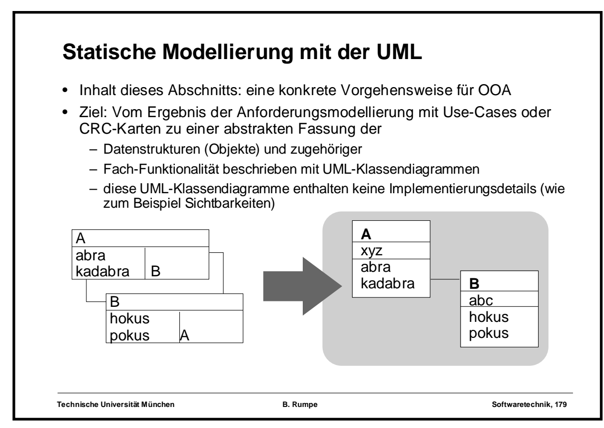


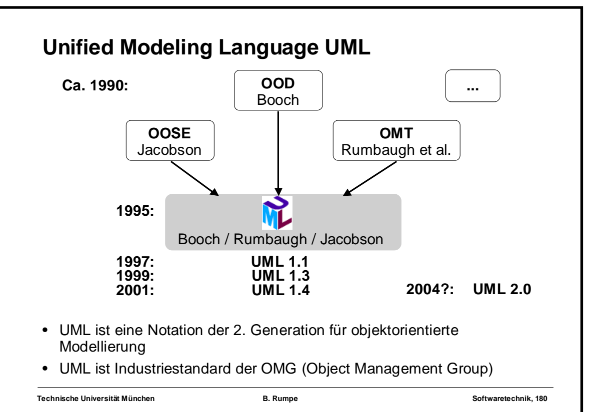

Entkopplung von Paketen
Personal- und Kundenverwaltung
Seminar-verwaltung
Dozent
Seminartyp
Veranstaltung
• Wichtigste Möglichkeit zur Entkopplung: Fachliche Zusammengehörigkeit sicherstellen
• Weitere Möglichkeiten zur Entkopplung (schon entwurfsnah):
– Dozentenklasse splitten in Personal- und seminarbezogene Information
– "Stellvertreter"-Klassen einführen (Entwurfsmuster Proxy, Half-Object-plus-Protocol)
– Schnittstellen-Objekte einführen (z.B. PersonalverwaltungsAPI) (Entwurfsmuster Facade)
Technische Universität München
B. Rumpe
Softwaretechnik, 175

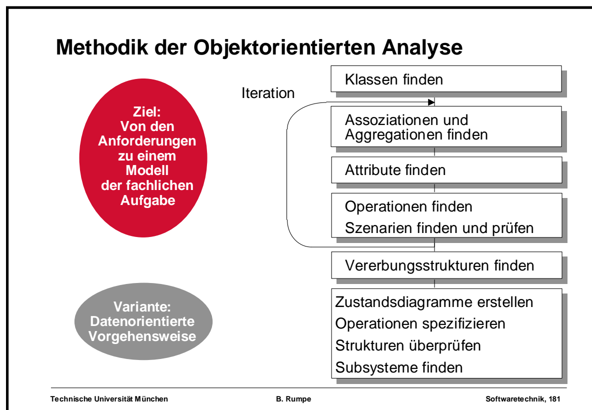

Zusammenfassung
4.2. Objektorientierte Analyse
• Objektorientierung bietet relativ gute Stabilität, und gewisse Vorzüge bei Änderungsfreundlichkeit und Wiederverwendbarkeit
• Alternative Vorgehensweisen:
– datenorientiert
– verhaltensorientiert
• Verschiedene Sichten und Detaillierungsebenen
– Kernstück: Klassendiagramm
– ergänzend: Zustandsdiagramme, Sequenzdiagramme, Use-Case-Diagramme
– Anbindung an Anforderungen durch Anwendungsfälle (use cases) und Szenarien
• Einheitliche Notation für Analyse (OOA) und Entwurf (OOD)
Technische Universität München
B. Rumpe
Softwaretechnik, 176


Zusammenfassung
4.2. Objektorientierte Analyse
TODAY IS THE THREE-YEAR ANNIVERSARY OF OUR FIRST MEETING TO DISCUSS PROJECT REQUIREMENTS.
AND WE'RE STILL DISCUSSING REQUIREMENTS. DOES ANYONE ELSE SEE A PROBLEM HERE?
WHEN YOU'RE DONE, CAN WE TALK ABOUT REQUIREMENTS?
www.dilbert.com
Copyright © 2002 United Feature Syndicate, Inc.
Technische Universität München
B. Rumpe
Softwaretechnik, 177

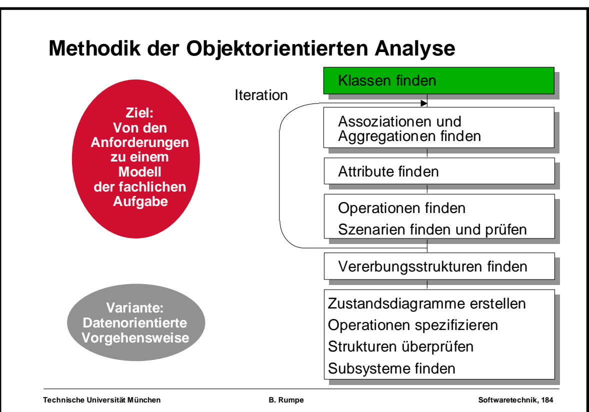

Softwaretechnik WS 02/03
4. Systemanalyse und Systemmodellierung
4.3. Statische Modellierung mit der UML (nach OMT)
Analyse
Entwurf
Implementierung
Test, Integration
Wartung
Anforderungs-ermittlung
System-modellierung
Literatur: Balzert LE 14
J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen: Object-oriented modeling and design (OMT), 1994
G. Booch, J. Rumbaugh, I. Jacobson: The Unified Modeling Language User Guide, 1999
Technische Universität München
B. Rumpe
Softwaretechnik, 178


Statische Modellierung mit der UML
• Inhalt dieses Abschnitts: eine konkrete Vorgehensweise für OOA
• Ziel: Vom Ergebnis der Anforderungsmodellierung mit Use-Cases oder CRC-Karten zu einer abstrakten Fassung der
– Datenstrukturen (Objekte) und zugehöriger
– Fach-Funktionalität beschrieben mit UML-Klassendiagrammen
– diese UML-Klassendiagramme enthalten keine Implementierungsdetails (wie zum Beispiel Sichtbarkeiten)
A
abra
kadabra
B
B
hokus
pokus
A
A
xyz
abra
kadabra
B
abc
hokus
pokus
Technische Universität München
B. Rumpe
Softwaretechnik, 179

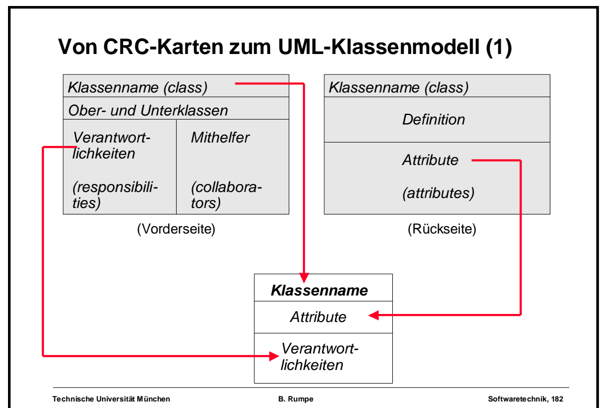

Unified Modeling Language UML
Ca. 1990:
OOD Booch
...
OOSE Jacobson
OMT Rumbaugh et al.
1995:
Booch / Rumbaugh / Jacobson
1997: UML 1.1
1999: UML 1.3
2001: UML 1.4
2004?: UML 2.0
• UML ist eine Notation der 2. Generation für objektorientierte Modellierung
• UML ist Industriestandard der OMG (Object Management Group)
Technische Universität München
B. Rumpe
Softwaretechnik, 180



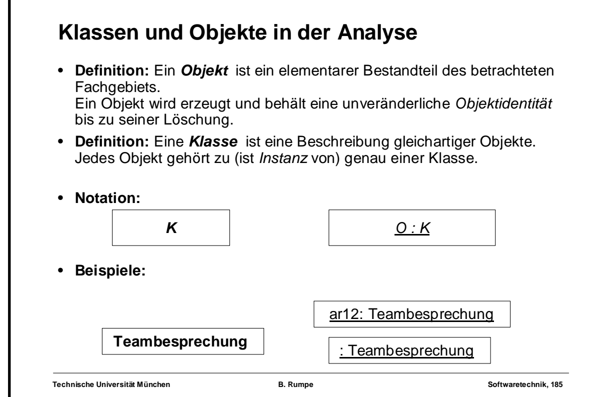
Methodik der Objektorientierten Analyse
Ziel: Von den Anforderungen zu einem Modell der fachlichen Aufgabe
Iteration
Klassen finden
Assoziationen und Aggregationen finden
Attribute finden
Operationen finden
Szenarien finden und prüfen
Vererbungsstrukturen finden
Variante: Datenorientierte Vorgehensweise
Zustandsdiagramme erstellen
Operationen spezifizieren
Strukturen überprüfen
Subsysteme finden
Technische Universität München
B. Rumpe
Softwaretechnik, 181



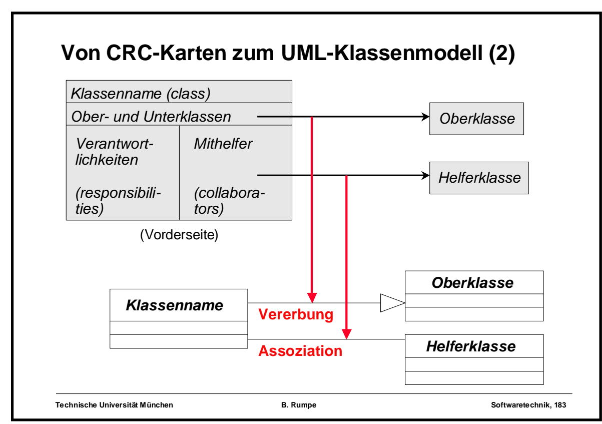
Methodik der Objektorientierten Analyse
Ziel: Von den Anforderungen zu einem Modell der fachlichen Aufgabe
Iteration
Klassen finden
Assoziationen und Aggregationen finden
Attribute finden
Operationen finden
Szenarien finden und prüfen
Vererbungsstrukturen finden
Variante: Datenorientierte Vorgehensweise
Zustandsdiagramme erstellen
Operationen spezifizieren
Strukturen überprüfen
Subsysteme finden
Technische Universität München
B. Rumpe
Softwaretechnik, 184



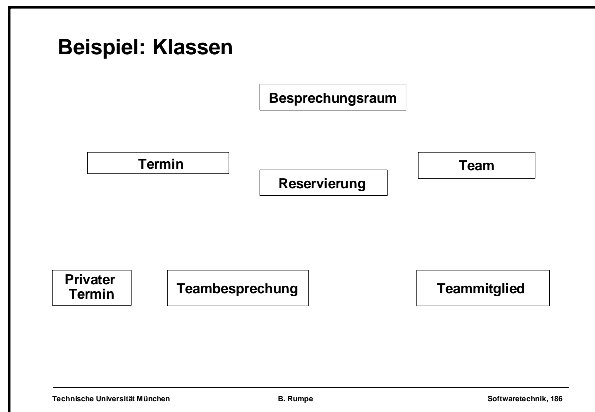
Von CRC-Karten zum UML-Klassenmodell (1)
Klassenname (class)
Ober- und Unterklassen
Verantwort-lichkeiten
(responsibili-ties)
Mithelfer
(collabora-tors)
(Vorderseite)
Klassenname (class)
Definition
Attribute
(attributes)
(Rückseite)
Klassenname
Attribute
Verantwort-lichkeiten
Technische Universität München
B. Rumpe
Softwaretechnik, 182



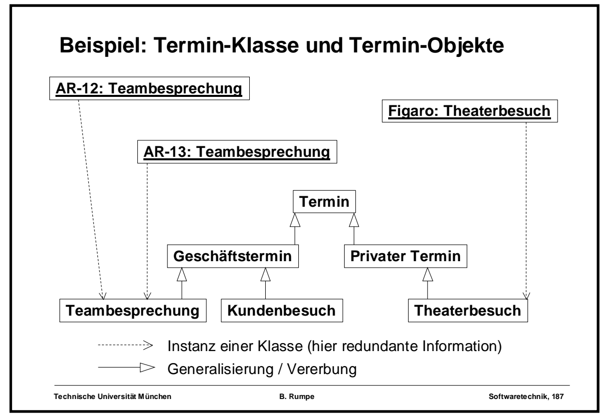
Klassen und Objekte in der Analyse
Definition: Ein Objekt ist ein elementarer Bestandteil des betrachteten Fachgebiets.
Ein Objekt wird erzeugt und behält eine unveränderliche Objektidentität bis zu seiner Löschung.
Definition: Eine Klasse ist eine Beschreibung gleichartiger Objekte.
Jedes Objekt gehört zu (ist Instanz von) genau einer Klasse.
Notation:
K
O : K
Beispiele:
Teambesprechung
ar12: Teambesprechung
: Teambesprechung
Technische Universität München
B. Rumpe
Softwaretechnik, 185

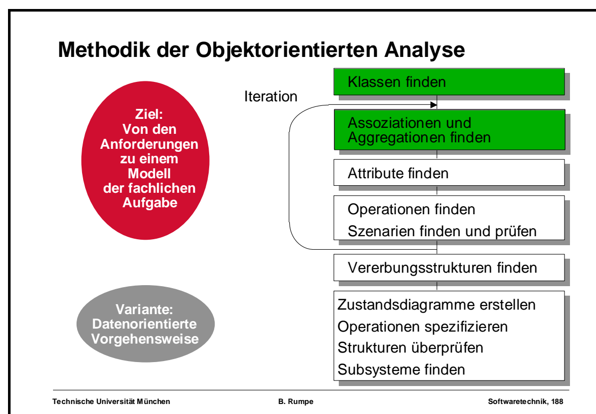


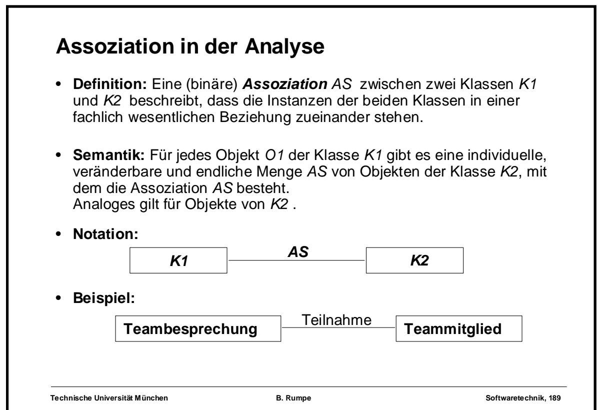
Von CRC-Karten zum UML-Klassenmodell (2)
Klassenname (class)
Ober- und Unterklassen
Verantwort-lichkeiten
(responsibili-ties)
Mithelfer
(collabora-tors)
(Vorderseite)
Oberklasse
Helferklasse
Klassenname
Vererbung
Assoziation
Oberklasse
Helferklasse
Technische Universität München
B. Rumpe
Softwaretechnik, 183

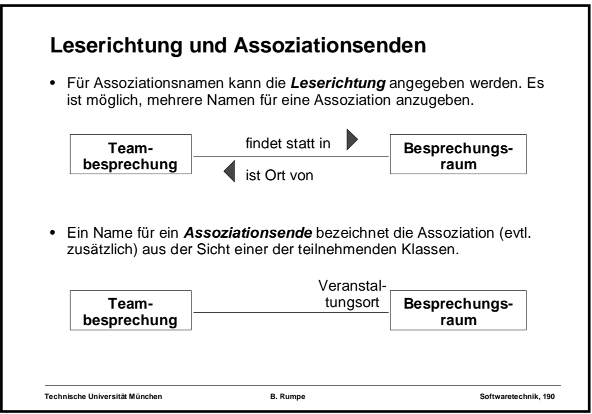


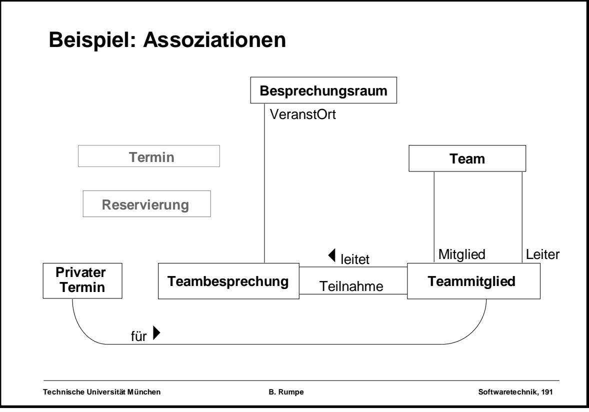
Beispiel: Klassen
Besprechungsraum
Termin
Reservierung
Team
Privater Termin
Teambesprechung
Teammitglied
Technische Universität München
B. Rumpe
Softwaretechnik, 186

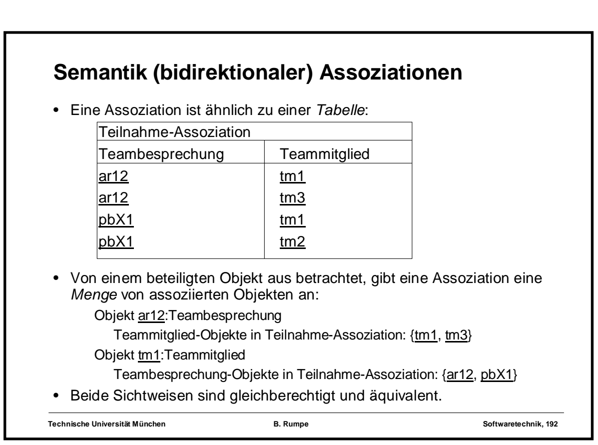

Beispiel: Termin-Klasse und Termin-Objekte
AR-12: Teambesprechung
Figaro: Theaterbesuch
AR-13: Teambesprechung
Termin
Geschäftstermin
Privater Termin
Teambesprechung
Kundenbesuch
Theaterbesuch
Instanz einer Klasse (hier redundante Information)
Generalisierung / Vererbung
Technische Universität München
B. Rumpe
Softwaretechnik, 187

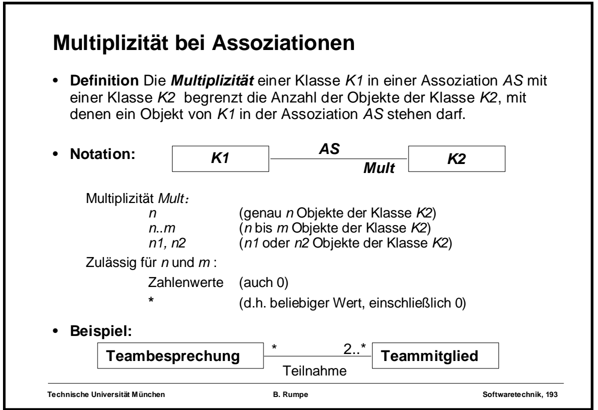

Methodik der Objektorientierten Analyse
Ziel: Von den Anforderungen zu einem Modell der fachlichen Aufgabe
Iteration
Klassen finden
Assoziationen und Aggregationen finden
Attribute finden
Operationen finden
Szenarien finden und prüfen
Vererbungsstrukturen finden
Variante: Datenorientierte Vorgehensweise
Zustandsdiagramme erstellen
Operationen spezifizieren
Strukturen überprüfen
Subsysteme finden
Technische Universität München
B. Rumpe
Softwaretechnik, 188

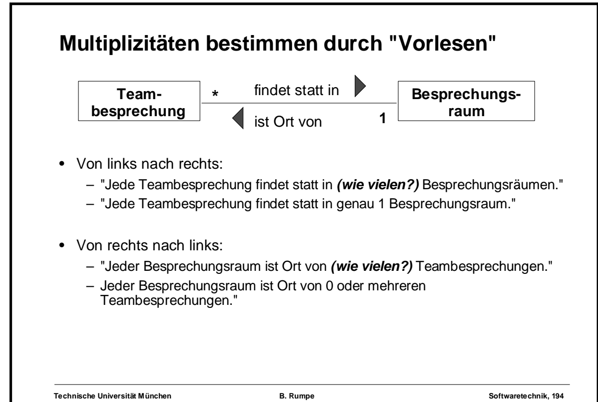

Assoziation in der Analyse
Definition: Eine (binäre) Assoziation AS zwischen zwei Klassen K1 und K2 beschreibt, dass die Instanzen der beiden Klassen in einer fachlich wesentlichen Beziehung zueinander stehen.
Semantik: Für jedes Objekt O1 der Klasse K1 gibt es eine individuelle, veränderbare und endliche Menge AS von Objekten der Klasse K2, mit dem die Assoziation AS besteht.
Analoges gilt für Objekte von K2 .
Notation:
K1
AS
K2
Beispiel:
Teambesprechung
Teilnahme
Teammitglied
Technische Universität München
B. Rumpe
Softwaretechnik, 189

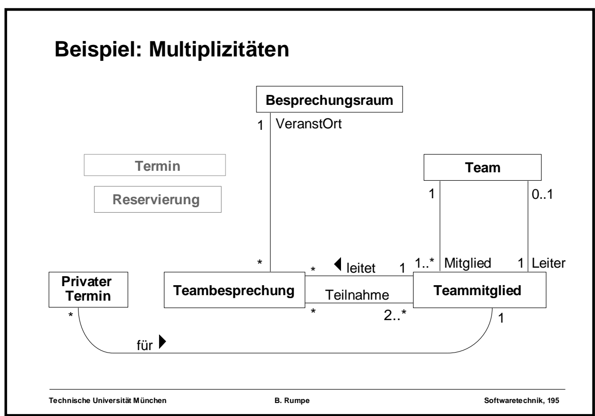

Leserichtung und Assoziationsenden
Für Assoziationsnamen kann die Leserichtung angegeben werden. Es ist möglich, mehrere Namen für eine Assoziation anzugeben.
Team-besprechung
findet statt in
ist Ort von
Besprechungs-raum
Ein Name für ein Assoziationsende bezeichnet die Assoziation (evtl. zusätzlich) aus der Sicht einer der teilnehmenden Klassen.
Team-besprechung
Veranstal-tungsort
Besprechungs-raum
Technische Universität München
B. Rumpe
Softwaretechnik, 190

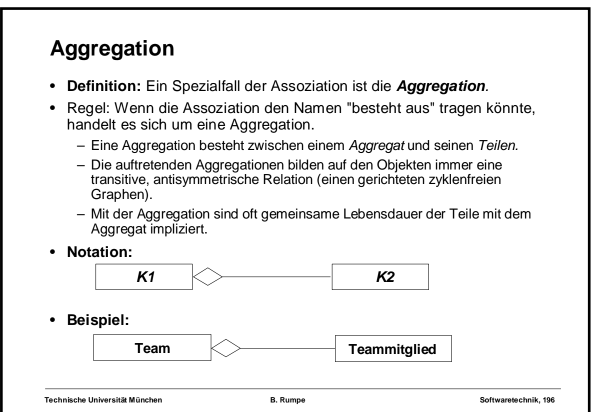

Beispiel: Assoziationen
Besprechungsraum
VeranstOrt
Termin
Team
Reservierung
leitet
Mitglied
Leiter
Privater Termin
Teambesprechung
Teilnahme
Teammitglied
für
Technische Universität München
B. Rumpe
Softwaretechnik, 191

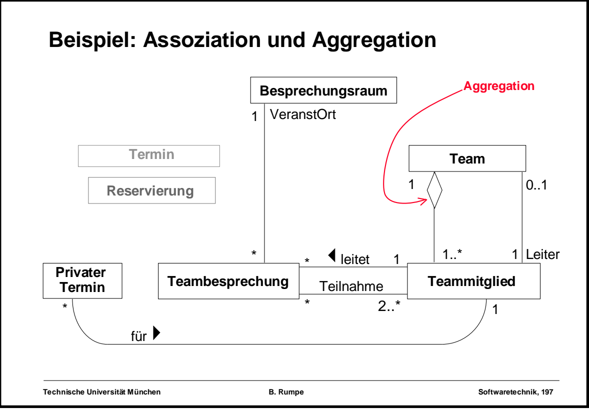

Semantik (bidirektionaler) Assoziationen
Eine Assoziation ist ähnlich zu einer Tabelle:
Teilnahme-Assoziation
Teambesprechung
Teammitglied
ar12
tm1
ar12
tm3
pbX1
tm1
pbX1
tm2
Von einem beteiligten Objekt aus betrachtet, gibt eine Assoziation eine Menge von assoziierten Objekten an:
Objekt ar12:Teambesprechung
Teammitglied-Objekte in Teilnahme-Assoziation: {tm1, tm3}
Objekt tm1:Teammitglied
Teambesprechung-Objekte in Teilnahme-Assoziation: {ar12, pbX1}
Beide Sichtweisen sind gleichberechtigt und äquivalent.
Technische Universität München
B. Rumpe
Softwaretechnik, 192

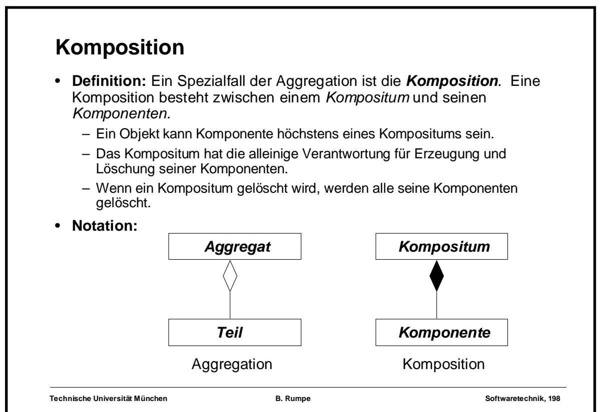

Multiplizität bei Assoziationen
• Definition Die Multiplizität einer Klasse K1 in einer Assoziation AS mit einer Klasse K2 begrenzt die Anzahl der Objekte der Klasse K2, mit denen ein Objekt von K1 in der Assoziation AS stehen darf.
• Notation:
K1
AS
Mult
K2
Multiplizität Mult:
n (genau n Objekte der Klasse K2)
n..m (n bis m Objekte der Klasse K2)
n1, n2 (n1 oder n2 Objekte der Klasse K2)
Zulässig für n und m :
Zahlenwerte (auch 0)
* (d.h. beliebiger Wert, einschließlich 0)
• Beispiel:
Teambesprechung
*
2..*
Teammitglied
Teilnahme
Technische Universität München
B. Rumpe
Softwaretechnik, 193

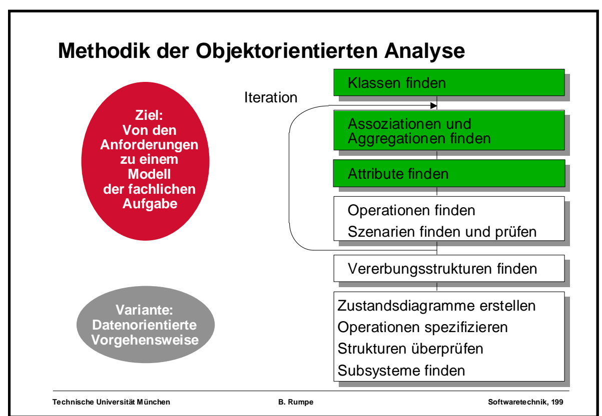

Multiplizitäten bestimmen durch "Vorlesen"
Team-
besprechung
*
findet statt in
ist Ort von
1
Besprechungs-
raum
• Von links nach rechts:
– "Jede Teambesprechung findet statt in (wie vielen?) Besprechungsräumen."
– "Jede Teambesprechung findet statt in genau 1 Besprechungsraum."
• Von rechts nach links:
– "Jeder Besprechungsraum ist Ort von (wie vielen?) Teambesprechungen."
– Jeder Besprechungsraum ist Ort von 0 oder mehreren Teambesprechungen."
Technische Universität München
B. Rumpe
Softwaretechnik, 194

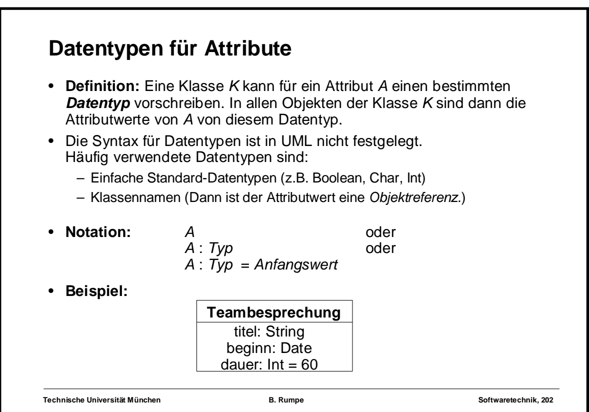

Beispiel: Multiplizitäten
Besprechungsraum
1
VeranstOrt
Termin
Reservierung
Team
1
0..1
*
*
leitet
1
1..*
Mitglied
1
Leiter
Privater Termin
Teambesprechung
Teilnahme
Teammitglied
*
*
2..*
1
für
Technische Universität München
B. Rumpe
Softwaretechnik, 195

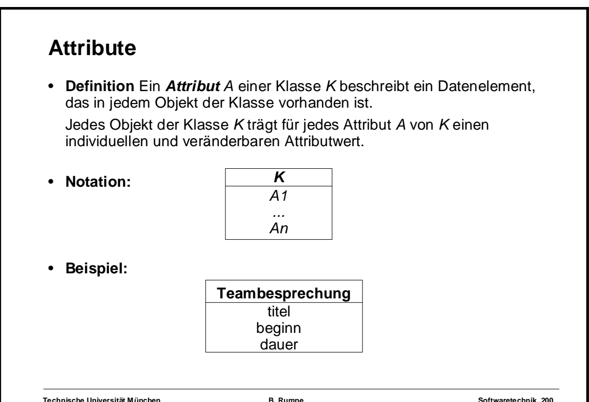

Aggregation
• Definition: Ein Spezialfall der Assoziation ist die Aggregation.
• Regel: Wenn die Assoziation den Namen "besteht aus" tragen könnte, handelt es sich um eine Aggregation.
– Eine Aggregation besteht zwischen einem Aggregat und seinen Teilen.
– Die auftretenden Aggregationen bilden auf den Objekten immer eine transitive, antisymmetrische Relation (einen gerichteten zyklenfreien Graphen).
– Mit der Aggregation sind oft gemeinsame Lebensdauer der Teile mit dem Aggregat impliziert.
• Notation:
K1
K2
• Beispiel:
Team
Teammitglied
Technische Universität München
B. Rumpe
Softwaretechnik, 196

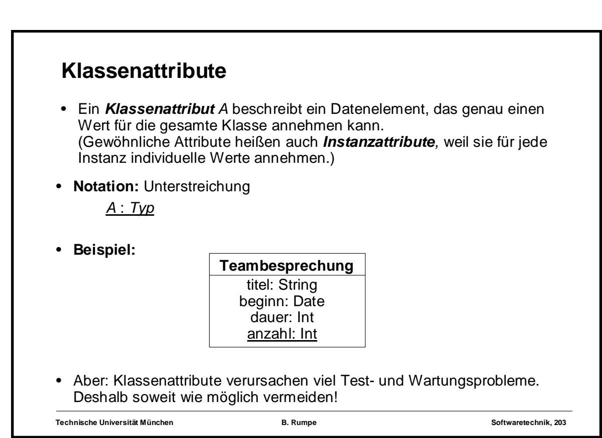

Beispiel: Assoziation und Aggregation
Besprechungsraum
Aggregation
1
VeranstOrt
Termin
Reservierung
Team
1
0..1
*
*
leitet
1
1..*
1
Leiter
Privater Termin
Teambesprechung
Teilnahme
Teammitglied
*
*
2..*
1
für
Technische Universität München
B. Rumpe
Softwaretechnik, 197

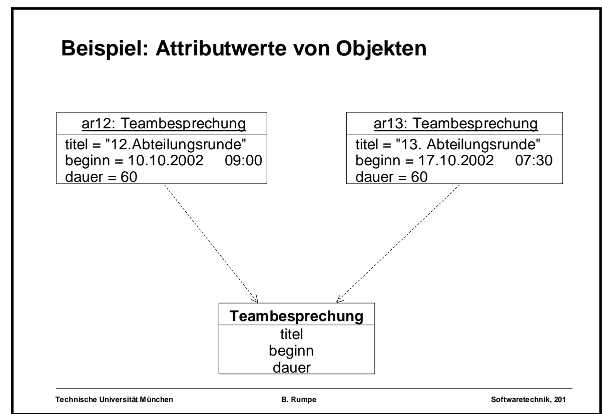

Komposition
• Definition: Ein Spezialfall der Aggregation ist die Komposition. Eine Komposition besteht zwischen einem Kompositum und seinen Komponenten.
– Ein Objekt kann Komponente höchstens eines Kompositums sein.
– Das Kompositum hat die alleinige Verantwortung für Erzeugung und Löschung seiner Komponenten.
– Wenn ein Kompositum gelöscht wird, werden alle seine Komponenten gelöscht.
• Notation:
Aggregat
Kompositum
Teil
Komponente
Aggregation
Komposition
Technische Universität München
B. Rumpe
Softwaretechnik, 198

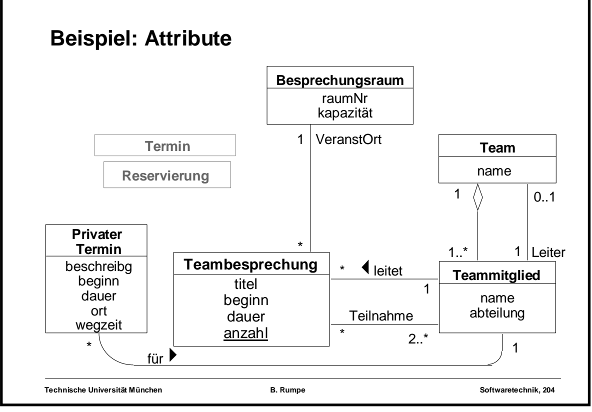

Methodik der Objektorientierten Analyse
Ziel: Von den Anforderungen zu einem Modell der fachlichen Aufgabe
Iteration
Klassen finden
Assoziationen und Aggregationen finden
Attribute finden
Operationen finden
Szenarien finden und prüfen
Vererbungsstrukturen finden
Variante: Datenorientierte Vorgehensweise
Zustandsdiagramme erstellen
Operationen spezifizieren
Strukturen überprüfen
Subsysteme finden
Technische Universität München
B. Rumpe
Softwaretechnik, 199

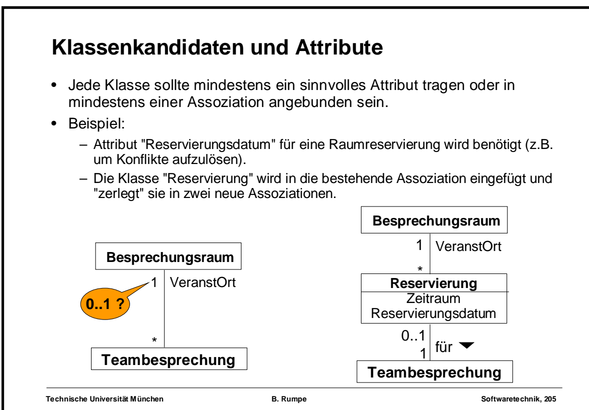

Datentypen für Attribute
Definition: Eine Klasse K kann für ein Attribut A einen bestimmten Datentyp vorschreiben. In allen Objekten der Klasse K sind dann die Attributwerte von A von diesem Datentyp.
Die Syntax für Datentypen ist in UML nicht festgelegt. Häufig verwendete Datentypen sind:
– Einfache Standard-Datentypen (z.B. Boolean, Char, Int)
– Klassennamen (Dann ist der Attributwert eine Objektreferenz.)
Notation:
A oder
A : Typ oder
A : Typ = Anfangswert
Beispiel:
Teambesprechung
titel: String
beginn: Date
dauer: Int = 60
Technische Universität München
B. Rumpe
Softwaretechnik, 202

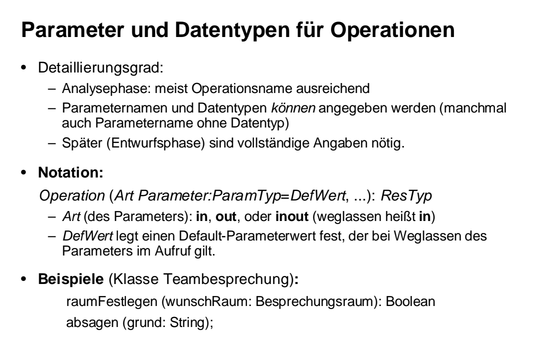

Attribute
Definition Ein Attribut A einer Klasse K beschreibt ein Datenelement, das in jedem Objekt der Klasse vorhanden ist.
Jedes Objekt der Klasse K trägt für jedes Attribut A von K einen individuellen und veränderbaren Attributwert.
Notation:
K
A1
...
An
Beispiel:
Teambesprechung
titel
beginn
dauer
Technische Universität München
B. Rumpe
Softwaretechnik, 200

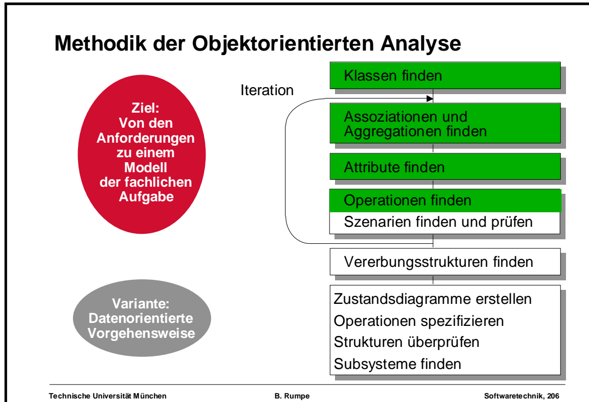

Klassenattribute
Ein Klassenattribut A beschreibt ein Datenelement, das genau einen Wert für die gesamte Klasse annehmen kann.
(Gewöhnliche Attribute heißen auch Instanzattribute, weil sie für jede Instanz individuelle Werte annehmen.)
Notation: Unterstreichung
A : Typ
Beispiel:
Teambesprechung
titel: String
beginn: Date
dauer: Int
anzahl: Int
Aber: Klassenattribute verursachen viel Test- und Wartungsprobleme.
Deshalb soweit wie möglich vermeiden!
Technische Universität München
B. Rumpe
Softwaretechnik, 203

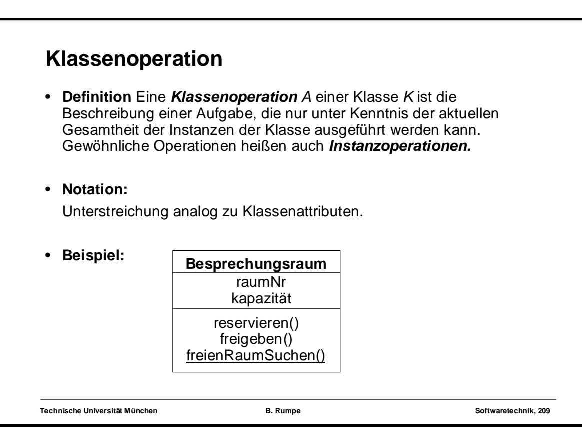

Beispiel: Attributwerte von Objekten
ar12: Teambesprechung
titel = "12.Abteilungsrunde"
beginn = 10.10.2002 09:00
dauer = 60
ar13: Teambesprechung
titel = "13. Abteilungsrunde"
beginn = 17.10.2002 07:30
dauer = 60
Teambesprechung
titel
beginn
dauer
Technische Universität München
B. Rumpe
Softwaretechnik, 201

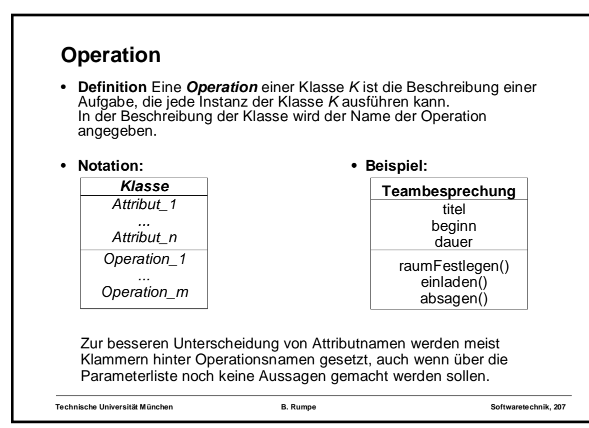

Beispiel: Attribute
Besprechungsraum
raumNr
kapazität
Termin
Reservierung
VeranstOrt
Team
name
Privater Termin
beschreibg
beginn
dauer
ort
wegzeit
Teambesprechung
titel
beginn
dauer
anzahl
leitet
Teammitglied
name
abteilung
Teilnahme
für
Leiter
1
0..1
1..*
2..*
*
Technische Universität München
B. Rumpe
Softwaretechnik, 204

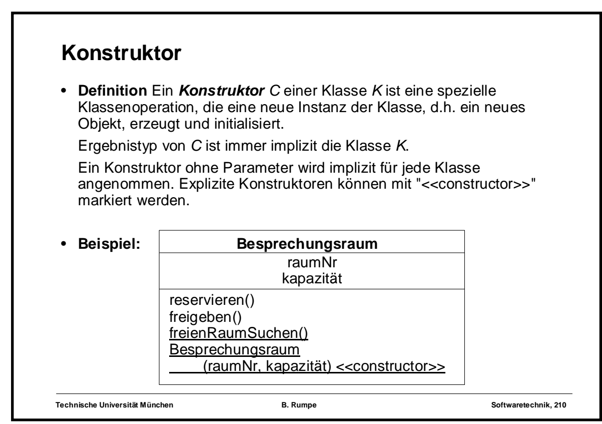

Klassenkandidaten und Attribute
• Jede Klasse sollte mindestens ein sinnvolles Attribut tragen oder in mindestens einer Assoziation angebunden sein.
• Beispiel:
– Attribut "Reservierungsdatum" für eine Raumreservierung wird benötigt (z.B. um Konflikte aufzulösen).
– Die Klasse "Reservierung" wird in die bestehende Assoziation eingefügt und "zerlegt" sie in zwei neue Assoziationen.
Besprechungsraum
1 VeranstOrt
0..1 ?
*
Teambesprechung
Besprechungsraum
1 VeranstOrt
*
Reservierung
Zeitraum
Reservierungsdatum
0..1
1 für
Teambesprechung
Technische Universität München B. Rumpe Softwaretechnik, 205

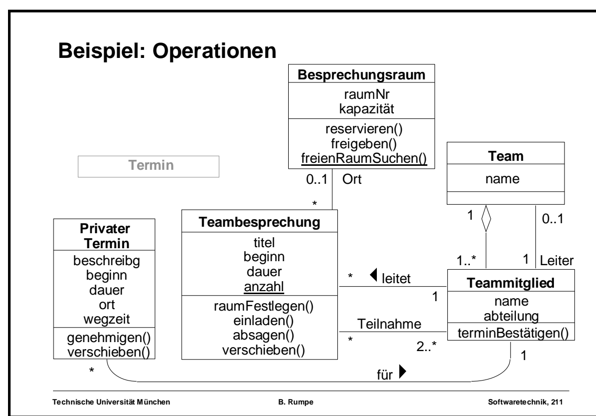

Parameter und Datentypen für Operationen
• Detaillierungsgrad:
– Analysephase: meist Operationsname ausreichend
– Parameternamen und Datentypen können angegeben werden (manchmal auch Parametername ohne Datentyp)
– Später (Entwurfsphase) sind vollständige Angaben nötig.
• Notation:
Operation (Art Parameter:ParamTyp=DefWert, ...): ResTyp
– Art (des Parameters): in, out, oder inout (weglassen heißt in)
– DefWert legt einen Default-Parameterwert fest, der bei Weglassen des Parameters im Aufruf gilt.
• Beispiele (Klasse Teambesprechung):
raumFestlegen (wunschRaum: Besprechungsraum): Boolean
absagen (grund: String);
Technische Universität München B. Rumpe Softwaretechnik, 208

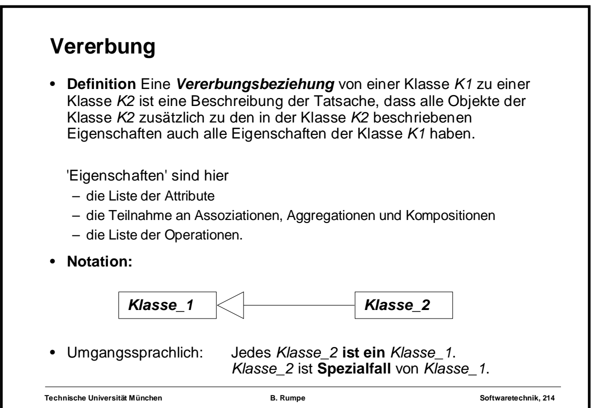

Methodik der Objektorientierten Analyse
Ziel: Von den Anforderungen zu einem Modell der fachlichen Aufgabe
Iteration
Klassen finden
Assoziationen und Aggregationen finden
Attribute finden
Operationen finden
Szenarien finden und prüfen
Vererbungsstrukturen finden
Zustandsdiagramme erstellen
Operationen spezifizieren
Strukturen überprüfen
Subsysteme finden
Variante: Datenorientierte Vorgehensweise
Technische Universität München B. Rumpe Softwaretechnik, 206

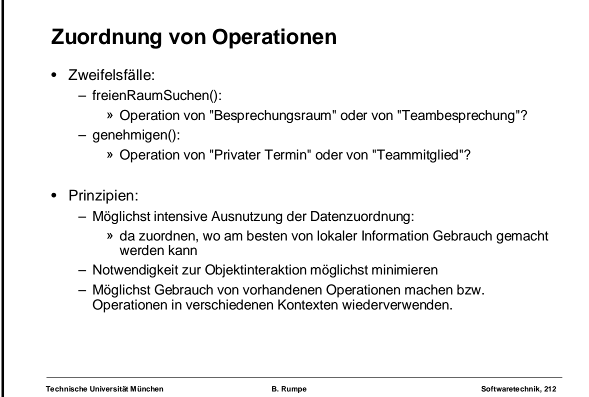

Klassenoperation
• Definition Eine Klassenoperation A einer Klasse K ist die Beschreibung einer Aufgabe, die nur unter Kenntnis der aktuellen Gesamtheit der Instanzen der Klasse ausgeführt werden kann. Gewöhnliche Operationen heißen auch Instanzoperationen.
• Notation:
Unterstreichung analog zu Klassenattributen.
• Beispiel:
Besprechungsraum
raumNr
kapazität
reservieren()
freigeben()
freienRaumSuchen()
Technische Universität München B. Rumpe Softwaretechnik, 209

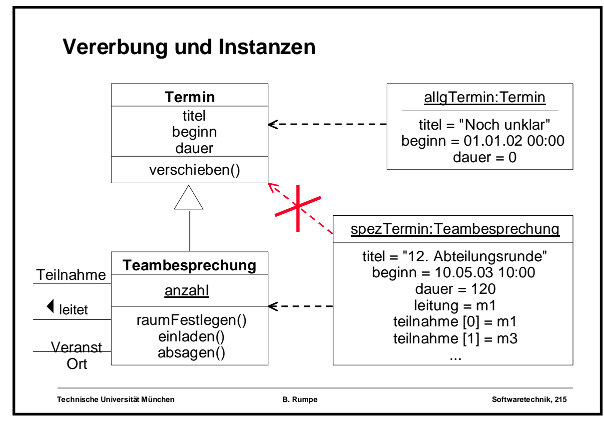

Operation
• Definition Eine Operation einer Klasse K ist die Beschreibung einer Aufgabe, die jede Instanz der Klasse K ausführen kann. In der Beschreibung der Klasse wird der Name der Operation angegeben.
• Notation:
Klasse
Attribut_1
...
Attribut_n
Operation_1
...
Operation_m
• Beispiel:
Teambesprechung
titel
beginn
dauer
raumFestlegen()
einladen()
absagen()
Zur besseren Unterscheidung von Attributnamen werden meist Klammern hinter Operationsnamen gesetzt, auch wenn über die Parameterliste noch keine Aussagen gemacht werden sollen.
Technische Universität München B. Rumpe Softwaretechnik, 207

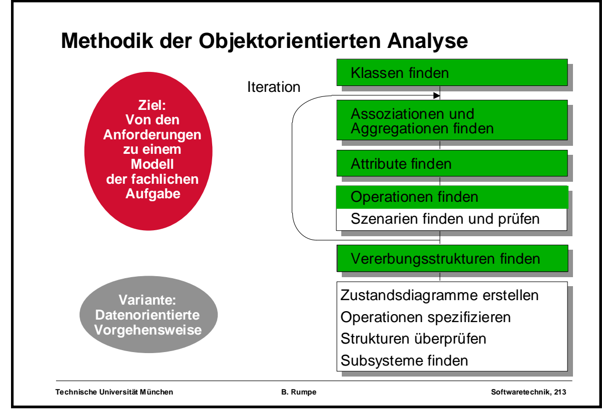

Konstruktor
• Definition Ein Konstruktor C einer Klasse K ist eine spezielle Klassenoperation, die eine neue Instanz der Klasse, d.h. ein neues Objekt, erzeugt und initialisiert.
Ergebnistyp von C ist immer implizit die Klasse K.
Ein Konstruktor ohne Parameter wird implizit für jede Klasse angenommen. Explizite Konstruktoren können mit "<<constructor>>" markiert werden.
• Beispiel:
Besprechungsraum
raumNr
kapazität
reservieren()
freigeben()
freienRaumSuchen()
Besprechungsraum
(raumNr, kapazität) <<constructor>>
Technische Universität München B. Rumpe Softwaretechnik, 210

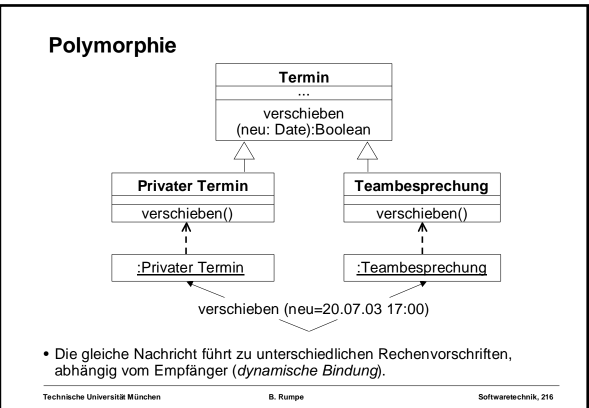

Beispiel: Operationen
Besprechungsraum
raumNr
kapazität
reservieren()
freigeben()
freienRaumSuchen()
Termin
Team
name
Privater Termin
beschreib
beginn
dauer
ort
wegzeit
genehmigen()
verschieben()
Teambesprechung
titel
beginn
dauer
anzahl
raumFestlegen()
einladen()
absagen()
verschieben()
Teammitglied
name
abteilung
terminBestätigen()
0..1
Ort
leitet
Teilnahme
Leiter
für
Technische Universität München
B. Rumpe
Softwaretechnik, 211

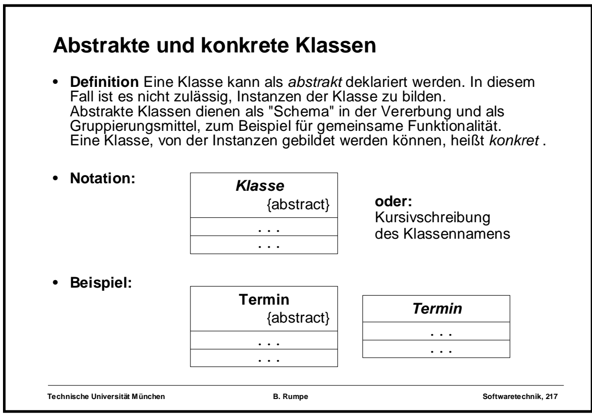

Vererbung
Definition Eine Vererbungsbeziehung von einer Klasse K1 zu einer Klasse K2 ist eine Beschreibung der Tatsache, dass alle Objekte der Klasse K2 zusätzlich zu den in der Klasse K2 beschriebenen Eigenschaften auch alle Eigenschaften der Klasse K1 haben.
'Eigenschaften' sind hier
– die Liste der Attribute
– die Teilnahme an Assoziationen, Aggregationen und Kompositionen
– die Liste der Operationen.
Notation:
Klasse_1
Klasse_2
Umgangssprachlich: Jedes Klasse_2 ist ein Klasse_1.
Klasse_2 ist Spezialfall von Klasse_1.
Technische Universität München
B. Rumpe
Softwaretechnik, 214

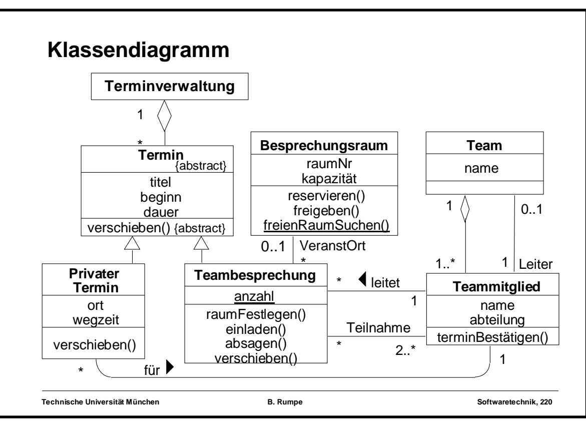

Zuordnung von Operationen
Zweifelsfälle:
– freienRaumSuchen():
» Operation von "Besprechungsraum" oder von "Teambesprechung"?
– genehmigen():
» Operation von "Privater Termin" oder von "Teammitglied"?
Prinzipien:
– Möglichst intensive Ausnutzung der Datenzuordnung:
» da zuordnen, wo am besten von lokaler Information Gebrauch gemacht werden kann
– Notwendigkeit zur Objektinteraktion möglichst minimieren
– Möglichst Gebrauch von vorhandenen Operationen machen bzw. Operationen in verschiedenen Kontexten wiederverwenden.
Technische Universität München
B. Rumpe
Softwaretechnik, 212

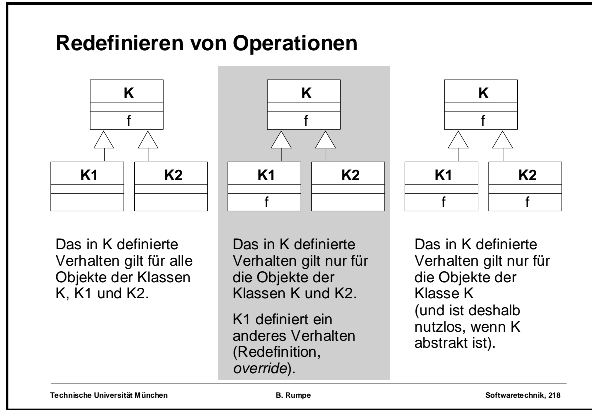

Vererbung und Instanzen
Termin
titel
beginn
dauer
verschieben()
allgTermin:Termin
titel = "Noch unklar"
beginn = 01.01.02 00:00
dauer = 0
spezTermin:Teambesprechung
titel = "12. Abteilungsrunde"
beginn = 10.05.03 10:00
dauer = 120
leitung = m1
teilnahme [0] = m1
teilnahme [1] = m3
...
Teilnahme
leitet
Veranst
Ort
Teambesprechung
anzahl
raumFestlegen()
einladen()
absagen()
Technische Universität München
B. Rumpe
Softwaretechnik, 215

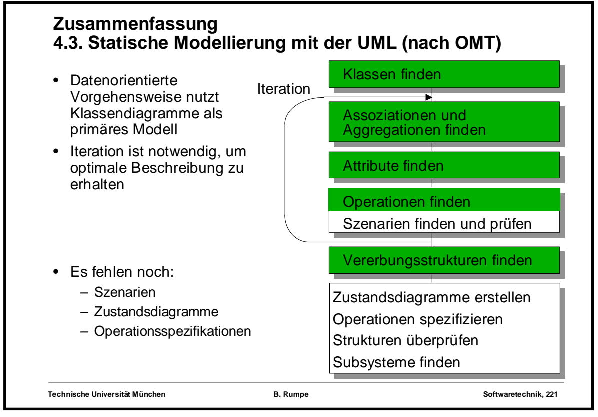

Methodik der Objektorientierten Analyse
Ziel: Von den Anforderungen zu einem Modell der fachlichen Aufgabe
Iteration
Klassen finden
Assoziationen und Aggregationen finden
Attribute finden
Operationen finden
Szenarien finden und prüfen
Vererbungsstrukturen finden
Zustandsdiagramme erstellen
Operationen spezifizieren
Strukturen überprüfen
Subsysteme finden
Variante: Datenorientierte Vorgehensweise
Technische Universität München
B. Rumpe
Softwaretechnik, 213

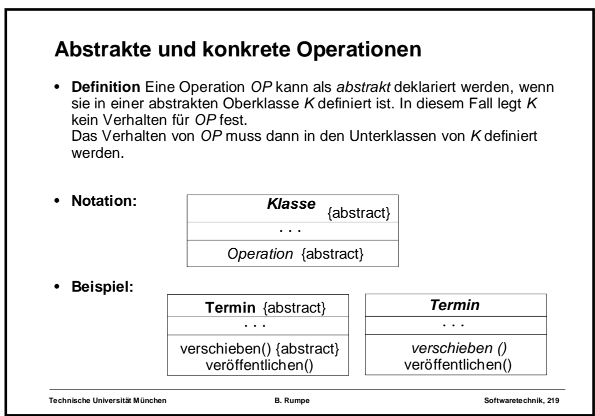

Polymorphie
Termin
...
verschieben
(neu: Date):Boolean
Privater Termin
verschieben()
Teambesprechung
verschieben()
:Privater Termin
:Teambesprechung
verschieben (neu=20.07.03 17:00)
Die gleiche Nachricht führt zu unterschiedlichen Rechenvorschriften, abhängig vom Empfänger (dynamische Bindung).
Technische Universität München
B. Rumpe
Softwaretechnik, 216

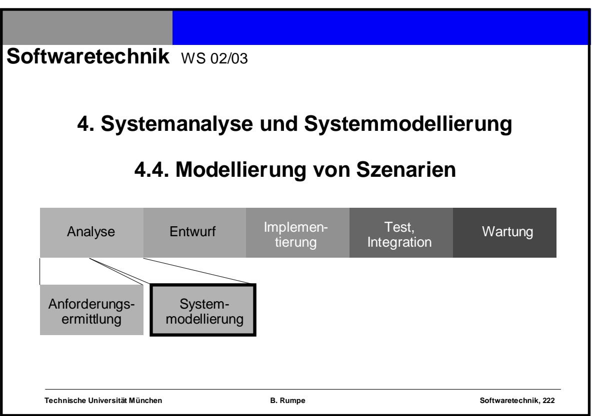

Abstrakte und konkrete Klassen
Definition Eine Klasse kann als abstrakt deklariert werden. In diesem Fall ist es nicht zulässig, Instanzen der Klasse zu bilden.
Abstrakte Klassen dienen als "Schema" in der Vererbung und als Gruppierungsmittel, zum Beispiel für gemeinsame Funktionalität.
Eine Klasse, von der Instanzen gebildet werden können, heißt konkret .
Notation:
Klasse
{abstract}
oder:
Kursivschreibung des Klassennamens
Beispiel:
Termin
{abstract}
Termin
Technische Universität München
B. Rumpe
Softwaretechnik, 217

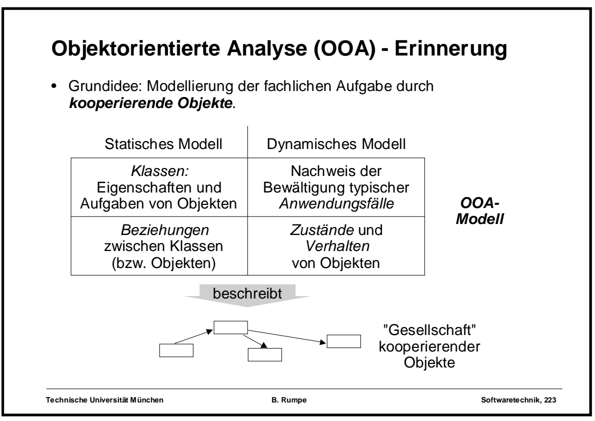

Klassendiagramm
Terminverwaltung
Termin
{abstract}
titel
beginn
dauer
verschieben() {abstract}
Besprechungsraum
raumNr
kapazität
reservieren()
freigeben()
freienRaumSuchen()
Team
name
Privater Termin
ort
wegzeit
verschieben()
Teambesprechung
anzahl
raumFestlegen()
einladen()
absagen()
verschieben()
Teammitglied
name
abteilung
terminBestätigen()
VeranstOrt
leitet
Teilnahme
Leiter
für
Technische Universität München
B. Rumpe
Softwaretechnik, 220


Redefinieren von Operationen
K
f
K1
K2
Das in K definierte Verhalten gilt für alle Objekte der Klassen K, K1 und K2.
Das in K definierte Verhalten gilt nur für die Objekte der Klassen K und K2.
K1 definiert ein anderes Verhalten (Redefinition, override).
Das in K definierte Verhalten gilt nur für die Objekte der Klasse K
(und ist deshalb nutzlos, wenn K abstrakt ist).
Technische Universität München
B. Rumpe
Softwaretechnik, 218

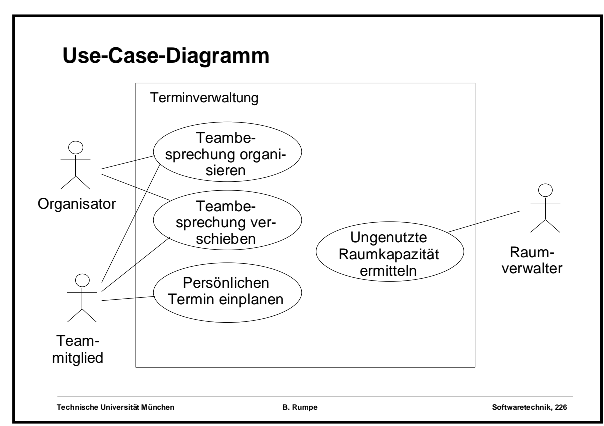

Zusammenfassung
4.3. Statische Modellierung mit der UML (nach OMT)
Datenorientierte Vorgehensweise nutzt Klassendiagramme als primäres Modell
Iteration ist notwendig, um optimale Beschreibung zu erhalten
Iteration
Klassen finden
Assoziationen und Aggregationen finden
Attribute finden
Operationen finden
Szenarien finden und prüfen
Vererbungsstrukturen finden
Es fehlen noch:
Szenarien
Zustandsdiagramme
Operationsspezifikationen
Zustandsdiagramme erstellen
Operationen spezifizieren
Strukturen überprüfen
Subsysteme finden
Technische Universität München
B. Rumpe
Softwaretechnik, 221


Abstrakte und konkrete Operationen
Definition Eine Operation OP kann als abstrakt deklariert werden, wenn sie in einer abstrakten Oberklasse K definiert ist. In diesem Fall legt K kein Verhalten für OP fest.
Das Verhalten von OP muss dann in den Unterklassen von K definiert werden.
Notation:
Klasse {abstract}
Operation {abstract}
Beispiel:
Termin {abstract}
verschieben() {abstract}
veröffentlichen()
Termin
verschieben ()
veröffentlichen()
Technische Universität München
B. Rumpe
Softwaretechnik, 219

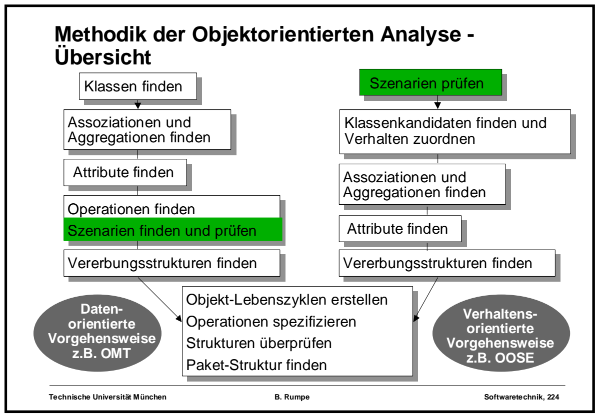

Softwaretechnik WS 02/03
4. Systemanalyse und Systemmodellierung
4.4. Modellierung von Szenarien
Analyse
Entwurf
Implementierung
Test, Integration
Wartung
Anforderungsermittlung
System-modellierung
Technische Universität München
B. Rumpe
Softwaretechnik, 222



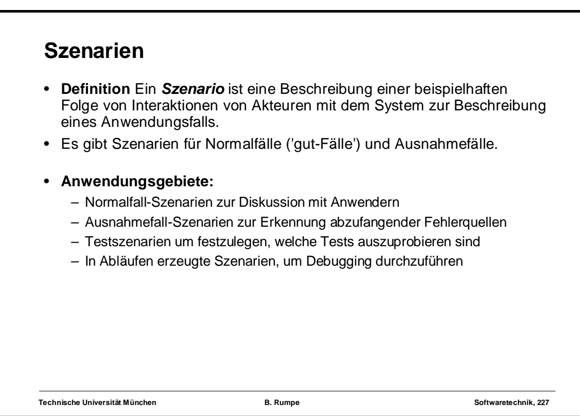
Objektorientierte Analyse (OOA) - Erinnerung
• Grundidee: Modellierung der fachlichen Aufgabe durch kooperierende Objekte.
Statisches Modell
Dynamisches Modell
Klassen: Eigenschaften und Aufgaben von Objekten
Nachweis der Bewältigung typischer Anwendungsfälle
Beziehungen zwischen Klassen (bzw. Objekten)
Zustände und Verhalten von Objekten
OOA-Modell
beschreibt
"Gesellschaft" kooperierender Objekte
Technische Universität München
B. Rumpe
Softwaretechnik, 223



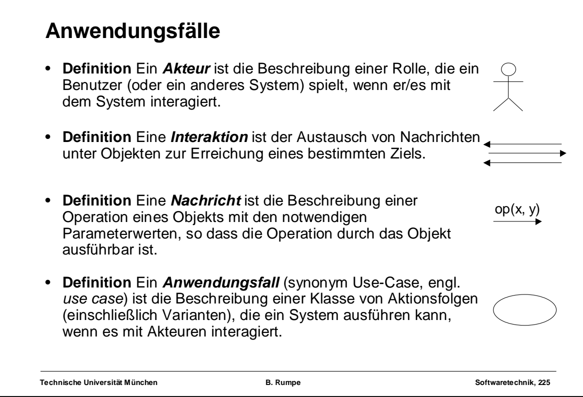
Use-Case-Diagramm
Terminverwaltung
Teambesprechung organisieren
Teambesprechung verschieben
Persönlichen Termin einplanen
Ungenutzte Raumkapazität ermitteln
Organisator
Teammitglied
Raumverwalter
Technische Universität München
B. Rumpe
Softwaretechnik, 226



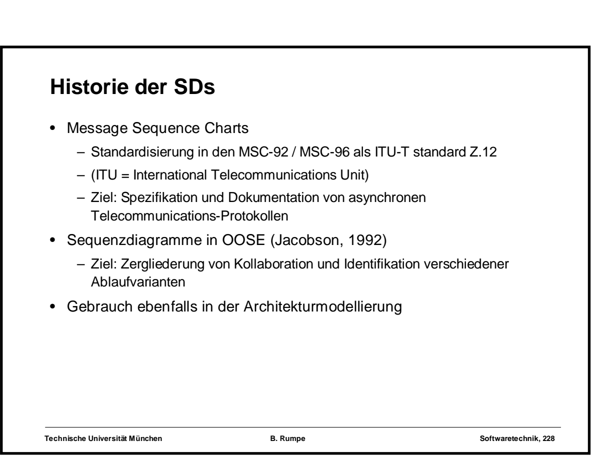
Methodik der Objektorientierten Analyse - Übersicht
Klassen finden
Assoziationen und Aggregationen finden
Attribute finden
Operationen finden
Szenarien finden und prüfen
Vererbungsstrukturen finden
Szenarien prüfen
Klassenkandidaten finden und Verhalten zuordnen
Assoziationen und Aggregationen finden
Attribute finden
Vererbungsstrukturen finden
Objekt-Lebenszyklen erstellen
Operationen spezifizieren
Strukturen überprüfen
Paket-Struktur finden
Daten-orientierte Vorgehensweise z.B. OMT
Verhaltens-orientierte Vorgehensweise z.B. OOSE
Technische Universität München
B. Rumpe
Softwaretechnik, 224



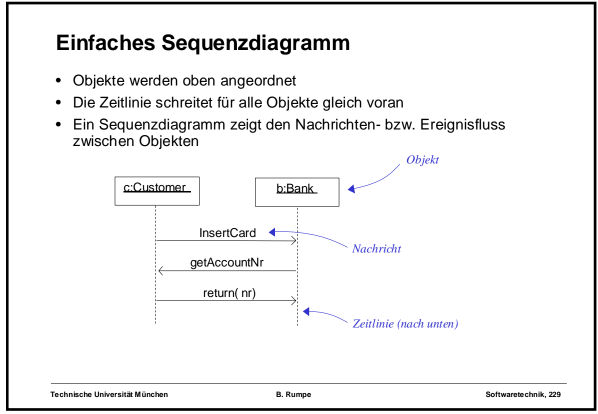
Szenarien
• Definition Ein Szenario ist eine Beschreibung einer beispielhaften Folge von Interaktionen von Akteuren mit dem System zur Beschreibung eines Anwendungsfalls.
• Es gibt Szenarien für Normalfälle ('gut-Fälle') und Ausnahmefälle.
• Anwendungsgebiete:
– Normalfall-Szenarien zur Diskussion mit Anwendern
– Ausnahmefall-Szenarien zur Erkennung abzufangender Fehlerquellen
– Testszenarien um festzulegen, welche Tests auszuprobieren sind
– In Abläufen erzeugte Szenarien, um Debugging durchzuführen
Technische Universität München
B. Rumpe
Softwaretechnik, 227

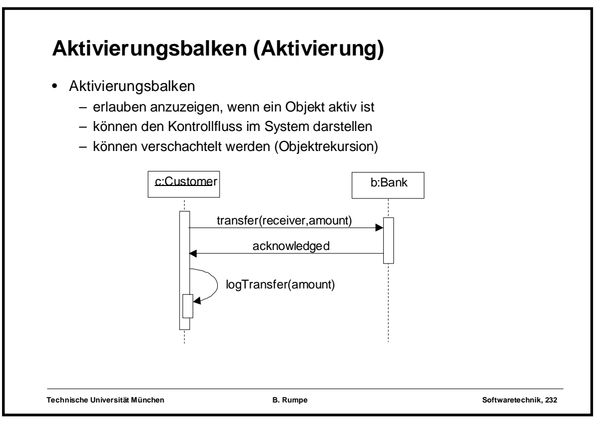


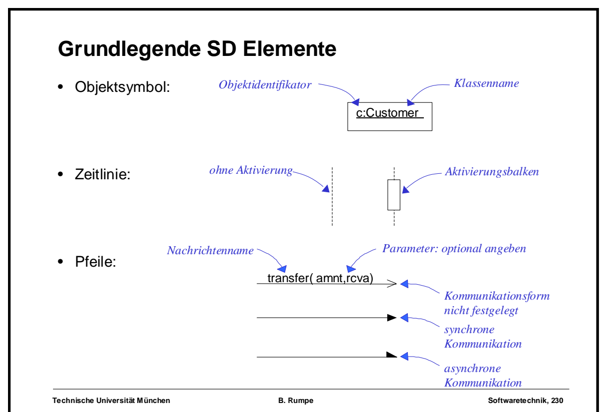
Anwendungsfälle
• Definition Ein Akteur ist die Beschreibung einer Rolle, die ein Benutzer (oder ein anderes System) spielt, wenn er/es mit dem System interagiert.
• Definition Eine Interaktion ist der Austausch von Nachrichten unter Objekten zur Erreichung eines bestimmten Ziels.
• Definition Eine Nachricht ist die Beschreibung einer Operation eines Objekts mit den notwendigen Parameterwerten, so dass die Operation durch das Objekt ausführbar ist.
op(x, y)
• Definition Ein Anwendungsfall (synonym Use-Case, engl. use case) ist die Beschreibung einer Klasse von Aktionsfolgen (einschließlich Varianten), die ein System ausführen kann, wenn es mit Akteuren interagiert.
Technische Universität München
B. Rumpe
Softwaretechnik, 225

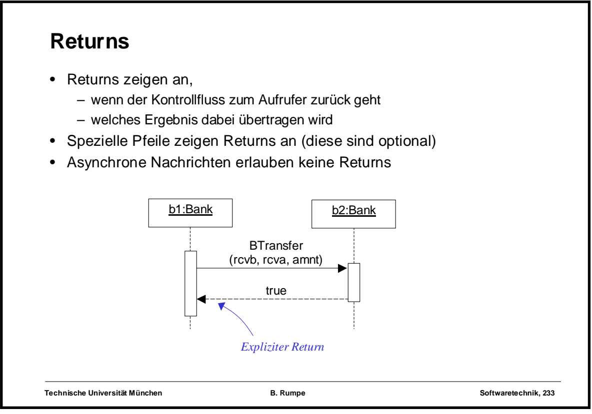


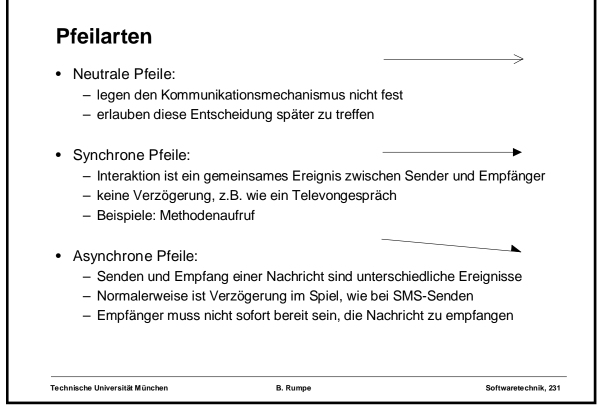
Historie der SDs
• Message Sequence Charts
– Standardisierung in den MSC-92 / MSC-96 als ITU-T standard Z.12
– (ITU = International Telecommunications Unit)
– Ziel: Spezifikation und Dokumentation von asynchronen Telecommunications-Protokollen
• Sequenzdiagramme in OOSE (Jacobson, 1992)
– Ziel: Zergliederung von Kollaboration und Identifikation verschiedener Ablaufvarianten
• Gebrauch ebenfalls in der Architekturmodellierung
Technische Universität München
B. Rumpe
Softwaretechnik, 228

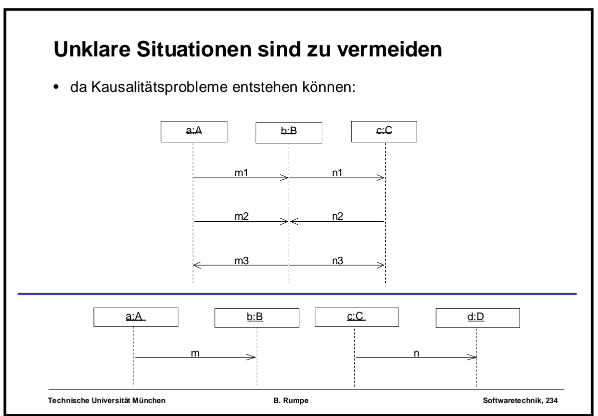

Einfaches Sequenzdiagramm
• Objekte werden oben angeordnet
• Die Zeitlinie schreitet für alle Objekte gleich voran
• Ein Sequenzdiagramm zeigt den Nachrichten- bzw. Ereignisfluss zwischen Objekten
Objekt
c:Customer
b:Bank
InsertCard
Nachricht
getAccountNr
return(nr)
Zeitlinie (nach unten)
Technische Universität München
B. Rumpe
Softwaretechnik, 229


Aktivierungsbalken (Aktivierung)
• Aktivierungsbalken
– erlauben anzuzeigen, wenn ein Objekt aktiv ist
– können den Kontrollfluss im System darstellen
– können verschachtelt werden (Objektrekursion)
c:Customer
b:Bank
transfer(receiver,amount)
acknowledged
logTransfer(amount)
Technische Universität München
B. Rumpe
Softwaretechnik, 232


Grundlegende SD Elemente
• Objektsymbol:
Objektidentifikator
Klassenname
c:Customer
• Zeitlinie:
ohne Aktivierung
Aktivierungsbalken
• Pfeile:
Nachrichtenname
Parameter: optional angeben
transfer(amnt,rcva)
Kommunikationsform nicht festgelegt
synchrone Kommunikation
asynchrone Kommunikation
Technische Universität München
B. Rumpe
Softwaretechnik, 230

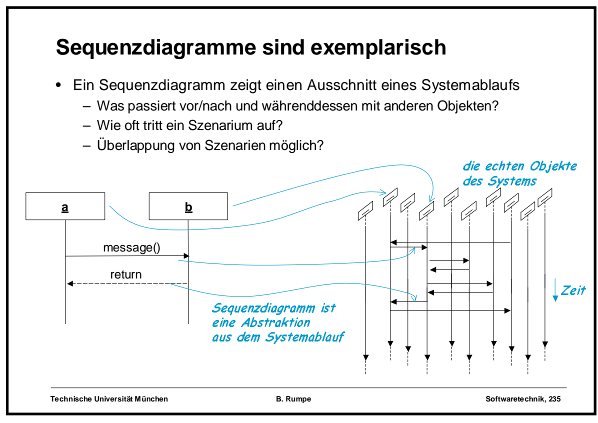

Returns
• Returns zeigen an,
– wenn der Kontrollfluss zum Aufrufer zurück geht
– welches Ergebnis dabei übertragen wird
• Spezielle Pfeile zeigen Returns an (diese sind optional)
• Asynchrone Nachrichten erlauben keine Returns
b1:Bank
b2:Bank
BTransfer
(rcvb, rcva, amnt)
true
Expliziter Return
Technische Universität München
B. Rumpe
Softwaretechnik, 233


Pfeilarten
• Neutrale Pfeile:
– legen den Kommunikationsmechanismus nicht fest
– erlauben diese Entscheidung später zu treffen
• Synchrone Pfeile:
– Interaktion ist ein gemeinsames Ereignis zwischen Sender und Empfänger
– keine Verzögerung, z.B. wie ein Televongespräch
– Beispiele: Methodenaufruf
• Asynchrone Pfeile:
– Senden und Empfang einer Nachricht sind unterschiedliche Ereignisse
– Normalerweise ist Verzögerung im Spiel, wie bei SMS-Senden
– Empfänger muss nicht sofort bereit sein, die Nachricht zu empfangen
Technische Universität München
B. Rumpe
Softwaretechnik, 231


Unklare Situationen sind zu vermeiden
• da Kausalitätsprobleme entstehen können:
a:A
b:B
c:C
m1
n1
m2
n2
m3
n3
a:A
b:B
c:C
d:D
m
n
Technische Universität München
B. Rumpe
Softwaretechnik, 234

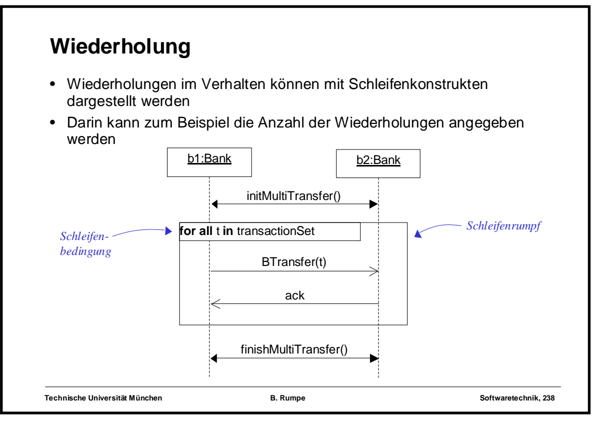
Sequenzdiagramme sind exemplarisch

- Ein Sequenzdiagramm zeigt einen Ausschnitt eines Systemablaufs
  - Was passiert vor/nach und währenddessen mit anderen Objekten?
  - Wie oft tritt ein Szenarium auf?
  - Überlappung von Szenarien möglich?



Wiederholung

- Wiederholungen im Verhalten können mit Schleifenkonstrukten dargestellt werden
- Darin kann zum Beispiel die Anzahl der Wiederholungen angegeben werden



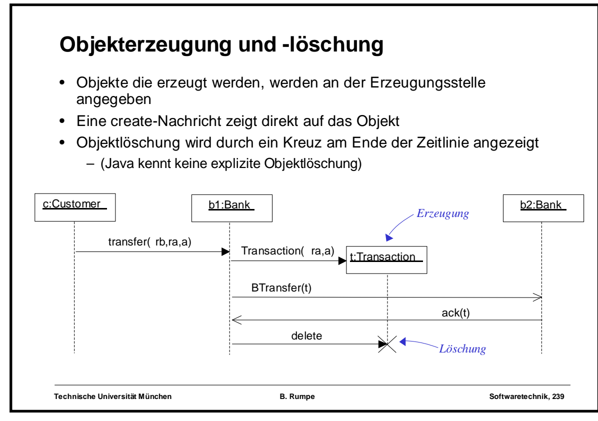
SD Erweiterung: Alternativen mit Guarded Messages

- Ein Guard ist ein boolescher Ausdruck der beschreibt unter welchen Umständen eine Nachricht auftritt
- Allerdings wird die Lesbarkeit durch solche Guards schnell reduziert

c:Customer
b:Bank
transfer(rcvb, rcva, amnt)
[amnt<limit] true
[amnt≥limit] false
Guard
Alternativen

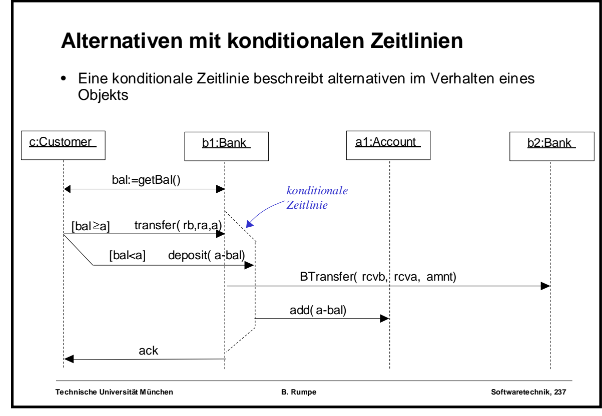
Objekterzeugung und -löschung

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



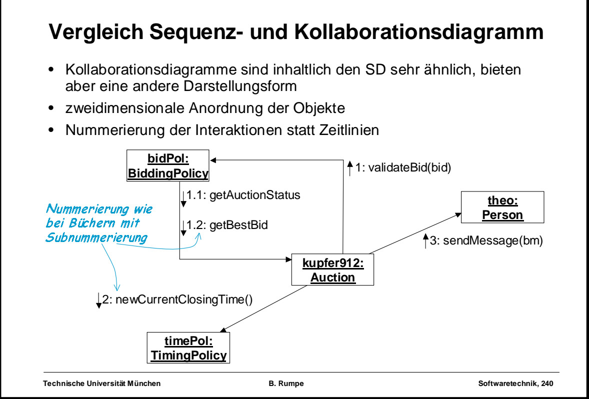
Alternativen mit konditionalen Zeitlinien

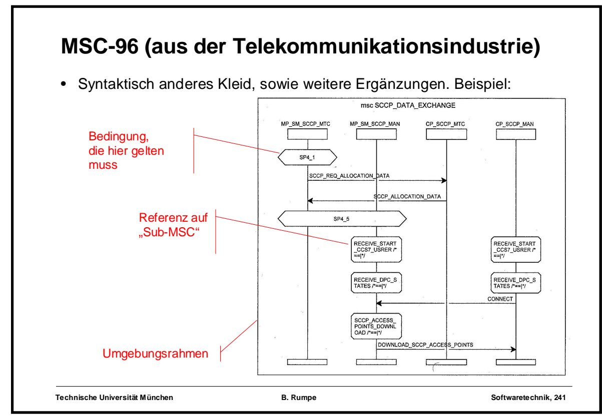
- Eine konditionale Zeitlinie beschreibt alternativen im Verhalten eines Objekts



Vergleich Sequenz- und Kollaborationsdiagramm

- Kollaborationsdiagramme sind inhaltlich den SD sehr ähnlich, bieten aber eine andere Darstellungsform
- zweidimensionale Anordnung der Objekte
- Nummerierung der Interaktionen statt Zeitlinien





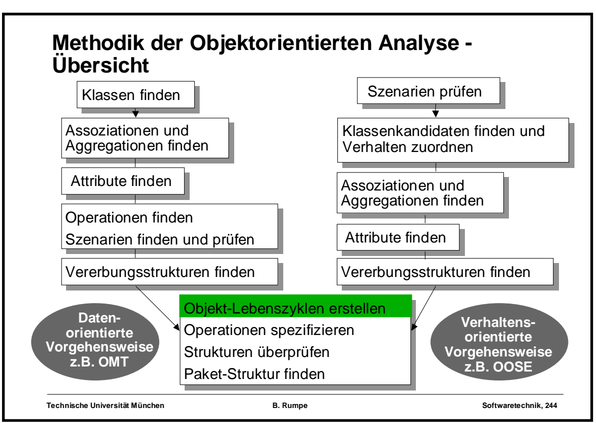
Zusammenfassung 4.4: Modellierung von Szenarien

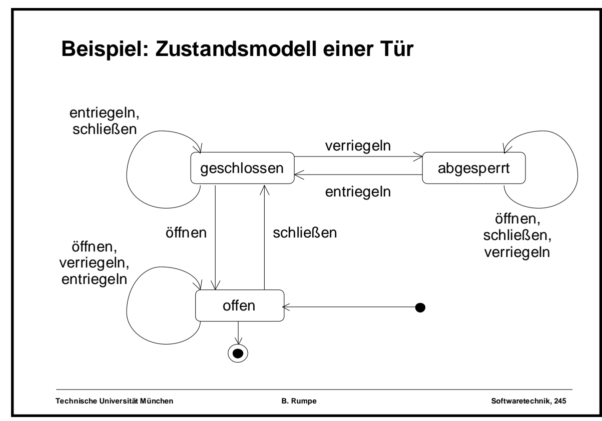
- Szenarien werden primär durch Sequenzdiagramme dargestellt
- Ein Sequenzdiagramm besteht aus
  - einer horizontal angeordneten Menge von Objekten
  - nach unten voranschreitenden Zeitlinien
  - und synchronen oder asynchronen Interaktionen zwischen den Objekten
- Anwendungen:
  - Systeminterne Kommunikation
  - Kommunikation zwischen System und Anwender
- Sequenzdiagramme sind exemplarisch und können daher keine vollständigen Verhaltensbeschreibungen sein.
- Erweiterungen wie Alternativen oder Wiederholung erlauben Mengen von Verhalten zu beschreiben, werden aber leichter unleserlich.

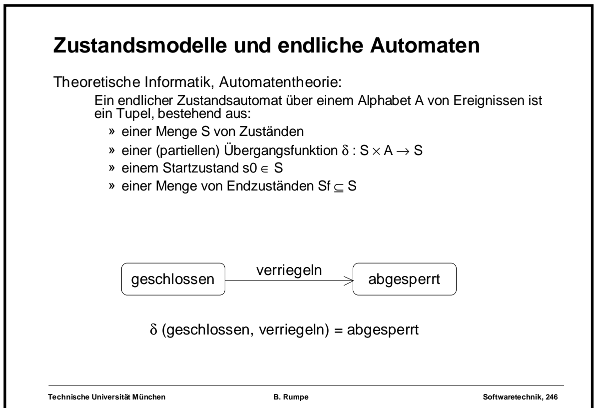
Softwaretechnik WS 02/03

4. Systemanalyse und Systemmodellierung

4.5. Dynamische Modellierung mit Statecharts (einschl. Umsetzung in Code)







Semantik eines Zustandsmodells

- Die Semantik eines Zustandsmodells ist definiert als Menge von Sequenzen:
  - in der Theoretischen Informatik: Menge von "akzeptierten Wörtern" (über Grundalphabet von Ereignissen)
  - in der Softwaretechnik: Menge von zulässigen *Ereignisfolgen*
- Wichtige Verallgemeinerung: "Automaten mit Ausgabe"
  - Mealy-Automaten: Ausgabe bei Übergang
    - » Softwaretechnik: *Aktion* bei Übergang
  - Moore-Automaten: Ausgabe bei Erreichen eines Zustands
    - » Softwaretechnik: *Eintrittsaktion (entry action)*

UML-Zustandsmodelle

- **Definition:** Ein ***Zustand*** ist eine Eigenschaft eines Systems, die über einen begrenzten Zeitraum besteht.
- **Notation:**

Z

- Was ist ein "System"?
  - Technisch: Ein Objekt oder eine Gruppe von Objekten
  - Praktisch:
    - » Eigenschaft eines einzelnen Objekts
    - » Eigenschaft eines komplexen Softwaresystems
    - » Eigenschaft eines Arbeitsprozesses
    - » Eigenschaft eines Produkts eines Arbeitsprozesses

Ereignisse

- **Definition** Ein ***Ereignis*** ist ein Geschehen von vernachlässigbarer Zeitdauer, das auf das betrachtete System Auswirkungen hat.

  Eine ***Ereignisklasse*** wird durch ihren Namen und evtl. weitere Parameter beschrieben.

  Eine ***Ereignisinstanz*** ist ein Objekt einer Ereignisklasse, das unter Umständen durch konkrete Werte für ***Parameter*** der Ereignisklasse näher beschrieben wird.

  Sprachgebrauch: Ereignis = Ereignisklasse = Ereignisinstanz

- **Notation:** *E*

  E (*P1*, ..., *Pn*)

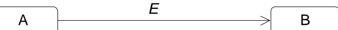
  E (*P1* : *T1*, ..., *Pn* : *Tn*)

Arten von Ereignissen

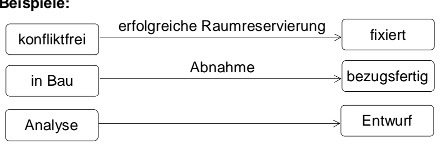
- Allgemeingültige Arten von Ereignissen:
  - Empfang einer Nachricht von außen
  - Ablaufen einer Zeitbedingung *(time-out)*
  - Veränderung einer (überwachten) Bedingung *(change event)*
  - Beendigung einer Folge von Aktionen ("namenloses" Ereignis)
- Spezielle Arten von Ereignissen für einzelne Objekte:
  - Eintreffen einer Nachricht bei einem Objekt
    - » Aufruf einer Operation (Methode)
    - » Signal (ohne zugehörige Methode, wird nur im Zustandsdiagramm behandelt)
  - Erzeugen oder Löschen des Objekts
- Detaillierungsgrad in der Analysephase:
  - Ereignisse allgemeingültiger Art
  - Textuelle Beschreibungen

Zustandsübergänge

- **Definition:**
  Ein ***Zustandsübergang*** von Zustand *A* nach Zustand *B* mit Ereignisnamen *E* besagt, dass im Zustand *A* bei Auftreten eines *E*–Ereignisses der neue Zustand *B* angenommen wird.
- **Notation:**



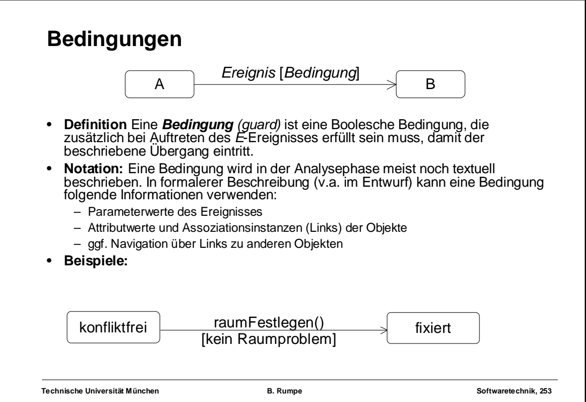
- **Beispiele:**

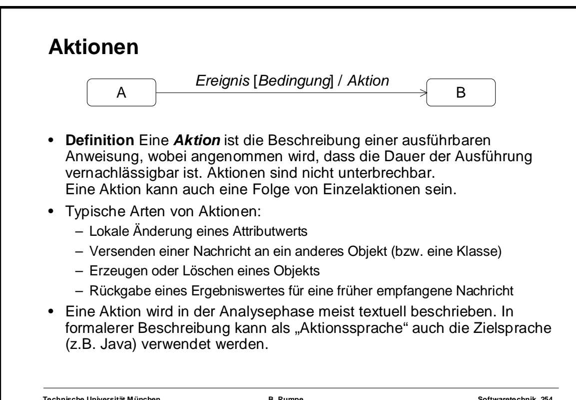


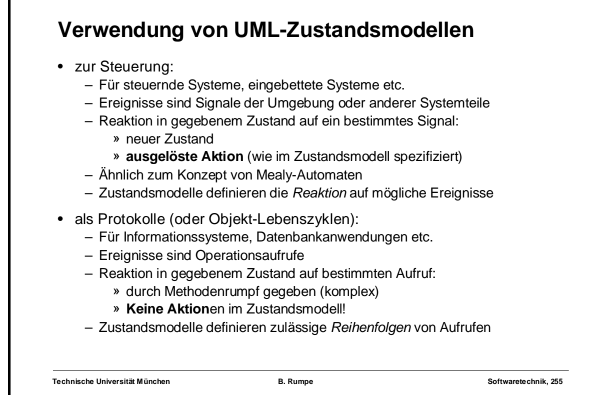
Start- und Endzustand

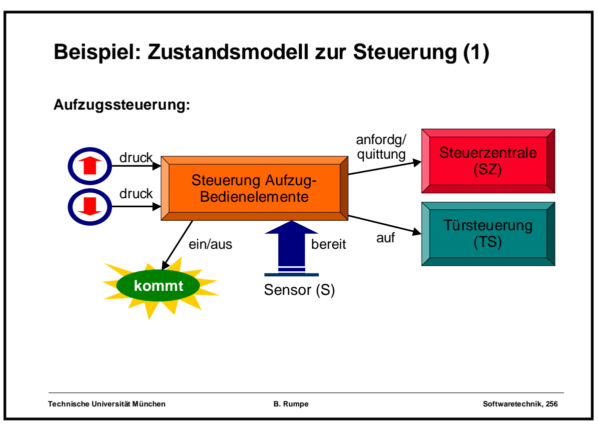
- Jedes Zustandsdiagramm sollte einen eindeutigen Startzustand haben. Der Startzustand ist ein "Pseudo-Zustand".
- **Notation:**

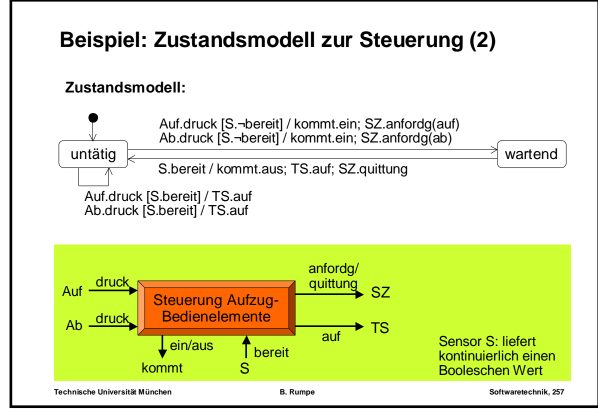
- Ein Zustandsdiagramm kann einen oder mehrere Endzustände haben. Die Angabe mindestens eines Endzustands ist wünschenswert.
- **Notation:** *("bull's eye")*

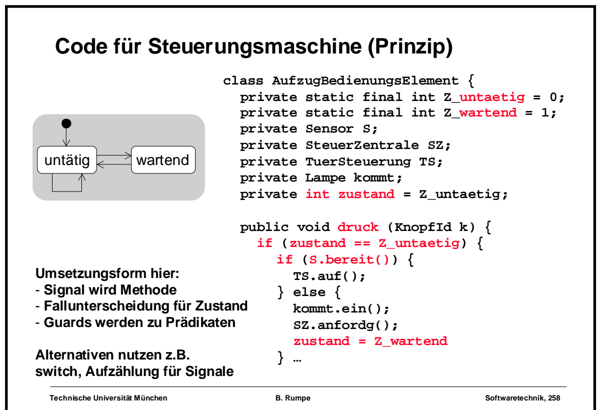

Bedingungen
A
Ereignis [Bedingung]
B
Definition Eine Bedingung (guard) ist eine Boolesche Bedingung, die zusätzlich bei Auftreten des E-Ereignisses erfüllt sein muss, damit der beschriebene Übergang eintritt.
Notation: Eine Bedingung wird in der Analysephase meist noch textuell beschrieben. In formalerer Beschreibung (v.a. im Entwurf) kann eine Bedingung folgende Informationen verwenden:
– Parameterwerte des Ereignisses
– Attributwerte und Assoziationsinstanzen (Links) der Objekte
– ggf. Navigation über Links zu anderen Objekten
Beispiele:
konfliktfrei
raumFestlegen()
[kein Raumproblem]
fixiert
Technische Universität München
B. Rumpe
Softwaretechnik, 253


Aktionen
A
Ereignis [Bedingung] / Aktion
B
Definition Eine Aktion ist die Beschreibung einer ausführbaren Anweisung, wobei angenommen wird, dass die Dauer der Ausführung vernachlässigbar ist. Aktionen sind nicht unterbrechbar.
Eine Aktion kann auch eine Folge von Einzelaktionen sein.
Typische Arten von Aktionen:
– Lokale Änderung eines Attributwerts
– Versenden einer Nachricht an ein anderes Objekt (bzw. eine Klasse)
– Erzeugen oder Löschen eines Objekts
– Rückgabe eines Ergebniswertes für eine früher empfangene Nachricht
Eine Aktion wird in der Analysephase meist textuell beschrieben. In formalerer Beschreibung kann als „Aktionssprache" auch die Zielsprache (z.B. Java) verwendet werden.
Technische Universität München
B. Rumpe
Softwaretechnik, 254


Verwendung von UML-Zustandsmodellen
zur Steuerung:
– Für steuernde Systeme, eingebettete Systeme etc.
– Ereignisse sind Signale der Umgebung oder anderer Systemteile
– Reaktion in gegebenem Zustand auf ein bestimmtes Signal:
» neuer Zustand
» ausgelöste Aktion (wie im Zustandsmodell spezifiziert)
– Ähnlich zum Konzept von Mealy-Automaten
– Zustandsmodelle definieren die Reaktion auf mögliche Ereignisse
als Protokolle (oder Objekt-Lebenszyklen):
– Für Informationssysteme, Datenbankanwendungen etc.
– Ereignisse sind Operationsaufrufe
– Reaktion in gegebenem Zustand auf bestimmten Aufruf:
» durch Methodenrumpf gegeben (komplex)
» Keine Aktionen im Zustandsmodell!
– Zustandsmodelle definieren zulässige Reihenfolgen von Aufrufen
Technische Universität München
B. Rumpe
Softwaretechnik, 255

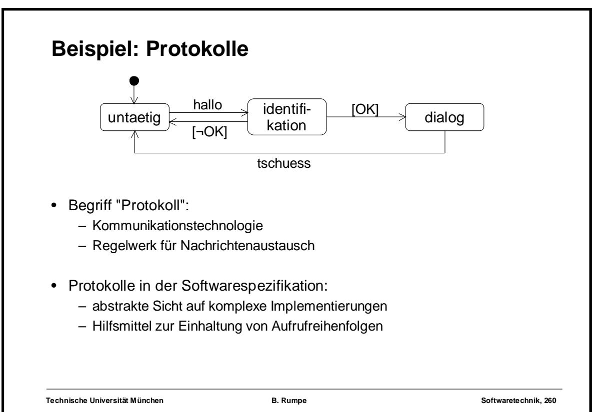

Beispiel: Zustandsmodell zur Steuerung (1)
Aufzugssteuerung:
druck
druck
Steuerung Aufzug-Bedienelemente
anfordg/
quittung
Steuerzentrale (SZ)
Türsteuerung (TS)
auf
ein/aus
bereit
kommt
Sensor (S)
Technische Universität München
B. Rumpe
Softwaretechnik, 256

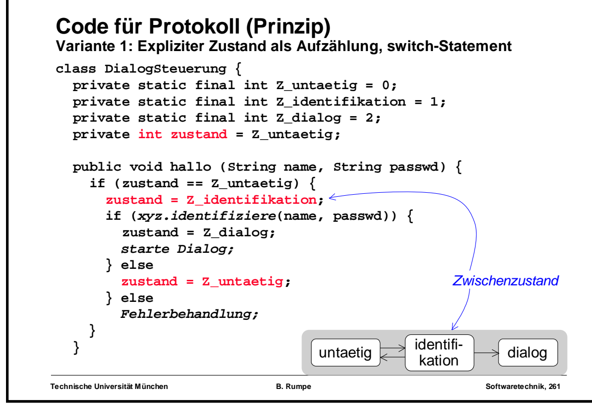

Beispiel: Zustandsmodell zur Steuerung (2)
Zustandsmodell:
Auf.druck [S.¬bereit] / kommt.ein; SZ.anfordg(auf)
Ab.druck [S.¬bereit] / kommt.ein; SZ.anfordg(ab)
untätig
wartend
S.bereit / kommt.aus; TS.auf; SZ.quittung
Auf.druck [S.bereit] / TS.auf
Ab.druck [S.bereit] / TS.auf
Auf
Ab
druck
druck
Steuerung Aufzug-Bedienelemente
anfordg/
quittung
SZ
TS
auf
ein/aus
kommt
bereit
S
Sensor S: liefert kontinuierlich einen Booleschen Wert
Technische Universität München
B. Rumpe
Softwaretechnik, 257


Code für Steuerungsmaschine (Prinzip)
untätig
wartend
class AufzugBedienungsElement {
private static final int Z_untaetig = 0;
private static final int Z_wartend = 1;
private Sensor S;
private SteuerZentrale SZ;
private TuerSteuerung TS;
private Lampe kommt;
private int zustand = Z_untaetig;
public void druck (KnopfId k) {
if (zustand == Z_untaetig) {
if (S.bereit()) {
TS.auf();
} else {
kommt.ein();
SZ.anfordg();
zustand = Z_wartend
} …
Umsetzungsform hier:
- Signal wird Methode
- Fallunterscheidung für Zustand
- Guards werden zu Prädikaten
Alternativen nutzen z.B. switch, Aufzählung für Signale
Technische Universität München
B. Rumpe
Softwaretechnik, 258

Codegenerierung für Steuerungs-Maschinen

- Vollständige Codegenerierung aus Zustandsmodell möglich
  - Voraussetzung: Präzise Spezifikation von Ereignissen, Bedingungen, Aktionen
  - Grundstruktur des Codes: Fallunterscheidungskaskaden
- Praktische Aspekte:
  - Reaktives System, d.h. Bereitschaft zur Reaktion auf Ereignisse nötig
  - Realzeit-Bedingungen: Zeit zur Ausführung von Aktionen begrenzt
    - » Realzeit-Betriebssysteme
  - Kommunikation: Statt direktem Methodenaufruf Erzeugung einer Nachricht zur Weitergabe an Empfänger
- Für diese Vorlesung:
  - nur Modellierungsaspekte relevant





Code für Protokoll (Prinzip)

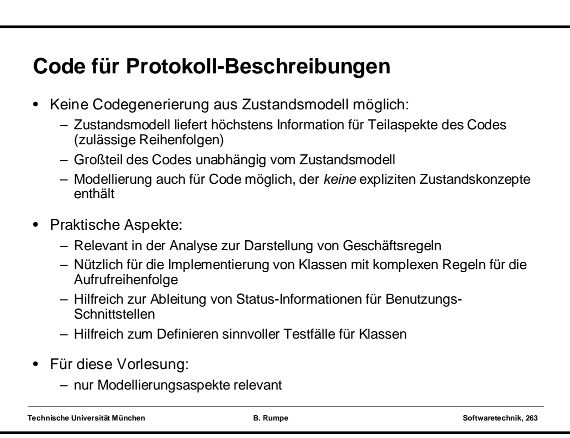
Variante 2: Impliziter Zustand durch Attribute

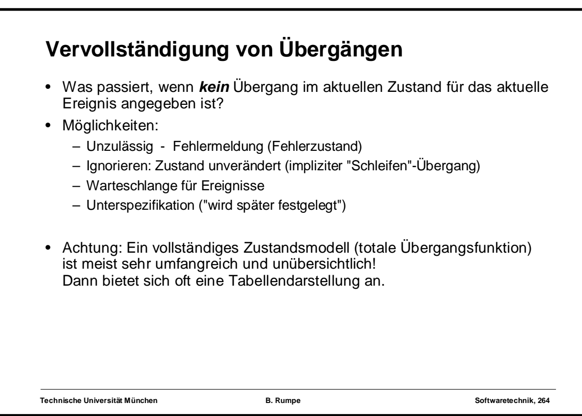
```
class DialogSteuerung {
  private boolean frei = true;
  private String user = "";

  public void hallo (String name, String passwd) {
    if (frei) {
      user = name;
      if (xyz.identifiziere(name, passwd)) {
        frei = false;
        starte Dialog;
      } else {
      frei = true;
      user = "";
    }
    else Fehlermeldung;
  }...
```

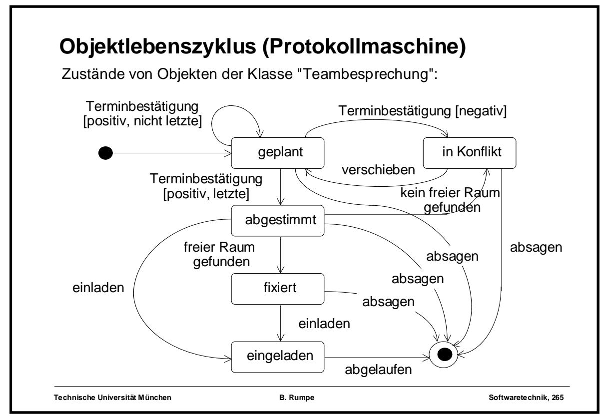
Zustand = Funktion mehrerer Attribute

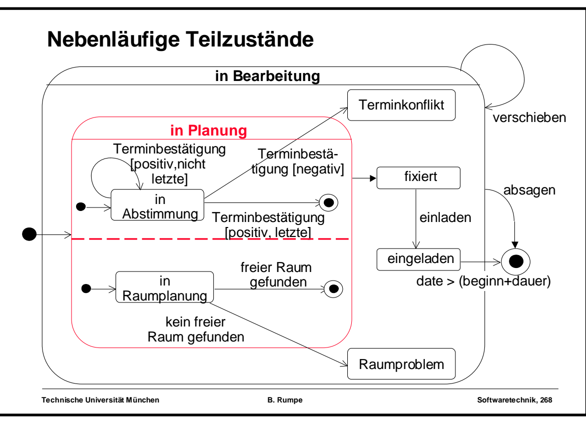
Zustand	frei	user
untaetig	true	""
identifikation	true	≠ ""
dialog	false	≠ ""

untaetig — identifi-kation — dialog





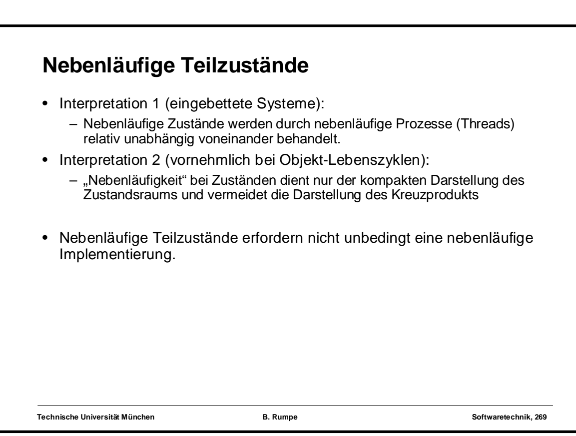




Ober- und Unterzustände

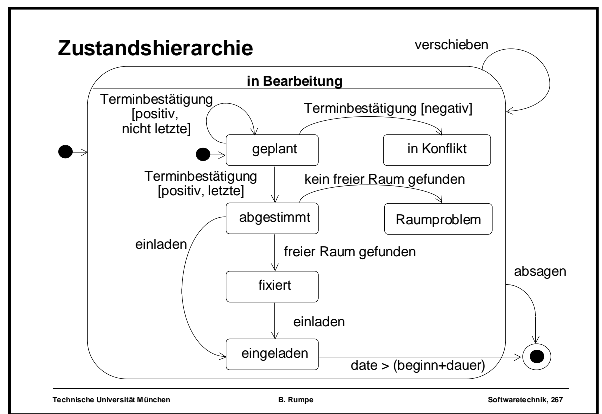
- Um eine ganze Gruppe von Zuständen einheitlich zu behandeln, können ***Oberzustände*** eingeführt werden.
  - Ein Objekt in einem Zustand ist gleichzeitig in allen Oberzuständen
  - Ein Zustand in den Oberzustand ist ein Übergang in den Startzustand des enthaltenen Zustandsdiagramms.
  - Transitionen können Zustandshierarchien kreuzen
  - „Äußere" Transitionen wirken für jeden Unterzustand

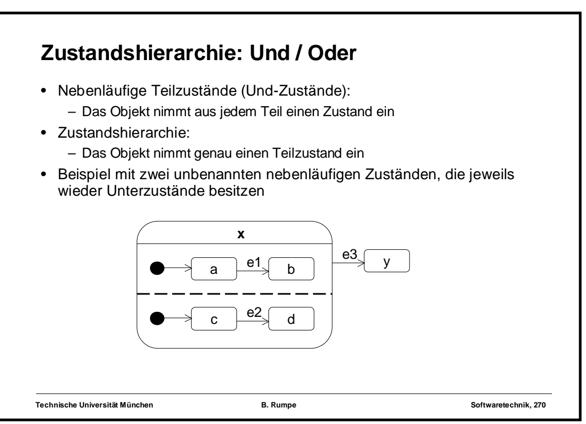
ab
a e1 b
e2 e4
e3 c
...

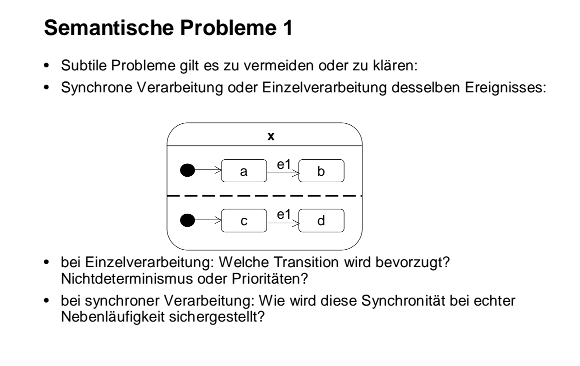


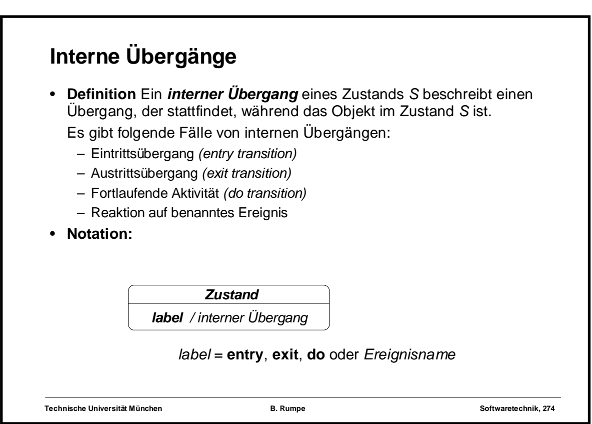
Nebenläufige Teilzustände

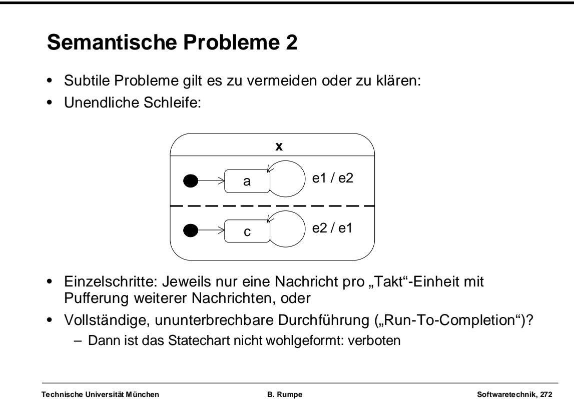
- Interpretation 1 (eingebettete Systeme):
  - Nebenläufige Zustände werden durch nebenläufige Prozesse (Threads) relativ unabhängig voneinander behandelt.
- Interpretation 2 (vornehmlich bei Objekt-Lebenszyklen):
  - „Nebenläufigkeit" bei Zuständen dient nur der kompakten Darstellung des Zustandsraums und vermeidet die Darstellung des Kreuzprodukts
- Nebenläufige Teilzustände erfordern nicht unbedingt eine nebenläufige Implementierung.

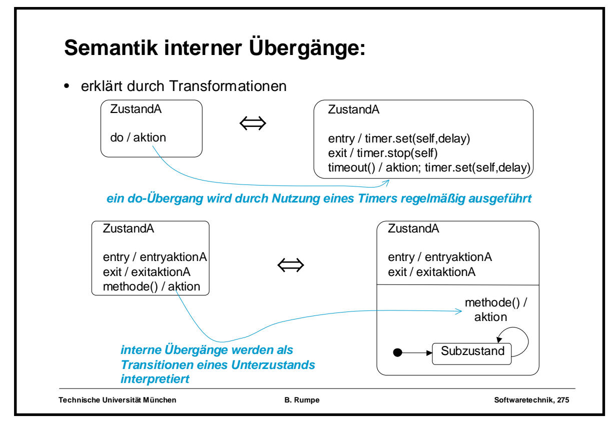


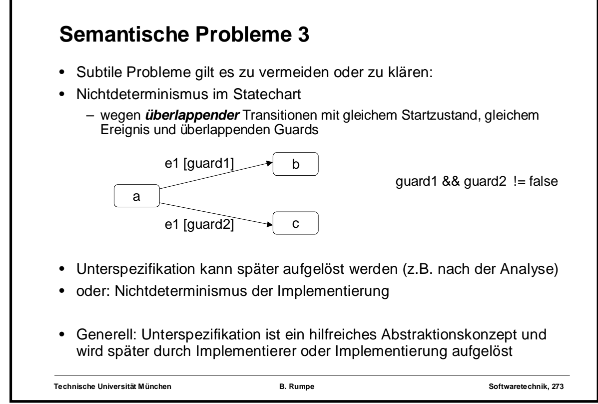


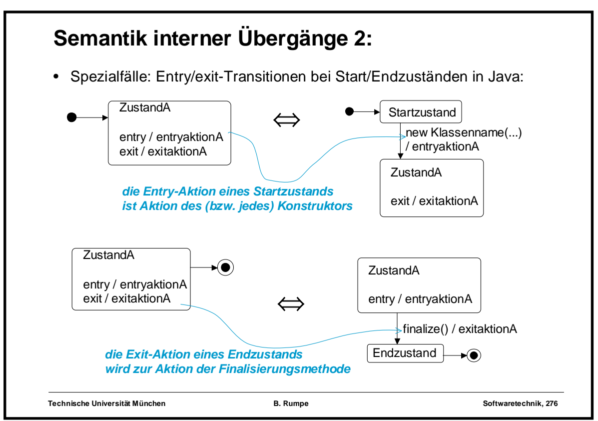

Semantische Probleme 1
Subtile Probleme gilt es zu vermeiden oder zu klären:
Synchrone Verarbeitung oder Einzelverarbeitung desselben Ereignisses:
x
a
e1
b
c
e1
d
bei Einzelverarbeitung: Welche Transition wird bevorzugt? Nichtdeterminismus oder Prioritäten?
bei synchroner Verarbeitung: Wie wird diese Synchronität bei echter Nebenläufigkeit sichergestellt?
Technische Universität München
B. Rumpe
Softwaretechnik, 271

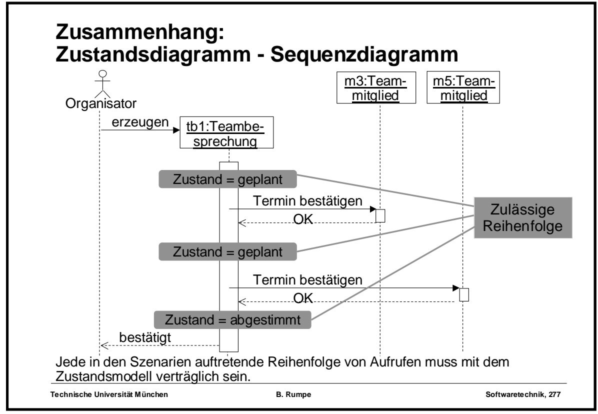

Interne Übergänge
Definition Ein interner Übergang eines Zustands S beschreibt einen Übergang, der stattfindet, während das Objekt im Zustand S ist.
Es gibt folgende Fälle von internen Übergängen:
– Eintrittsübergang (entry transition)
– Austrittsübergang (exit transition)
– Fortlaufende Aktivität (do transition)
– Reaktion auf benanntes Ereignis
Notation:
Zustand
label / interner Übergang
label = entry, exit, do oder Ereignisname
Technische Universität München
B. Rumpe
Softwaretechnik, 274


Semantische Probleme 2
Subtile Probleme gilt es zu vermeiden oder zu klären:
Unendliche Schleife:
x
a
e1 / e2
c
e2 / e1
Einzelschritte: Jeweils nur eine Nachricht pro „Takt“-Einheit mit Pufferung weiterer Nachrichten, oder
Vollständige, ununterbrechbare Durchführung („Run-To-Completion“)?
– Dann ist das Statechart nicht wohlgeformt: verboten
Technische Universität München
B. Rumpe
Softwaretechnik, 272


Semantik interner Übergänge:
erklärt durch Transformationen
ZustandA
do / aktion
⇔
ZustandA
entry / timer.set(self,delay)
exit / timer.stop(self)
timeout() / aktion; timer.set(self,delay)
ein do-Übergang wird durch Nutzung eines Timers regelmäßig ausgeführt
ZustandA
entry / entryaktionA
exit / exitaktionA
methode() / aktion
⇔
ZustandA
entry / entryaktionA
exit / exitaktionA
methode() / aktion
Subzustand
interne Übergänge werden als Transitionen eines Unterzustands interpretiert
Technische Universität München
B. Rumpe
Softwaretechnik, 275


Semantische Probleme 3
Subtile Probleme gilt es zu vermeiden oder zu klären:
Nichtdeterminismus im Statechart
– wegen überlappender Transitionen mit gleichem Startzustand, gleichem Ereignis und überlappenden Guards
e1 [guard1]
b
a
e1 [guard2]
c
guard1 && guard2 != false
Unterspezifikation kann später aufgelöst werden (z.B. nach der Analyse)
oder: Nichtdeterminismus der Implementierung
Generell: Unterspezifikation ist ein hilfreiches Abstraktionskonzept und wird später durch Implementierer oder Implementierung aufgelöst
Technische Universität München
B. Rumpe
Softwaretechnik, 273

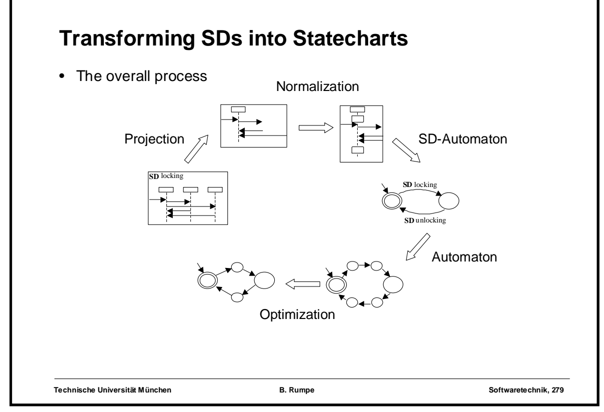

Semantik interner Übergänge 2:
Spezialfälle: Entry/exit-Transitionen bei Start/Endzuständen in Java:
ZustandA
entry / entryaktionA
exit / exitaktionA
⇔
Startzustand
new Klassenname(...)
/ entryaktionA
ZustandA
exit / exitaktionA
die Entry-Aktion eines Startzustands ist Aktion des (bzw. jedes) Konstruktors
ZustandA
entry / entryaktionA
exit / exitaktionA
⇔
ZustandA
entry / entryaktionA
finalize() / exitaktionA
Endzustand
die Exit-Aktion eines Endzustands wird zur Aktion der Finalisierungsmethode
Technische Universität München
B. Rumpe
Softwaretechnik, 276

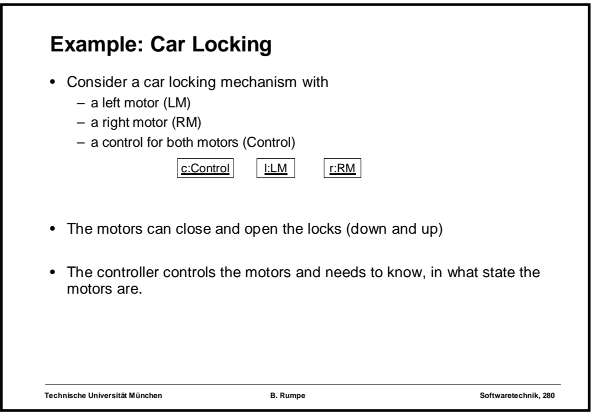


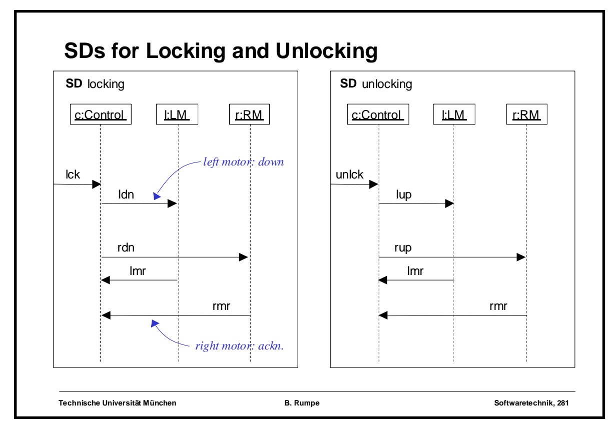
Zusammenhang: Zustandsdiagramm – Sequenzdiagramm

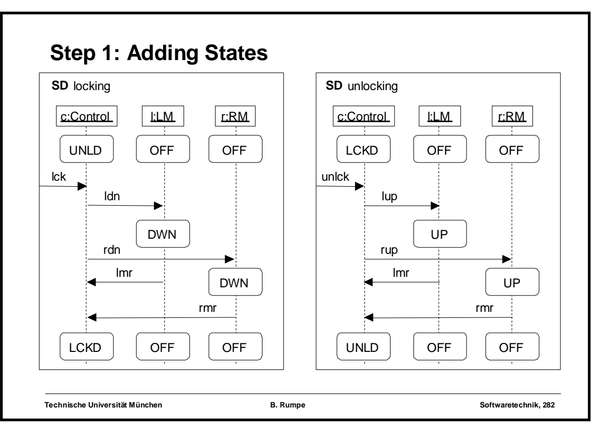
- Jede in den Szenarien auftretende Reihenfolge von Aufrufen muss mit dem Zustandsmodell verträglich sein
- Sequenzdiagramme besitzen keine Zustandsinformation, beschreiben aber Abläufe mehrerer Objekte gleichzeitig
- Nachfolgend eine Beispielanwendung für ein Verfahren, das in fünf Schritten aus mehreren Sequenzdiagrammen ein Statechart entwickelt
- Das Verfahren stammt aus der Dissertation von Ingolf Krüger und wurde in ähnlicher Form auch am Lehrstuhl Broy patentiert
- (Englische Original-Folien eines Tutorials)

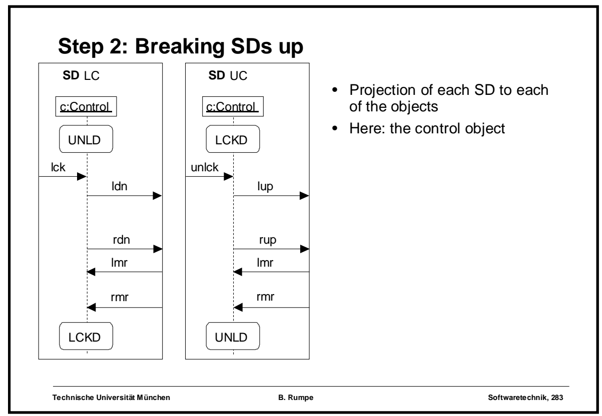
Technische Universität München B. Rumpe Softwaretechnik, 278

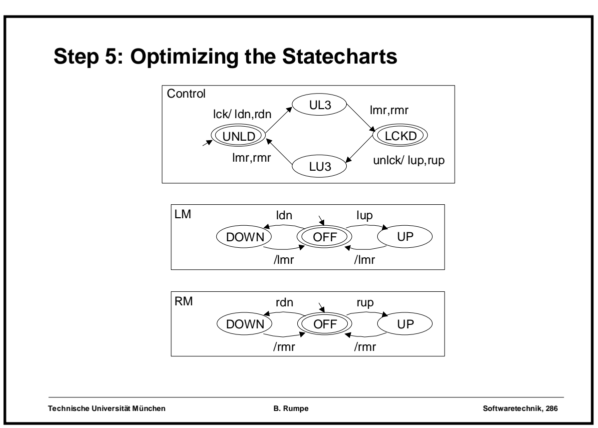


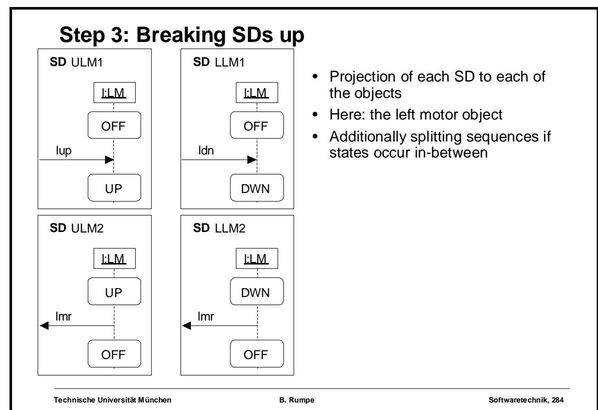


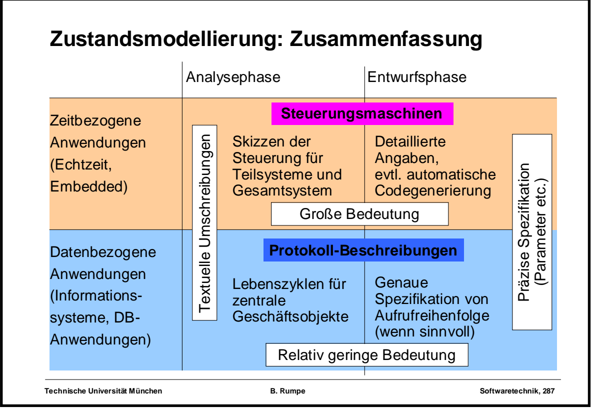


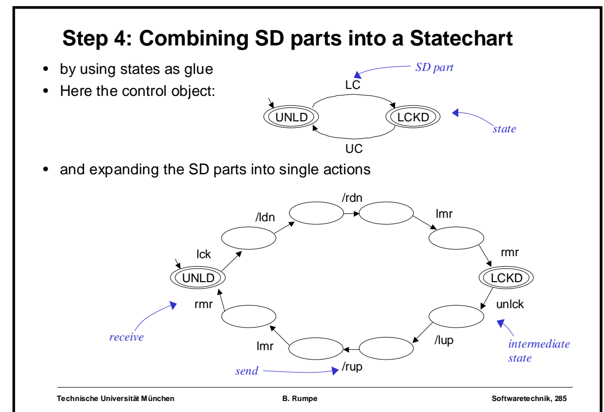


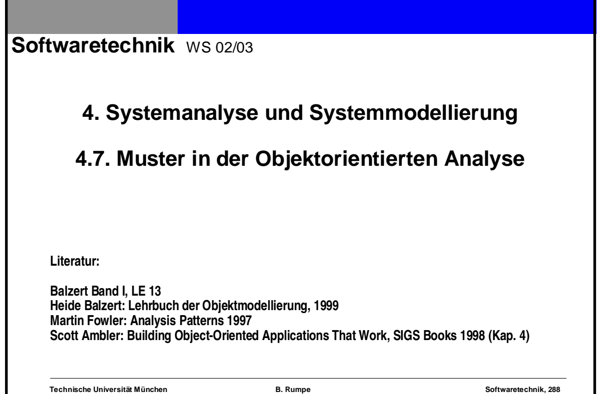

Step 2: Breaking SDs up
SD LC
c:Control
UNLD
lck
ldn
rdn
lmr
rmr
LCKD
SD UC
c:Control
LCKD
unlck
lup
rup
lmr
rmr
UNLD
Projection of each SD to each of the objects
Here: the control object
Technische Universität München
B. Rumpe
Softwaretechnik, 283


Step 5: Optimizing the Statecharts
Control
UL3
lck/ ldn,rdn
lmr,rmr
UNLD
LCKD
lmr,rmr
LU3
unlck/ lup,rup
LM
ldn
lup
DOWN
OFF
UP
/lmr
/lmr
RM
rdn
rup
DOWN
OFF
UP
/rmr
/rmr
Technische Universität München
B. Rumpe
Softwaretechnik, 286


Step 3: Breaking SDs up
SD ULM1
l:LM
OFF
lup
UP
SD LLM1
l:LM
OFF
ldn
DWN
SD ULM2
l:LM
UP
lmr
OFF
SD LLM2
l:LM
DWN
lmr
OFF
Projection of each SD to each of the objects
Here: the left motor object
Additionally splitting sequences if states occur in-between
Technische Universität München
B. Rumpe
Softwaretechnik, 284


Zustandsmodellierung: Zusammenfassung
Analysephase
Entwurfsphase
Zeitbezogene Anwendungen (Echtzeit, Embedded)
Steuerungsmaschinen
Skizzen der Steuerung für Teilsysteme und Gesamtsystem
Detaillierte Angaben, evtl. automatische Codegenerierung
Große Bedeutung
Datenbezogene Anwendungen (Informations-systeme, DB-Anwendungen)
Protokoll-Beschreibungen
Lebenszyklen für zentrale Geschäftsobjekte
Genaue Spezifikation von Aufrufreihenfolge (wenn sinnvoll)
Relativ geringe Bedeutung
Textuelle Umschreibungen
Präzise Spezifikation (Parameter etc.)
Technische Universität München
B. Rumpe
Softwaretechnik, 287


Step 4: Combining SD parts into a Statechart
by using states as glue
Here the control object:
SD part
LC
UNLD
LCKD
state
UC
and expanding the SD parts into single actions
/rdn
/ldn
lmr
lck
rmr
UNLD
LCKD
rmr
unlck
receive
lmr
/lup
intermediate state
send
/rup
Technische Universität München
B. Rumpe
Softwaretechnik, 285


Softwaretechnik WS 02/03
4. Systemanalyse und Systemmodellierung
4.7. Muster in der Objektorientierten Analyse
Literatur:
Balzert Band I, LE 13
Heide Balzert: Lehrbuch der Objektmodellierung, 1999
Martin Fowler: Analysis Patterns 1997
Scott Ambler: Building Object-Oriented Applications That Work, SIGS Books 1998 (Kap. 4)
Technische Universität München
B. Rumpe
Softwaretechnik, 288

Wiederverwendung mit Musterkatalogen

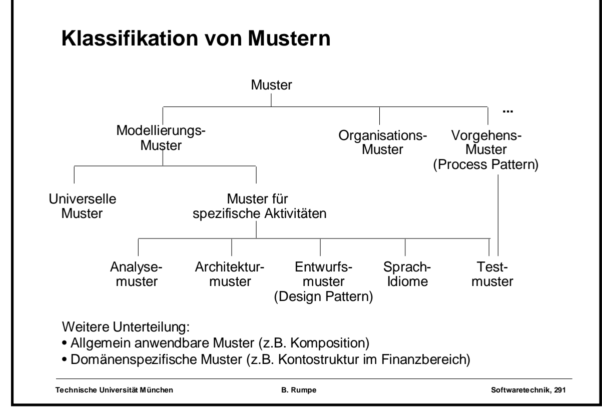
- Ein ***Muster*** ist eine schematische Lösung für eine Klasse verwandter Probleme. ***Muster*** beschreiben Erfahrungswissen.
- Darstellung eines Musters
  - Name, evtl. Synonyme
  - Problem
    - » Motivation, Anwendungsbereich
  - Lösung(en)
    - » Struktur (Klassendiagramm)
    - » Bestandteile (schematische Klassen- und Objektnamen)
    - » Beschreibung, evtl. Abläufe, z.B. Sequenzdiagramm
  - Diskussion
    - » Vor- und Nachteile, Abhängigkeiten, Einschränkungen
  - Beispielanwendungen
  - Verwandte Muster (Ähnlichkeiten)

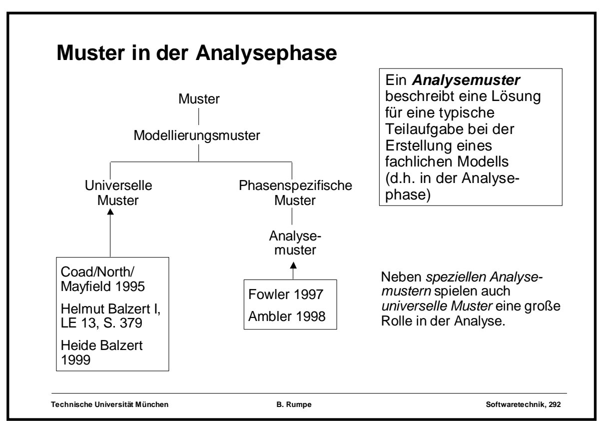
Häufige Mißverständnisse zu Mustern

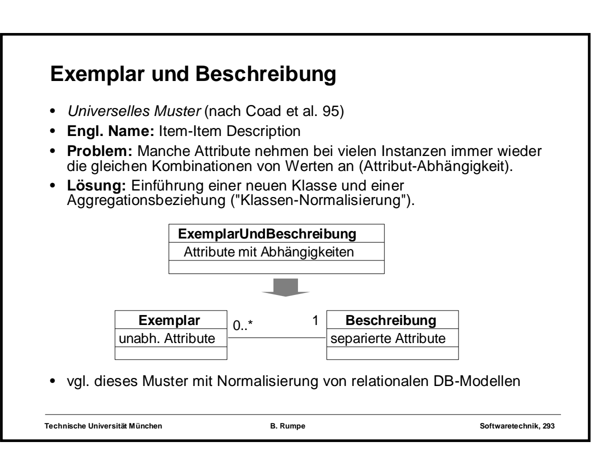
Diese Aussagen sind falsch:

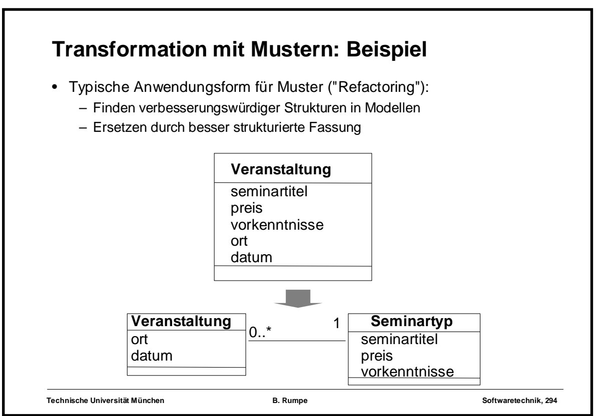
- „Ein Muster ist eine Lösung für ein Problem in einem gegebenen Kontext.“
  - Stattdessen ist der Schema-Charakter, „erzieherischer Wert“, Name wichtig
- „Muster sind nur Jargon, Regeln, Programmiertricks, ...“
- „Alle Muster sehen gleich aus.“
- „Muster brauchen Werkzeugunterstützung.“
- „Die Anwendung von Analyse- bzw. Entwurfsmustern erzeugt eine komplette Systemanalyse bzw. Systemarchitektur.“
- „Muster können nur mit objektorientierten Methoden eingesetzt werden.“

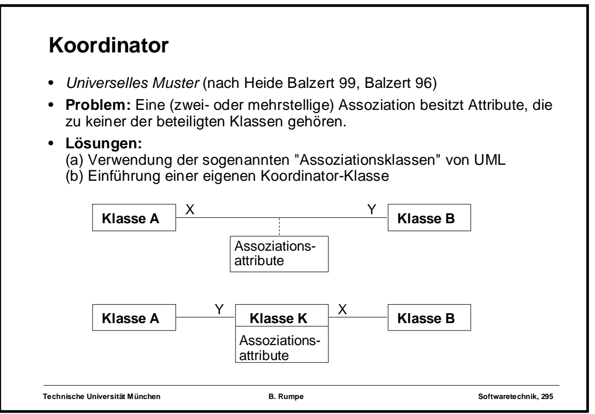
- Muster lassen sich nur verstehen, wenn sie praktisch angewendet wurden oder die Anwendbarkeit an praktischen Beispielen diskutiert wird!

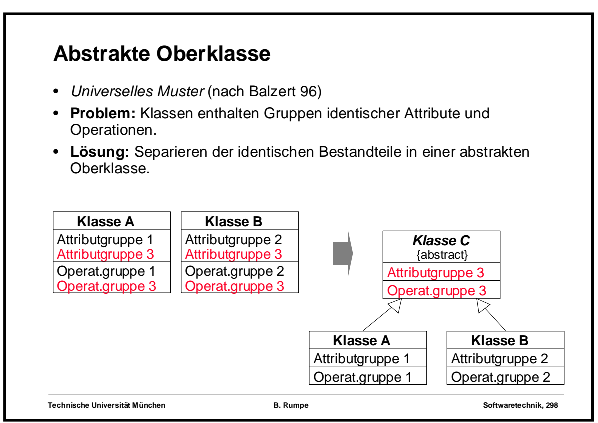


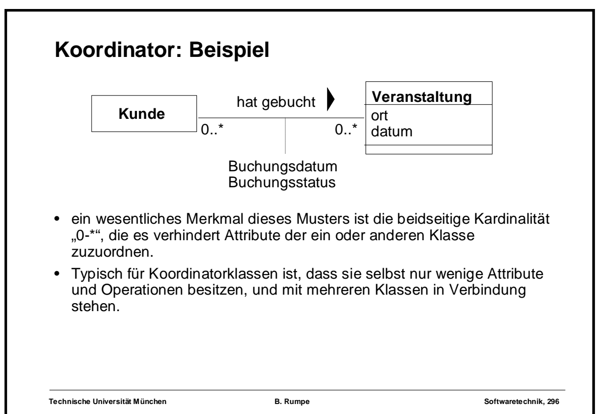


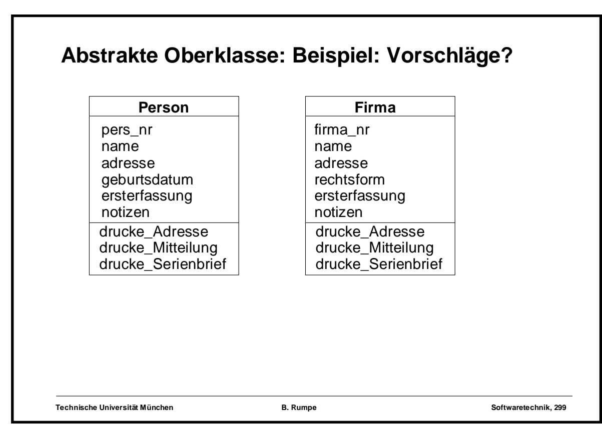


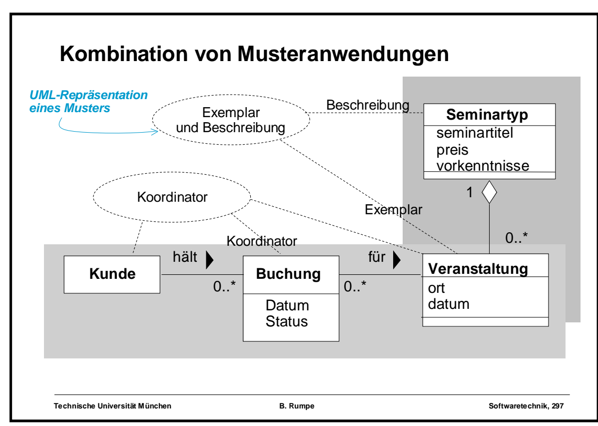


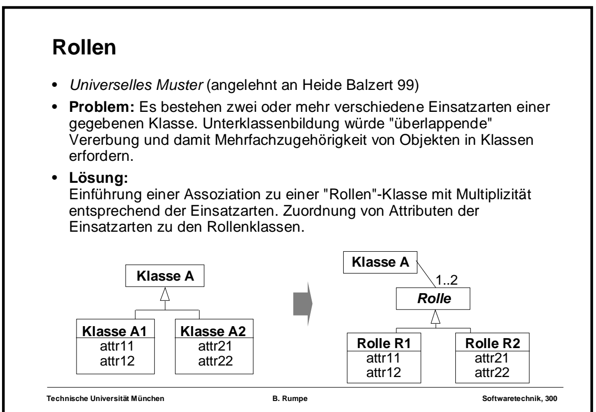

Koordinator
Universelles Muster (nach Heide Balzert 99, Balzert 96)
Problem: Eine (zwei- oder mehrstellige) Assoziation besitzt Attribute, die zu keiner der beteiligten Klassen gehören.
Lösungen:
(a) Verwendung der sogenannten "Assoziationsklassen" von UML
(b) Einführung einer eigenen Koordinator-Klasse
Klasse A
X
Y
Klasse B
Assoziations-attribute
Klasse A
Y
Klasse K
X
Klasse B
Assoziations-attribute
Technische Universität München
B. Rumpe
Softwaretechnik, 295

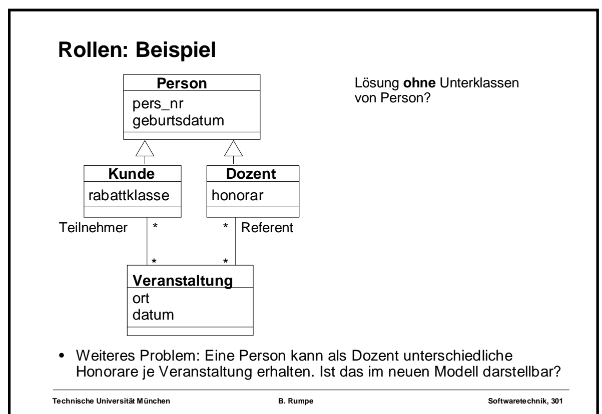

Abstrakte Oberklasse
Universelles Muster (nach Balzert 96)
Problem: Klassen enthalten Gruppen identischer Attribute und Operationen.
Lösung: Separieren der identischen Bestandteile in einer abstrakten Oberklasse.
Klasse A
Attributgruppe 1
Attributgruppe 3
Operat.gruppe 1
Operat.gruppe 3
Klasse B
Attributgruppe 2
Attributgruppe 3
Operat.gruppe 2
Operat.gruppe 3
Klasse C
{abstract}
Attributgruppe 3
Operat.gruppe 3
Klasse A
Attributgruppe 1
Operat.gruppe 1
Klasse B
Attributgruppe 2
Operat.gruppe 2
Technische Universität München
B. Rumpe
Softwaretechnik, 298

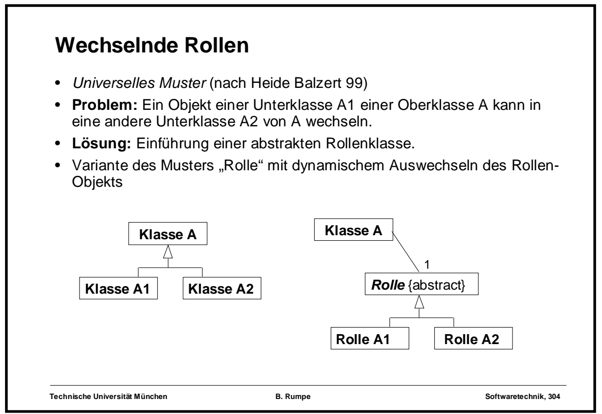

Koordinator: Beispiel
Kunde
hat gebucht
0..*
0..*
Veranstaltung
ort
datum
Buchungsdatum
Buchungsstatus
ein wesentliches Merkmal dieses Musters ist die beidseitige Kardinalität „0-*“, die es verhindert Attribute der ein oder anderen Klasse zuzuordnen.
Typisch für Koordinatorklassen ist, dass sie selbst nur wenige Attribute und Operationen besitzen, und mit mehreren Klassen in Verbindung stehen.
Technische Universität München
B. Rumpe
Softwaretechnik, 296

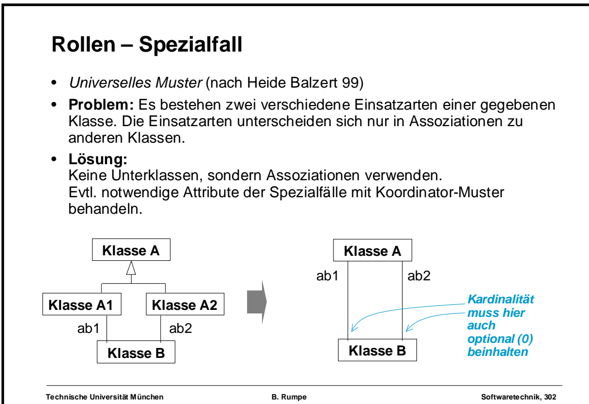

Abstrakte Oberklasse: Beispiel: Vorschläge?
Person
pers_nr
name
adresse
geburtsdatum
ersterfassung
notizen
drucke_Adresse
drucke_Mitteilung
drucke_Serienbrief
Firma
firma_nr
name
adresse
rechtsform
ersterfassung
notizen
drucke_Adresse
drucke_Mitteilung
drucke_Serienbrief
Technische Universität München
B. Rumpe
Softwaretechnik, 299

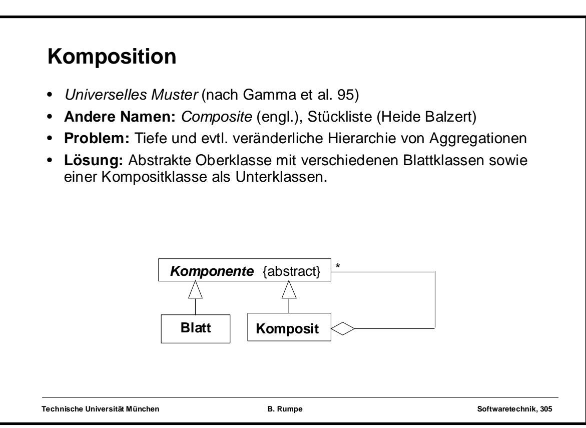

Kombination von Musteranwendungen
UML-Repräsentation eines Musters
Exemplar und Beschreibung
Beschreibung
Seminartyp
seminartitel
preis
vorkenntnisse
Koordinator
Exemplar
1
0..*
Koordinator
Kunde
hält
0..*
Buchung
Datum
Status
für
0..*
Veranstaltung
ort
datum
Technische Universität München
B. Rumpe
Softwaretechnik, 297

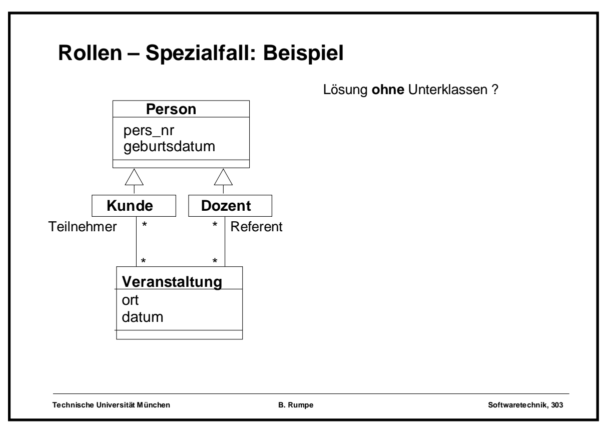

Rollen
Universelles Muster (angelehnt an Heide Balzert 99)
Problem: Es bestehen zwei oder mehr verschiedene Einsatzarten einer gegebenen Klasse. Unterklassenbildung würde "überlappende" Vererbung und damit Mehrfachzugehörigkeit von Objekten in Klassen erfordern.
Lösung:
Einführung einer Assoziation zu einer "Rollen"-Klasse mit Multiplizität entsprechend der Einsatzarten. Zuordnung von Attributen der Einsatzarten zu den Rollenklassen.
Klasse A
Klasse A1
attr11
attr12
Klasse A2
attr21
attr22
Klasse A
1..2
Rolle
Rolle R1
attr11
attr12
Rolle R2
attr21
attr22
Technische Universität München
B. Rumpe
Softwaretechnik, 300

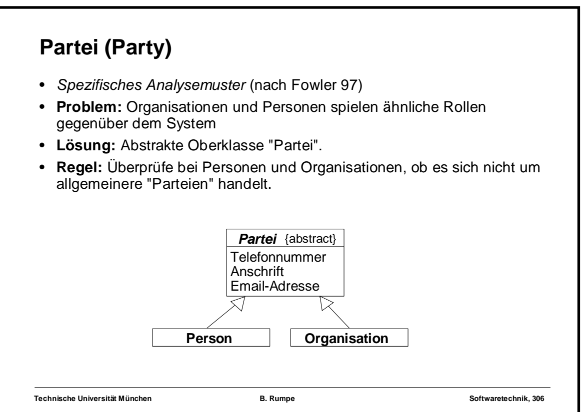

Rollen: Beispiel
Person
pers_nr
geburtsdatum
Lösung ohne Unterklassen von Person?
Kunde
rabattklasse
Dozent
honorar
Teilnehmer
Referent
Veranstaltung
ort
datum
Weiteres Problem: Eine Person kann als Dozent unterschiedliche Honorare je Veranstaltung erhalten. Ist das im neuen Modell darstellbar?
Technische Universität München
B. Rumpe
Softwaretechnik, 301

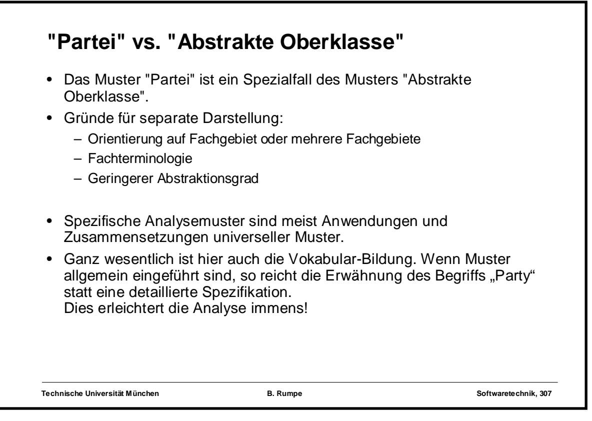

Wechselnde Rollen
Universelles Muster (nach Heide Balzert 99)
Problem: Ein Objekt einer Unterklasse A1 einer Oberklasse A kann in eine andere Unterklasse A2 von A wechseln.
Lösung: Einführung einer abstrakten Rollenklasse.
Variante des Musters „Rolle“ mit dynamischem Auswechseln des Rollen-Objekts
Klasse A
Klasse A1
Klasse A2
Klasse A
1
Rolle {abstract}
Rolle A1
Rolle A2
Technische Universität München
B. Rumpe
Softwaretechnik, 304

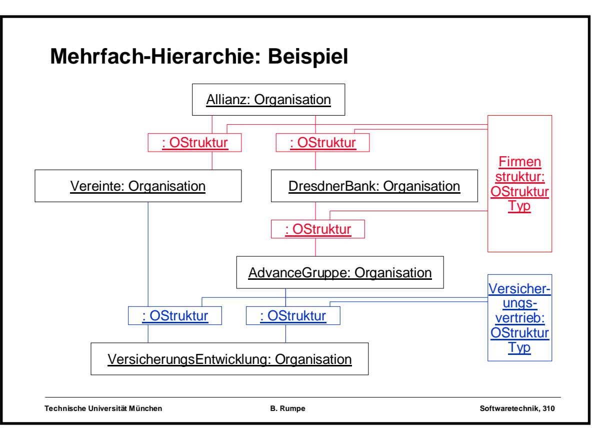

Rollen – Spezialfall
Universelles Muster (nach Heide Balzert 99)
Problem: Es bestehen zwei verschiedene Einsatzarten einer gegebenen Klasse. Die Einsatzarten unterscheiden sich nur in Assoziationen zu anderen Klassen.
Lösung:
Keine Unterklassen, sondern Assoziationen verwenden.
Evtl. notwendige Attribute der Spezialfälle mit Koordinator-Muster behandeln.
Klasse A
Klasse A1
Klasse A2
ab1
ab2
Klasse B
Klasse A
ab1
ab2
Klasse B
Kardinalität muss hier auch optional (0) beinhalten
Technische Universität München
B. Rumpe
Softwaretechnik, 302


Komposition
Universelles Muster (nach Gamma et al. 95)
Andere Namen: Composite (engl.), Stückliste (Heide Balzert)
Problem: Tiefe und evtl. veränderliche Hierarchie von Aggregationen
Lösung: Abstrakte Oberklasse mit verschiedenen Blattklassen sowie einer Kompositklasse als Unterklassen.
Komponente {abstract}
*
Blatt
Komposit
Technische Universität München
B. Rumpe
Softwaretechnik, 305


Rollen – Spezialfall: Beispiel
Lösung ohne Unterklassen ?
Person
pers_nr
geburtsdatum
Kunde
Dozent
Teilnehmer
Referent
Veranstaltung
ort
datum
Technische Universität München
B. Rumpe
Softwaretechnik, 303


Partei (Party)
Spezifisches Analysemuster (nach Fowler 97)
Problem: Organisationen und Personen spielen ähnliche Rollen gegenüber dem System
Lösung: Abstrakte Oberklasse "Partei".
Regel: Überprüfe bei Personen und Organisationen, ob es sich nicht um allgemeinere "Parteien" handelt.
Partei {abstract}
Telefonnummer
Anschrift
Email-Adresse
Person
Organisation
Technische Universität München
B. Rumpe
Softwaretechnik, 306





Organisations-Hierarchie

- *Spezifisches Analysemuster* (nach Fowler 97)
- **Engl. Name:** Organization Hierarchies
- **Problem:** Eine Organisation ist hierarchisch in Unterorganisationen gegliedert.
- **Lösung:** Flexible Ordnungsassoziation
- Spezialfall von "Komposition".

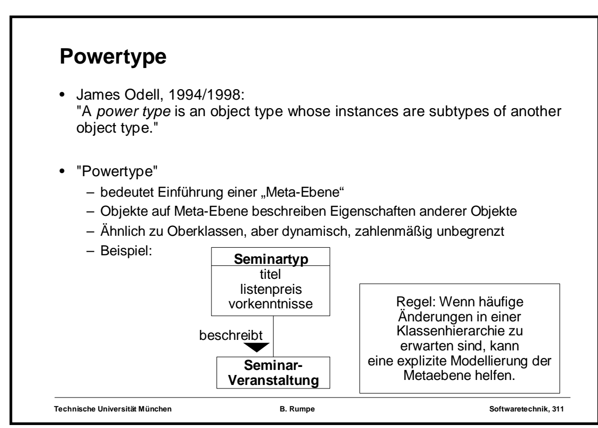
Organisation

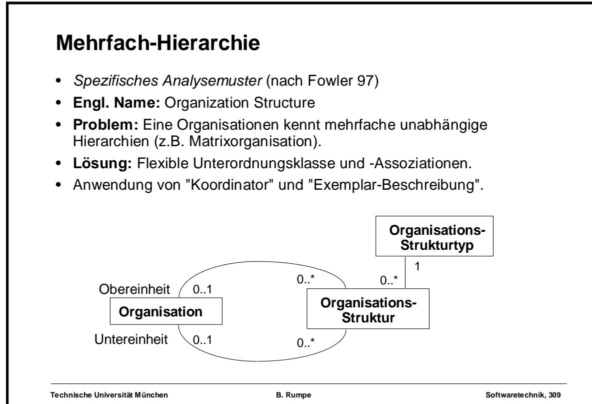
Untereinheit 0..*

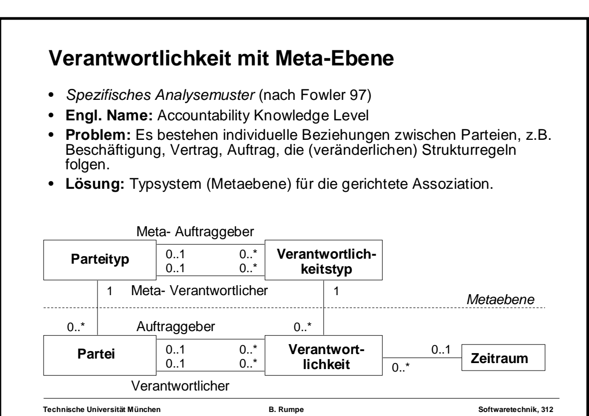
Obereinheit 0..1

{ Hierarchie }

Stereotyp (tag) der informell anzeigt, dass mit dieser Assoziation nur Baumstrukturen erlaubt sind







Verantwortlichkeit mit Meta-Ebene - Diskussion

- Die Einführung des "Verantwortlichkeitstyps" hat den Charakter eines ***Typsystems***, das die einzelnen Beziehungen zwischen Organisationen klassifiziert.
- Es ist zu beachten, dass das Klassendiagramm alleine noch nicht die wesentlichen Einschränkungen ausdrückt, die sicherstellen, dass auch nur "typkorrekte" Verantwortlichkeiten aufgebaut werden. Solche Einschränkungen lassen sich z.B. mit der *Object Constraint Language (OCL)* definieren.
- Beispiel: Die Metaebene legt fest, dass der Verantwortlichkeitstyp „Erziehungsberechtigter von“ nur zwischen Personen auftreten darf, der Erziehungsberechtigte 18 Jahre sein muss, etc.
- Die Einführung einer Meta-Ebene analog zu einem Typsystem ist wie der „Powertype“ auch ein universelles Muster.

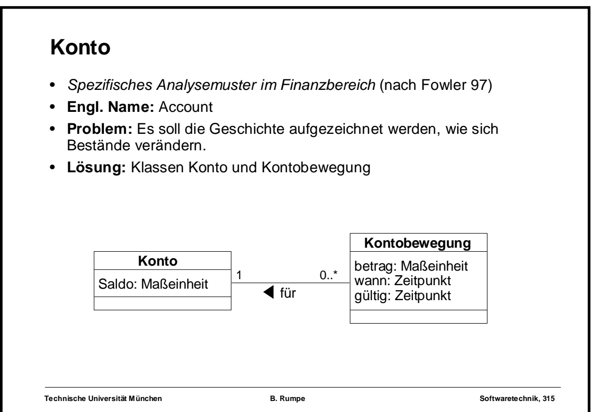
Ort

- *Spezifisches Analysemuster* (nach Ambler 98)
- **Engl. Name:** Place
- **Problem:** Ein Ort ist zu beschreiben.
- **Lösung:** Folgende Lösung enthält einiges an Detailinformation zu dem
- komplexen Thema und bietet sich zur Wiederverwendung/Zuschnitt an.

Zeitzone: Name, Abkürzung, Differenz zu UTC, Sommerzeitregel; Zeitdifferenz berechnen
gehört zu 1..* 0..*
liegt in 0..* 0..1
Ort: interner Name, Ortsbezeichnung, offizieller Name, Hemisphäre
feiert 1..* 0..*
Feiertag: Name, Datum, Berechnungsregel, Betroffene
betrifft 1..* 0..*
Steuer: Steuersatz, Regeln, Steuerart, Zahlungsmodus
Region
Land: Ländercode

Konto

- *Spezifisches Analysemuster im Finanzbereich* (nach Fowler 97)
- **Engl. Name:** Account
- **Problem:** Es soll die Geschichte aufgezeichnet werden, wie sich Bestände verändern.
- **Lösung:** Klassen Konto und Kontobewegung



Buchung

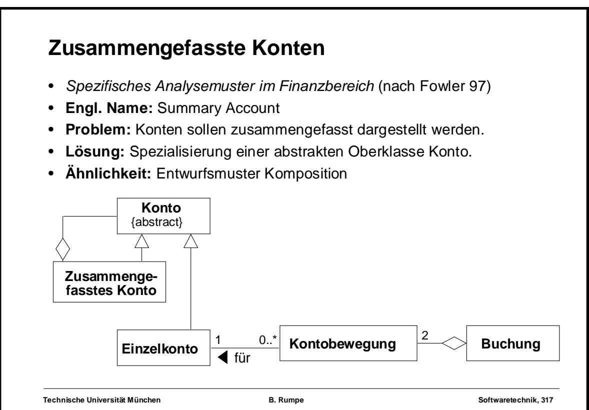
- *Spezifisches Analysemuster im Finanzbereich* (nach Fowler 97)
- **Engl. Name:** Transaction
- **Problem:** Es soll sichergestellt werden, dass bei einer Kontobewegung niemals Bestände verloren gehen.
- **Lösung:** Klasse Buchung mit Einschränkungen *(constraints)*

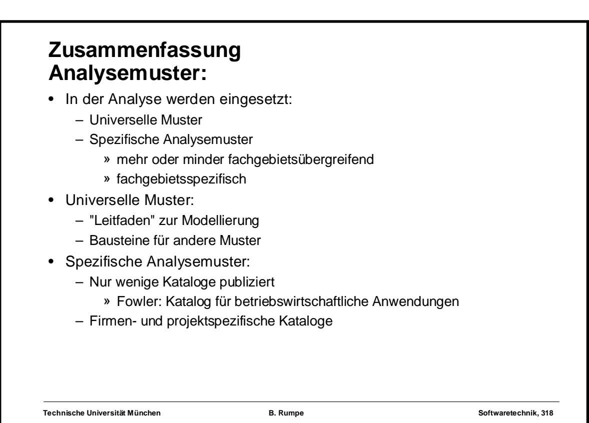
Konto 1 — für — 0..* Kontobewegung (betrag: Maßeinheit) 2 — Buchung

kontobewegung[1].betrag
+
kontobewegung[2].betrag = 0

Zusammengefasste Konten

- *Spezifisches Analysemuster im Finanzbereich* (nach Fowler 97)
- **Engl. Name:** Summary Account
- **Problem:** Konten sollen zusammengefasst dargestellt werden.
- **Lösung:** Spezialisierung einer abstrakten Oberklasse Konto.
- **Ähnlichkeit:** Entwurfsmuster Komposition





Zusammenfassung Analysemuster:

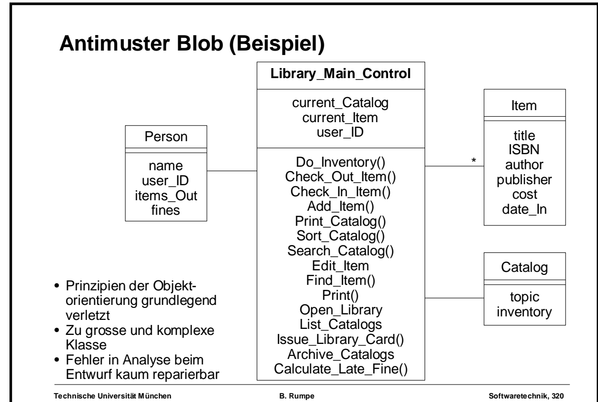
- In der Analyse werden eingesetzt:
  - Universelle Muster
  - Spezifische Analysemuster
    - » mehr oder minder fachgebietsübergreifend
    - » fachgebietsspezifisch
- Universelle Muster:
  - "Leitfaden" zur Modellierung
  - Bausteine für andere Muster
- Spezifische Analysemuster:
  - Nur wenige Kataloge publiziert
    - » Fowler: Katalog für betriebswirtschaftliche Anwendungen
  - Firmen- und projektspezifische Kataloge

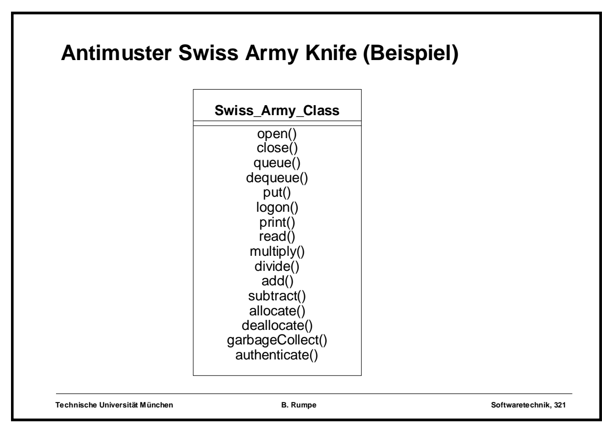
Muster und Antimuster

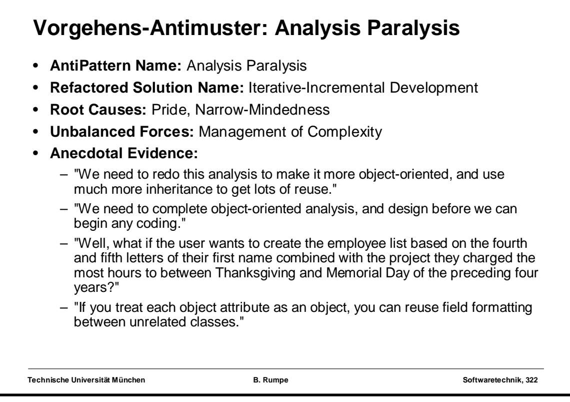
- Muster:
  - Beschreibung einer angestrebten "guten" Lösung für ein typisches Problem
- Anti-Muster:
  - Beschreibung einer existierenden "schlechten" Lösung für ein typisches Problem
  - Identifikation von Problemstellen

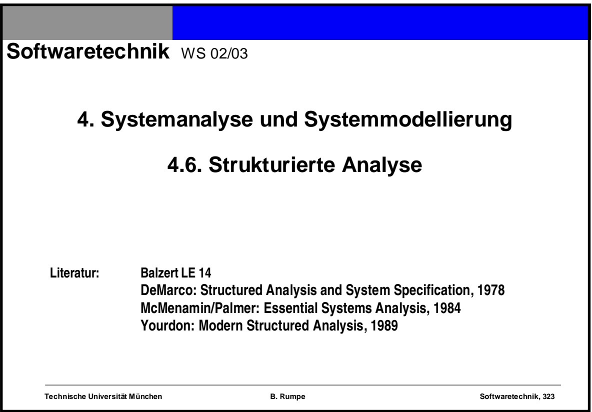
  Brown et al: Anti-Patterns, Wiley 1998

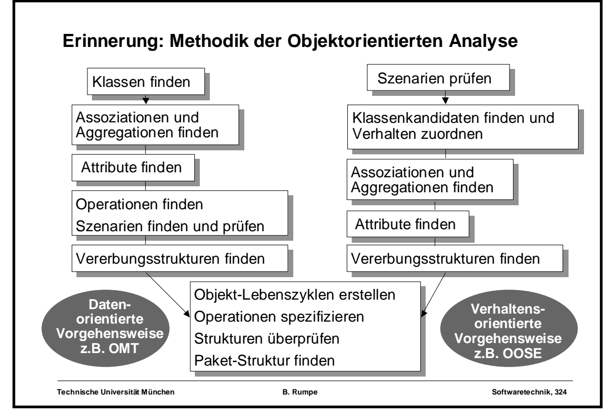
- Anti-Muster gibt es für:
  - Organisation
  - Vorgehen
  - Struktur
  - ...











Notationen in der Strukturierten Analyse (SA)

- Datenflussdiagramm (DFD)
  - Strukturierung des Systems
  - Funktionen und Daten und ihr Zusammenwirken
- Data Dictionary (DD)
  - Strukturierung der Daten
  - Detaillierte Datenbeschreibung
- Funktionsbaum
  - Strukturierung der Funktionen
- Mini-Spezifikationen
  - Detaillierte Funktionsbeschreibung

Acht Schritte zum SA-Modell

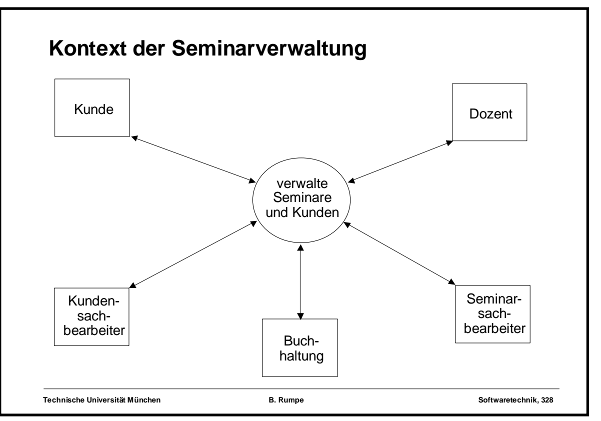
1. Schnittstellen und Ein/Ausgaben finden
2. Funktionen finden
3. Speicher finden
4. Datenflüsse finden
5. Data Dictionary erstellen
6. Konsolidieren des Modells
7. Iterative Verfeinerung
8. Mini-Spezifikationen erstellen

(nach Balzert)

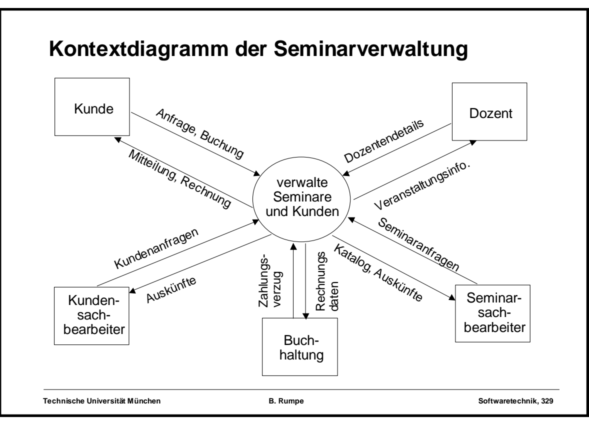
Schnittstellen und Ein/Ausgaben

- ***Schnittstellen*** sind Stellen, an denen das System Daten mit seiner Umwelt austauscht (Datenquelle, Datensenke).
- Anwendungsbereich des Systems = Angabe aller Schnittstellen
- Je Schnittstelle:
  - Name
  - Datenflüsse zum/vom System (Richtung, Name)
  - Keine Datenflüsse zwischen Schnittstellen
- Schnittstellen werden im Kontextdiagramm beschrieben
- **Kontextdiagramm:**
  - ist der einfachste Fall eines DFD
  - besitzt nur eine Funktion für das Gesamtsystem (Ebene 0)
  - zeigt alle Schnittstellen
  - beschreibt keine Speicher

Kontext der Seminarverwaltung

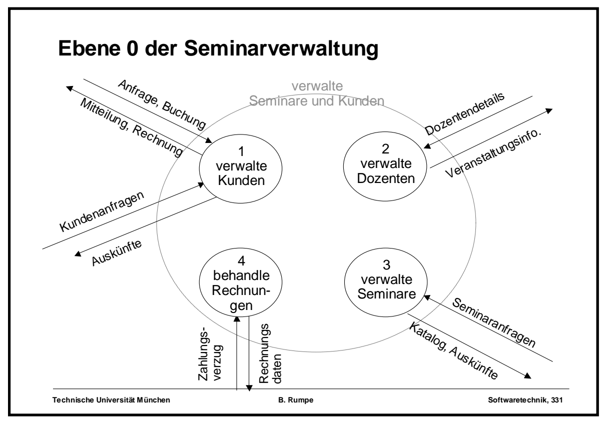


Kontextdiagramm der Seminarverwaltung



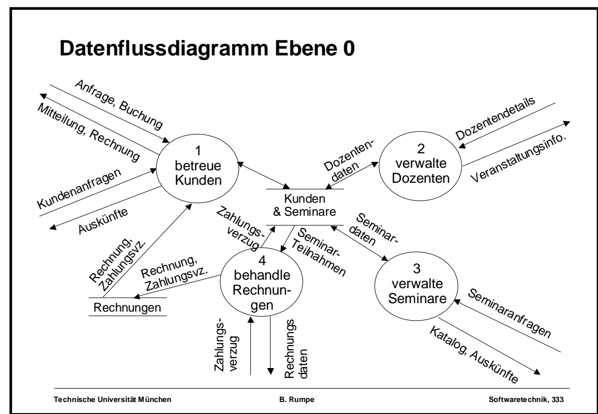
Hierarchische Zerlegung der Systemfunktionalität

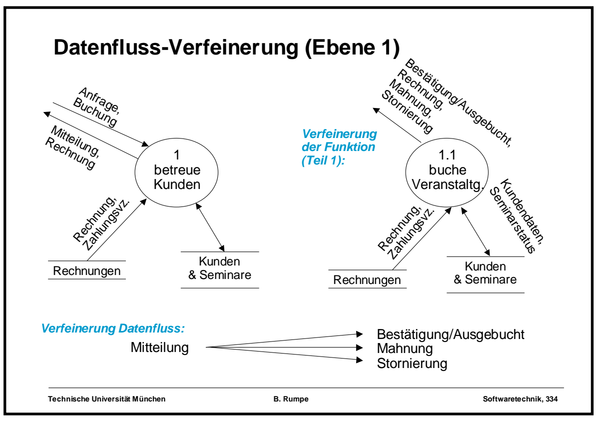
- Ebene 0-Diagramm zerlegt das System in eine erste Schicht von Teilfunktionen
- Kontextdiagramm und Ebene-0-Diagramm in SA zusammen sind sehr ähnlich zu den Use-Case-Diagrammen von UML.
- Ziel ist die hierarchische Funktions-Verfeinerung
  - Die einzige Funktion der Ebene 0 enthält Unterfunktionen 1, 2, 3, ...
  - Funktion i enthält Unterfunktionen i.1, i.2, i.3, ...
  - Aus der Praxis motiviert: Maximal ca. 3 Ebenen
  - Idealerweise: max. 7 Unterfunktionen
- Hierarchie ist Übersichtshilfe
  - Nicht vergleichbar mit Prozedurkonzept
  - Keine Sichtbarkeitsbegrenzung



Speicher im Datenfluss

- Funktionen haben in SA ein "Gedächtnis" (Datenbank, Objekte)
- ***Speicher*** repräsentieren Gedächtnis
  - Pfeil mit Datenfluss in Speicher ist *Schreiben*
  - Pfeil mit Datenfluss aus Speicher ist *Lesen*
- Speicher wird zu Funktionen zugeordnet:
  - Speicher s wird auf Ebene i definiert
  - Speicher s steht Funktionen i.xxxx zur Verfügung
- Datenfluss drückt benötigte oder gelieferte Daten einer Funktion aus
- Datenfluss ist statisch, nicht verwechseln mit Kontrollfluss!
  - So muss eine Funktion die zur Verfügung stehenden Speicher nicht nutzen
  - Die Reihenfolge der Funktionsaufrufe ist damit noch keineswegs festgelegt
  - Tatsächlich: für komplexere Datenflüsse empfiehlt sich die Konzentration des Ablaufwissens in eigenständige Komponenten („Sachbearbeiter“, „Steuerungsobjekte“)





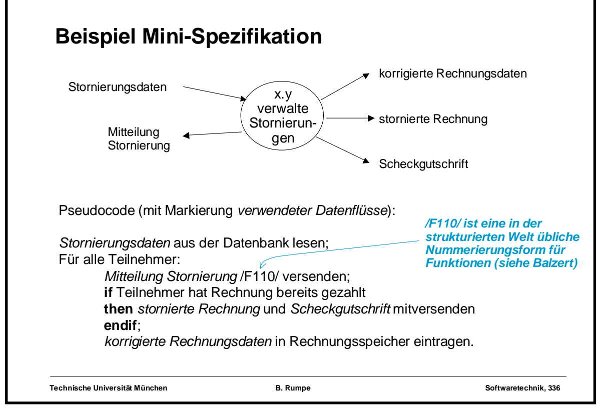
Data Dictionary und DFD (Ausschnitt)

Anfrage, Buchung — Mitteilung, Rechnung — 1 betreue Kunden — Rechnung, Zahlungsvz. — Rechnungen — Kunden & Seminare

Bestätigung/Ausgebucht, Rechnung, Mahnung, Stornierung — 1.1 buche Veranstaltg. — Kundendaten, Seminarstatus — Rechnung, Zahlungsvz. — Rechnungen — Kunden & Seminare

Data Dictionary:

Mitteilung	= [Bestätigung/Ausgebucht | Mahnung | Stornierung]
Kunden & Seminare	= Kundendaten + Seminarstatus + ...
Kundendaten	= KdNr + Name + Adresse + GebDatum
Seminarstatus	= VeranstNr + FreiePlätze
Rechnung	= RechnNr + KdNr + Betrag + VeranstNr + Rabatt + RechnDatum



Entity-Relationship-Modelle in SA

- Modernere Versionen von SA berücksichtigen Datenmodellierung
  - z.B. Yourdon, McMenamic/Palmer, SSADM
- Datenmodellierung als separate Aktivität
  - Entity-Relationship-Diagramm
- Konsolidierung mit Speichern des DFDs
  - Ein Speicher besteht dann oft aus mehreren Entities und Relationships
- Z.B. (SSADM) Korrespondenztabelle Speicher/ER-Diagramm:

Speicher	Entities	Relationships
Kunden & Seminare	Kunde Seminar	nimmt_teil
Rechnungen	Rechnung	

SD: Von der SA-Analyse zum Entwurf

- "Structured Design" (SD)
  - Stevens, Yourdon/Constantine, Myers, Page-Jones 1974-88
  - Definition von *Transformationsstrukturen* auf Basis von DFDs
- Darstellungsform: *Structure Charts* (SCs)

Aufgabe
C D
Hole C Schreibe D
B B C
Hole B Transf. B->C
A A B
Lese A Transf. A->B

korr. DFD: A B C D

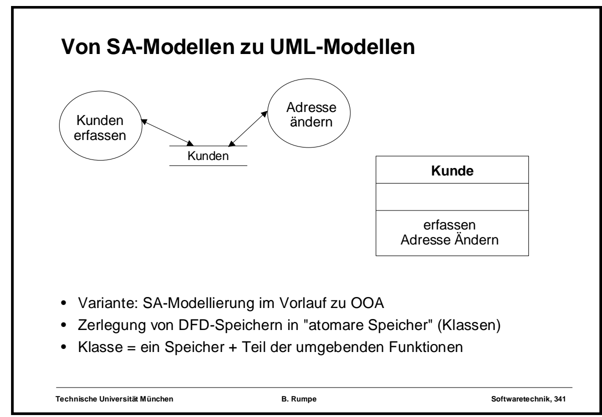
Transaktions- und Transformationsanalyse

- Transaktionsanalyse
  - Finden von Ketten
    - » auslösendes Ereignis *(event, stimulus)*
    - » Systemaktivität *(activity)*
    - » Systemantwort *(response)*
    - » Effekt z.B. auf dem Speicher *(effect)*
- Transformationsanalyse
  - Für jede Transformationskette von Eingabe in Ausgabe:
    - » "Zentrale Transformation" finden
    - » Funktion der zentralen Transformation oder neue Funktion wird Koordinator (Wurzel eines SC)
    - » Erstellen eines SC entsprechend der Daten-Transformation

Zusammenfassung SA

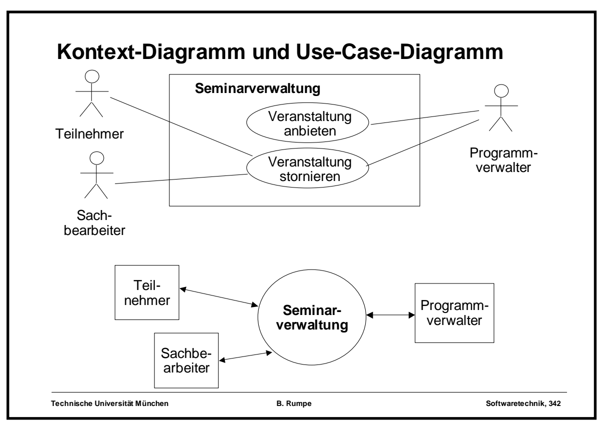
- Ausgangspunkt: Kontextdiagramm (ähnlich Use-Cases)
- Ergebnis: Abgestimmte Modelle von Funktionalität und Datenhaltung
- Methodisches Vorgehen der schrittweisen Verfeinerung
- In verschiedenen Detaillierungsgraden anwendbar
- Unbefriedigende Integration mit Datenmodellierung
- Sprachebenenwechsel nötig für Entwurf ("Strukturbruch")
  - Eingeschränkte Systemarchitektur (Baumstrukturen)
- Keine Unterstützung des Lokalitätsprinzips
  - demgegenüber gruppiert OO Daten und Funktionen in Einheiten und erlaubt so bessere Lokalisierung und auch Wiederverwendung

Von SA-Modellen zu UML-Modellen

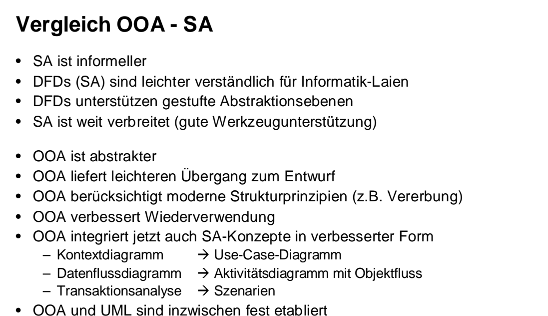


- Variante: SA-Modellierung im Vorlauf zu OOA
- Zerlegung von DFD-Speichern in "atomare Speicher" (Klassen)
- Klasse = ein Speicher + Teil der umgebenden Funktionen

Kontext-Diagramm und Use-Case-Diagramm



Vergleich OOA - SA



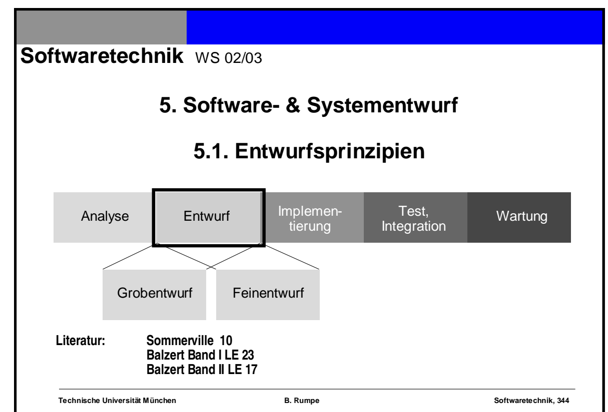
- SA ist informeller
- DFDs (SA) sind leichter verständlich für Informatik-Laien
- DFDs unterstützen gestufte Abstraktionsebenen
- SA ist weit verbreitet (gute Werkzeugunterstützung)

- OOA ist abstrakter
- OOA liefert leichteren Übergang zum Entwurf
- OOA berücksichtigt moderne Strukturprinzipien (z.B. Vererbung)
- OOA verbessert Wiederverwendung
- OOA integriert jetzt auch SA-Konzepte in verbesserter Form
  - Kontextdiagramm → Use-Case-Diagramm
  - Datenflussdiagramm → Aktivitätsdiagramm mit Objektfluss
  - Transaktionsanalyse → Szenarien
- OOA und UML sind inzwischen fest etabliert (schnelle Entwicklung bei den Werkzeugen)

Softwaretechnik WS 02/03

5. Software- & Systementwurf

5.1. Entwurfsprinzipien

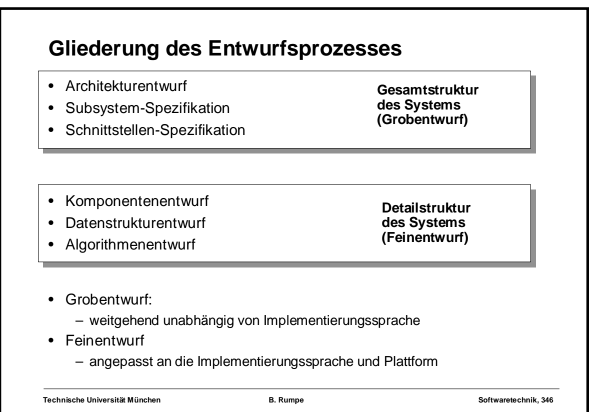


Literatur: **Sommerville 10**
Balzert Band I LE 23
Balzert Band II LE 17

Software-Entwurf

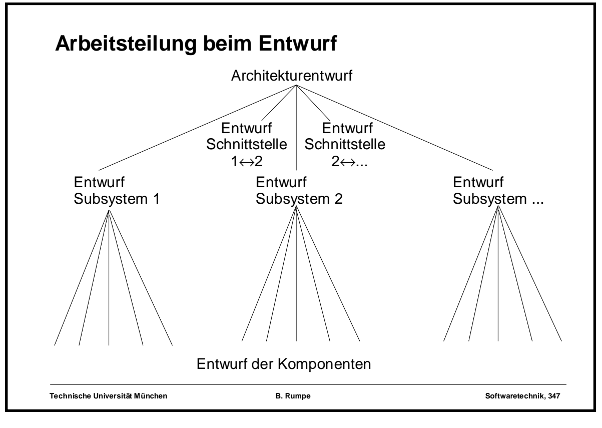
- Ausgangspunkt:
  - Anforderungsspezifikation (Pflichtenheft) und
  - Funktionale Spezifikation (Produktdefinition)
- Ziel:
  - Vom "WAS" zum "WIE": Vorgabe für Implementierung
- Zentrale Begriffe:
  - Subsystem:
    - » in sich geschlossen
    - » eigenständig funktionsfähig mit definierten Schnittstellen
    - » besteht aus Komponenten
  - Komponente
    - » Baustein für ein Softwaresystem (z.B. Modul, Klasse, Paket)
    - » benutzt andere Komponenten
    - » wird von anderen Komponenten benutzt
    - » kann auch aus Unterkomponenten bestehen

Gliederung des Entwurfsprozesses



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

Arbeitsteilung beim Entwurf



Kriterien für "guten" Entwurf

- Korrektheit
  - Erfüllung der Anforderungen
  - Wiedergabe aller Funktionen des Systemmodells
  - Sicherstellung der nichtfunktionalen Anforderungen
- Verständlichkeit & Präzision
  - Gute Dokumentation
- Anpassbarkeit
- Hohe Kohäsion innerhalb der Komponenten
- Schwache Kopplung zwischen den Komponenten
- Wiederverwendung

- Diese Kriterien sind "fraktal", d.h. sie gelten auf allen Ebenen des Entwurfs (Architektur, Subsysteme, Komponenten)

Kohäsion

- Kohäsion ist ein Maß für die Zusammengehörigkeit der Bestandteile einer Komponente.
- Hohe Kohäsion einer Komponente erleichtert Verständnis, Wartung und Anpassung.
- Frühere Ansätze zur Kohäsion wie:
  - ähnliche Funktionalitäten zusammenfassen
    - » führten *nicht* unbedingt zu stabiler Systemstruktur
- Bessere Kohäsion wird erreicht durch:
  - Prinzipien der Objektorientierung (Datenkapselung)
  - Einhaltung von Regeln zur Paketbildung
  - Verwendung geeigneter Muster zu Kopplung und Entkopplung
  - "Kohärente" Klasse:
    - » Es gibt keine Partitionierung in Untergruppen von zusammengehörigen Operationen und Attributen

Kopplung

- Kopplung ist ein Maß für die Abhängigkeiten zwischen Komponenten.
- Geringe Kopplung erleichtert die Wartbarkeit und macht Systeme stabiler.
- Arten der Kopplung:
  - Datenkopplung (gemeinsame Daten)
  - Schnittstellenkopplung (gegenseitiger Aufruf)
  - Strukturkopplung (gemeinsame Strukturelemente)
- Reduktion der Kopplung:
  - Kopplung kann nie auf Null reduziert werden!
  - Schnittstellenkopplung ist akzeptabel, da höhere Flexibilität
  - Datenkopplung vermeiden!
    - » aber durch Objektorientierung meist gegeben
  - Strukturkopplung vermeiden !
    - » z.B. keine Vererbung über Paketgrenzen hinweg
- Entkopplungsbeispiel: get/set-Methoden statt Attributzugriff

Interne Wiederverwendung

- Interne Wiederverwendung (*reuse*) ist ein Maß für die Ausnutzung von Gemeinsamkeiten zwischen Komponenten
- Reduktion der Redundanz
- Erhöhung der Stabilität und Ergonomie
- Hilfsmittel für Wiederverwendung:
  - im objektorientierten Entwurf: Vererbung, Parametrisierung
  - im modularen und objektorientierten Entwurf: Module/Objekte mit allgemeinen Schnittstellen (Interfaces)
- Aber: Wiederverwendung kann die Kopplung erhöhen:
  - Schnittstellenkopplung und Strukturkopplung

Metriken für modulare Entwürfe

- Nach Henry/Kafura 1981, Card/Glass 1990: *Fan-in/fan-out-Metrik*
- Fan-in:
  - Anzahl von Stellen, wo Kontrollfluss auf das betrachtete Modul M übergeht (Aufrufe von Funktionen/Prozeduren in M), plus
  - Anzahl globaler Variablen, die in M zugänglich sind
- Fan-out:
  - Anzahl von Stellen, an denen M andere Module aufruft, plus
  - Anzahl der globalen Variablen, die von M verändert werden
- Kopplung: Hoher Fan-out ist ein Indikator für hohe Kopplung
  - » Fan-out ist *soweit möglich* zu minimieren
- Kohäsion:
  - » Hoher Fan-in *kann* auf geringe Kohäsion von M hindeuten
  - » Informelle Skala (Kohäsionsgrade) verwenden

Metriken für objektorientierte Entwürfe

- Nach Chidamber/Kemerer 1994:
- Kohäsion
  - *Lack of Cohesion in Methods (LCOM):* Anzahl von Methodenpaaren, die keine gemeinsamen Daten verwenden, skaliert auf Anzahl von Methoden und Attributen
    (Diverse Definitionsvarianten, hier von Henderson-Sellers 1996)
- Kopplung
  - *Coupling between Objects (CBO):* Anzahl der Objekte, die von Objekten einer Klasse C benutzt werden
  - *Response for a Class (RFC):* Anzahl der Methoden, die durch eine Nachricht an ein Objekt der Klasse C aufgerufen werden können
- Wiederverwendung
  - *Depth of Inheritance Tree (DIT)*
  - *Number of (Subclass) Children (NOC)*

Problematik von Entwurfsmetriken

- Entwurfsmetrik: Messbare Kenngröße eines Entwicklungs-Produkts oder des Software-Prozesses
- Existenz von Metriken täuscht oft Präzision nur vor.
- Ziele des Einsatzes von Metriken oft unklar:
  - Qualitätsüberprüfung fertiger Entwürfe/Programme
    - » Fehlerfreiheit, Änderungsfreundlichkeit, Verständlichkeit, ...
  - Präzisierung theoretischer Begriffe
  - Hilfsmittel zur Aufwandsschätzung
- Werkzeuge existieren meist nur für Programme (nicht für Entwürfe)
- Signifikanz von Metriken schwer überprüfbar
  - Stichprobenmaterial
  - Vergleichsmetriken
- Unangenehm: Die „gemessenen" Entwickler optimieren ihre Ergebnisse bzgl. der verwendeten Metriken!

OO-Metriken und Fehlerwahrscheinlichkeit

- Aussagekraft der Metriken ach Basili/Briand/Melo 96:
- Korrelation zwischen Fehlerwahrscheinlichkeit und OO-Entwurfsmetriken, gemessen an C++-Programmen

Metrik	Wirkung hoher Werte auf Fehlerwahrscheinlichkeit	statistische Signifikanz
Kopplung:		
CBO	höher	mittel
RFC	höher	hoch
Kohäsion:		
LCOM	---	keine
Wiederverwendung:		
DIT	höher	hoch
NOC	niedriger (außer bei GUIs)	hoch

- Aktuelles Forschungsgebiet

  Weitere Information z.B. in Henderson-Sellers: Object-Oriented Metrics, Prentice-Hall 1996

Vererbung und Fehlerwahrscheinlichkeit

- Hypothese 1:
  Tiefe Vererbungshierarchien erhöhen Fehlerwahrscheinlichkeit.
  - Basili/Briand/Melo 96
  - Cartwright/Shepperd 00
- Hypothese 2:
  Vererbung kann die Fehlerwahrscheinlichkeit reduzieren, wenn sparsam und nicht auf hohen Ebenen des Systems angewendet.
  - Abreu/Melo 96
- Diskussion:
  - Viele "objektorientierte" Programme beachten wichtige Prinzipien, wie das Substitutions-Prinzip nicht.
  - Weitere empirische Forschung unbedingt nötig.

Metriken auf Zustandsmodell-Basis

- Zustandsmodelle zur Abschätzung der Fehleranfälligkeit und Komplexität einer Klasse
- Nach Cartwright/Shepperd 00:
  - *States (STATES):*
    Anzahl von Zuständen pro Klasse im Zustandsmodell
  - *Events (EVNT):*
    Anzahl von Ereignissen pro Klasse im Zustandsmodell
- STATES scheint gute Korrelation mit der Größe des Klassencodes (LOC) zu zeigen.
- EVNT scheint gute Korrelation mit der Anzahl der Defekte zu zeigen.
- Vorteil:
  - Metriken bereits vor Codefertigstellung anwendbar.
- Hinweis:
  - Es gibt keine Korrelation zwischen LOC und Defektanzahl!
  - Pareto-Regel: 20% (10%?) des Codes verursachen 80% der Fehler.

Zusammenfassung: 5.1. Entwurfsprinzipien

- Hierarchische Zerlegung des Systems in Subsysteme und Komponenten
- Dabei auf Kohäsion zusammengefasster Elemente achten
- Kopplung zwischen Elementen so gering wie möglich
  - Insbesondere Daten- und Strukturkopplung reduzieren
- Entwurfsmetrik: Messbare Kenngröße eines Entwicklungs-Produkts oder des Software-Prozesses
- Aussagekraft einer Metrik beachten und nicht überbewerten!

Softwaretechnik WS 02/03

5. Software- & Systementwurf

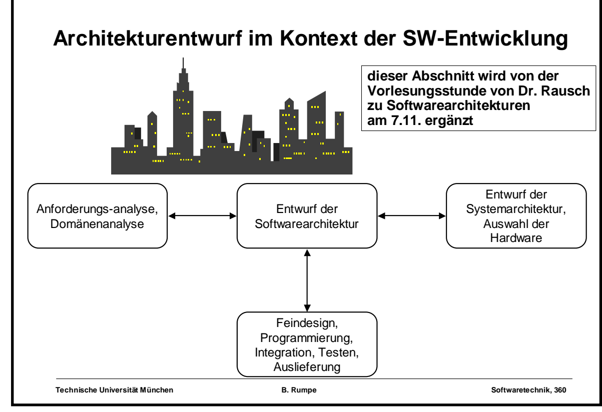
5.2. Softwarearchitektur

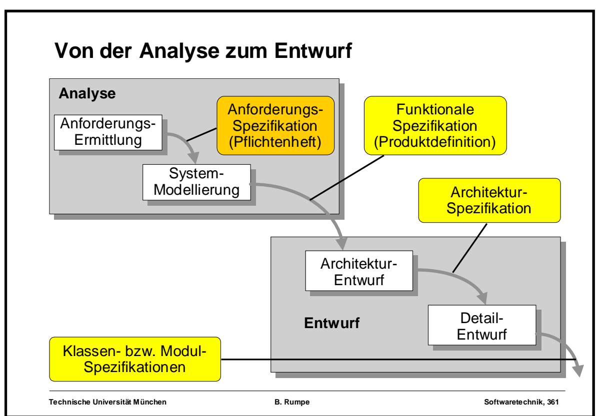
Analyse | **Entwurf** | Implementierung | Test, Integration | Wartung

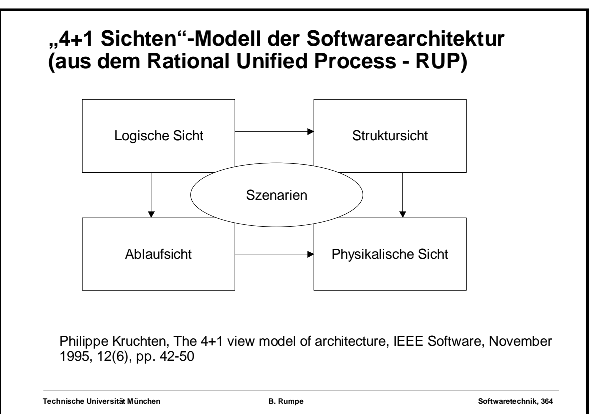
Grobentwurf | Feinentwurf

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

Architekturentwurf im Kontext der SW-Entwicklung

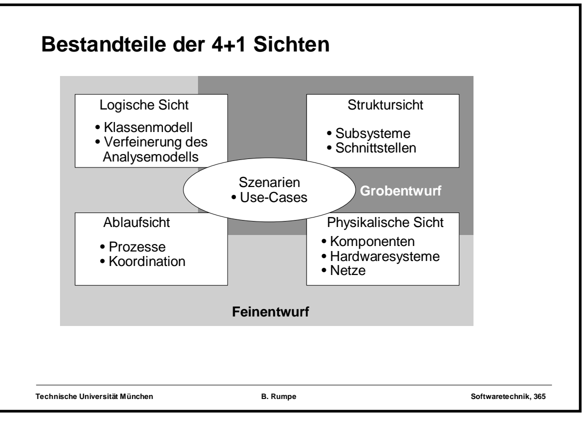


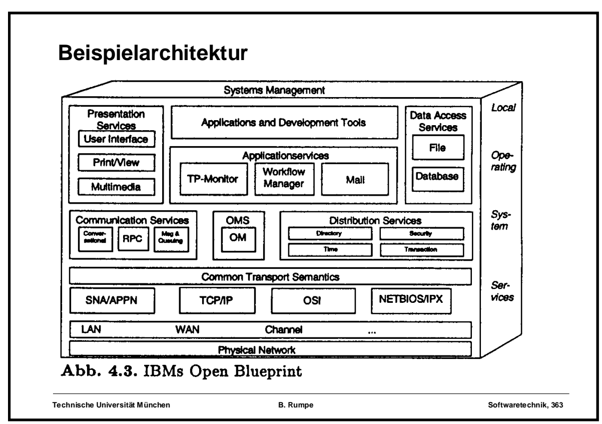


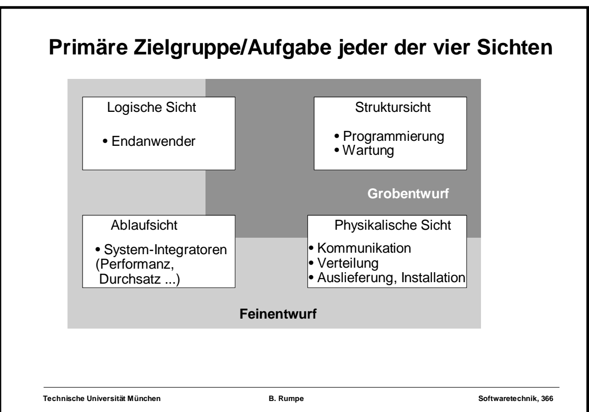


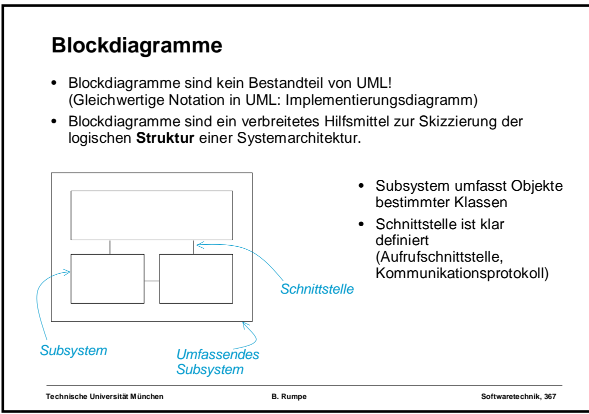
Softwarearchitektur in der Praxis

- Architekturspezifikation wird oft nicht als separates Dokument gefordert.
- Häufig wird funktionale Spezifikation und Architekturspezifikation in einem Dokument realisiert.
  - denn „WAS“ zu spezifizieren, ohne auf grobe Strukturen des „WIE“ einzugehen ist oft nicht möglich.
  - Dennoch: die grobe Systemarchitektur wird der Entwurfs-Aktivität zugeordnet
- Ist Hardware involviert (Steuergeräte im Auto, Telekommunikations-Anlagen etc.), so wird oft bereits dadurch eine Architektur vorgegeben. So kann es in einem systemorientierten Projekt durchaus sinnvoll sein, erste Architekturskizzen bereits in die Anforderungsbeschreibung aufzunehmen.

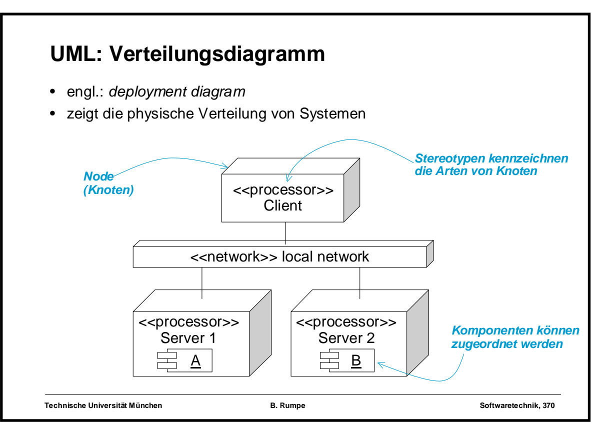




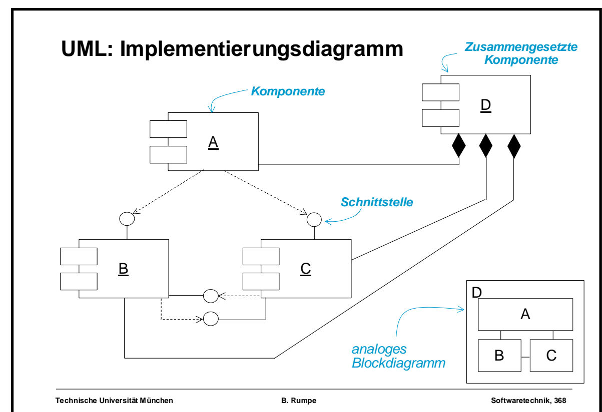




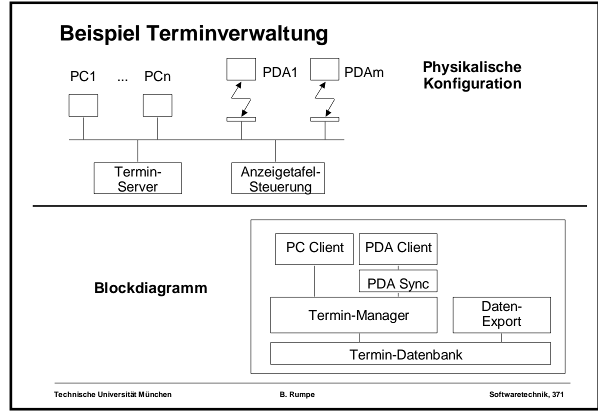
Blockdiagramme
Blockdiagramme sind kein Bestandteil von UML!
(Gleichwertige Notation in UML: Implementierungsdiagramm)
Blockdiagramme sind ein verbreitetes Hilfsmittel zur Skizzierung der logischen Struktur einer Systemarchitektur.
Subsystem umfasst Objekte bestimmter Klassen
Schnittstelle ist klar definiert (Aufrufschnittstelle, Kommunikationsprotokoll)
Schnittstelle
Subsystem
Umfassendes Subsystem
Technische Universität München
B. Rumpe
Softwaretechnik, 367



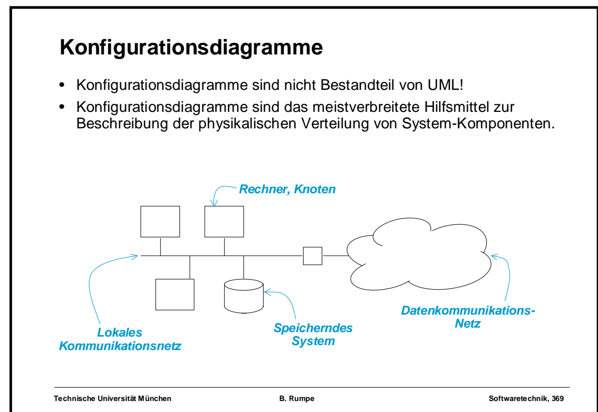
UML: Verteilungsdiagramm
engl.: deployment diagram
zeigt die physische Verteilung von Systemen
Node (Knoten)
Stereotypen kennzeichnen die Arten von Knoten
<<processor>> Client
<<network>> local network
<<processor>> Server 1
A
<<processor>> Server 2
B
Komponenten können zugeordnet werden
Technische Universität München
B. Rumpe
Softwaretechnik, 370



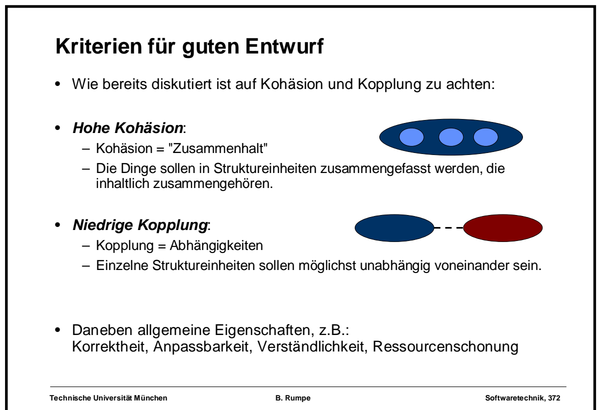
UML: Implementierungsdiagramm
Zusammengesetzte Komponente
Komponente
A
D
Schnittstelle
B
C
D
A
B
C
analoges Blockdiagramm
Technische Universität München
B. Rumpe
Softwaretechnik, 368



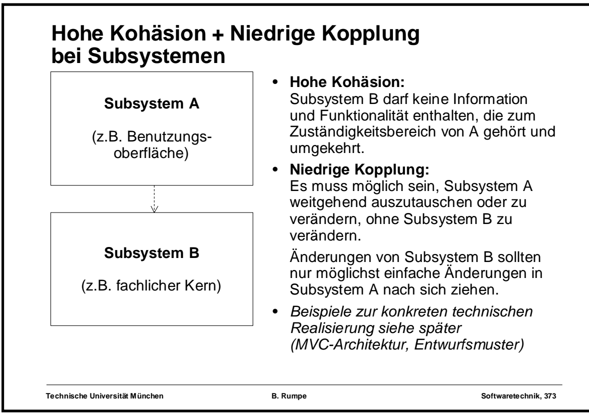
Beispiel Terminverwaltung
PC1 ... PCn
PDA1
PDAm
Physikalische Konfiguration
Termin-Server
Anzeigetafel-Steuerung
Blockdiagramm
PC Client
PDA Client
PDA Sync
Termin-Manager
Daten-Export
Termin-Datenbank
Technische Universität München
B. Rumpe
Softwaretechnik, 371

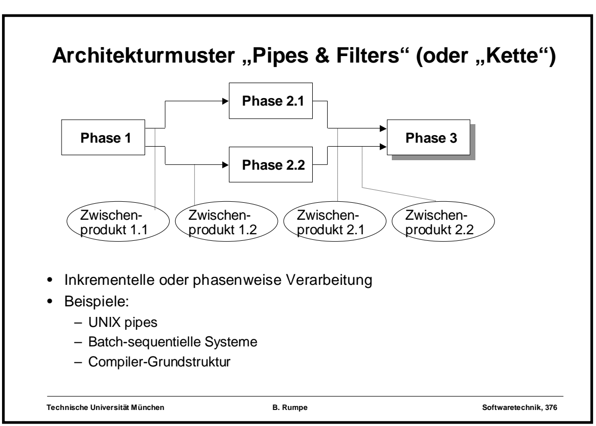


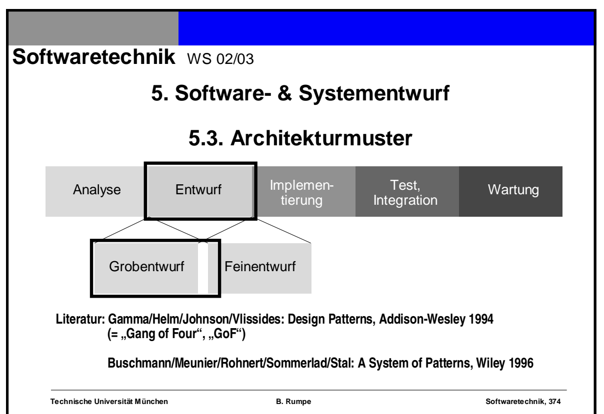
Konfigurationsdiagramme
Konfigurationsdiagramme sind nicht Bestandteil von UML!
Konfigurationsdiagramme sind das meistverbreitete Hilfsmittel zur Beschreibung der physikalischen Verteilung von System-Komponenten.
Rechner, Knoten
Lokales Kommunikationsnetz
Speicherndes System
Datenkommunikations-Netz
Technische Universität München
B. Rumpe
Softwaretechnik, 369

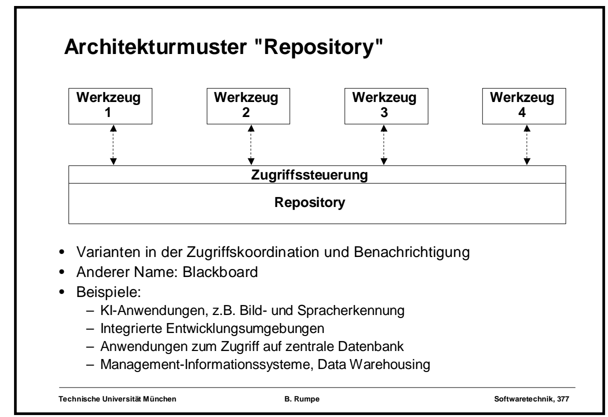


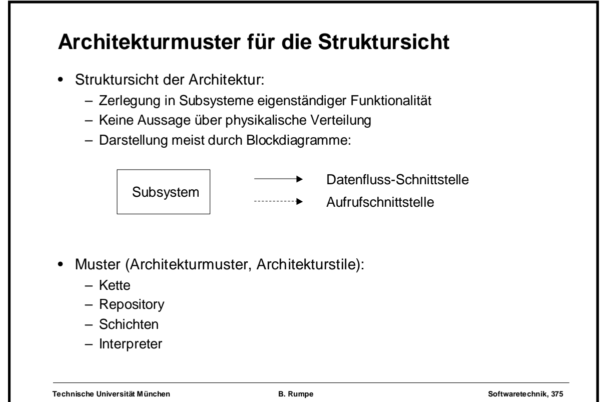
Kriterien für guten Entwurf
Wie bereits diskutiert ist auf Kohäsion und Kopplung zu achten:
Hohe Kohäsion:
– Kohäsion = "Zusammenhalt"
– Die Dinge sollen in Struktureinheiten zusammengefasst werden, die inhaltlich zusammengehören.
Niedrige Kopplung:
– Kopplung = Abhängigkeiten
– Einzelne Struktureinheiten sollen möglichst unabhängig voneinander sein.
Daneben allgemeine Eigenschaften, z.B.:
Korrektheit, Anpassbarkeit, Verständlichkeit, Ressourcenschonung
Technische Universität München
B. Rumpe
Softwaretechnik, 372

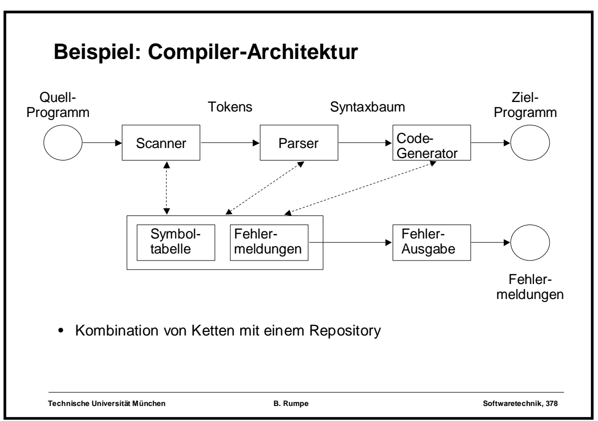

Hohe Kohäsion + Niedrige Kopplung bei Subsystemen
Subsystem A
(z.B. Benutzungs-oberfläche)
Subsystem B
(z.B. fachlicher Kern)
• Hohe Kohäsion:
Subsystem B darf keine Information und Funktionalität enthalten, die zum Zuständigkeitsbereich von A gehört und umgekehrt.
• Niedrige Kopplung:
Es muss möglich sein, Subsystem A weitgehend auszutauschen oder zu verändern, ohne Subsystem B zu verändern.
Änderungen von Subsystem B sollten nur möglichst einfache Änderungen in Subsystem A nach sich ziehen.
• Beispiele zur konkreten technischen Realisierung siehe später (MVC-Architektur, Entwurfsmuster)
Technische Universität München
B. Rumpe
Softwaretechnik, 373

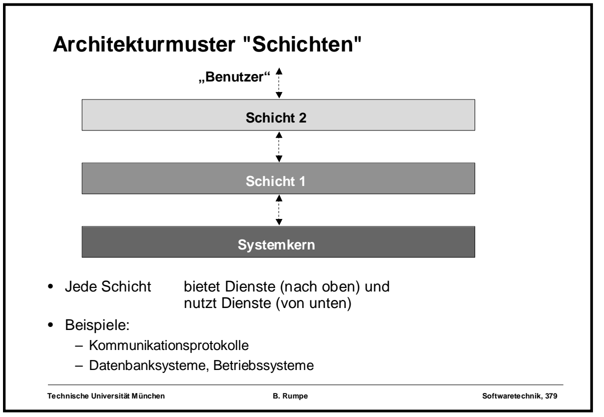

Architekturmuster „Pipes & Filters“ (oder „Kette“)
Phase 1
Phase 2.1
Phase 2.2
Phase 3
Zwischen-produkt 1.1
Zwischen-produkt 1.2
Zwischen-produkt 2.1
Zwischen-produkt 2.2
• Inkrementelle oder phasenweise Verarbeitung
• Beispiele:
– UNIX pipes
– Batch-sequentielle Systeme
– Compiler-Grundstruktur
Technische Universität München
B. Rumpe
Softwaretechnik, 376

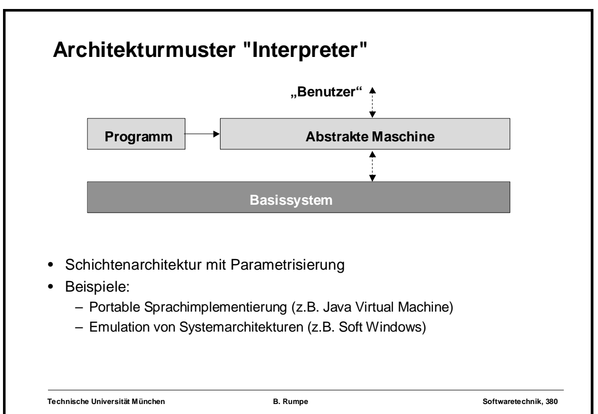

Softwaretechnik WS 02/03
5. Software- & Systementwurf
5.3. Architekturmuster
Analyse
Entwurf
Implemen-tierung
Test, Integration
Wartung
Grobentwurf
Feinentwurf
Literatur: Gamma/Helm/Johnson/Vlissides: Design Patterns, Addison-Wesley 1994
(= „Gang of Four“, „GoF“)
Buschmann/Meunier/Rohnert/Sommerlad/Stal: A System of Patterns, Wiley 1996
Technische Universität München
B. Rumpe
Softwaretechnik, 374

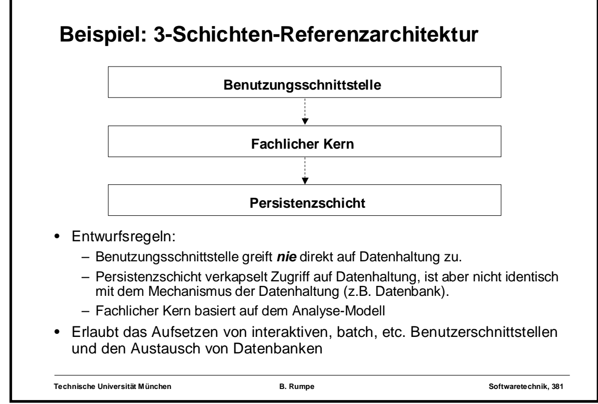

Architekturmuster "Repository"
Werkzeug 1
Werkzeug 2
Werkzeug 3
Werkzeug 4
Zugriffssteuerung
Repository
• Varianten in der Zugriffskoordination und Benachrichtigung
• Anderer Name: Blackboard
• Beispiele:
– KI-Anwendungen, z.B. Bild- und Spracherkennung
– Integrierte Entwicklungsumgebungen
– Anwendungen zum Zugriff auf zentrale Datenbank
– Management-Informationssysteme, Data Warehousing
Technische Universität München
B. Rumpe
Softwaretechnik, 377

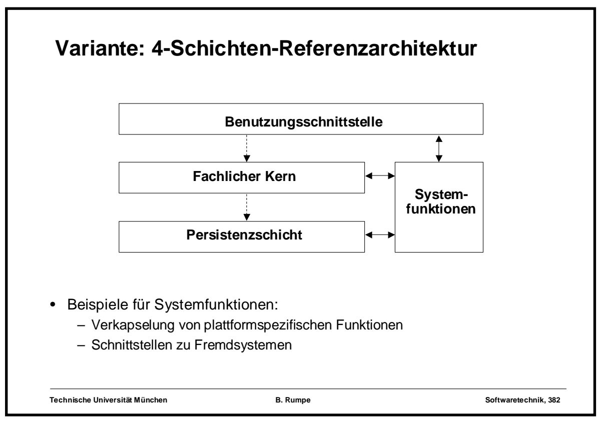

Architekturmuster für die Struktursicht
• Struktursicht der Architektur:
– Zerlegung in Subsysteme eigenständiger Funktionalität
– Keine Aussage über physikalische Verteilung
– Darstellung meist durch Blockdiagramme:
Subsystem
Datenfluss-Schnittstelle
Aufrufschnittstelle
• Muster (Architekturmuster, Architekturstile):
– Kette
– Repository
– Schichten
– Interpreter
Technische Universität München
B. Rumpe
Softwaretechnik, 375

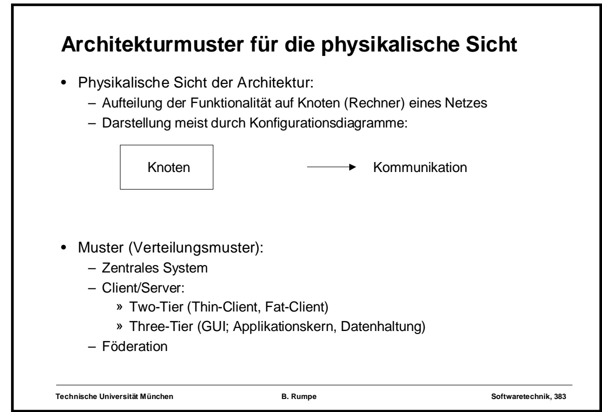

Beispiel: Compiler-Architektur
Quell-Programm
Tokens
Syntaxbaum
Ziel-Programm
Scanner
Parser
Code-Generator
Symbol-tabelle
Fehler-meldungen
Fehler-Ausgabe
Fehler-meldungen
• Kombination von Ketten mit einem Repository
Technische Universität München
B. Rumpe
Softwaretechnik, 378

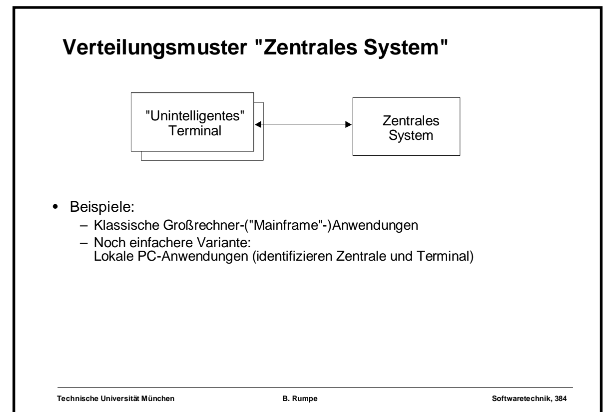

Architekturmuster "Schichten"
„Benutzer“
Schicht 2
Schicht 1
Systemkern
• Jede Schicht bietet Dienste (nach oben) und nutzt Dienste (von unten)
• Beispiele:
– Kommunikationsprotokolle
– Datenbanksysteme, Betriebssysteme
Technische Universität München
B. Rumpe
Softwaretechnik, 379

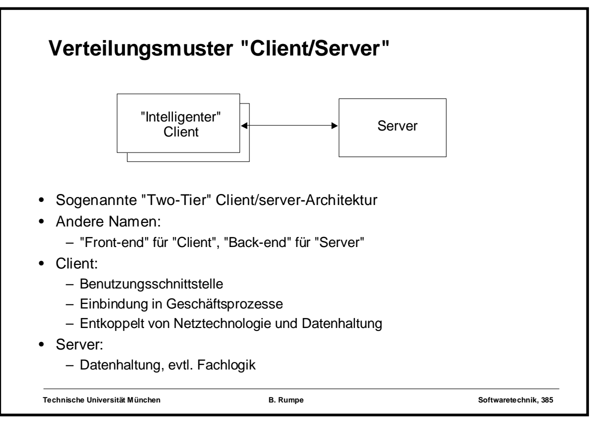

Architekturmuster "Interpreter"
„Benutzer“
Programm
Abstrakte Maschine
Basissystem
• Schichtenarchitektur mit Parametrisierung
• Beispiele:
– Portable Sprachimplementierung (z.B. Java Virtual Machine)
– Emulation von Systemarchitekturen (z.B. Soft Windows)
Technische Universität München
B. Rumpe
Softwaretechnik, 380

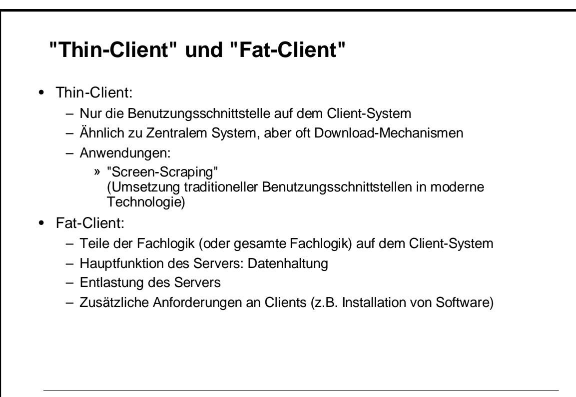

Beispiel: 3-Schichten-Referenzarchitektur
Benutzungsschnittstelle
Fachlicher Kern
Persistenzschicht
• Entwurfsregeln:
– Benutzungsschnittstelle greift nie direkt auf Datenhaltung zu.
– Persistenzschicht verkapselt Zugriff auf Datenhaltung, ist aber nicht identisch mit dem Mechanismus der Datenhaltung (z.B. Datenbank).
– Fachlicher Kern basiert auf dem Analyse-Modell
• Erlaubt das Aufsetzen von interaktiven, batch, etc. Benutzerschnittstellen und den Austausch von Datenbanken
Technische Universität München
B. Rumpe
Softwaretechnik, 381

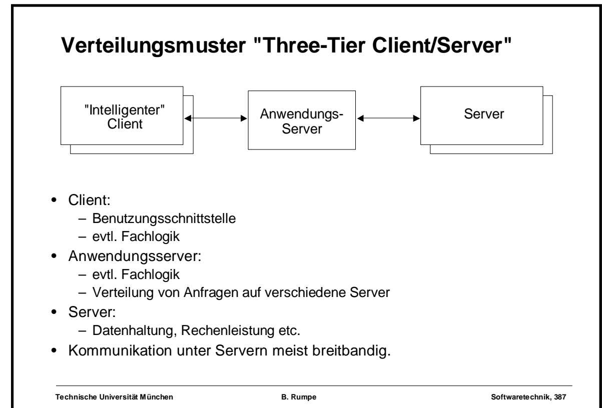

Variante: 4-Schichten-Referenzarchitektur
Benutzungsschnittstelle
Fachlicher Kern
Persistenzschicht
System-funktionen
• Beispiele für Systemfunktionen:
– Verkapselung von plattformspezifischen Funktionen
– Schnittstellen zu Fremdsystemen
Technische Universität München
B. Rumpe
Softwaretechnik, 382

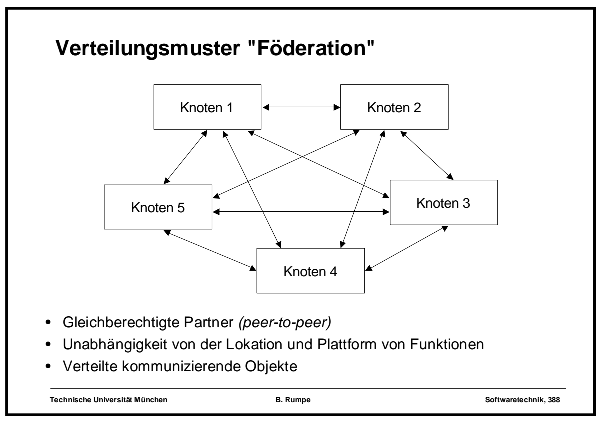

Architekturmuster für die physikalische Sicht
• Physikalische Sicht der Architektur:
– Aufteilung der Funktionalität auf Knoten (Rechner) eines Netzes
– Darstellung meist durch Konfigurationsdiagramme:
Knoten
Kommunikation
• Muster (Verteilungsmuster):
– Zentrales System
– Client/Server:
» Two-Tier (Thin-Client, Fat-Client)
» Three-Tier (GUI; Applikationskern, Datenhaltung)
– Föderation
Technische Universität München
B. Rumpe
Softwaretechnik, 383

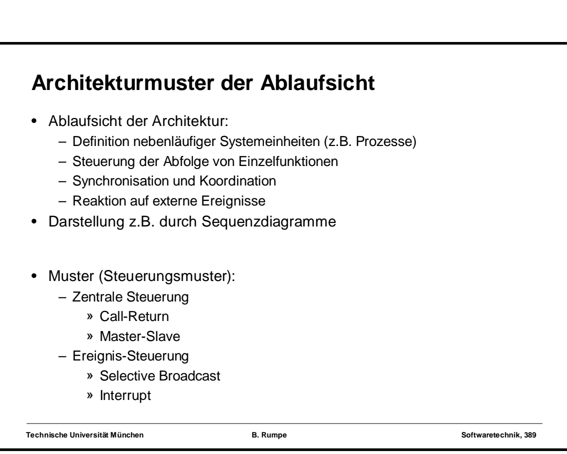

Verteilungsmuster "Zentrales System"
"Unintelligentes" Terminal
Zentrales System
• Beispiele:
– Klassische Großrechner-("Mainframe"-)Anwendungen
– Noch einfachere Variante:
Lokale PC-Anwendungen (identifizieren Zentrale und Terminal)
Technische Universität München
B. Rumpe
Softwaretechnik, 384

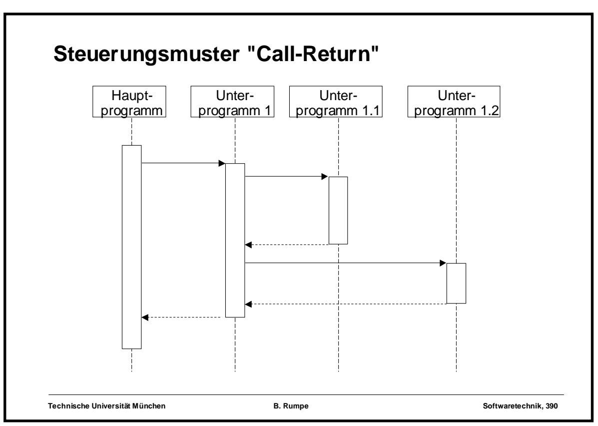

Verteilungsmuster "Client/Server"
"Intelligenter" Client
Server
• Sogenannte "Two-Tier" Client/server-Architektur
• Andere Namen:
– "Front-end" für "Client", "Back-end" für "Server"
• Client:
– Benutzungsschnittstelle
– Einbindung in Geschäftsprozesse
– Entkoppelt von Netztechnologie und Datenhaltung
• Server:
– Datenhaltung, evtl. Fachlogik
Technische Universität München
B. Rumpe
Softwaretechnik, 385

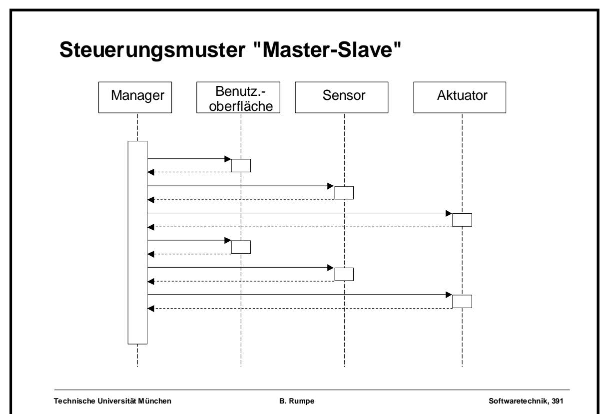

"Thin-Client" und "Fat-Client"
• Thin-Client:
– Nur die Benutzungsschnittstelle auf dem Client-System
– Ähnlich zu Zentralem System, aber oft Download-Mechanismen
– Anwendungen:
» "Screen-Scraping" (Umsetzung traditioneller Benutzungsschnittstellen in moderne Technologie)
• Fat-Client:
– Teile der Fachlogik (oder gesamte Fachlogik) auf dem Client-System
– Hauptfunktion des Servers: Datenhaltung
– Entlastung des Servers
– Zusätzliche Anforderungen an Clients (z.B. Installation von Software)
Technische Universität München
B. Rumpe
Softwaretechnik, 386

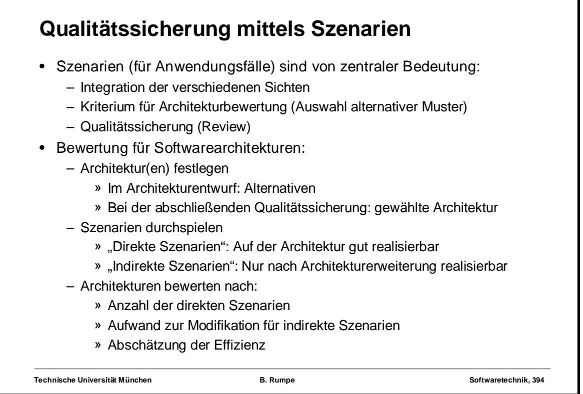

Verteilungsmuster "Three-Tier Client/Server"
"Intelligenter" Client
Anwendungs-Server
Server
• Client:
– Benutzungsschnittstelle
– evtl. Fachlogik
• Anwendungsserver:
– evtl. Fachlogik
– Verteilung von Anfragen auf verschiedene Server
• Server:
– Datenhaltung, Rechenleistung etc.
• Kommunikation unter Servern meist breitbandig.
Technische Universität München
B. Rumpe
Softwaretechnik, 387

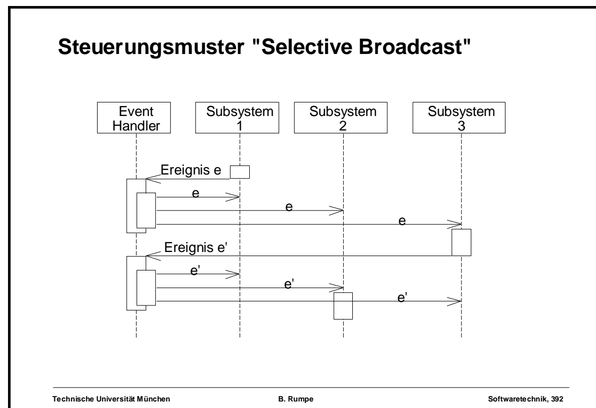

Verteilungsmuster "Föderation"
Knoten 1
Knoten 2
Knoten 5
Knoten 3
Knoten 4
• Gleichberechtigte Partner (peer-to-peer)
• Unabhängigkeit von der Lokation und Plattform von Funktionen
• Verteilte kommunizierende Objekte
Technische Universität München
B. Rumpe
Softwaretechnik, 388

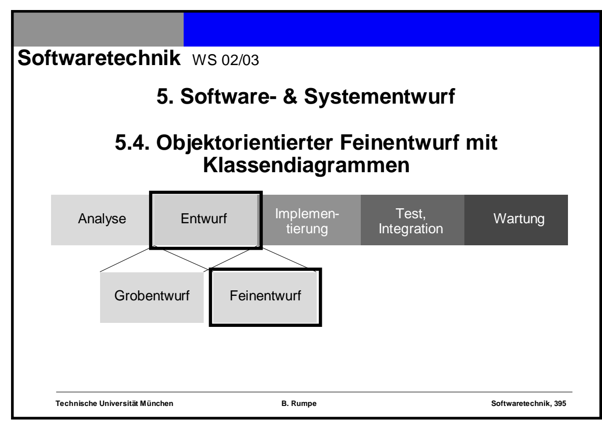

Architekturmuster der Ablaufsicht
• Ablaufsicht der Architektur:
– Definition nebenläufiger Systemeinheiten (z.B. Prozesse)
– Steuerung der Abfolge von Einzelfunktionen
– Synchronisation und Koordination
– Reaktion auf externe Ereignisse
• Darstellung z.B. durch Sequenzdiagramme
• Muster (Steuerungsmuster):
– Zentrale Steuerung
» Call-Return
» Master-Slave
– Ereignis-Steuerung
» Selective Broadcast
» Interrupt
Technische Universität München
B. Rumpe
Softwaretechnik, 389

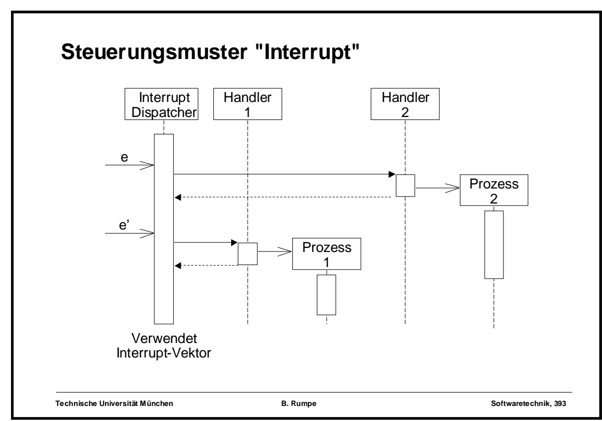

Steuerungsmuster "Call-Return"
Haupt-programm
Unter-programm 1
Unter-programm 1.1
Unter-programm 1.2
Technische Universität München
B. Rumpe
Softwaretechnik, 390

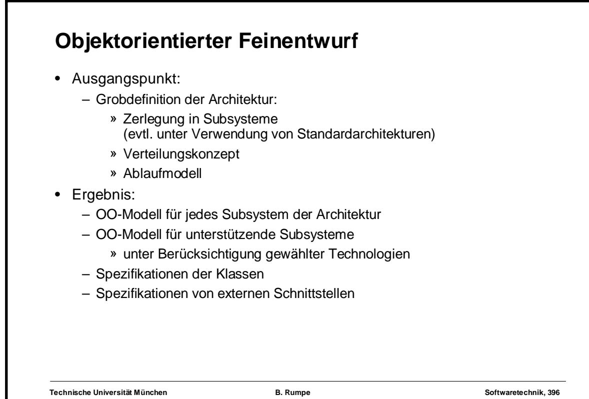

Steuerungsmuster "Master-Slave"
Manager
Benutz.-oberfläche
Sensor
Aktuator
Technische Universität München
B. Rumpe
Softwaretechnik, 391

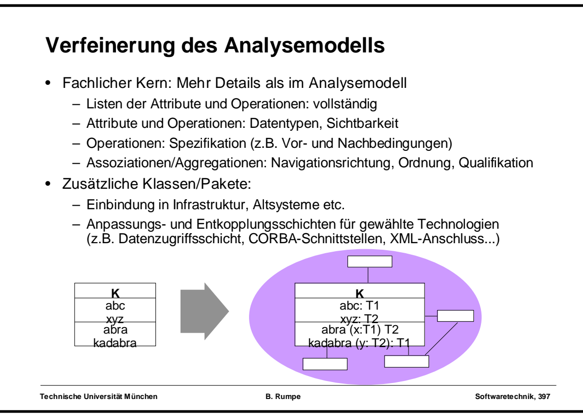

Qualitätssicherung mittels Szenarien
• Szenarien (für Anwendungsfälle) sind von zentraler Bedeutung:
– Integration der verschiedenen Sichten
– Kriterium für Architekturbewertung (Auswahl alternativer Muster)
– Qualitätssicherung (Review)
• Bewertung für Softwarearchitekturen:
– Architektur(en) festlegen
» Im Architekturentwurf: Alternativen
» Bei der abschließenden Qualitätssicherung: gewählte Architektur
– Szenarien durchspielen
» „Direkte Szenarien“: Auf der Architektur gut realisierbar
» „Indirekte Szenarien“: Nur nach Architekturerweiterung realisierbar
– Architekturen bewerten nach:
» Anzahl der direkten Szenarien
» Aufwand zur Modifikation für indirekte Szenarien
» Abschätzung der Effizienz
Technische Universität München
B. Rumpe
Softwaretechnik, 394

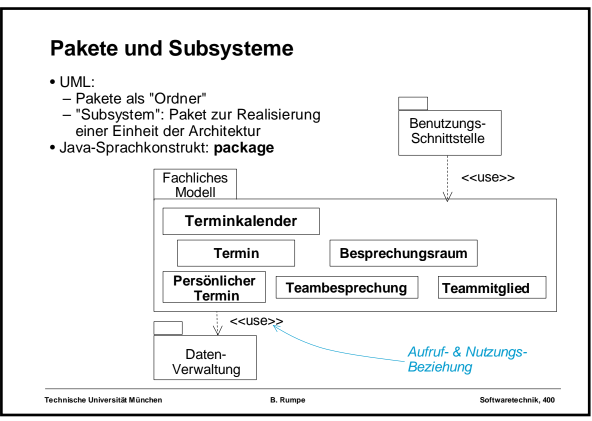

Steuerungsmuster "Selective Broadcast"
Event Handler
Subsystem 1
Subsystem 2
Subsystem 3
Ereignis e
e
e
e
Ereignis e'
e'
e'
e'
Technische Universität München
B. Rumpe
Softwaretechnik, 392


Softwaretechnik WS 02/03
5. Software- & Systementwurf
5.4. Objektorientierter Feinentwurf mit Klassendiagrammen
Analyse
Entwurf
Implemen-tierung
Test, Integration
Wartung
Grobentwurf
Feinentwurf
Technische Universität München
B. Rumpe
Softwaretechnik, 395


Steuerungsmuster "Interrupt"
Interrupt Dispatcher
Handler 1
Handler 2
e
e'
Prozess 2
Prozess 1
Verwendet Interrupt-Vektor
Technische Universität München
B. Rumpe
Softwaretechnik, 393

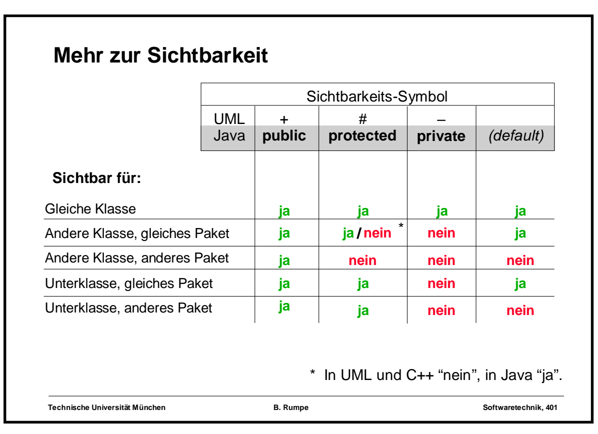

Objektorientierter Feinentwurf
• Ausgangspunkt:
– Grobdefinition der Architektur:
» Zerlegung in Subsysteme
(evtl. unter Verwendung von Standardarchitekturen)
» Verteilungskonzept
» Ablaufmodell
• Ergebnis:
– OO-Modell für jedes Subsystem der Architektur
– OO-Modell für unterstützende Subsysteme
» unter Berücksichtigung gewählter Technologien
– Spezifikationen der Klassen
– Spezifikationen von externen Schnittstellen
Technische Universität München
B. Rumpe
Softwaretechnik, 396





UML im Entwurf

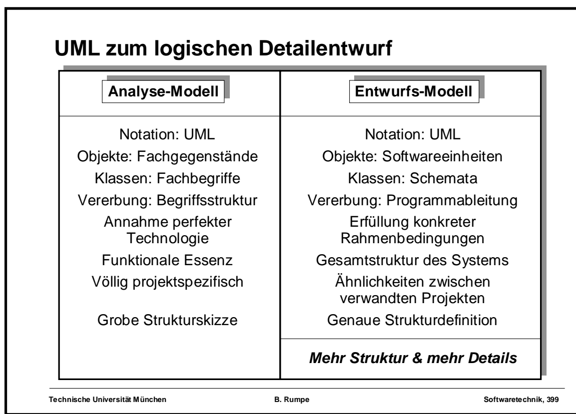
- generell: Analysemodelle werden im Entwurf umgebaut
- Insbesondere Klassendiagramme erhalten dazu eine neue Bedeutung:
  - In der Analyse repräsentieren Klassen meist Einheiten der realen Welt
  - Im Entwurf stellen Klassen einen Teil des Softwaresystems dar
  - Es findet eine Detaillierung und Präzisierung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

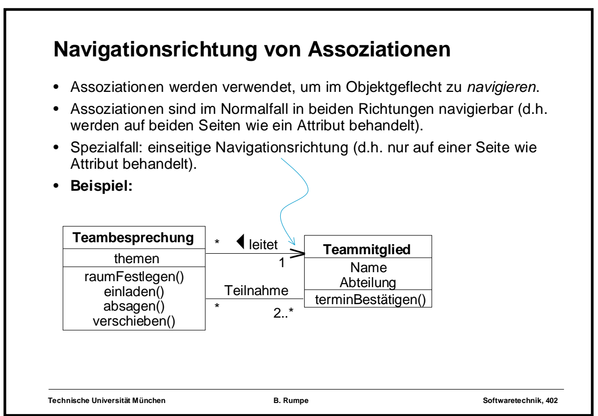


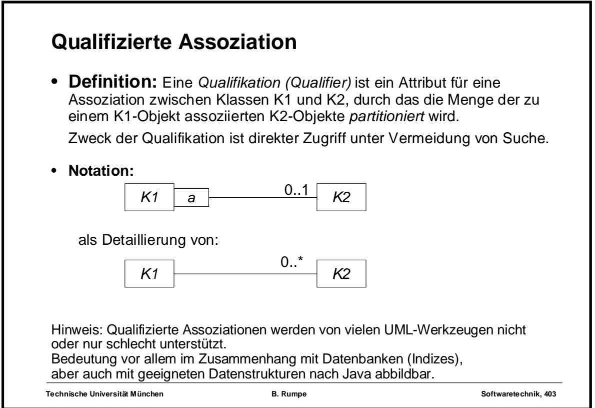
Mehr zur Sichtbarkeit

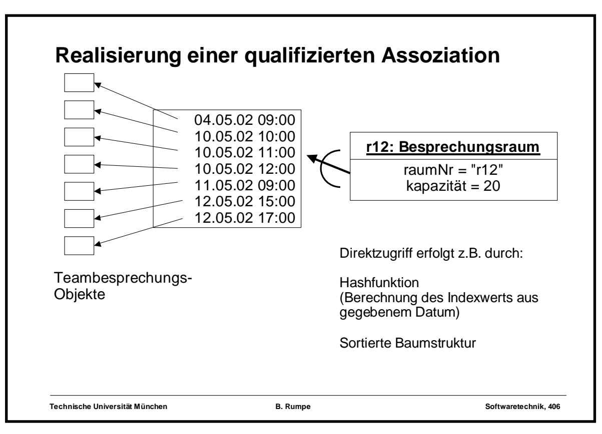
	Sichtbarkeits-Symbol			
UML / Java	+ public	# protected	– private	(default)
Sichtbar für:				
Gleiche Klasse	ja	ja	ja	ja
Andere Klasse, gleiches Paket	ja	ja / nein *	nein	ja
Andere Klasse, anderes Paket	ja	nein	nein	nein
Unterklasse, gleiches Paket	ja	ja	nein	ja
Unterklasse, anderes Paket	ja	ja	nein	nein

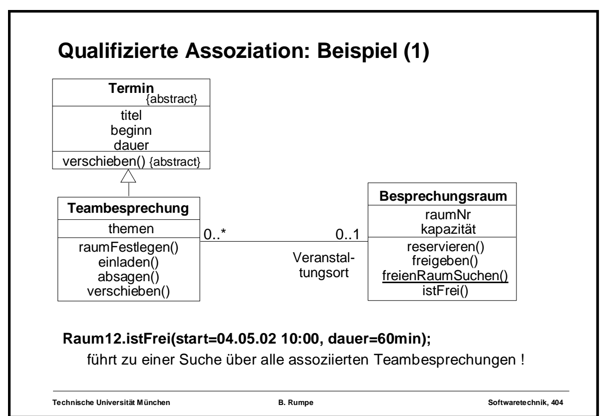
* In UML und C++ “nein”, in Java “ja”.

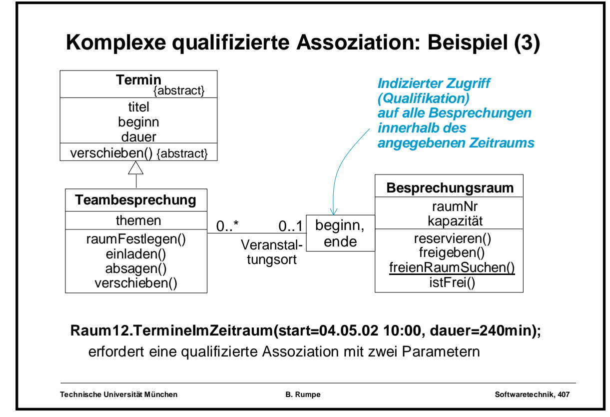


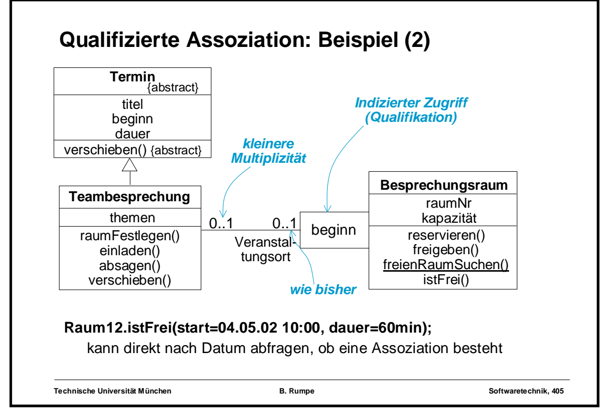


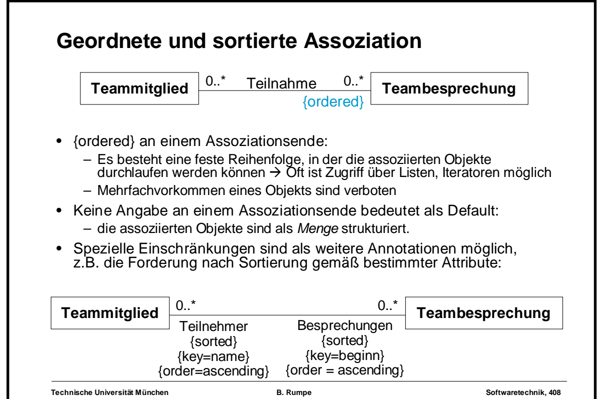

Qualifizierte Assoziation
Definition: Eine Qualifikation (Qualifier) ist ein Attribut für eine Assoziation zwischen Klassen K1 und K2, durch das die Menge der zu einem K1-Objekt assoziierten K2-Objekte partitioniert wird.
Zweck der Qualifikation ist direkter Zugriff unter Vermeidung von Suche.
Notation:
K1
a
0..1
K2
als Detaillierung von:
K1
0..*
K2
Hinweis: Qualifizierte Assoziationen werden von vielen UML-Werkzeugen nicht oder nur schlecht unterstützt.
Bedeutung vor allem im Zusammenhang mit Datenbanken (Indizes), aber auch mit geeigneten Datenstrukturen nach Java abbildbar.
Technische Universität München
B. Rumpe
Softwaretechnik, 403

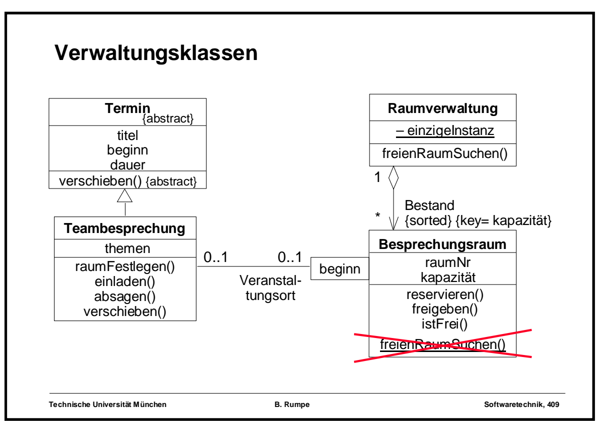

Realisierung einer qualifizierten Assoziation
04.05.02 09:00
10.05.02 10:00
10.05.02 11:00
10.05.02 12:00
11.05.02 09:00
12.05.02 15:00
12.05.02 17:00
r12: Besprechungsraum
raumNr = "r12"
kapazität = 20
Teambesprechungs-Objekte
Direktzugriff erfolgt z.B. durch:
Hashfunktion
(Berechnung des Indexwerts aus gegebenem Datum)
Sortierte Baumstruktur
Technische Universität München
B. Rumpe
Softwaretechnik, 406

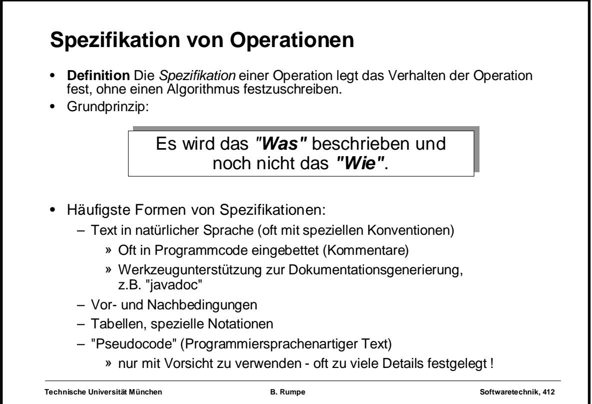

Qualifizierte Assoziation: Beispiel (1)
Termin
{abstract}
titel
beginn
dauer
verschieben() {abstract}
Teambesprechung
themen
raumFestlegen()
einladen()
absagen()
verschieben()
0..*
0..1
Veranstal-tungsort
Besprechungsraum
raumNr
kapazität
reservieren()
freigeben()
freienRaumSuchen()
istFrei()
Raum12.istFrei(start=04.05.02 10:00, dauer=60min);
führt zu einer Suche über alle assoziierten Teambesprechungen !
Technische Universität München
B. Rumpe
Softwaretechnik, 404

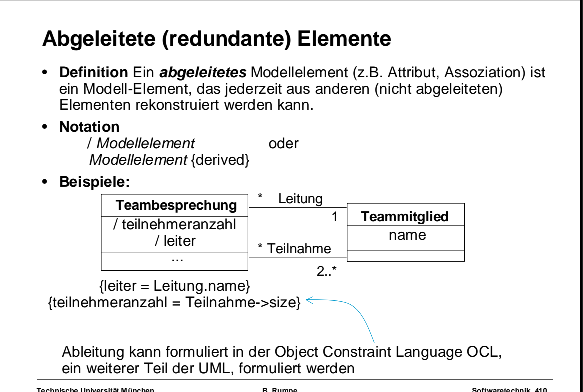

Komplexe qualifizierte Assoziation: Beispiel (3)
Termin
{abstract}
titel
beginn
dauer
verschieben() {abstract}
Indizierter Zugriff (Qualifikation) auf alle Besprechungen innerhalb des angegebenen Zeitraums
Teambesprechung
themen
raumFestlegen()
einladen()
absagen()
verschieben()
0..*
0..1
beginn, ende
Veranstal-tungsort
Besprechungsraum
raumNr
kapazität
reservieren()
freigeben()
freienRaumSuchen()
istFrei()
Raum12.TermineImZeitraum(start=04.05.02 10:00, dauer=240min);
erfordert eine qualifizierte Assoziation mit zwei Parametern
Technische Universität München
B. Rumpe
Softwaretechnik, 407

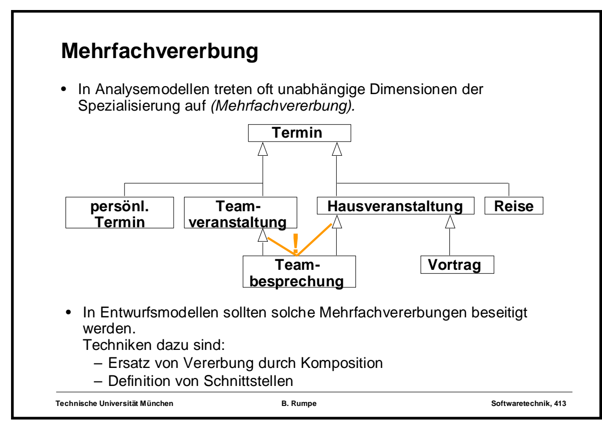

Qualifizierte Assoziation: Beispiel (2)
Termin
{abstract}
titel
beginn
dauer
verschieben() {abstract}
Indizierter Zugriff (Qualifikation)
kleinere Multiplizität
Teambesprechung
themen
raumFestlegen()
einladen()
absagen()
verschieben()
0..1
0..1
beginn
Veranstal-tungsort
wie bisher
Besprechungsraum
raumNr
kapazität
reservieren()
freigeben()
freienRaumSuchen()
istFrei()
Raum12.istFrei(start=04.05.02 10:00, dauer=60min);
kann direkt nach Datum abfragen, ob eine Assoziation besteht
Technische Universität München
B. Rumpe
Softwaretechnik, 405

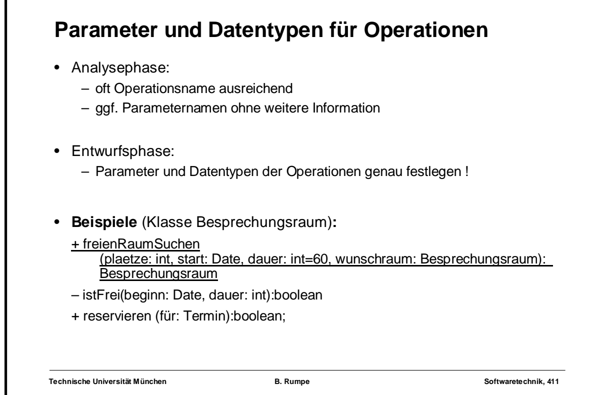

Geordnete und sortierte Assoziation
Teammitglied
0..*
Teilnahme
0..*
{ordered}
Teambesprechung
{ordered} an einem Assoziationsende:
– Es besteht eine feste Reihenfolge, in der die assoziierten Objekte durchlaufen werden können → Oft ist Zugriff über Listen, Iteratoren möglich
– Mehrfachvorkommen eines Objekts sind verboten
Keine Angabe an einem Assoziationsende bedeutet als Default:
– die assoziierten Objekte sind als Menge strukturiert.
Spezielle Einschränkungen sind als weitere Annotationen möglich, z.B. die Forderung nach Sortierung gemäß bestimmter Attribute:
Teammitglied
0..*
Teilnehmer
{sorted}
{key=name}
{order=ascending}
0..*
Besprechungen
{sorted}
{key=beginn}
{order = ascending}
Teambesprechung
Technische Universität München
B. Rumpe
Softwaretechnik, 408

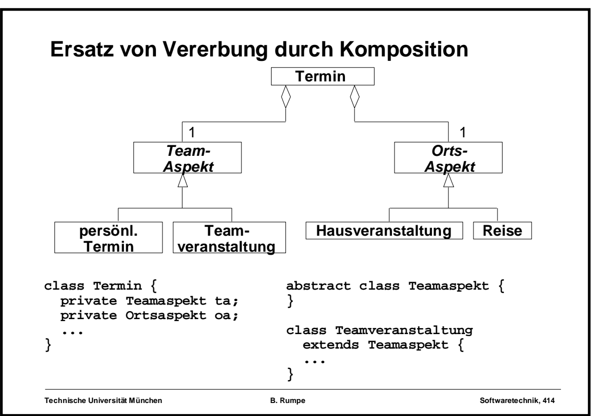

Verwaltungsklassen
Termin {abstract}
titel
beginn
dauer
verschieben() {abstract}
Teambesprechung
themen
raumFestlegen()
einladen()
absagen()
verschieben()
0..1
0..1
Veranstal-
tungsort
beginn
Raumverwaltung
– einzigeInstanz
freienRaumSuchen()
1
*
Bestand
{sorted} {key= kapazität}
Besprechungsraum
raumNr
kapazität
reservieren()
freigeben()
istFrei()
freienRaumSuchen()
Technische Universität München
B. Rumpe
Softwaretechnik, 409

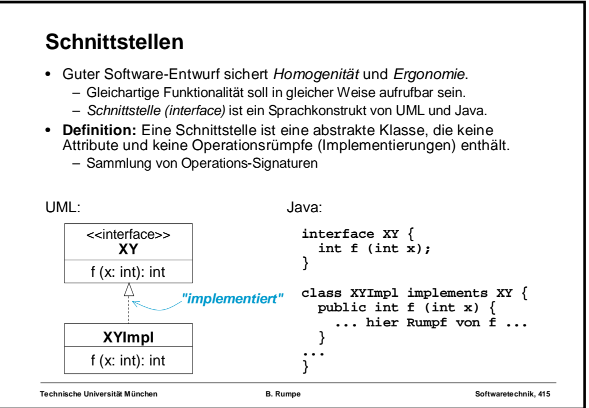

Spezifikation von Operationen
• Definition Die Spezifikation einer Operation legt das Verhalten der Operation fest, ohne einen Algorithmus festzuschreiben.
• Grundprinzip:
Es wird das "Was" beschrieben und noch nicht das "Wie".
• Häufigste Formen von Spezifikationen:
– Text in natürlicher Sprache (oft mit speziellen Konventionen)
» Oft in Programmcode eingebettet (Kommentare)
» Werkzeugunterstützung zur Dokumentationsgenerierung, z.B. "javadoc"
– Vor- und Nachbedingungen
– Tabellen, spezielle Notationen
– "Pseudocode" (Programmiersprachenartiger Text)
» nur mit Vorsicht zu verwenden - oft zu viele Details festgelegt !
Technische Universität München
B. Rumpe
Softwaretechnik, 412

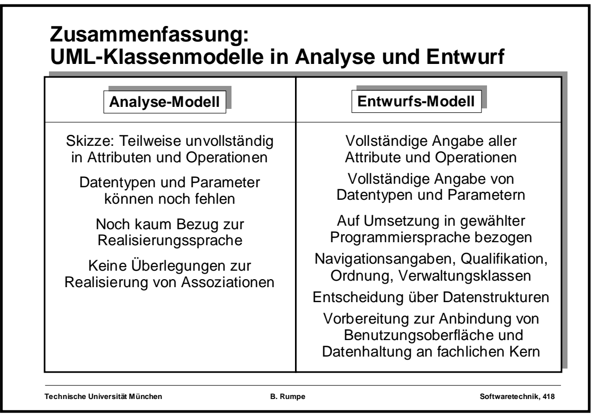

Abgeleitete (redundante) Elemente
• Definition Ein abgeleitetes Modellelement (z.B. Attribut, Assoziation) ist ein Modell-Element, das jederzeit aus anderen (nicht abgeleiteten) Elementen rekonstruiert werden kann.
• Notation
/ Modellelement oder
Modellelement {derived}
• Beispiele:
Teambesprechung
/ teilnehmeranzahl
/ leiter
...
* Leitung
1
Teammitglied
name
* Teilnahme
2..*
{leiter = Leitung.name}
{teilnehmeranzahl = Teilnahme->size}
Ableitung kann formuliert in der Object Constraint Language OCL, ein weiterer Teil der UML, formuliert werden
Technische Universität München
B. Rumpe
Softwaretechnik, 410

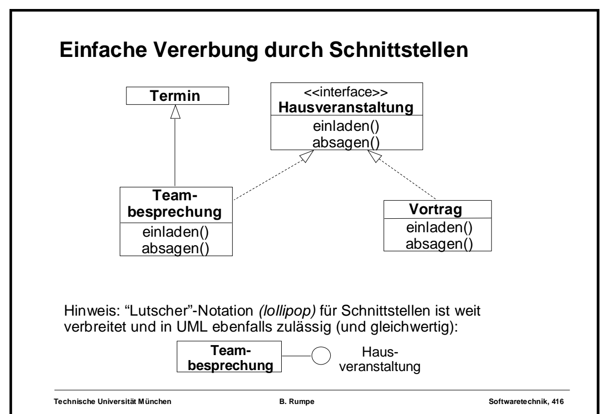

Mehrfachvererbung
• In Analysemodellen treten oft unabhängige Dimensionen der Spezialisierung auf (Mehrfachvererbung).
Termin
persönl. Termin
Team-veranstaltung
Hausveranstaltung
Reise
Team-besprechung
Vortrag
!
• In Entwurfsmodellen sollten solche Mehrfachvererbungen beseitigt werden.
Techniken dazu sind:
– Ersatz von Vererbung durch Komposition
– Definition von Schnittstellen
Technische Universität München
B. Rumpe
Softwaretechnik, 413

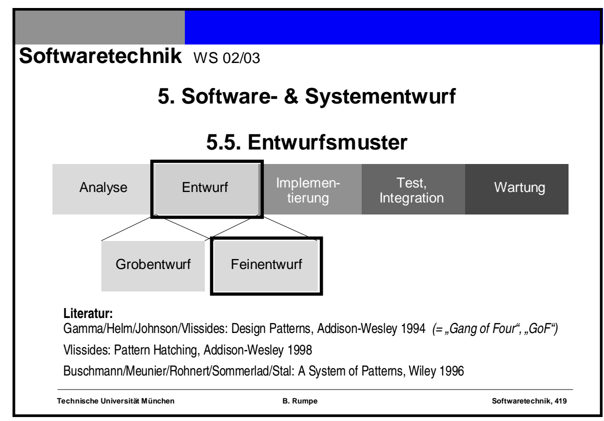

Parameter und Datentypen für Operationen
• Analysephase:
– oft Operationsname ausreichend
– ggf. Parameternamen ohne weitere Information
• Entwurfsphase:
– Parameter und Datentypen der Operationen genau festlegen !
• Beispiele (Klasse Besprechungsraum):
+ freienRaumSuchen
(plaetze: int, start: Date, dauer: int=60, wunschraum: Besprechungsraum): Besprechungsraum
– istFrei(beginn: Date, dauer: int):boolean
+ reservieren (für: Termin):boolean;
Technische Universität München
B. Rumpe
Softwaretechnik, 411


Ersatz von Vererbung durch Komposition
Termin
1
1
Team-Aspekt
Orts-Aspekt
persönl. Termin
Team-veranstaltung
Hausveranstaltung
Reise
class Termin {
private Teamaspekt ta;
private Ortsaspekt oa;
...
}
abstract class Teamaspekt {
}
class Teamveranstaltung
extends Teamaspekt {
...
}
Technische Universität München
B. Rumpe
Softwaretechnik, 414



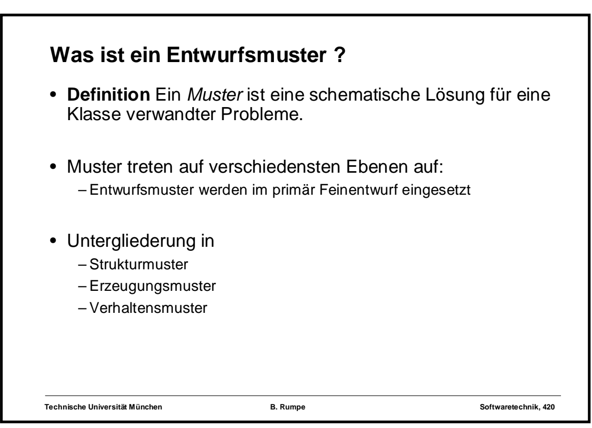






Schnittstellen und abstrakte Klassen

Abstrakte Klasse	Schnittstelle
Enthält Attribute und Operationen	Enthält nur Operationen (und ggf. Konstante)
Kann Default-Verhalten festlegen	Kann kein Default-Verhalten festlegen
Default-Verhalten kann in Unterklassen redefiniert werden	Redefinition unsinnig, da keine Definition existiert
Java: Unterklasse kann nur von einer Klasse erben	Java: Eine Klasse kann mehrere Schnittstellen implementieren
	Schnittstelle ist eine spezielle Sicht auf eine Klasse



Beschreibung eines Entwurfsmusters (nach GoF)

- Name
- Problem
  - Motivation
  - Anwendungsbereich
- Lösung
  - Struktur (Klassendiagramm)
  - Bestandteile (meist Klassen-, Assoziations- und Operationsnamen):
    - » "Rollennamen", d.h. Platzhalter für Bestandteile der Anwendung
    - » feste Bestandteile der Implementierung
  - Objektinteraktion (Abläufe, evtl. Sequenzdiagramm)
- Diskussion
  - Vor- und Nachteile
  - Abhängigkeiten, Einschränkungen
  - Spezialfälle
  - Bekannte Verwendung

Beispiel: Text zu "Adapter" (1)

Ausschnitt aus "Design Patterns":

ADAPTER

Also known as: Wrapper

Motivation:

Sometimes a toolkit class that's designed for reuse isn't reusable only because its interface doesn't match the domain-specific interface an application requires.

Consider for example a drawing editor that lets users draw and arrange graphical elements (lines, polygons, text etc) into pictures and diagrams. ... (1 Seite Text)

Applicability:

Use the Adapter pattern when

–you want to use an existing class, and its interface does not match the one you need.

–you want to create a reusable class that cooperates with unrelated or unforeseen classes ...

Beispiel: Text zu "Adapter" (2)

Ausschnitt aus "Design Patterns":

ADAPTER (Fortsetzung)

Structure: (Zwei Klassendiagramme + weitere Information)
Consequences: (1 Seite Text)
Implementation: (2 Seiten Text mit Klassendiagrammen)
Sample Code: (3 Seiten Text mit C++-Beispielen)
Known Uses: (1 Seite Text)
Related Patterns: (1/4 Seite Text)

Erzeugungsmuster (nach „Gang of Four“)

- *Mustername:* *variabel gemachter Aspekt*

- Factory Method: Implementierungsvarianten
- Abstract Factory: Familien von Implementierungsvarianten
- Builder: Repräsentation komplexer Objekte
- Prototype: Repräsentation komplexer Objekte
- Singleton: globaler Konstruktorzugriff mit Sicherstellung, dass nur eine Instanz erzeugt wird

Strukturmuster (nach „Gang of Four“)

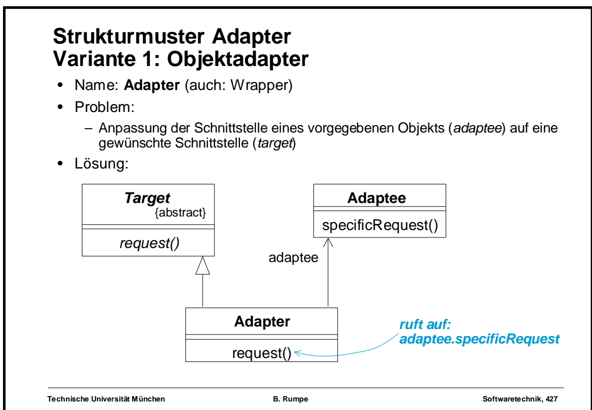
- *Mustername:* *variabel gemachter Aspekt*

- Adapter: Schnittstelle zu konkretem Objekt
- Bridge: Implementierung eines Objekts
- Composite: Objektstruktur eines komplexen Objekts
- Decorator: Verantwortlichkeiten, Schnittstelle
- Facade: Schnittstelle eines Teilsystems
- Flyweight: Speicheraufwand für ein Objekt
- Proxy: Zugang zu einem Objekt und Ort eines Objekts

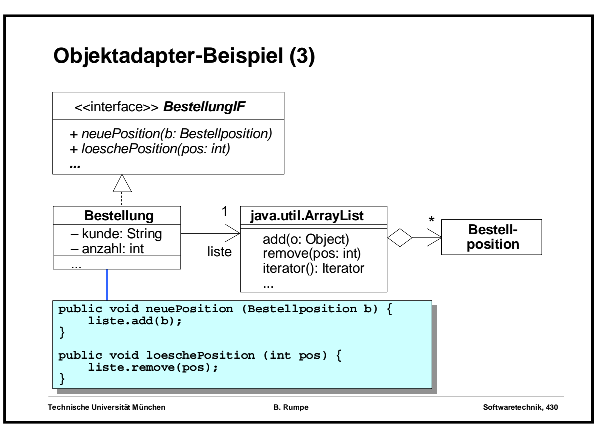
Verhaltensmuster (nach „Gang of Four“)

- *Mustername:* *variabel gemachter Aspekt*

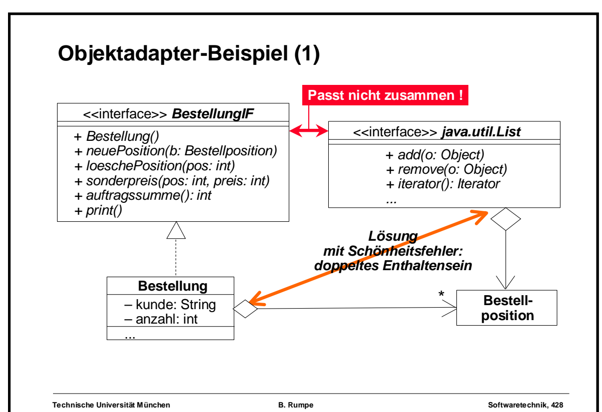
- Chain of Responsibility: Objektkopplung
- Command: Konstruktion von Methodenaufrufen
- Interpreter: Abstraktionsebene
- Iterator: Durchlauf(reihenfolge) bei Kollektionen
- Mediator: Objektkopplung
- Memento: Reversibilität
- Observer: Objektkopplung (Zustandsveränderungen)
- State: zustandsabhängiges Verhalten
- Strategy: Algorithmenverwendung
- Template Method: Verwendung von Funktionsbausteinen
- Visitor: Durchlauf komplexer Objektstrukturen



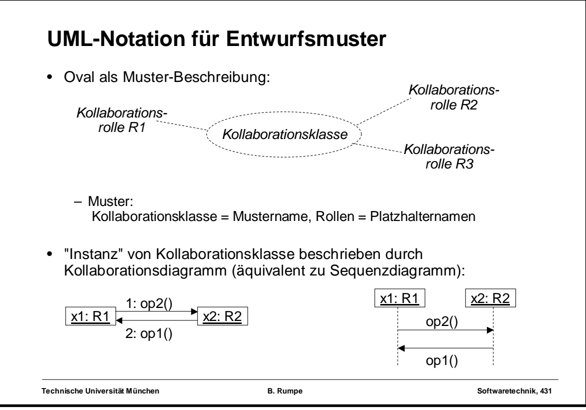
Strukturmuster Adapter
Variante 1: Objektadapter
Name: Adapter (auch: Wrapper)
Problem:
Anpassung der Schnittstelle eines vorgegebenen Objekts (adaptee) auf eine gewünschte Schnittstelle (target)
Lösung:
Target {abstract}
request()
Adaptee
specificRequest()
adaptee
Adapter
request()
ruft auf: adaptee.specificRequest
Technische Universität München
B. Rumpe
Softwaretechnik, 427



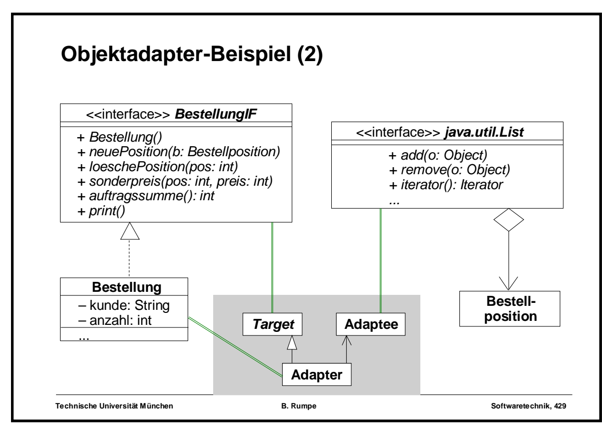
Objektadapter-Beispiel (3)
<<interface>> BestellungIF
+ neuePosition(b: Bestellposition)
+ loeschePosition(pos: int)
...
Bestellung
– kunde: String
– anzahl: int
...
1
liste
java.util.ArrayList
add(o: Object)
remove(pos: int)
iterator(): Iterator
...
*
Bestell-position
public void neuePosition (Bestellposition b) {
liste.add(b);
}
public void loeschePosition (int pos) {
liste.remove(pos);
}
Technische Universität München
B. Rumpe
Softwaretechnik, 430



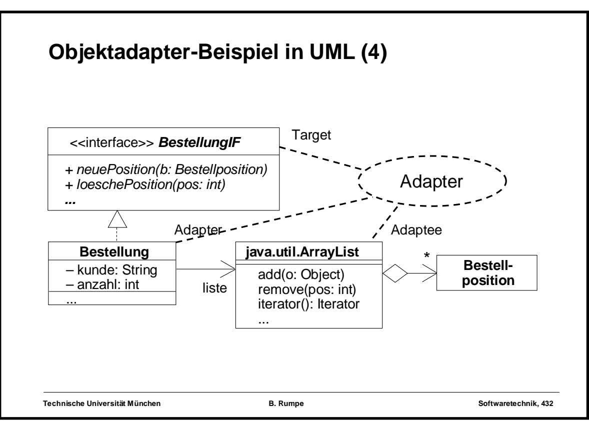
Objektadapter-Beispiel (1)
Passt nicht zusammen !
<<interface>> BestellungIF
+ Bestellung()
+ neuePosition(b: Bestellposition)
+ loeschePosition(pos: int)
+ sonderpreis(pos: int, preis: int)
+ auftragssumme(): int
+ print()
<<interface>> java.util.List
+ add(o: Object)
+ remove(o: Object)
+ iterator(): Iterator
...
Lösung
mit Schönheitsfehler:
doppeltes Enthaltensein
Bestellung
– kunde: String
– anzahl: int
...
*
Bestell-position
Technische Universität München
B. Rumpe
Softwaretechnik, 428



UML-Notation für Entwurfsmuster
Oval als Muster-Beschreibung:
Kollaborations-rolle R1
Kollaborationsklasse
Kollaborations-rolle R2
Kollaborations-rolle R3
Muster:
Kollaborationsklasse = Mustername, Rollen = Platzhalternamen
"Instanz" von Kollaborationsklasse beschrieben durch Kollaborationsdiagramm (äquivalent zu Sequenzdiagramm):
x1: R1
1: op2()
x2: R2
2: op1()
x1: R1
x2: R2
op2()
op1()
Technische Universität München
B. Rumpe
Softwaretechnik, 431



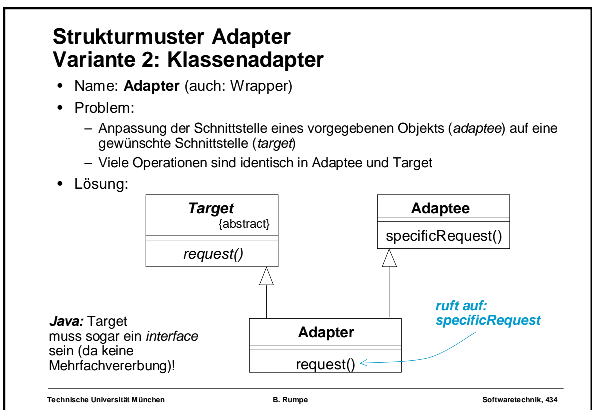
Objektadapter-Beispiel (2)
<<interface>> BestellungIF
+ Bestellung()
+ neuePosition(b: Bestellposition)
+ loeschePosition(pos: int)
+ sonderpreis(pos: int, preis: int)
+ auftragssumme(): int
+ print()
<<interface>> java.util.List
+ add(o: Object)
+ remove(o: Object)
+ iterator(): Iterator
...
Bestellung
– kunde: String
– anzahl: int
...
Target
Adaptee
Adapter
Bestell-position
Technische Universität München
B. Rumpe
Softwaretechnik, 429

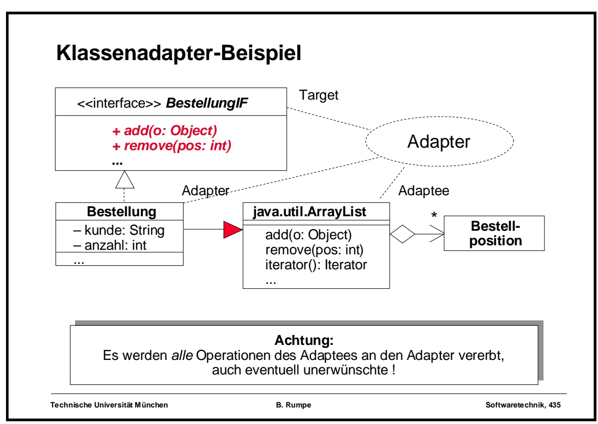


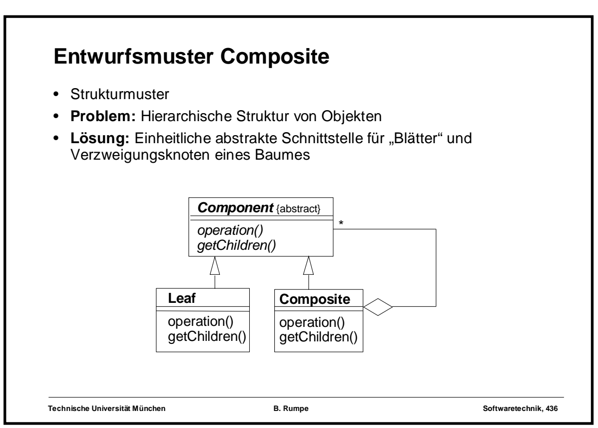
Objektadapter-Beispiel in UML (4)
<<interface>> BestellungIF
+ neuePosition(b: Bestellposition)
+ loeschePosition(pos: int)
...
Target
Adapter
Adapter
Adaptee
Bestellung
– kunde: String
– anzahl: int
...
liste
java.util.ArrayList
add(o: Object)
remove(pos: int)
iterator(): Iterator
...
*
Bestell-position
Technische Universität München
B. Rumpe
Softwaretechnik, 432

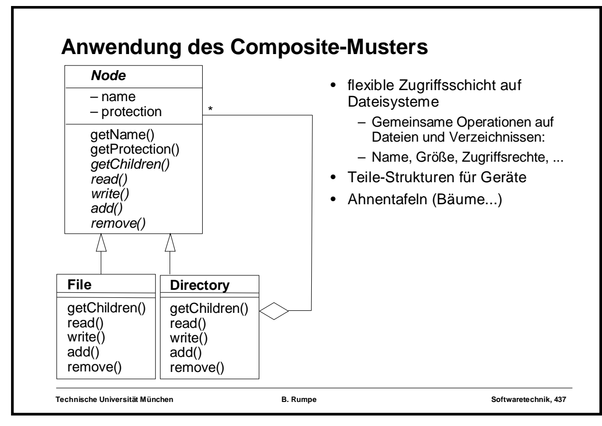
Anwendung eines Musters

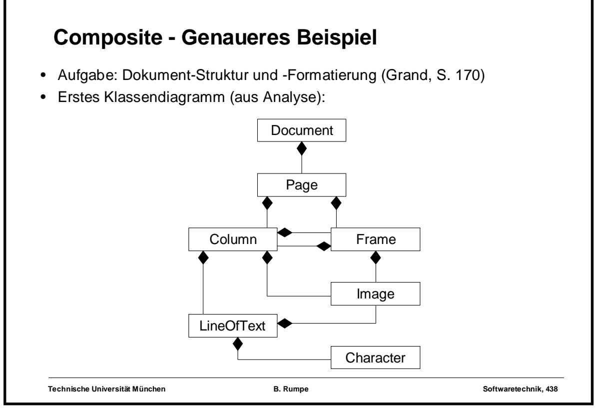
- Kein mechanisches "Pattern Matching"!
  - Eher Übertragung der Idee des Musters
- Grundstruktur des Musters sollte sich wiederfinden lassen
  - ggf. vorliegenden Entwurf etwas anpassen bzw. anders darstellen
- Auch Verhaltensschema muss im Code ähnlich zur Musteridee auftreten:
  - Muster:
    Adapter.request() ruft Adaptee.specificRequest() auf
    (und tut nicht wesentlich mehr)
  - Konkreter Fall:
    Bestellung.neuePosition() ruft ArrayList.add() auf
    Bestellung.loeschePosition() ruft ArrayList.remove() auf
    ...

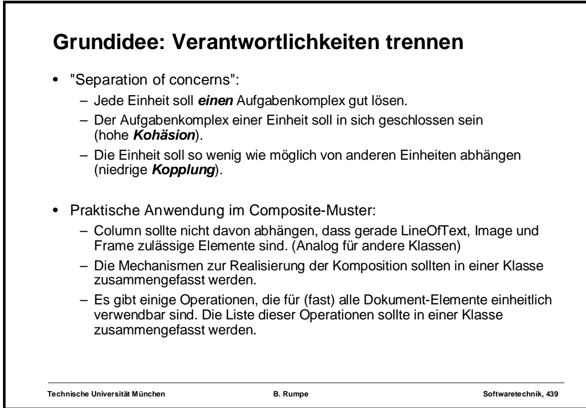


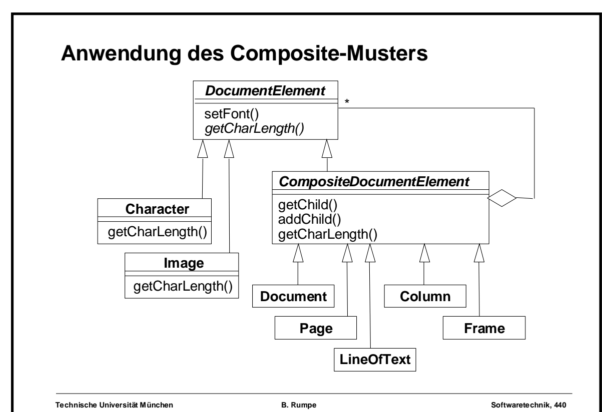


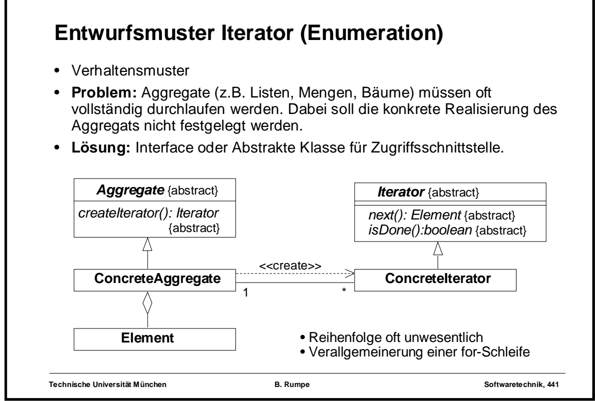


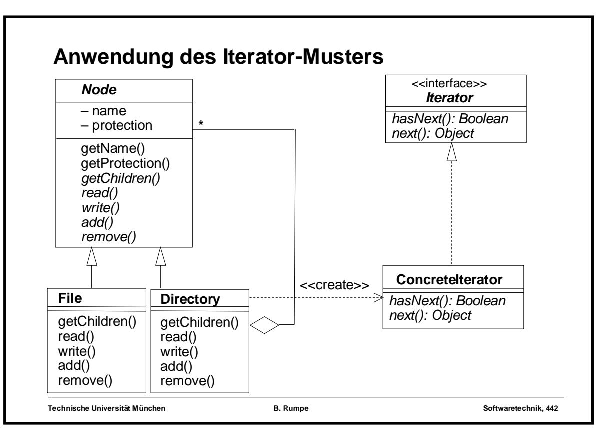


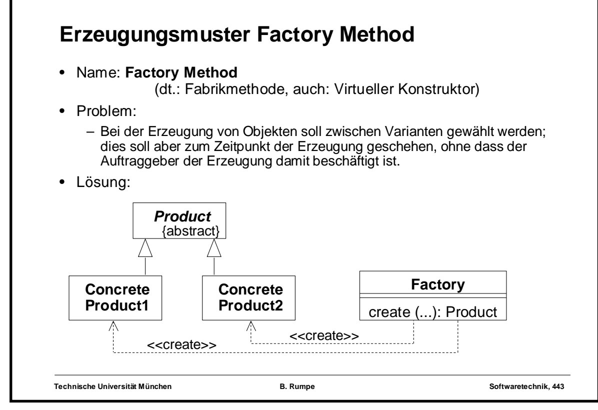


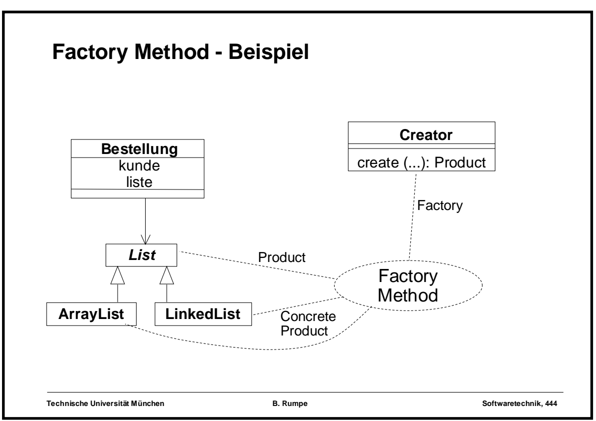

Grundidee: Verantwortlichkeiten trennen
• "Separation of concerns":
– Jede Einheit soll einen Aufgabenkomplex gut lösen.
– Der Aufgabenkomplex einer Einheit soll in sich geschlossen sein (hohe Kohäsion).
– Die Einheit soll so wenig wie möglich von anderen Einheiten abhängen (niedrige Kopplung).
• Praktische Anwendung im Composite-Muster:
– Column sollte nicht davon abhängen, dass gerade LineOfText, Image und Frame zulässige Elemente sind. (Analog für andere Klassen)
– Die Mechanismen zur Realisierung der Komposition sollten in einer Klasse zusammengefasst werden.
– Es gibt einige Operationen, die für (fast) alle Dokument-Elemente einheitlich verwendbar sind. Die Liste dieser Operationen sollte in einer Klasse zusammengefasst werden.
Technische Universität München
B. Rumpe
Softwaretechnik, 439


Anwendung des Composite-Musters
DocumentElement
setFont()
getCharLength()
*
Character
getCharLength()
Image
getCharLength()
CompositeDocumentElement
getChild()
addChild()
getCharLength()
Document
Column
Page
Frame
LineOfText
Technische Universität München
B. Rumpe
Softwaretechnik, 440

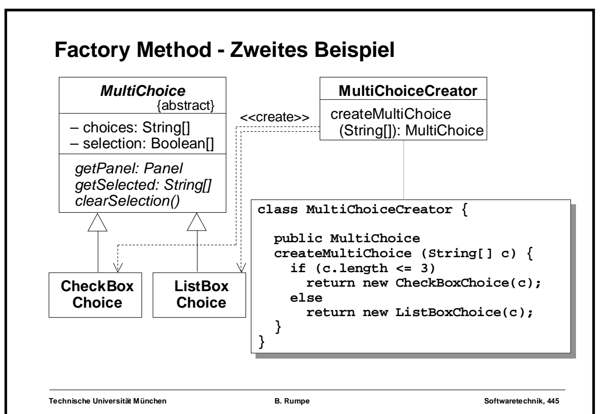

Entwurfsmuster Iterator (Enumeration)
• Verhaltensmuster
• Problem: Aggregate (z.B. Listen, Mengen, Bäume) müssen oft vollständig durchlaufen werden. Dabei soll die konkrete Realisierung des Aggregats nicht festgelegt werden.
• Lösung: Interface oder Abstrakte Klasse für Zugriffsschnittstelle.
Aggregate {abstract}
createIterator(): Iterator {abstract}
Iterator {abstract}
next(): Element {abstract}
isDone():boolean {abstract}
ConcreteAggregate
<<create>>
ConcreteIterator
1
*
Element
• Reihenfolge oft unwesentlich
• Verallgemeinerung einer for-Schleife
Technische Universität München
B. Rumpe
Softwaretechnik, 441


Anwendung des Iterator-Musters
Node
– name
– protection
getName()
getProtection()
getChildren()
read()
write()
add()
remove()
*
<<interface>>
Iterator
hasNext(): Boolean
next(): Object
File
getChildren()
read()
write()
add()
remove()
Directory
getChildren()
read()
write()
add()
remove()
<<create>>
ConcreteIterator
hasNext(): Boolean
next(): Object
Technische Universität München
B. Rumpe
Softwaretechnik, 442


Erzeugungsmuster Factory Method
• Name: Factory Method
(dt.: Fabrikmethode, auch: Virtueller Konstruktor)
• Problem:
– Bei der Erzeugung von Objekten soll zwischen Varianten gewählt werden; dies soll aber zum Zeitpunkt der Erzeugung geschehen, ohne dass der Auftraggeber der Erzeugung damit beschäftigt ist.
• Lösung:
Product
{abstract}
Concrete Product1
Concrete Product2
Factory
create (...): Product
<<create>>
<<create>>
Technische Universität München
B. Rumpe
Softwaretechnik, 443

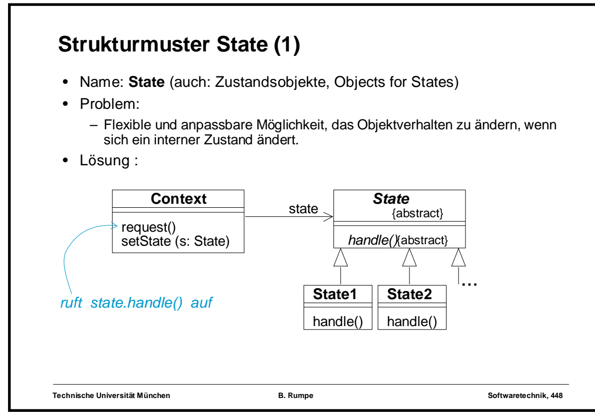

Factory Method - Beispiel
Bestellung
kunde
liste
Creator
create (...): Product
Factory
List
Product
Factory Method
ArrayList
LinkedList
Concrete Product
Technische Universität München
B. Rumpe
Softwaretechnik, 444

Factory Method - Zweites Beispiel



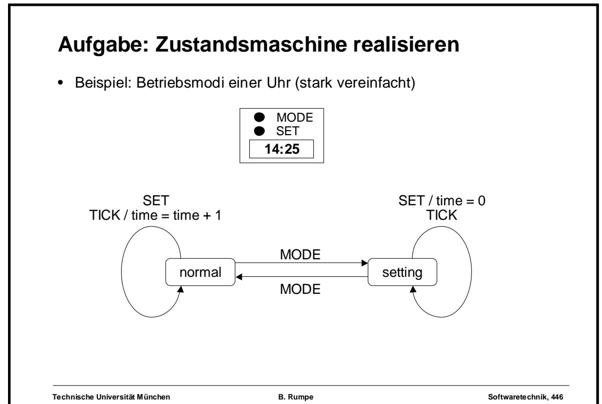
Strukturmuster State (1)

- Name: **State** (auch: Zustandsobjekte, Objects for States)
- Problem:
  - Flexible und anpassbare Möglichkeit, das Objektverhalten zu ändern, wenn sich ein interner Zustand ändert.
- Lösung :

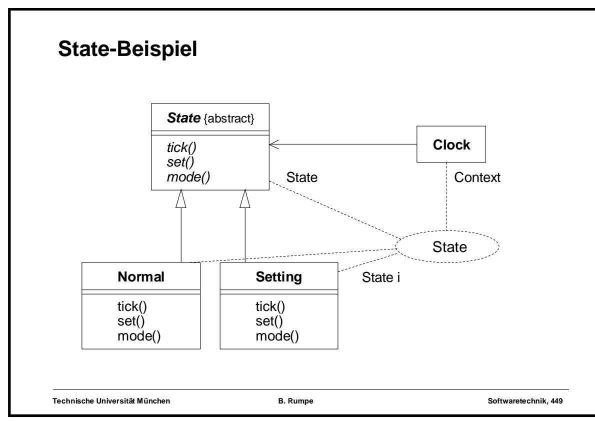


Aufgabe: Zustandsmaschine realisieren

- Beispiel: Betriebsmodi einer Uhr (stark vereinfacht)



State-Beispiel

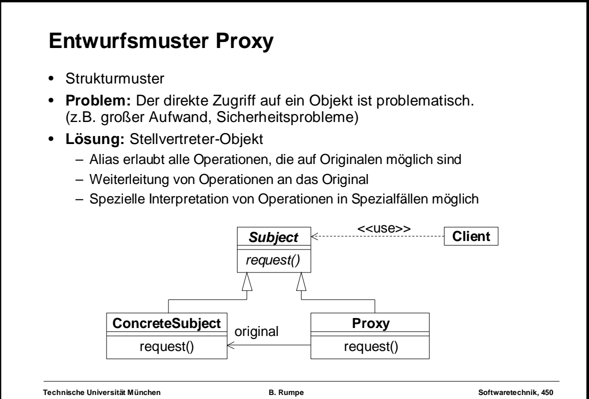


Implementierung mit Fallunterscheidungen

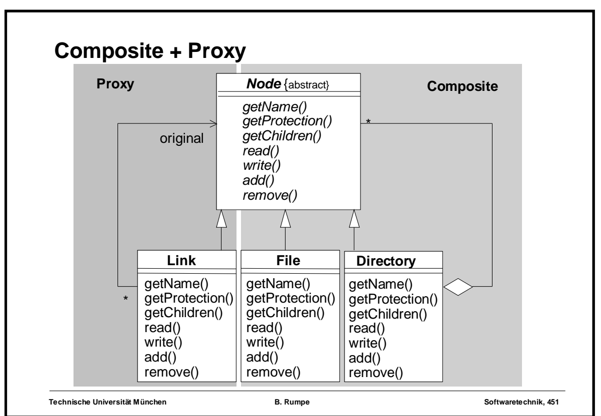
```
class Clock {

   private int time = 0;
   private static final int NORMAL = 0;
   private static final int SETTING = 1;

   private int state = NORMAL;

   public void set () {
       switch (state) {
             case NORMAL: {
                   break;
             };
             case SETTING: {
                   time = 0;
                   setChanged();
                   break;
             };
       };
   }

   ...// analog tick(), mode()
}
```


Entwurfsmuster Proxy

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

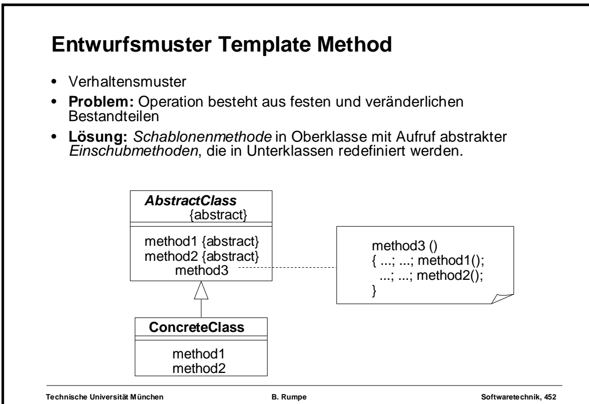


Composite + Proxy



Entwurfsmuster Template Method

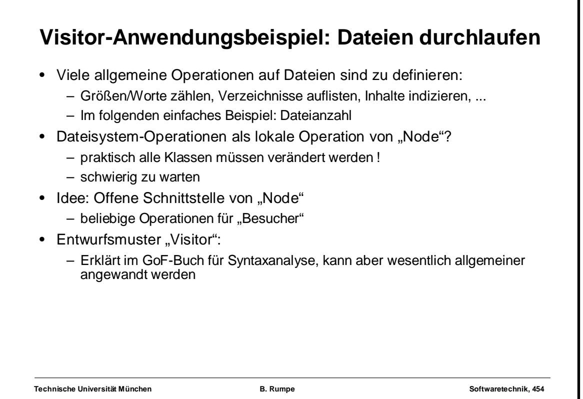
- Verhaltensmuster
- **Problem:** Operation besteht aus festen und veränderlichen Bestandteilen
- **Lösung:** *Schablonenmethode* in Oberklasse mit Aufruf abstrakter *Einschubmethoden*, die in Unterklassen redefiniert werden.



Template Method:

- Anwendungsgebiete bei Frameworks, die durch Subklassenbildung angepasst werden
- Beispiel: Schutz gegen versehentliches Überschreiben
  - Operation zum Überschreiben von Datei / Verzeichnis / Link:
  - unveränderlicher Anteil für alle Knoten
    - » Schreibvorgang
    - » Grundsätzliche Logik
  - unterschiedlicher Anteil je nach Unterklasse
    - » genaue Kriterien für Zulässigkeit
    - » spezielle Fehlermeldung
  - Beispiel:
    - » aDirectory.write(x) liefert immer Fehlermeldung „directory can't be written“
    - » aFile.write(x) prüft lokale Zugriffsrechte und gibt ggf. Fehlermeldung „file is read-only“

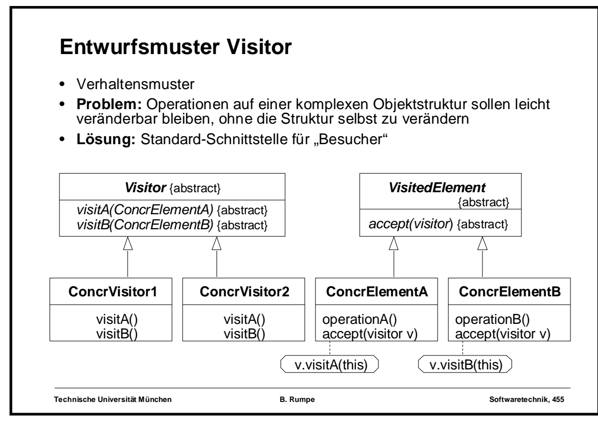
Visitor-Anwendungsbeispiel: Dateien durchlaufen



- Viele allgemeine Operationen auf Dateien sind zu definieren:
  - Größen/Worte zählen, Verzeichnisse auflisten, Inhalte indizieren, ...
  - Im folgenden einfaches Beispiel: Dateianzahl
- Dateisystem-Operationen als lokale Operation von „Node“?
  - praktisch alle Klassen müssen verändert werden !
  - schwierig zu warten
- Idee: Offene Schnittstelle von „Node“
  - beliebige Operationen für „Besucher“
- Entwurfsmuster „Visitor“:
  - Erklärt im GoF-Buch für Syntaxanalyse, kann aber wesentlich allgemeiner angewandt werden

Entwurfsmuster Visitor

- Verhaltensmuster
- **Problem:** Operationen auf einer komplexen Objektstruktur sollen leicht veränderbar bleiben, ohne die Struktur selbst zu verändern
- **Lösung:** Standard-Schnittstelle für „Besucher“



Beispiel-Code zu Visitor (1)

- Entwurfsmuster haben direkte Implikationen für den Code.
- Sie können aber oft in Varianten realisiert werden. Hier ein Beispiel zum Visitor:

```
abstract class Node { ...
 public abstract void accept(NodeVisitor v);
}
class File extends Node { ...
  void accept (NodeVisitor v) {
    v.visit(this);
} …}
class Directory extends Node { ...
  void accept (NodeVisitor v) {
    v.visit(this);
} …}
class Link extends Node { ...
  void accept (NodeVisitor v) {
    v.visit(this);
} …}
interface NodeVisitor {
  void visit (File f);
  void visit (Directory d);
  void visit (Link l);
}
```

Beispiel-Code zu Visitor (2)

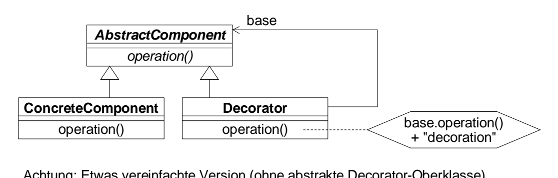
```
class FileCountVisitor implements NodeVisitor {
  private int fileCount = 0;
  public void visit(File f) {
    fileCount ++;
  }
  public void visit(Directory d) {
    Iterator it = d.getChildren();
    while (it.hasNext()) {
      ((Node)it.next()).accept(this);
    }
  }
  public void visit(Link l) {
    l.getOriginal().accept(this);
  }
  public void resetFileCount() {
    fileCount = 0;
  }
  public int getFileCount() {
    return fileCount;
  }
}
```


Mehrfachzählungen verhindern

- In vorhergehendem Beispiel-Code werden Dateien so oft gezählt, wie sie (über irgendwelche Alias-Pfade) erreichbar sind.
- Möglichst flexible Umschaltung zwischen den Strategien "auf jedem Pfad besuchen" und "nur einmal besuchen" ist wünschenswert.
  - Allgemein anwendbar für verschiedenste Besucher
  - Eventuell sogar dynamisch umschaltbar...

Entwurfsmuster Decorator

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



Achtung: Etwas vereinfachte Version (ohne abstrakte Decorator-Oberklasse)

Entwurfsmuster Singleton

- Erzeugungsmuster
- **Problem:** Manche Klassen sind nur sinnvoll, wenn sichergestellt ist, dass immer höchstens eine Instanz der Klasse besteht (und diese bei Bedarf erzeugt wird).
- **Lösung:**
  - Modellebene: Klassen als Singleton auszeichnen
  - Programmebene: Sprachabhängig
- Beispiel (Java):

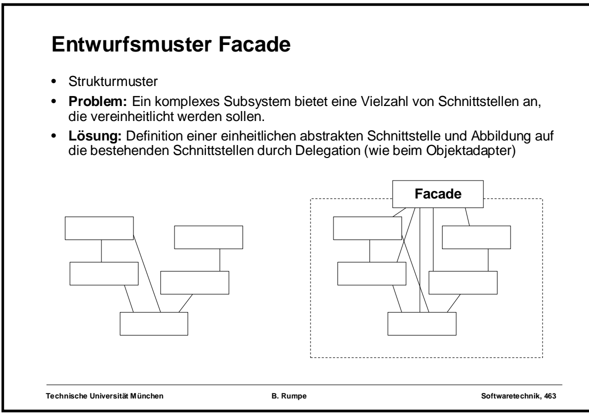
```
class Singleton {
  private static Singleton theInstance;
  private Singleton () {
  }
  public static Singleton getInstance() {
    if (theInstance==null)
      theInstance = new Singleton();
    return theInstance;
  }
}
```


Entwurfsmuster Singleton

- Beispiele:
  - Protokoll-, DB- Schnittstellen
  - „Manager“-Klassen sind meist Singletons.
- Lösungen sind technisch verschieden je nach Programmiersprache, umfassen aber immer die selben wesentlichen Elemente (Variable theInstance, Modifikationen beim Konstruktor).

Schnittstellen für Dateisysteme

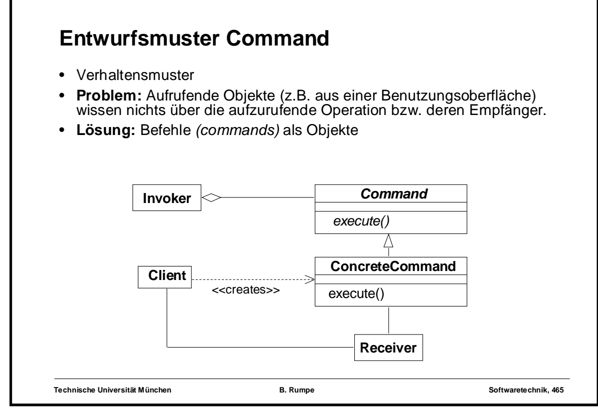
- Dateisystem-API enthält verschiedenste Klassen, deren Zusammenwirken nicht ganz einfach zu verstehen ist.
- Insbesondere die zulässigen Folgen zum Erzeugen von Dateistrukturen sind nicht ganz klar.
- In echten Dateisystemen: Einheitliche Schnittstelle zum Umgang mit den verschiedenen Phänomenen Datei, Verzeichnis, Alias, ...

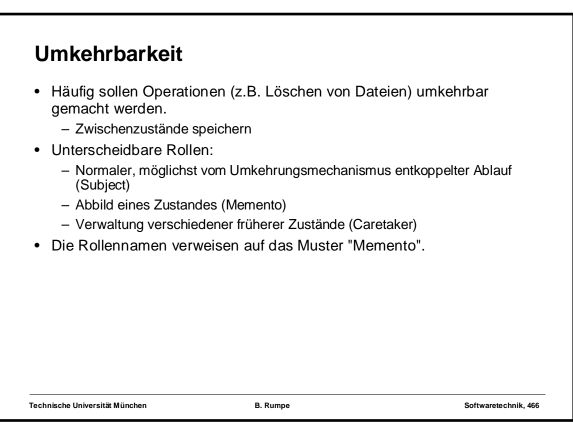


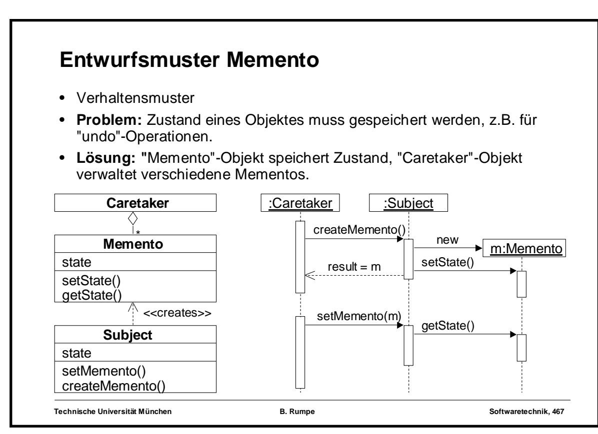
Speicherbare Aufrufe

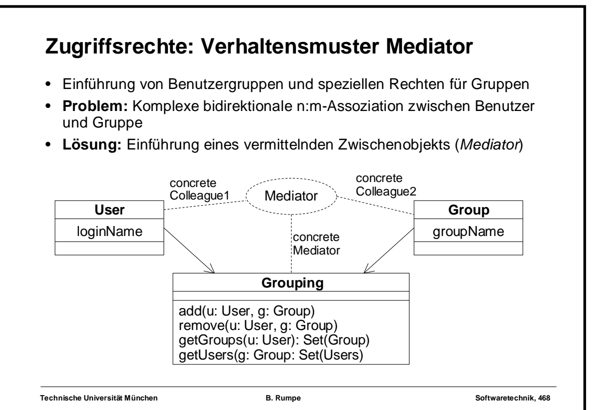
- Es kann notwendig sein, Aufrufe an universellen Fassaden zu speicherbaren und übertragbaren Objekten zu machen:
  - Methodenaufrufe zur späteren Ausführung abspeichern (Callback)
  - Verteilte Systeme: Aufrufe übermitteln und in Warteschlangen verwalten
  - Aufzeichnen von Aufrufsequenzen (Recovery-Vorsorge)
  - Implizite Beschreibung eines komplexen Zustands durch Änderungen gegenüber einem Grundzustand.
- Command-Muster: Aufrufe als Objekte
- Command ist (zusammen mit Memento) das wichtigste Muster für die Realisierung von "undo"-Operationen.

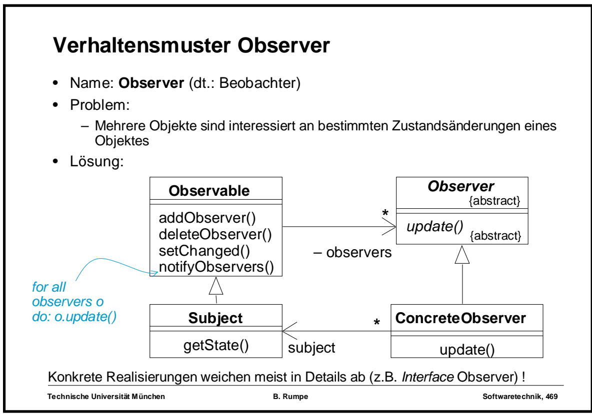
Technische Universität München B. Rumpe Softwaretechnik, 464

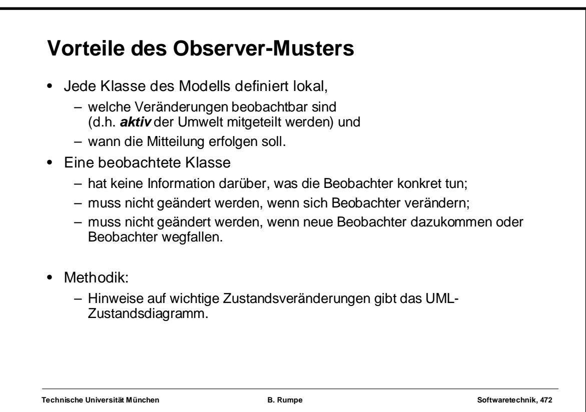








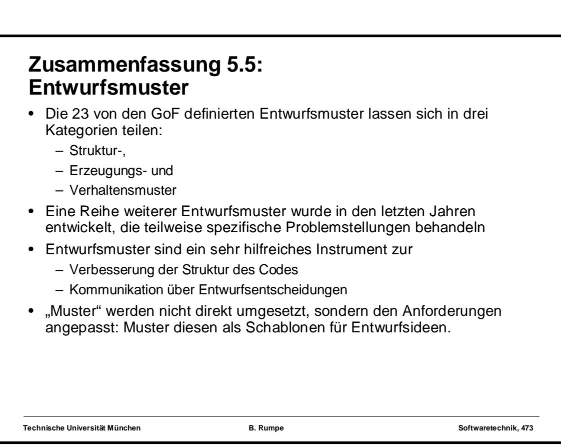


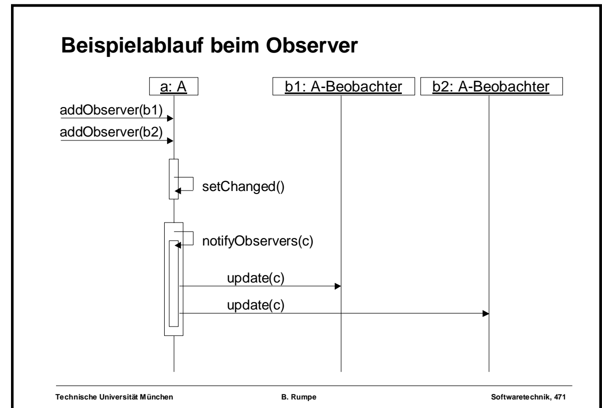


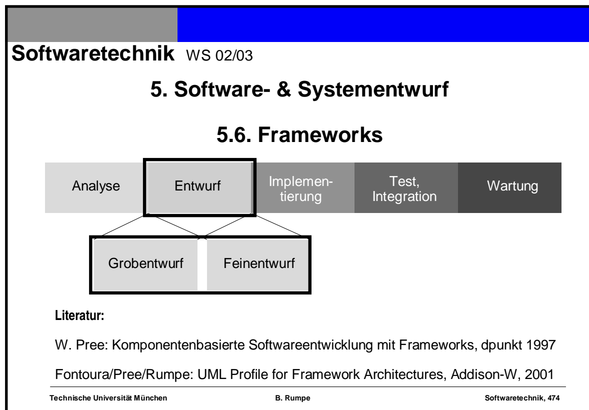
java.util.Observable, java.util.Observer

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

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

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

Argumente für notifyObservers():
– meist nur Art der Änderung, nicht gesamte Zustandsinformation
– Beobachter können normale Methodenaufrufe nutzen, um sich näher zu informieren.







Wiederverwendung vs. Neuentwicklung

- Typische Programmiereransicht:
  "Wiederverwendung kostet genauso viel Aufwand wie neu schreiben."
- Diese Ansicht ist teilweise richtig !

Aufwand für Wiederverwendung

(als Anteil des Neuentwicklungs-Aufwands)

100%

55%

12%

100%

Umfang der Anpassungen

(an wiederverwendeter Komponente)

Quelle: NASA-Studie 1994

Technische Universität München — B. Rumpe — Softwaretechnik, 475

Technologien für Wiederverwendung

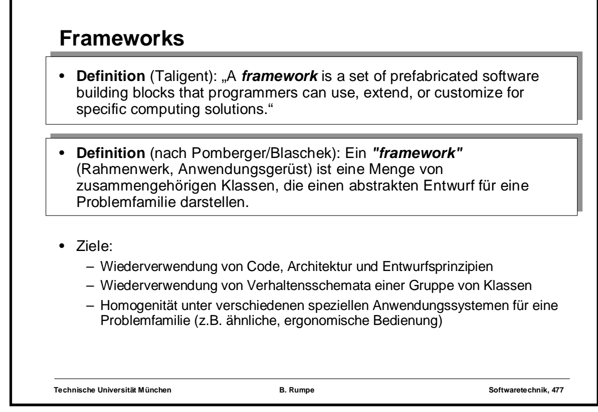
- Bausteinsysteme (*componentware*)
- Frameworks
- Architekturmuster und Referenzarchitekturen
- Entwurfsmuster
- Klassenbibliotheken

Abnehmende Festlegung der Architektur

- Grundidee: Softwareentwurf *für* Wiederverwendung
- Vielfältiger Einsatz von Mustern:

Anwendungs-programm → Muster-verwendung → Application Framework → Muster-verwendung → Baustein-system

Technische Universität München — B. Rumpe — Softwaretechnik, 476



Klassifikation nach Einsatzart

- Anwendungs-Framework (*Application Framework*):
  - Rahmen für eine ganze Anwendung
  - Beinhalten Expertenwissen zur Systemarchitektur, das auf ähnlich geartete Programme anwendbar ist
    - » z.B. GUI-Frameworks
    - » "micro-framework" = Rahmenlösung für Teilsystem (z.B. Datenbankanbindung)
- Bereichsspezifisches Framework (*Domain Framework*):
  - Beinhalten Expertenwissen zu speziellem Anwendungsbereich
    - » z.B. Framework für Anlagensteuerungen
    - » z.B. Framework für betriebswirtschaftliche Anwendungen
    - » z.B. Multimedia-Framework
- Infrastruktur-Framework (*Support Framework*):
  - Bereitstellung von Systemdiensten
    - » z.B. Framework zur Anpassung von Gerätetreibern

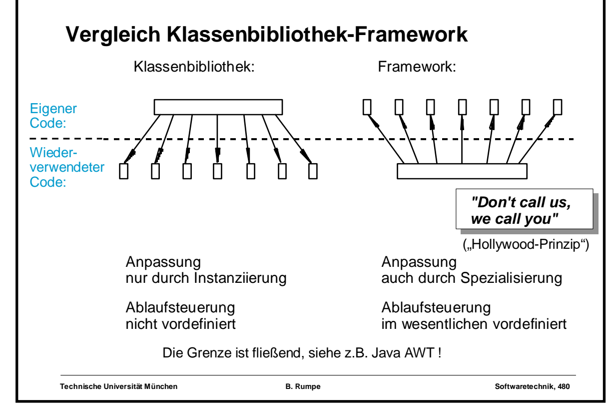
Technische Universität München — B. Rumpe — Softwaretechnik, 478

Klassifikation nach Architektur

- Architektur-getriebenes Framework (*architecture-driven*):
  - Anpassung durch Vererbung und Redefinieren
  - Komplexe Klassenhierarchien und Muster
  - Relativ viel neuer Code zu schreiben
  - Anpassung erfordert sehr hohen Einarbeitungsaufwand
- Daten-getriebenes Framework (*data-driven*):
  - Anpassung durch Objektkonfiguration und Parametereinstellung
  - Weitergeleitete Objekte bestimmen Verhalten (z.B. Ereignis-Objekte)
  - Relativ einfach anzupassen
  - Eingeschränkte Flexibilität — *vgl. Komponentensysteme!*
- Möglicher und sinnvoller Kompromiss:
  - Zwei-Schichten-Architektur:

daten-getrieben
architektur-getrieben

Technische Universität München — B. Rumpe — Softwaretechnik, 479



Beispiel: Schließbares Fenster in Java AWT

```
import java.awt.*;
import java.awt.event.*;
class WindowCloser extends WindowAdapter {
  public void windowClosing(WindowEvent event) {
    System.exit(0);
  }
}
class ExampleFrame extends Frame {
  public ExampleFrame () {
      setTitle("untitled");
      setSize(150, 50);
      addWindowListener(new WindowCloser());
      setVisible(true);
   }
}
class GUI1 {
public static void main (String[] argv) {
  ExampleFrame f = new ExampleFrame();}}
```

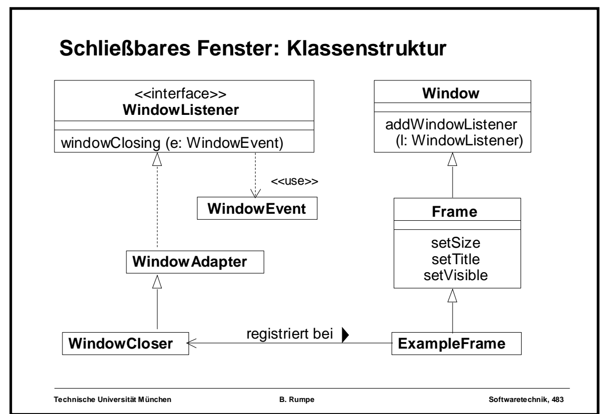

java.awt.event.WindowListener

```
public interface WindowListener extends EventListener {
  public void windowClosed (WindowEvent ev);
  public void windowOpened (WindowEvent ev);
  public void windowIconified (WindowEvent ev);
  public void windowDeiconified (WindowEvent ev);
  public void windowActivated (WindowEvent ev);
  public void windowDeactivated (WindowEvent ev);
  public void windowClosing (WindowEvent ev);
}
```

java.util.EventListener:

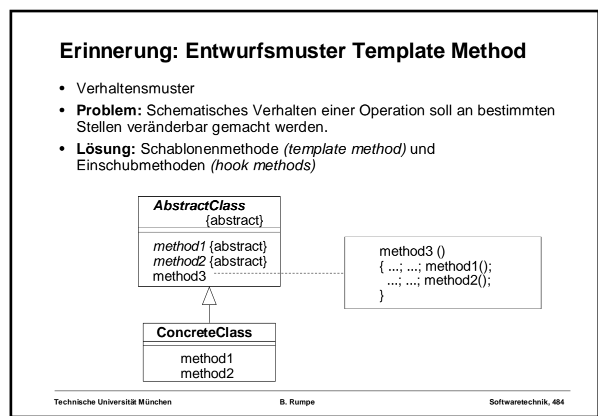
Basis für alle "Listener"
Keine Operationen

Schließbares Fenster: Klassenstruktur



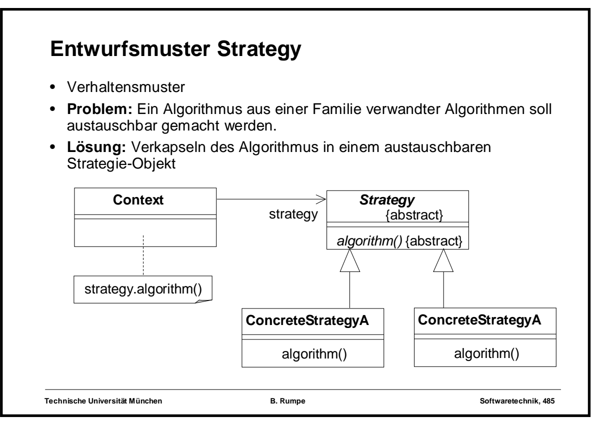
Erinnerung: Entwurfsmuster Template Method

- Verhaltensmuster
- **Problem:** Schematisches Verhalten einer Operation soll an bestimmten Stellen veränderbar gemacht werden.
- **Lösung:** Schablonenmethode *(template method)* und Einschubmethoden *(hook methods)*

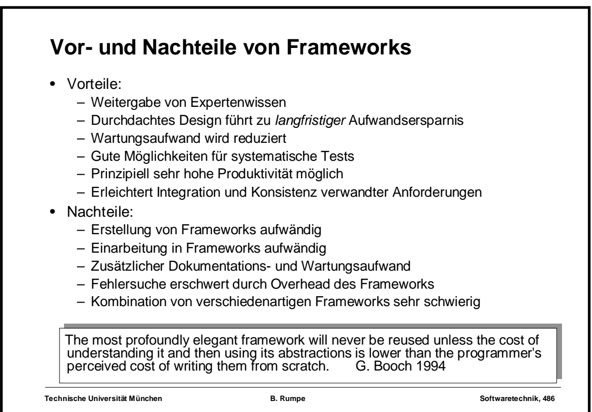


Entwurfsmuster Strategy

- Verhaltensmuster
- **Problem:** Ein Algorithmus aus einer Familie verwandter Algorithmen soll austauschbar gemacht werden.
- **Lösung:** Verkapseln des Algorithmus in einem austauschbaren Strategie-Objekt



Vor- und Nachteile von Frameworks



- Vorteile:
  - Weitergabe von Expertenwissen
  - Durchdachtes Design führt zu *langfristiger* Aufwandsersparnis
  - Wartungsaufwand wird reduziert
  - Gute Möglichkeiten für systematische Tests
  - Prinzipiell sehr hohe Produktivität möglich
  - Erleichtert Integration und Konsistenz verwandter Anforderungen
- Nachteile:
  - Erstellung von Frameworks aufwändig
  - Einarbeitung in Frameworks aufwändig
  - Zusätzlicher Dokumentations- und Wartungsaufwand
  - Fehlersuche erschwert durch Overhead des Frameworks
  - Kombination von verschiedenartigen Frameworks sehr schwierig

> The most profoundly elegant framework will never be reused unless the cost of understanding it and then using its abstractions is lower than the programmer's perceived cost of writing them from scratch. G. Booch 1994

Zusammenfassung: 5.6 Frameworks

- Ein Framework ist eine zusammenhängende Gruppe von Klassen, die ihren Kontrollfluss selbst festlegt:
  - „Don‘t call us, we call you.“ (Hollywood-Prinzip)
- Die Entwicklung eines Frameworks ist komplex, die Nutzung guter Frameworks aber von immensem Wert:
- Frameworks unterstützen Wiederverwendung, ohne selbst geändert werden zu müssen
- Angewendet werden Frameworks durch
  - Anpassung in Subklassen und durch
  - Übergabe von aufzurufenden Objekten (Beispiel: Listener's)
- Frameworks nutzen eine Reihe von Entwurfsmustern zur Flexibilisierung

Softwaretechnik WS 02/03

5. Software- & Systementwurf

5.7. Komponenten

Literatur:

C. Szyperski: Component Software - Beyond Object-oriented Programming, Addison-Wesley 1997

V. Gruhn, A. Thiel: Komponentenmodelle, Addison-Wesley 2000

D. Flanagan: Java in a Nutshell, Kap. 10, O'Reilly 1997 (2nd ed.)

M. Johnson: A beginner's guide to Enterprise JavaBeans, JavaWorld 10/98, http://www.javaworld.com/javaworld/jw-10-1998/jw-10-beans_p.html

R. Monson-Haefel: Enterprise JavaBeans, O'Reilly 1999

Bausteinorientierte Programmierung *(Component-Ware)*

- Idealzustand der Softwareentwicklung ist die Konstruktion aus vorgegebenen Bausteinen ("Software-ICs")
- Generische Bausteine *(components)*
  - anpassbar
  - zusammensetzbar ("Verdrahtung")
- Einfache Kompositionsmechanismen:
  - werkzeuggestützt
  - graphisch
- Infrastruktur zur Komponenteninteraktion: *"object bus"*
- Beispiele:
  - Visuelle Programmierung von GUIs
  - Java Beans, Enterprise Java Beans (EJBs)
  - CORBAcomponents
  - Microsoft COM+, DCOM
- Praxis: Oft projektspezifische Komponenten-Entwicklung
  - Wiederverwendung der Infrastruktur, nicht der Komponenten!

Komponenten

- **Definition**: „A ***software component*** is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.“
  ECOOP 1996, Workshop on Component-oriented Prog. 1997 & Clemens Szyperski

- Schnittstellen (*interfaces*):
  - Explizit definierte Interaktionen mit Komponenten und Umgebung
- Kontextabhängigkeiten (*context dependencies*):
  - benötigte Komponenten-Infrastruktur
  - Systemressourcen
- Unabhängige Einsetzbarkeit (*independent deployment*):
  - Alle Bestandteile enthalten (Archiv-Dateien), als Ganzes eingesetzt
- Komposition durch Dritte (*composition by third parties*):
  - Endbenutzer, Komponenten-Hersteller und Komponenten-Integrator
  - Meist nur als compilierter Code verfügbar, nicht als Quellcode

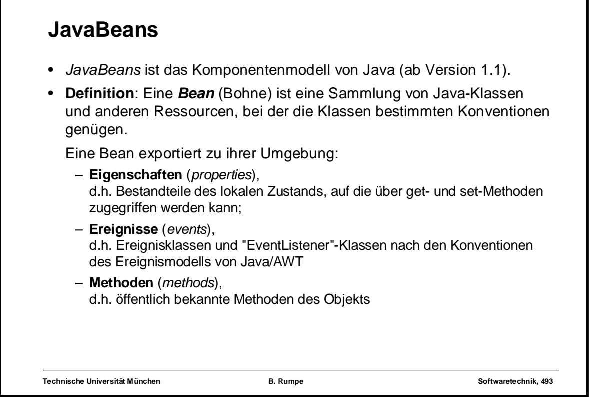
Sind Komponenten Objekte?

Widersprüchliche Stellungnahmen in der Literatur:

- Evolutionäre Position:
  - Komponenten als spezielle (komplexe) Objekte
  - Komponentenorientierte Programmierung als Abstraktionsschicht über objektorientierter Programmmierung
  - Komponentenbaukästen als datengetriebene Frameworks
  - z.B. JavaBeans, Enterprise Java Beans
    - » im folgenden näher erklärt
- Revolutionäre Position:
  - Komponentenorientierte Programmierung als Revolution
  - „Objektorientierte Programmierung ist gescheitert.“
  - Komponentenorientierte Programmierung benötigt die Konzepte der Objektorientierung nicht.
  - Komponenten ähnlich zu klassischen Modulen
- Allerdings scheint die evolutionäre Position stärker zu werden

Client- und Server-Komponenten

- Komponenten für eigenständige Anwendungen und Clients:
  - orientiert auf graphische Benutzungsoberfläche
  - Fachlogik ohne Berücksichtigung von Mehrbenutzer-Betrieb
- Komponenten für Client/Server-Anwendungen:
  - Serverseitige Komponenten
  - Komponentenbasierte "Application-Server"-Software
  - "Unsichtbare" Komponenten
  - Sicherstellung von Eigenschaften für Server-Software:
    - » Performance
    - » Sicherheit
    - » Mehrbenutzerbetrieb



Beispiel: SimpleBean

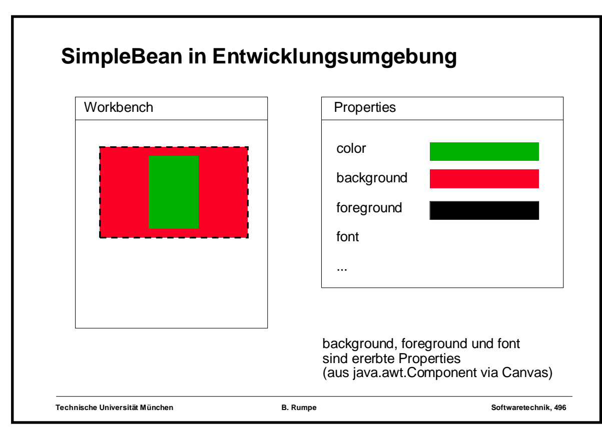
```
public class SimpleBean extends Canvas
  implements Serializable {

  private Color color = Color.green;

  public Color getColor(){
      return color;
  }

  public void setColor(Color newColor){
      color = newColor;
      repaint();
  }

  public void paint(Graphics g) {
      g.setColor(color);
      g.fillRect(20, 5, 20, 30);
  }

  public SimpleBean(){
      setSize(60,40);
      setBackground(Color.red);
  }
}
```

Technische Universität München B. Rumpe Softwaretechnik, 494

Wann ist eine Java-Klasse eine JavaBean?

- Es gibt keine Wurzelklasse aller JavaBeans und kein spezifisches Interfaces für alle JavaBeans!
- Eine Klasse wird zur JavaBean(-Klasse) durch Einhaltung bestimmter ***Namens-Konventionen***:
  - Konventionen für Eigenschaften (Properties):
    - » public set*PropertyName* (*PropertyType* p);
    - » public *PropertyType* get*PropertyName*();
- Bean-basierte Entwicklungswerkzeuge unterstützen:
  - Instanziieren von Bean-Objekten
  - Manipulation von Properties
  - Ereignisbehandlung
    - » Verwendung von "Meta-Programmierung" (Reflektion)
- Die graphischen Elemente der Java-GUI-Toolkits AWT und Swing erfüllen die JavaBean-Konventionen.

Technische Universität München B. Rumpe Softwaretechnik, 495



java.awt.Button

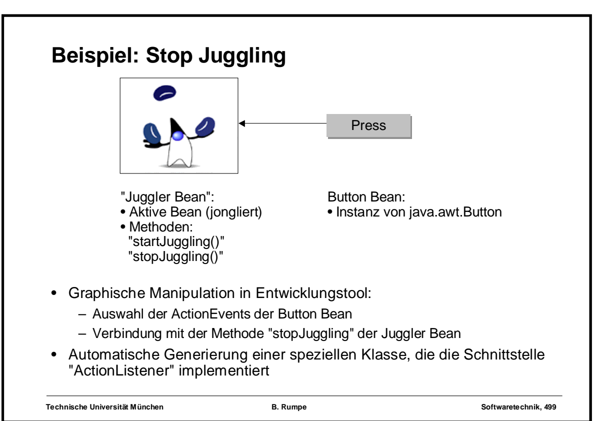
```
public class Button extends Component {
  ...
  public String getLabel();
  public void setLabel (String label);

  public void addActionListener (ActionListener l);
  public void removeActionListener
    (ActionListener l);

  ...
}
```

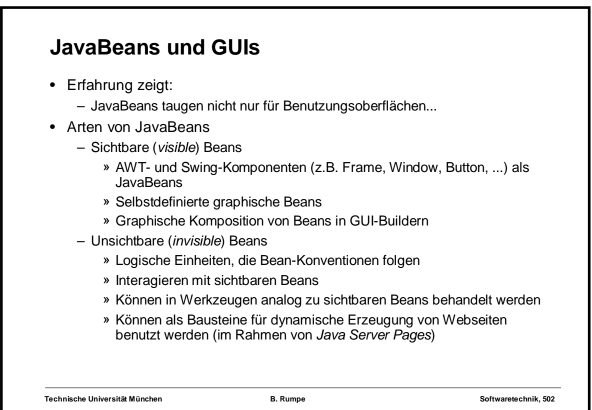
Technische Universität München B. Rumpe Softwaretechnik, 497

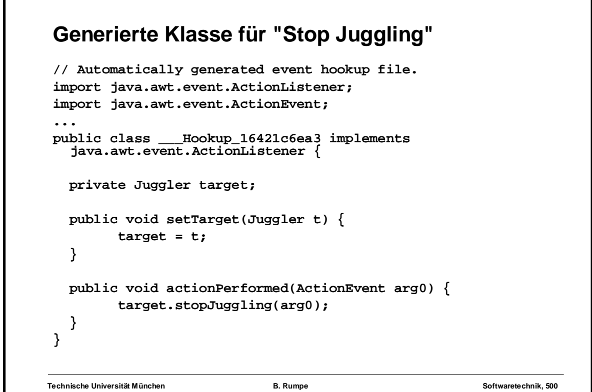
java.awt.event.ActionEvent, ActionListener

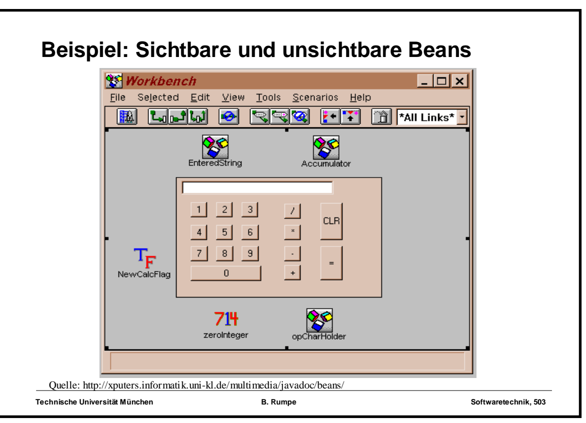
```
public interface ActionListener extends EventListener {
  public void actionPerformed (ActionEvent ev);
}

public class ActionEvent extends AWTEvent {
  ...
  public ActionEvent (object source, ...);
  ...
  public Object getSource ()
}
```

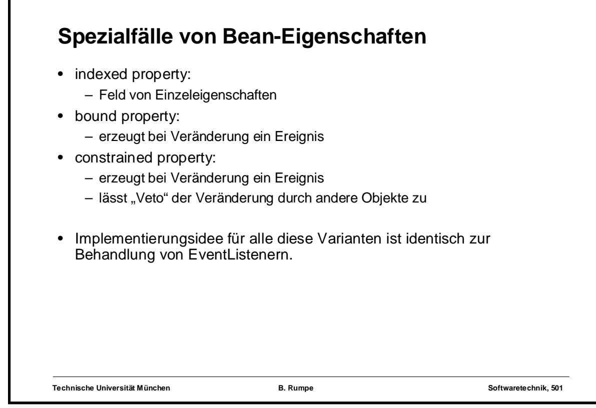
Technische Universität München B. Rumpe Softwaretechnik, 498

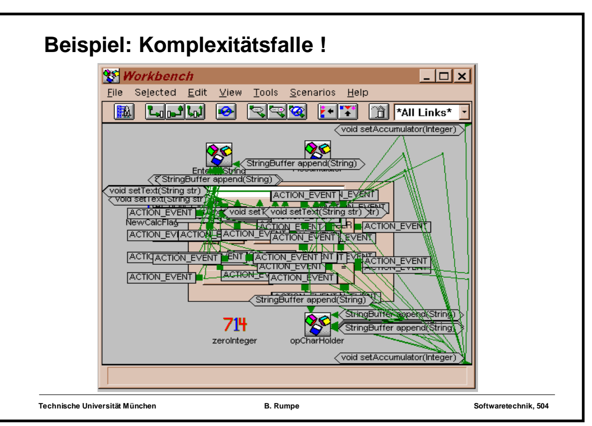

Beispiel: Stop Juggling
Press
"Juggler Bean":
• Aktive Bean (jongliert)
• Methoden:
"startJuggling()"
"stopJuggling()"
Button Bean:
• Instanz von java.awt.Button
• Graphische Manipulation in Entwicklungstool:
– Auswahl der ActionEvents der Button Bean
– Verbindung mit der Methode "stopJuggling" der Juggler Bean
• Automatische Generierung einer speziellen Klasse, die die Schnittstelle "ActionListener" implementiert
Technische Universität München
B. Rumpe
Softwaretechnik, 499


JavaBeans und GUIs
• Erfahrung zeigt:
– JavaBeans taugen nicht nur für Benutzungsoberflächen...
• Arten von JavaBeans
– Sichtbare (visible) Beans
» AWT- und Swing-Komponenten (z.B. Frame, Window, Button, ...) als JavaBeans
» Selbstdefinierte graphische Beans
» Graphische Komposition von Beans in GUI-Buildern
– Unsichtbare (invisible) Beans
» Logische Einheiten, die Bean-Konventionen folgen
» Interagieren mit sichtbaren Beans
» Können in Werkzeugen analog zu sichtbaren Beans behandelt werden
» Können als Bausteine für dynamische Erzeugung von Webseiten benutzt werden (im Rahmen von Java Server Pages)
Technische Universität München
B. Rumpe
Softwaretechnik, 502


Generierte Klasse für "Stop Juggling"
// Automatically generated event hookup file.
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
...
public class ___Hookup_16421c6ea3 implements
java.awt.event.ActionListener {
private Juggler target;
public void setTarget(Juggler t) {
target = t;
}
public void actionPerformed(ActionEvent arg0) {
target.stopJuggling(arg0);
}
}
Technische Universität München
B. Rumpe
Softwaretechnik, 500


Beispiel: Sichtbare und unsichtbare Beans
Workbench
File Selected Edit View Tools Scenarios Help
All Links
EnteredString
Accumulator
NewCalcFlag
CLR
zeroInteger
opCharHolder
Quelle: http://xputers.informatik.uni-kl.de/multimedia/javadoc/beans/
Technische Universität München
B. Rumpe
Softwaretechnik, 503

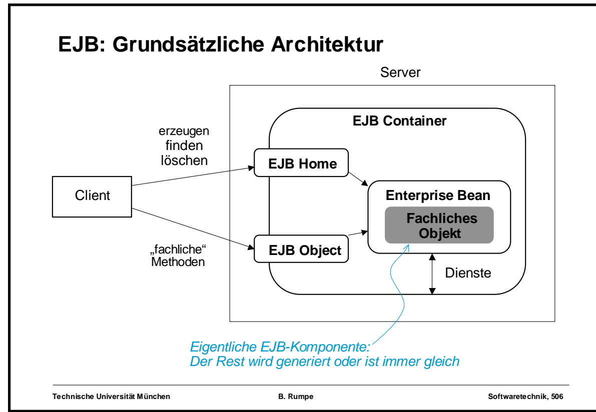

Spezialfälle von Bean-Eigenschaften
• indexed property:
– Feld von Einzeleigenschaften
• bound property:
– erzeugt bei Veränderung ein Ereignis
• constrained property:
– erzeugt bei Veränderung ein Ereignis
– lässt „Veto“ der Veränderung durch andere Objekte zu
• Implementierungsidee für alle diese Varianten ist identisch zur Behandlung von EventListenern.
Technische Universität München
B. Rumpe
Softwaretechnik, 501


Beispiel: Komplexitätsfalle !
Workbench
File Selected Edit View Tools Scenarios Help
All Links
void setAccumulator(Integer)
StringBuffer append(String)
ACTION_EVENT
zeroInteger
opCharHolder
Technische Universität München
B. Rumpe
Softwaretechnik, 504

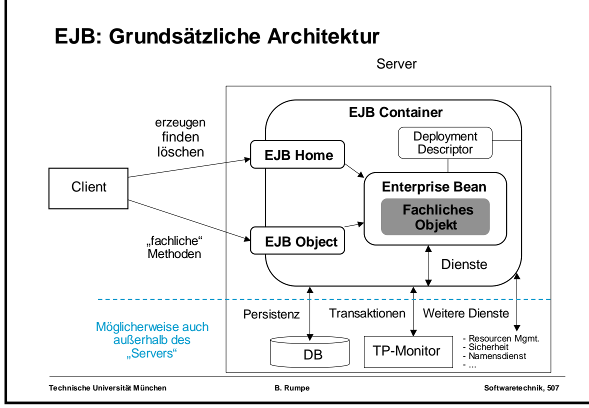
Client- und Server-Komponenten

- Wichtige Eigenschaften für Client/Server-Anwendungen:
  - Transaktionen
  - Sicherheit
  - Ressourcenverwaltung
  - Persistenz
- Komponentenkonzept für **Server**-Komponenten:
  - meist unsichtbare Komponenten
  - standardisierte Realisierung der wichtigen Eigenschaften für Client/Server-Anwendungen
  - Spezielle Infrastruktur: „*Transaktionsmonitor*“ für Komponenten
- Beispiele für Server-Komponentenmodelle:
  - Enterprise Java Beans (EJBs)
  - Microsoft Transaction Server (MTS)

EJB: Grundsätzliche Architektur



EJB: Grundsätzliche Architektur



EJB‘s Eigenschaften

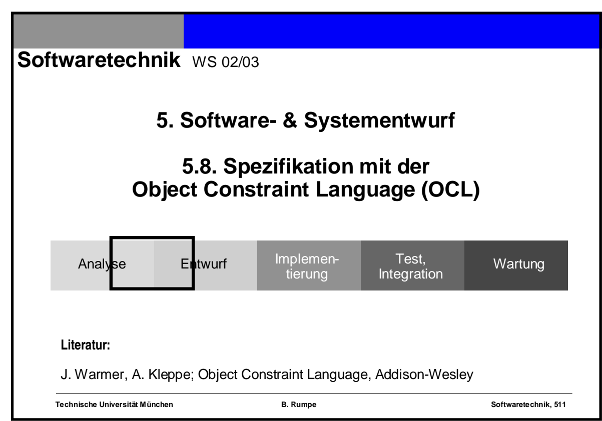
- Alles außerhalb des„Fachlichen Objekts“ wird entweder fertig bereitgestellt oder automatisch generiert.
- Jede EJB-Komponente hat ein „Home Interface“, über die Instanzen auf dem Server erzeugt, initialisiert, gelöscht und lokalisiert werden können.
- Für die Erzeugung und das Finden von Server-seitigen Beans wird das Java Naming and Directory Interface (JNDI), einen standardisierten Verzeichnisdienst, eingesetzt.
- Das Interface „EJB Object“ ist als notwendige Ergänzung der normalen Fachmethoden einer EJB-Klasse gedacht. Die (wenigen) Operationen hier dienen z.B. zur eindeutigen Identifikation eines Objekts.
- Das Home- und das Object-Interface, in dem die fachliche Schnittstelle eingebettet ist, werden auf der Client-Seite, über sogenannte „Stubs“ (Entwurfsmuster Proxy) zugänglich gemacht.
- Kommunikation Object-Container:
  - Java Remote Method Invocation (RMI) oder
  - CORBA/IIOP

Arten von Enterprise Java Beans

- Entity Bean:
  - Repräsentiert persistente Information aus einer Datenbank
  - Typischerweise mit mehreren Transaktionen assoziiert
  - Mehrbenutzerzugriff möglich
- Session Bean:
  - EJB-Instanz, die mit einem einzelnen Client assoziiert ist
  - Typischerweise nicht persistent
  - Beispiele:
    - » Benutzersitzung innerhalb einer Web-Site
    - » Buchungsvorgang
  - „Stateful Session Bean“
    - » Dialogzustände innerhalb der Session
  - „Stateless Session Bean“
    - » keine Dialogzustände, hohe Effizienz

Verpackungsform von Komponenten

- Eine Komponente ist eine geschlossene Einheit:
  - Möglicherweise viele Klassen
  - Dokumentation
  - Mechanismen zur Anpassung
- JavaBeans:
  - Bean wird ausgeliefert in Java-Archiv-Datei (JAR-Datei)
  - Metainformation in der JAR-Datei („Manifest“) beschreibt, welche Klassen Beans sind
  - Archiv enthält alle möglicherweise zusätzlich notwendigen Ressourcen
    - » Klassen (als Bytecode) (auch Ereignisklassen und Listener-Klassen
    - » Serialisierte Objekte, Daten, Medienobjekte
    - » ...
- Enterprise JavaBeans:
  - Bean wird ausgeliefert in Java-Archiv-Datei (JAR-Datei)
  - Deployment Descriptor zur Konfiguration der EJB
  - Archiv enthält alle möglicherweise zusätzlich notwendigen Ressourcen
    - » z.B. Stubs und Skeletons zur CORBA-Kommunikation
    - » ...



Constraints

- Def: Ein ***Constraint*** ist eine einschränkende Bedingung an das System.

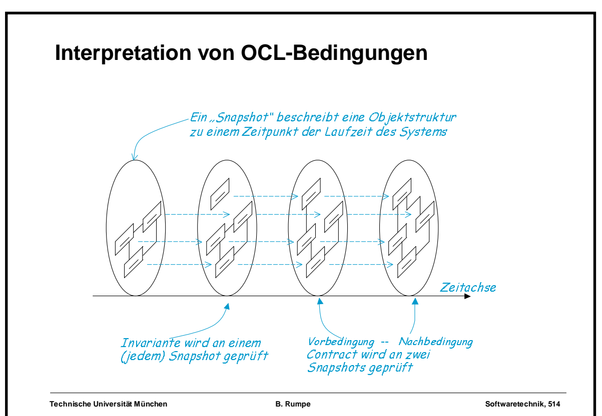
- Ein Constraint kann formuliert sein
  - über den möglichen Zustandsraum eines Systems, mehrerer Objekte oder nur eines einzelnen Objekts
    - » Beispiel: „Alle Personen sind älter als 18“
  - als zeitliche Bedingung:
    - » „Personen werden im System nicht jünger“
    - » „Diese Komposition bleibt über die Existenz des Kompositions-Objekts unverändert.“
  - als Vorbedingung für eine Methode
  - zur Beschreibung des Effekts einer Methode: Nachbedingung
  - als Guard bei einer Transition oder im Sequenzdiagramm

Constraint-Formulierungen

- Problem: Die Diagramme der UML ist nicht beliebig mächtig und komfortabel, um bestimmte Eigenschaften zu beschreiben.

- Formulierung von Constraints erfolgt daher
  - in natürlicher Sprache
  - in Pseudocode
  - in formalen Sprachen, zum Beispiel: Temporale Logik für zeitbehaftete Aussagen,
  - oder in der Object Constraint Language (OCL)

- Die OCL ist ein textuelles Addendum zu den UML-Diagrammen, das logische Aussagen formulierbar macht.
- OCL ist geeignet für Invarianten und Contracts
- OCL ist ungeeignet für allgemeine zeitbehaftete Bedingungen



Eigenschaften der OCL

- Ähnlich einer First-Order Logik:
  - Boolesche Operationen
  - Quantoren über endliche Objektmengen
- Nutzung des zugrundeliegenden Typsystems und der dortigen seiteneffektfreien Methoden
- Spezielle Konstrukte zur Navigation über Assoziationen
- Grunddatentypen: Boolean, Integer, Real, ...
- Container für Mengen, Sequenzen, Multimengen
- Design by Contract-Unterstützung

- OCL ist eine eingeschränkte Logiksprache, deren konkrete Syntax ähnlich einer Programmiersprache aussieht.
  - In dieser Vorlesung: eine an Java angepasste OCL-Variante: <<Java>>OCL
- OCL kann zur Spezifikation, zum Entwurf und zum Testen eingesetzt werden:
  - OCL ist ausführbar

Invarianten -1

- Eine Invariante ist eine allgemeingültige Aussage.
- Invarianten können Attribute und seiteneffektfreie Methoden (sog. “Queries”) nutzen.

- Eine Klasseninvariante:
  Alle Passagiere sind älter als 16 Jahre:

```
context Passenger inv:
        age > 16
```

- Eine Implikation:

```
context Passenger inv:
    age > 90   implies   getsAssistance() == true
```

Passenger
name: String age: Integer
getsAssistance(): Bool

Kontext:
das der OCL zugrundeliegende Klassendiagramm

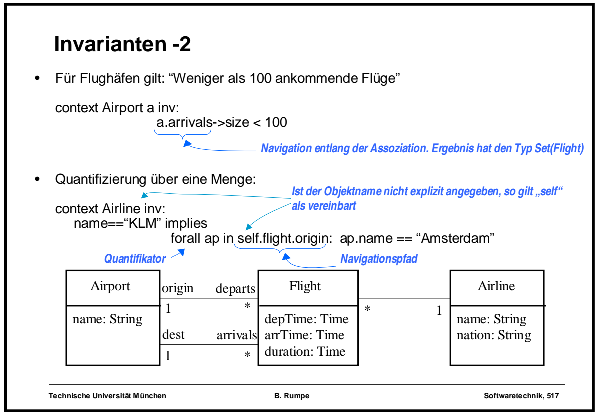
Invarianten -2

- Für Flughäfen gilt: "Weniger als 100 ankommende Flüge"

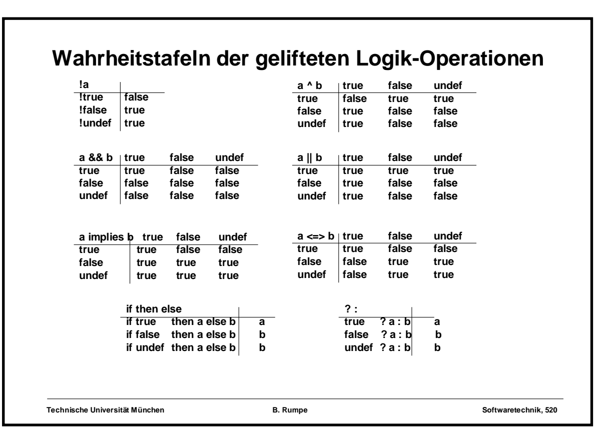
context Airport a inv:
a.arrivals->size < 100

- Quantifizierung über eine Menge:

context Airline inv:
name=="KLM" implies
forall ap in self.flight.origin: ap.name == "Amsterdam"



Wahrheitstafeln der geliften Logik-Operationen



!a	
!true	false
!false	true
!undef	true

a ^ b	true	false	undef
true	false	true	true
false	true	false	false
undef	true	false	false

a && b	true	false	undef
true	true	false	false
false	false	false	false
undef	false	false	false

a || b	true	false	undef
true	true	true	true
false	true	false	false
undef	true	false	false

a implies b	true	false	undef
true	true	false	false
false	true	true	true
undef	true	true	true

a <=> b	true	false	undef
true	true	false	false
false	false	true	true
undef	false	true	true

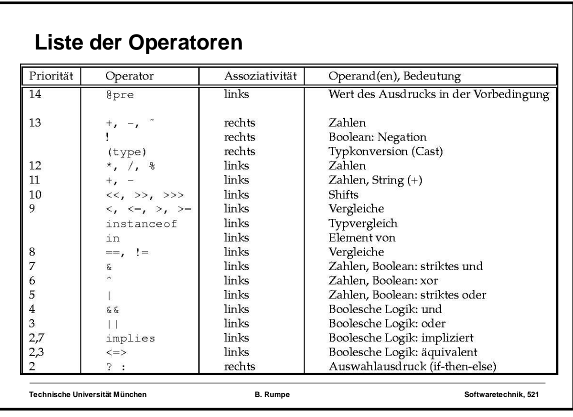
if then else	
if true then a else b	a
if false then a else b	b
if undef then a else b	b

? :	
true ? a : b	a
false ? a : b	b
undef ? a : b	b

Die Logik in der OCL – Beispiel Konjunktion

- Problem: Wie werden undefinierte Werte behandelt?
- Antwort in der Spezifikation:
  - Undefinierter Wert = False: Das ist ein „Lifting" in eine zweiwertige Logik
- Antwort beim Testen:
  - Es wird versucht undefinierte Werte abzufangen (Exceptions) und als „False" zu behandeln.
  - Das kann allerdings schief gehen: Dadurch unterschiedliche Semantik der OCL bei Spezifikation und Test!
- Ideal für die Praxis:
  - In der Analyse den undefinierten Fall ignorieren und auf die Funktionalität konzentrieren.
  - In der Implementierung alles in robuster Form umsetzen, z.B. durch explizite Exceptions

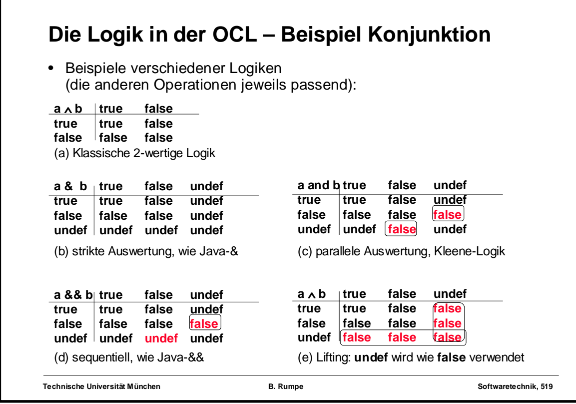
Liste der Operatoren



Priorität	Operator	Assoziativität	Operand(en), Bedeutung
14	@pre	links	Wert des Ausdrucks in der Vorbedingung
13	+, -, ~	rechts	Zahlen
	!	rechts	Boolean: Negation
	(type)	rechts	Typkonversion (Cast)
12	*, /, %	links	Zahlen
11	+, -	links	Zahlen, String (+)
10	<<, >>, >>>	links	Shifts
9	<, <=, >, >=	links	Vergleiche
	instanceof	links	Typvergleich
	in	links	Element von
8	==, !=	links	Vergleiche
7	&	links	Zahlen, Boolean: striktes und
6	^	links	Zahlen, Boolean: xor
5	|	links	Zahlen, Boolean: striktes oder
4	&&	links	Boolesche Logik: und
3	||	links	Boolesche Logik: oder
2,7	implies	links	Boolesche Logik: impliziert
2,3	<=>	links	Boolesche Logik: äquivalent
2	? :	rechts	Auswahlausdruck (if-then-else)

Die Logik in der OCL – Beispiel Konjunktion

- Beispiele verschiedener Logiken
  (die anderen Operationen jeweils passend):



a ∧ b	true	false
true	true	false
false	false	false

(a) Klassische 2-wertige Logik

a & b	true	false	undef
true	true	false	undef
false	false	false	undef
undef	undef	undef	undef

(b) strikte Auswertung, wie Java-&

a and b	true	false	undef
true	true	false	undef
false	false	false	false
undef	undef	false	undef

(c) parallele Auswertung, Kleene-Logik

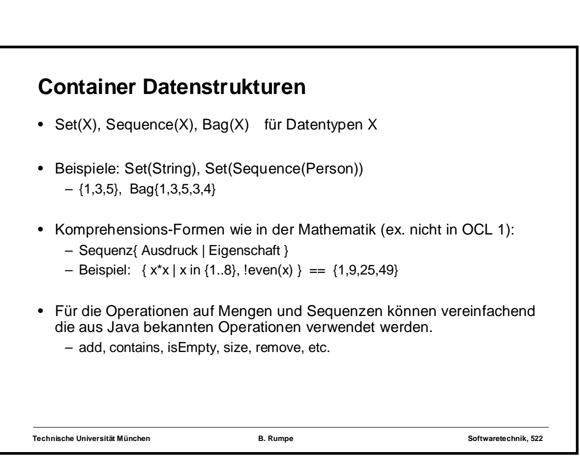
a && b	true	false	undef
true	true	false	undef
false	false	false	false
undef	undef	undef	undef

(d) sequentiell, wie Java-&&

a ∧ b	true	false	undef
true	true	false	false
false	false	false	false
undef	false	false	false

(e) Lifting: **undef** wird wie **false** verwendet

Container Datenstrukturen



- Set(X), Sequence(X), Bag(X) für Datentypen X
- Beispiele: Set(String), Set(Sequence(Person))
  - {1,3,5}, Bag{1,3,5,3,4}
- Komprehensions-Formen wie in der Mathematik (ex. nicht in OCL 1):
  - Sequenz{ Ausdruck | Eigenschaft }
  - Beispiel: { x*x | x in {1..8}, !even(x) } == {1,9,25,49}
- Für die Operationen auf Mengen und Sequenzen können vereinfachend die aus Java bekannten Operationen verwendet werden.
  - add, contains, isEmpty, size, remove, etc.

Navigation entlang Assoziationen

- Kontext:

- Auction a, Set(Auction) sa
- a.policy ist vom Typ Policy
- a.person ist vom Typ Set(Person)
- a.message ist vom Typ Sequence(Message)
- sa.person ist vom Typ Bag(Person)

- Navigation mit Containern als Ausgangsbasis entstehen zum Beispiel durch Navigationsketten. Diese werden aus pragmatischen Gründen „flachgedrückt“: sa.person ist keine Menge von Mengen, sondern nur eine Multimenge (Bag)

Design By Contract

- wurde in Eiffel (B. Meyer) zur Reife gebracht
- Idee:
  - Spezifikation einer Methode durch Vor- und Nachbedingungen
  - Vorbedingung beschreibt notwendige Eigenschaften des Objekts, der Umgebung und der Aufrufparameter unter denen die Methode garantiert erfolgreich sein wird.
  - Nachbedingung beschreibt die möglichen Änderungen am Objekt, der Umgebung und das Ergebnis.
  - Die getroffene Aussage ist:
    - » Wenn die Vorbedingung gilt, so garantiert die Methode die Nachbedingung.
- Allerdings: Für Contracts werden Informationen über inneren Objektzustand benötig und es ist nicht möglich über parallel stattfindende Ereignisse, die das eigene Ergebnis beeinflussen, zu sprechen.

Quantoren (wieder syntaktisch nicht OCL konform)

- Allquantor
  - forall Typ x: Eigenschaft
  - forall var in Container: Eigenschaft
- Existenzquantor:
  - exists Typ x: Eigenschaft
  - exists var in Typ: Eigenschaft

- Die Aussage
  - exists Person p: p.age > 18
- wird über einem festen Systemzustand (Snapshot) interpretiert und kann aufgrund der Endlichkeit der quantifizierten Menge berechnet werden.
- Daher ist OCL prinzipiell für Tests geeignet.
- Aber: Quantoren über integer sind dann zu vermeiden:
  - exists int n,a,b,c: a^n + b^n == c^n && n>2

Beispiele: OCL für Operationen

- Vor- und Nachbedingungen für Operationen:

Operation die beschrieben wird

```
context Class::operation(parameters:Types): Resulttype
pre:        precondition
post:       postcondition
```

- Beispiele:

```
context Math::sqrt(n : Integer): Integer
pre:        n >= 0
post:       result*result == n
```

Ist diese Methode gut spezifiziert?

```
context Airline::addFlight(fl:Flight)
pre:        not flights->contains(fl)
post:       flights == flights@pre->add(fl)
```

Wert des Attributs vor Aufruf

Spezialoperatoren

- any Mengenausdruck
  - liefert einen(!) Wert der Menge. Vorsicht: Unterspezifiziert.

- let var = Expr in Ausdruck
  - Definition von Hilfsvariablen
  - Beispiel: let y=x*x-2x in y>0 ? y : -y

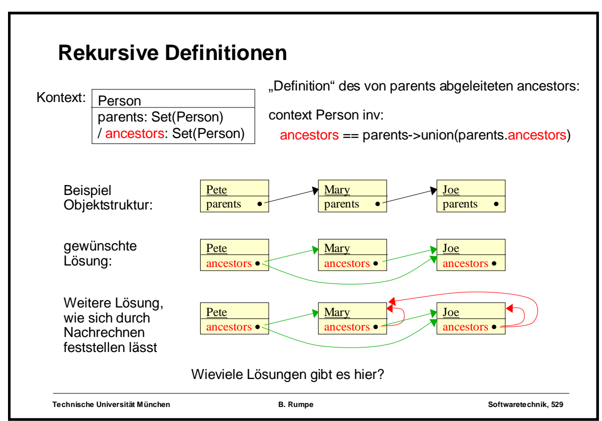
- defined(Ausdruck)
  - liefert True genau dann, wenn der Ausdruck einen definierten Wert abgibt.
  - Vorsicht: dieser Operator ist nur partiell implementierbar

Vorbedingung nicht erfüllt?

- Interpretation 1:
  - Wenn die Vorbedingung nicht erfüllt ist, dann sagt der Contract nichts aus
  - Die Methode darf beliebiges Verhalten haben.
  - Idealerweise: Robuste Implementierung
- Interpretation 2:
  - Es ist ein Fehler der Umgebung die Methode so aufzurufen.
  - Böswillig: Die Methode sollte sogar „abstürzen“ oder Fehler melden
  - Die Vorbedingung ist dann also eine Aussage über das erwartete Verhalten der Umgebung vor dem Aufruf.

- Interpretation 1 erlaubt die Zerlegung von Contracts:

context X:	context X:		context X:
pre: A1	pre: A2	*entsprechen*	pre: A1 || A2
post: B1	post: B2		post: (A1 implies B1) && (A2 implies B2)



Zusammenfassung
5.7 Object Constraint Language

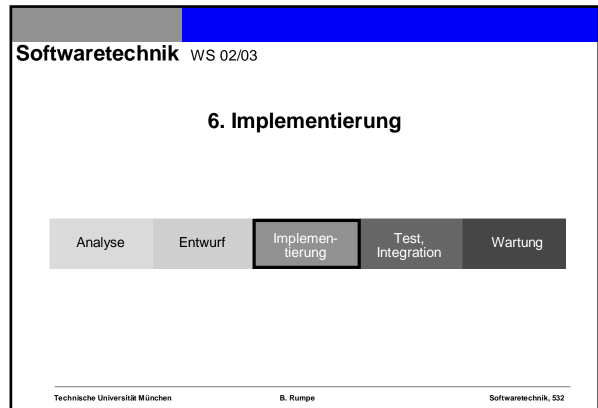
- OCL ist eine textuelle Sprache zur Definition von Bedingungen.
- OCL besitzt Operatoren der First-Order-Logik (Boolesche Operatoren und Quantoren)
- Das Typsystem und Methoden werden vom zugrundliegenden UML-Modell bereit gestellt.
- Queries sind Methoden ohne Seiteneffekte und können in OCL verwendet werden.
- OCL beschreibt Invarianten, Guards von Transitionen, und Contracts
- Ein Contract ist ein Paar aus Vor- und Nachbedingung, das eine Methode spezifiziert.
- Es gibt Container für Mengen, Sequenzen und Multimengen.
- Navigationskonstrukte erlauben über Assoziationen zu anderen Objekten zu gelangen.

- Der vorgestellte Ausschnitt der OCL ist auf Java-Syntax zugeschnitten!

Implementierungen:

- Es gibt für die OCL eigene Parser und teilweise Codegeneratoren.
- Für Java gibt es jedoch mit der „Java Modeling Language“ von Gary Leavens et.al. eine Erweiterung von Java, die noch mehr leistet:
  - JML kennt den Contract-Stil
  - JML hat eine assertion-Sprache für geltende Eigenschaften im Code
  - JML bietet viele Shortcuts
  - JML integriert perfekt mit Java, weil alle JML-Erweiterungen als Java-Kommentare deklariert werden

- Deshalb: JML ist einen Einsatz wert:
  - http://www.jml.org/

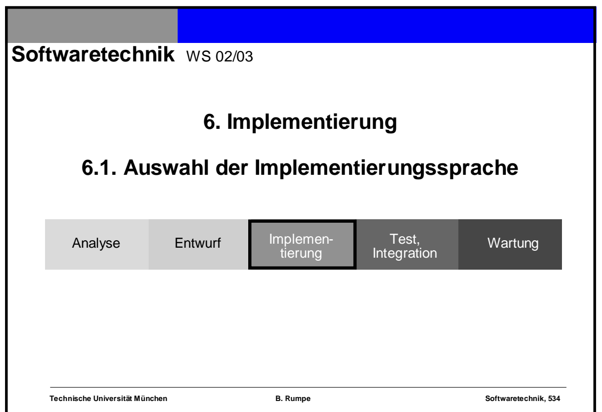


Definition: Implementierung

- **Definition**: Implementierung ist die Menge aller Programmier-Aktivitäten.

- Die Implementierung geht von einer gegebenen System-Architektur und detaillierten Spezifikation der Funktionalität aus.
- Ihre Ergebnisse sind:
  - Quellprogramm einschließlich integrierter Dokumentation
  - Lauffähiges System
  - Testplanung und Testdokumentation

- Die Implementierungs-Aktivitäten sind daher stark mit den Test-Aktivitäten verzahnt, um die Qualität des Systems sicher zu stellen.
- Varianten der „Implementierung“:
  - Legacy-System ist zu erweitern oder anzupassen
  - Rapid Prototyping: auf Design und Detailspezifikation wird dabei u.U. verzichtet



Funktionale Programmiersprachen

- Beispiele: Gofer, Haskell, ML
- Seiteneffektfrei und dadurch leicht verstehbar
- Sehr mächtiges Typsystem (bei den Beispielen)
- Patternmatching auf Argumenten
- Effiziente Definition von Datenstrukturen und Funktionen
  - `data Tree = Leaf(Int) | Node(Tree,Int,Tree)`
- Funktionen höherer Ordnung (Funktionen auf Funktionen anwendbar)
  - `twice f x = f(f(x))`
- Kompakte Formulierung

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

Prozedurale Programmiersprachen

- Beispiele: Modula-2, (Pascal, C, Fortran)
- Ideen zum Modul-Konzept teilweise vorhanden
- Komfortable Definition von Datenstrukturen
- Trennung von Datenstruktur und Funktionen macht Wartbarkeit schwieriger

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

Objektorientierte Programmierung

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

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

Spezialsprachen

- Logikprogramme: Prolog
  - Logische Aussagen in Hornklauselform als Programm
- Visuelle Programmierung
  - a) Komposition des Programms aus Bausteinen
  - b) Modellierung z.B. mit ausführbaren Statecharts
- Programmiersprachen mit inhärentem Verteilungskonzept
  - für massiv verteilte Systeme
- Skriptsprachen
  - Beispiel: Python
  - Meist kein festes Typsystem
  - effektive Module zur Textbearbeitung (z.B. reguläre Ausdrücke)
  - Neue: immer bessere Integration mit anderen Sprachen (z.B. Python + Java)
- Weitere Spezialsprachen: HTML, JSP, XML, SQL, ...

Weitere Auswahlkriterien

- Welche technische Umgebung ist gegeben?
  - Legacy-System? Gibt es eine Vorgabe des Unternehmens?

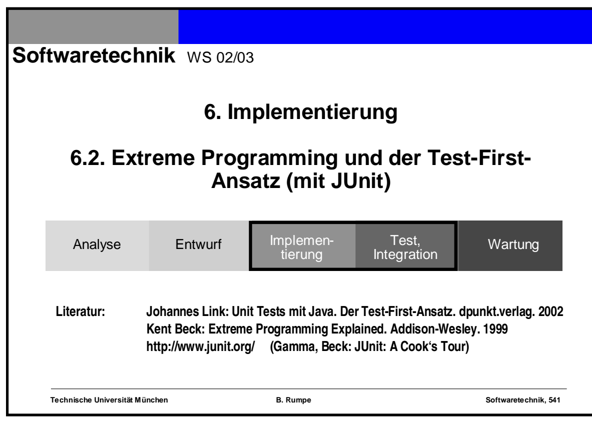
- Human Factor: Welche Kenntnisse/Vorlieben besitzen die Programmierer?

- Welche Bibliotheksfunktionen werden benötigt?

- Wie gut ist die Werkzeugunterstützung?
  - Compiler, Debugger, IDE, ...

Beobachtungen

- Die Kooperationsfähigkeiten der Sprachen erlauben die Anwendung der jeweils besten Sprache:
  - Corba, .NET, Embedded SQL (z.B. in Java)
  - Java Server Pages (JSP) = Java + HTML
  - Modul-Import über API's (Python in Java)
  - Generatoren wie Yacc: für Grammatik -> Parser
- Fehlende Spracheigenschaften werden durch standardisierte Klassenbibliotheken abgedeckt:
  - I/O wurde in C als erstes durch Bibliotheksfunktionen standardisiert
  - Threads/Nebenläufigkeit wird in Java über die Bibliothek realisiert
  - Kommunikation, Datenspeicherung wird nicht über Sprachprimitive, sondern Bibliotheksfunktionen angeboten
  - Security (z.B. Java Sandbox oder Verschlüsselung)
- Werkzeuge für das Management der Implementierung und Wartung erlauben wesentlich flexiblere Entwicklungsprozesse:
  - Dokumentation, Testen, Versionsverwaltung, Evolution, Generierung, Installation, ...



Extreme Programming (XP) ist

- eine leichtgewichtige, agile Methode, bestehend aus mehreren Prinzipien und daraus abgeleiteten Entwicklungspraktiken.
- Verzicht auf einige Elemente der Softwareentwicklung:
  - Dokumentation, Trennung in (längere) Phasen, ...
- Fokussierung stattdessen auf
  - Code, Tests, Kommunikation, ...
  - Sehr kurze Iterationszyklen, ...
- XP kann daher nur auf bestimmte Projektarten angewendet werden.
- Entstehung und Literatur:
  - Bibel: Kent Beck „Extreme Programming Explained", Addison Wesley, 1999
  - Mittlerweile Serien von Büchern und Artikeln zum Thema XP.
  - Auf mehreren Web-Sites aktiver Erfahrungsaustausch (wie unser SWIKI)
- XP ist als explizites Gegenstück zu RUP, Catalysis, UML gedacht.

Extreme Programming

- Motivation
  - In kurzer Zeit ein qualitativ hochwertiges Software-System entwickeln
  - Flexibel Anforderungen einbringen und priorisieren
  - Programmierer konzentrieren sich frühzeitig auf wesentliche Elemente der Softwareentwicklung: Codierung.
  - Hohe Innovationsgeschwindigkeit in der IT und sich ändernde Anforderungen machen flexibles Projektmanagement notwendig.
- Rahmenbedingungen
  - Maximal 5-10 Programmierer sind an einem Projekt beteiligt.
  - Kunde steht zur intensiven Einbindung in das Projekt zur Verfügung.
  - Räumlichkeiten erlauben intensive Kommunikation und Kontakt zwischen den Entwicklern.
  - Kunde verzichtet auf ausführliche Dokumentation von Analyse- und Entwurfsaktivitäten.
  - Entwickler-Team ist motiviert, eigenverantwortlich zu handeln.

Die wesentlichen Rollen in XP

- Projektleiter:
  - verantwortlich für Management und Koordination des Projekts.
  - verwaltet Ressourcen, Kosten, Zeitpläne,
  - versucht damit optimale Qualität zu erreichen.
- Kunde:
  - Wenigstens ein Kunde ist permanent ansprechbar, um schnell aufkommende Fragen zu klären.
  - Er entwirft funktionale Tests für die Software.
- Entwickler:
  - tragen die Hauptlast der Projekts.
  - Grundaktivitäten: Codierung, Testen, Zuhören, Design.

Aktivitäten (1)

Codierung

- In kleinen Schritten wird das System sukzessive erweitert
- Codierungsstandards
- Regelmäßige Durchläufe aller Tests
- Test werden gemeinsam mit dem Code entwickelt und fortgeschrieben.
- Zur Modifikation existenten Codes wird „Refactoring" eingesetzt.

Testen

- Jedes Programmelement besitzt einen automatisierten Test. Jede Schleife und jede Verzweigung werden dabei durchlaufen.
- Komponententests entstehen gemeinsam mit dem Code.
- Kunden entwerfen funktionale Tests, die die Geschäftslogik prüfen.

Aktivitäten (2)

Zuhören

- Kommunikation unter den Entwicklern, sowie zwischen Entwicklern und dem Kunden ist essentiell:
  - sie erlaubt flexibel die Erhebung und Weiterentwicklung von Anforderungen und
  - verhindert Irrwege bei der Entwicklung.

Design

- organisiert die Systemlogik in einer Form, die Änderungen lokal hält,
- hält Funktionalität und Daten zusammen,
- erlaubt die lokalisierte Erweiterung des Systems,
- ist schlank und verhindert unnötige Komplexität.
- Design ist Teil der täglichen Programmiertätigkeit.

Fundamentale Prinzipien

– **schnelles Feedback**
  – kontinuierliche Projektsteuerung

– **inkrementelle Änderungen**
  – keinen Big-Bang, sondern messbarer Fortschritt

– **qualitativ hochwertige Arbeit**
  – gesichert durch geeignete Massnahmen

– **Einfachheit**
  – Klarheit, Eleganz

– **Änderbarkeit unterstützen**
  – um Flexibilität zu erreichen
  – Fehlerkosten zu reduzieren

Entwicklungspraktiken

- Eine Entwicklungspraktik ist die operationelle Umsetzung von Prinzipien
- Hier die wichtigsten:

– **Planning game**
– 40-Std. Woche
– ständig verfügbarer Kunde
– kleine Releases
– kontinuierliche Integration
– simples Design
– **Testen**
– **Refactoring**
– **Pair programming**
– gemeinsamer Code-Besitz
– Codierungsstandards

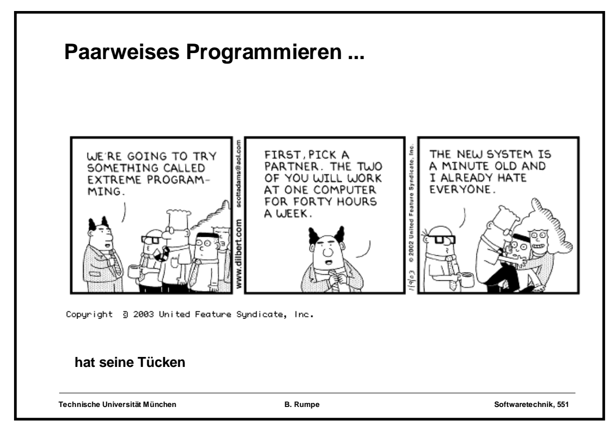
Planning Game

- ein evolutionärer, permanenter Dialog zwischen Notwendigem und Machbarem.
- Der Kunde entscheidet darüber, was zu implementieren ist:
  – Prioritäten
  – Zusammensetzung der Releases
  – Daten der Releases
- Entwickler entscheiden über:
  – Schätzungen Aufwand, Ressourcen
  – Konsequenzen
  – Vorgehensweise und Organisation

Paarweises Programmieren

- Vier Augen sehen mehr als zwei.
- Deshalb ist es in besonders kritischen Bereichen sinnvoll, dass zwei Personen gemeinsam an einem Terminal wesentliche Code-Stücke in Angriff nehmen.
- Zwei Personen an einem Rechner: eine tippt, die andere schaut über die Schulter – abwechselnd.
- Kritisch:
  – Muss zur Persönlichkeit der Beteiligten passen
  – Erste quantitative Versuche mit Studenten: (Paar vs. single)
    » Mehraufwand von ca. 30%
    » Time-to-finish: um ca. 15 % reduziert
    » Qualität (korrekte Testläufe): 97% pair, vs. 83% single

Paarweises Programmieren ...

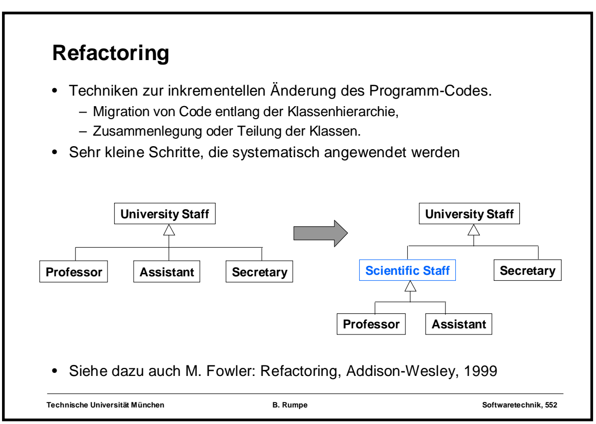


Copyright © 2003 United Feature Syndicate, Inc.

hat seine Tücken

Refactoring

- Techniken zur inkrementellen Änderung des Programm-Codes.
  – Migration von Code entlang der Klassenhierarchie,
  – Zusammenlegung oder Teilung der Klassen.
- Sehr kleine Schritte, die systematisch angewendet werden



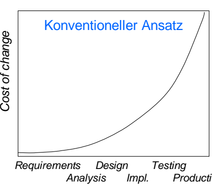
- Siehe dazu auch M. Fowler: Refactoring, Addison-Wesley, 1999

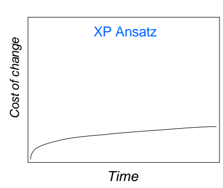
Testen

- ist eine der wichtigsten Tätigkeiten in XP
- Parallele Entwicklung von Code und Tests garantiert „Testüberdeckung“
- Automatische Tests um Wiederholbarkeit zu sichern
- Tests als „Modell“ mit Prüfcharakter
- Tests sichern (weitestgehend) die Korrektheit von Refactorings
- Testarten:
  - Black-Box-Tests (entstehen bevor die Funktion geschrieben wird)
  - White-Box-Tests (decken die Kontrollflüsse ab)
  - Unit-Tests (vom Anwender entworfen)
- Framework *junit* unterstützt die Testdefinition

Annahme von XP: Lineare Kostenkurve bei Fehlerbehebung?

- „cost of change may not rise dramatically over time“





- Konventioneller Ansatz wurde in den 70er'n gemessen für prozeduralen Sprachen, viel Dokumentation und kleineren Systemen. (u.A. Barry Boehm)
- XP Annahme ist nicht empirisch belegt. Wenn diese Annahme nicht korrekt ist, dann ist z.B. Refactoring zu teuer.

Test-Situation mit JUnit

Fixture: Gruppe von Objekten, die einen Zustand beschreibt

o1 o2 o3 o4

Methode operiert auf Fixture

Zu testende Methode

Fixture: Ergebnis des Aufrufs

o1 o2 o3 o4 o5

Schnittstellen zur Umgebung sind mit „Dummies“ zu simulieren, um Nebeneffekte zu erfassen

DB-Anschluss

Protokoll

GUI

Test-First Ansatz mit Einsatz von JUnit

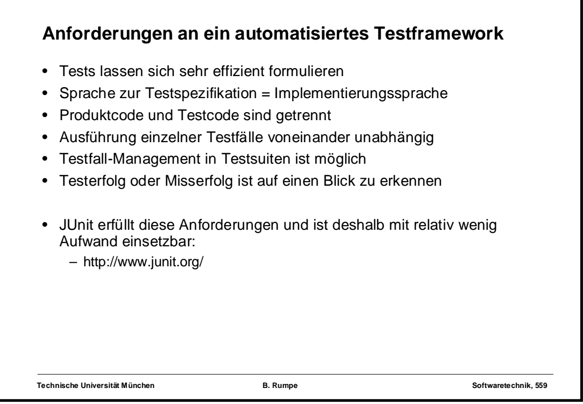
- (entwickelt aus XP)
- Grundidee:
  - „Test a little, code a little“
- Soll ein Stück Produktionscode entwickelt werden, so werden dafür zuerst Tests geschrieben, die die Funktionalität des Codes demonstrieren
  - Tests legen die Signatur des Codes fest
  - Tests sollten die Features des Codes überdecken
  - Kein überflüssiger Code, der nicht durch Tests gefordert wird
- Kunden formulieren Akzeptanztests quasi als Spezifikationsersatz
- Wichtig: Testautomatisierung, um effektives Wiederholen der Tests zu ermöglichen

Pro & Contra XP:

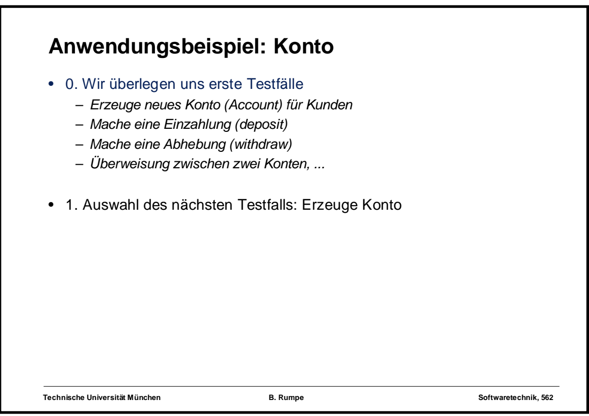
- Vorteile von XP:
  - Basiert auf „Best Practices“ und Entwickler-Erfahrung.
  - Betont die menschliche Komponente (die oft Fehlschläge verschuldet)
    - » z.B. 40h Woche, Kommunikation
    - » Systematisiert „Hacking“ in einen Prozess
    - » Kommt dem menschlichen Hacker-Naturell entgegen
  - Effizienz der Softwareentwicklung im Focus
  - Qualität durch integrierten Test-Prozess erreicht
- Probleme von XP:
  - Hohe Akzeptanzprobleme konventionell trainierter Entwickler
  - Skaliert nur bis ca. 10 Personen
  - Manche Projekte schreiben Dokumentationsleistung vor
  - Anwendung auf manche Einsatzgebiete unklar, z.B. Embedded Systems

Test-First-Entwicklungszyklus

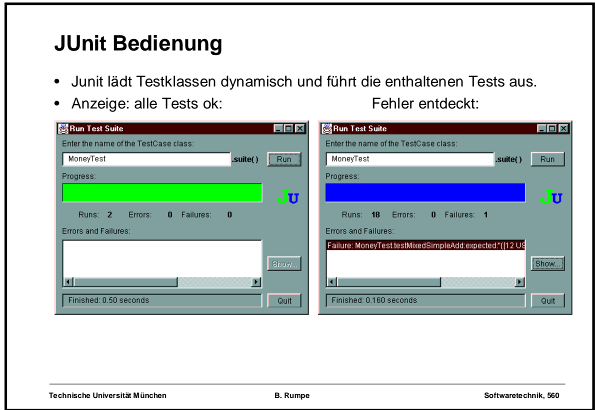
- *0. Wir überlegen uns erste Testfälle*
- *Wiederholung der Schritte 1-6:*
- *1. Auswahl des nächsten Testfalls.*
- *2. Wir entwerfen einen Test, der zunächst fehlschlagen sollte.*
- *3. Wir schreiben gerade soviel Code, dass sich der Test übersetzen lässt (Signatur).*
- *4. Wir prüfen, ob der Test fehlschlägt.*
- *5. Wir schreiben gerade soviel Code, dass der Test erfüllt sein sollte.*
- *6. Wir prüfen, ob der Test durchläuft.*
- *7. Die Entwicklung ist abgeschlossen, wenn uns keine weiteren Tests mehr einfallen, die fehlschlagen können*



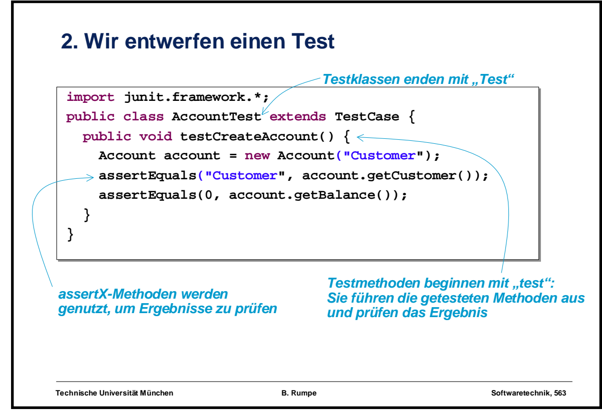
Anforderungen an ein automatisiertes Testframework
Tests lassen sich sehr effizient formulieren
Sprache zur Testspezifikation = Implementierungssprache
Produktcode und Testcode sind getrennt
Ausführung einzelner Testfälle voneinander unabhängig
Testfall-Management in Testsuiten ist möglich
Testerfolg oder Misserfolg ist auf einen Blick zu erkennen
JUnit erfüllt diese Anforderungen und ist deshalb mit relativ wenig Aufwand einsetzbar:
http://www.junit.org/
Technische Universität München
B. Rumpe
Softwaretechnik, 559



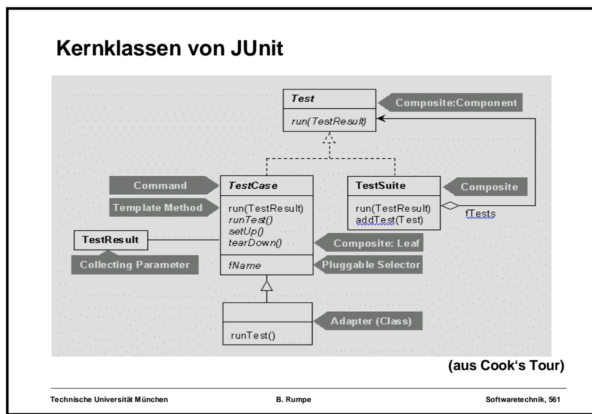
Anwendungsbeispiel: Konto
0. Wir überlegen uns erste Testfälle
Erzeuge neues Konto (Account) für Kunden
Mache eine Einzahlung (deposit)
Mache eine Abhebung (withdraw)
Überweisung zwischen zwei Konten, ...
1. Auswahl des nächsten Testfalls: Erzeuge Konto
Technische Universität München
B. Rumpe
Softwaretechnik, 562



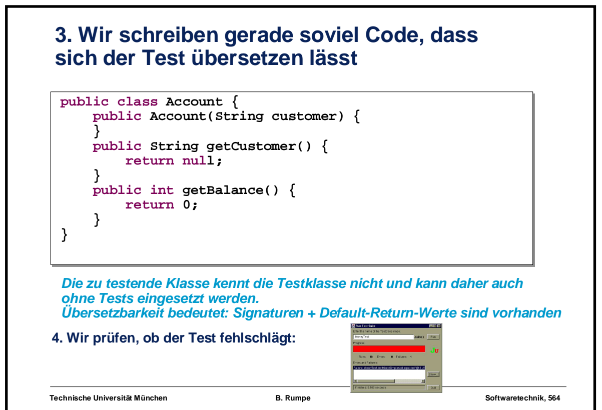
JUnit Bedienung
Junit lädt Testklassen dynamisch und führt die enthaltenen Tests aus.
Anzeige: alle Tests ok:
Fehler entdeckt:
Run Test Suite
Enter the name of the TestCase class:
MoneyTest
.suite()
Run
Progress:
Runs: 2 Errors: 0 Failures: 0
Runs: 10 Errors: 0 Failures: 1
Errors and Failures:
Show
Finished: 0.50 seconds
Finished: 0.160 seconds
Quit
Technische Universität München
B. Rumpe
Softwaretechnik, 560



2. Wir entwerfen einen Test
import junit.framework.*;
public class AccountTest extends TestCase {
public void testCreateAccount() {
Account account = new Account("Customer");
assertEquals("Customer", account.getCustomer());
assertEquals(0, account.getBalance());
}
}
Testklassen enden mit „Test“
assertX-Methoden werden genutzt, um Ergebnisse zu prüfen
Testmethoden beginnen mit „test“: Sie führen die getesteten Methoden aus und prüfen das Ergebnis
Technische Universität München
B. Rumpe
Softwaretechnik, 563



Kernklassen von JUnit
Test
run(TestResult)
Composite:Component
Command
Template Method
TestCase
run(TestResult)
runTest()
setUp()
tearDown()
fName
TestSuite
run(TestResult)
addTest(Test)
Composite
fTests
TestResult
Collecting Parameter
Composite: Leaf
Pluggable Selector
runTest()
Adapter (Class)
(aus Cook's Tour)
Technische Universität München
B. Rumpe
Softwaretechnik, 561



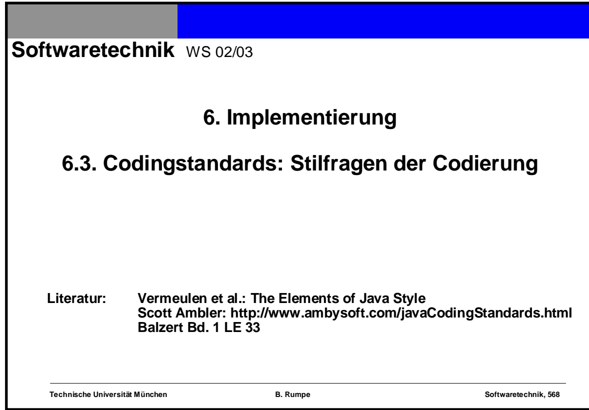
3. Wir schreiben gerade soviel Code, dass sich der Test übersetzen lässt
public class Account {
public Account(String customer) {
}
public String getCustomer() {
return null;
}
public int getBalance() {
return 0;
}
}
Die zu testende Klasse kennt die Testklasse nicht und kann daher auch ohne Tests eingesetzt werden.
Übersetzbarkeit bedeutet: Signaturen + Default-Return-Werte sind vorhanden
4. Wir prüfen, ob der Test fehlschlägt:
Technische Universität München
B. Rumpe
Softwaretechnik, 564

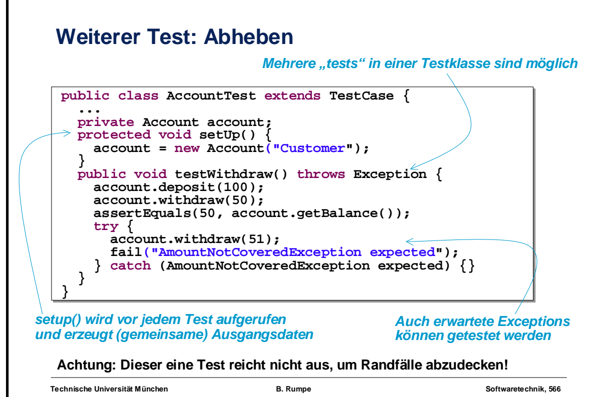
5. Wir schreiben gerade soviel Code, dass der Test erfüllt sein sollte

```java
public class Account {
    private String customer;
    public Account(String customer) {
        this.customer = customer;
    }
    public String getCustomer() {
        return customer;
    }
    public int getBalance() {
        return 0;
    }
}
```

Im Beispiel werden also der Konstruktor und getCustomer umgesetzt sowie die notwendige Datenstruktur (String customer) definiert.

6. Wir prüfen, ob der Test durchläuft:





Programmierstil

- Qualität von Programmcode:
  - Funktionalität
  - Verständlichkeit, Wartbarkeit
  - Effizienz
  - Eleganz
- Im Ablauf identische Programme können in der Verständlichkeit erheblich differieren.
  - Programmierkonventionen, z.B.:
    - Verwendete Konstrukte
    - Reihenfolgen
    - Klammerung
  - Bezeichnerwahl
  - Layout
    - Einrückung
- "Style Guide": Standard-Konventionen (z.B. für Projekt, Firma)

JUnit Tests

- Testsuite = Sammlung von Tests und Suiten
- Jede Testklasse stellt eine Suite dar
  - Durch Reflection werden die realisierten Tests erkannt
  - Explizite suite-Definition ist aber möglich mit der Methode suite()
- Übergabe von Test-Ergebnissen erfolgt unsichtbar für den Entwickler durch die assert-Methoden
- Ein voller Zyklus eines Tests besteht aus
  - setup(), testX(), teardown()
- Eine Anzahl assertX()-Methoden erlauben die Prüfung auf Gleichheit, null-Referenzen, etc. und haben optional zusätzliche Argumente, die bei Fehlermeldungen ausgegeben werden.

Was tut dieses Java-Programm?

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

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

Formatierungs-Richtlinien

```
public class Z {

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

}
```


Hinweise zur Formatierung

- Einheitliche Formatierung verwenden !
  - Werkzeuge ("pretty printer", "beautifier")
- Gemäß Schachtelungstiefe einrücken
  - Genau festgelegte Anzahl von Leerzeichen (besser als Tabulatoren!)
  - Formatierungsprobleme bei zu tiefer Schachtelung deuten oft auf Strukturprobleme des Codes hin !
- Leerzeilen verwenden (einheitlich)
  - z.B. vor und nach Methoden
  - Aber: Zusammenhängender Code soll auf einem normalen Bildschirm darstellbar bleiben!
- Leerzeichen verwenden (einheitlich):
  - z.B. Operatoren und Operanden durch ein Leerzeichen trennen
  - z.B. keine Leerzeichen vor Methodenparametern, nach Casts
- Einheitliche Dateistruktur verwenden
  - z.B.: Je Klasse eine .java-Datei, je Package ein Verzeichnis
  - "package"-Statement immer als erste Zeile (noch vor "import")

Beispiele zur Einrückung

- JavaSoft-Konvention zu Einrückungstiefe:
  - 4 Zeichen normale Einrückung
  - 8 Zeichen Einrückung zu Sonderzwecken
- Lange Methodenköpfe:

```
void methode (int x, Object y, String z, Xyz v,
              float p);  //konventionell
private static synchronized void etwasLang (int x;
        Object y, String z, Xyz v, float p) {
        // Acht Zeichen Einrückung ist besser
    x = ... // Methodenrumpf nur um vier eingerückt
```

- Fallunterscheidung:

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


Beispiele zu Klammern und Separatoren

- Relativ schlecht wartbarer Code (Java):

```
if (bedingung)
  methode();
```

- Besser wartbarer Code:

```
if (bedingung) {
  methode();
}
```

- Relativ schlecht wartbarer Code (Pascal):

```
if xyz then
  begin
    statement1;
    statement2
  end;
```

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

Wahl von Bezeichnern

- Einheitliche Namenskonvention verwenden !
- Bezeichner sollen:
  - natürlicher Sprache entnommen sein (bevorzugt Englisch)
    - » Ausnahmen: Schleifenvariablen, manche Größen in Formeln
  - aussagekräftig sein
  - leicht zu merken sein
  - nicht zu lang sein, wenn häufig verwendet
  - Kurze Bezeichner nur bei sehr kleinem Scope (Schleifenvariablen)
- Beispiele:
  - x1, x2, i, j, k
  - customername, CustomerName, customerName, cust_name
  - kontoOeffnen, oeffneKonto, kontoOeffnung, kontoGeoeffnet
  - CONSTANT

Beispiele für Namenskonventionen

- Klasse:
  - Substantiv, erster Buchstabe groß, Rest klein
  - Ganze Worte, Zusammensetzung durch Großschreibung
  - Bsp: `Account, StandardTemplate`
- Methode:
  - Verb, Imperativ (Aufforderung), erster Buchstabe klein
  - Lesen und Schreiben von Attributen mit get/set-Präfix im Namen
  - Bsp: `checkAvailability(), doMaintenance(), getDate()`
- Konstante:
  - Nur Großbuchstaben, Worte mit "_" zusammengesetzt
  - Standardpräfixe: "MIN_", "MAX_", "DEFAULT_", …
  - Bsp.: `NORTH, BLUE, MIN_WIDTH, MAX_WIDTH, DEFAULT_SIZE,`
- Attribute
  - Mit führendem Underscore
  - Bsp: `_availability, _date`

Lesbarkeit durch Bezeichnerwahl

```
public class Zinstabelle {

  public static void main(String[] args) {
    double betrag;        //zu verzinsender Betrag
    double zinssatzJahr; //jaehrlicher Zins als Faktor
    int laufzeit;         //Laufzeit in Jahren
    betrag = Console.readDouble("Betrag:");
    zinssatzJahr = Console.readDouble("Zinssatz:") / 100;
    laufzeit = Console.readInt("Laufzeit:");
    double y;

    for (y = zinssatzJahr - 0.01;
        y <= zinssatzJahr + 0.01; y += 0.00125){
      double zinssatzMonat = y/12;
      double zahlung = betrag * zinssatzMonat /
          (1 - (Math.pow(1/(1 + zinssatzMonat),
          laufzeit * 12)));
      System.out.println(100*y+" : "+zahlung);
    }
  }

}
```


Änderungsfreundlicher Code (1)

- Wahl von Variablen, Konstanten und Typen orientiert an der fachlichen Aufgabe, nicht an der Implementierung:
  - Gutes Beispiel (C):
    `typedef char name [nameLength]`
    `typedef char firstName [firstNameLength]`
  - Schlechtes Beispiel (C):
    `typedef char string10 [10]`
- Symbolische Konstante statt literale Werte verwenden, wenn spätere Änderung denkbar.
- Algorithmen, Formeln, Standardkonzepte in Methoden/Prozeduren kapseln.
- An den Leser denken:
  - Zusammenhängende Einheit möglichst etwa Größe eines typischen Editorfensters (40-60 Zeilen, 70 Zeichen breit)
  - Text probehalber vorlesen ("Telefon-Test")

Änderungsfreundlicher Code (2)

- Strukturierte Programmierung
  - Kein "goto" verwenden (in anderen Sprachen als Java)
  - "switch" nur mit "break"-Anweisung nach jedem Fall
  - "break" nur in "switch"-Anweisungen verwenden
  - "continue" nicht verwenden
  - "return" nur zur Rückgabe des Werts, nicht als Rücksprung
- Übersichtliche Ausdrücke:
  - Möglichst seiteneffektfreie Ausdrücke
    - » Schlechtes Bsp.: `y += 12*x++;`
    - » Inkremementierung/Dekrementierung besser in separaten Anweisungen
  - Fallunterscheidungsoperator (ternäres "? : ") sparsam einsetzen
- Sichtbarkeitsprüfungen des Compilers ausnutzen:
  - Variablen möglichst lokal und immer "private" deklarieren
  - Wiederverwendung "äußerer" Namen (Verschattung) vermeiden

Stilistisch verbessertes Programm

```
public class Zinstabelle {

  private static double zinsFormel
      (double betrag, double zinssatzMonat, double laufzeit) {
    return betrag * zinssatzMonat
       / (1 - (Math.pow(1/(1 + zinssatzMonat), laufzeit * 12)));
  }
  static final double BEREICH = 0.01;
  static final double SCHRITT = 0.00125;
  public static void main(String[] args) {

    double betrag;       //zu verzinsender Betrag
    double zinssatzJahr; //jaehrlicher Zins als Faktor
    int laufzeit;        //Laufzeit in Jahren
  // Werte einlesen
    betrag = Console.readDouble("Betrag:");
    zinssatzJahr = Console.readDouble("Zinssatz:") / 100;
    laufzeit = Console.readInt("Laufzeit:");
  // Ergebnisse ausgeben
    double y;
    for (y = zinssatzJahr - BEREICH;
        y <= zinssatzJahr + BEREICH; y += SCHRITT){
      System.out.println(100*y+" : "+zinsFormel(betrag,y/12,laufzeit));
    }
  }
}
```


Kommentare

- Prinzip der integrierten Dokumentation:
  - Kommentare im Code sind leichter zu warten
  - Kommentare sollten parallel zum Code entstehen
    - » "Nach-Dokumentation" funktioniert in der Praxis nie!
  - Werkzeuge zur Generierung von Dokumentation (z.B. javadoc)
- Idealzustand:
  - Kommentare zu Klassen und Methoden stellen eine vollständige und eindeutige Spezifikation des Codes dar
- Kommentare sollen *nicht:*
  - den Code unlesbar machen
    - » z.B. durch Verzerrung des Layouts
  - redundante Information zum Code enthalten
  - Schlechter Kommentar:
    `i++; // i wird hochgezaehlt`
- Lesbarer kommentarfreier Code ist besser als formal kommentierter, aber unlesbarer Code.

Typischer Einsatz von Kommentaren

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

Hinweise zum Verfassen von Kommentaren

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

Professionell kommentierter Code

```
/*
 * @(#)Observer.java 1.14 98/06/29
 * Copyright 1994-1998 by Sun Microsystems, Inc., ...
 */
package java.util;

/**
 * A class can implement the <code>Observer</code> interface when it
 * wants to be informed of changes in observable objects.
 *
 * @author  Chris Warth
 * @version 1.14, 06/29/98
 * @see     java.util.Observable
 * @since   JDK1.0
 */
public interface Observer {
    /**
     * This method is called whenever the observed object is changed. An
     * application calls an <tt>Observable</tt> object's
     * <code>notifyObservers</code> method to have all the object's
     * observers notified of the change.
     *
     * @param   o     the observable object.
     * @param   arg   an argument passed to the
     *                <code>notifyObservers</code> method.
     */
    void update(Observable o, Object arg);
}
```


Anhang: Elements of Programming Style

Auszüge aus: Kernighan/Plauger,
The Elements of Programming Style, McGraw-Hill 1978 (!)

- Write clearly - don't be too clever.
- Don't sacrifice clarity for efficiency.
- Each module should do one thing well.
- Make sure every module hides something.
- Use the good features of a language, avoid the bad ones.
- Use the "telephone test" for readability.
- Don't stop with your first draft.
- Don't patch bad code - rewrite it.
- Don't comment bad code - rewrite it.
- Make it right before you make it faster.
- Make it clear before you make it faster.
- Keep it right when you make it faster.
- Let your compiler do the simple optimizations.
- Measure your program before making "efficiency" changes.

- Dies wurde z.B. in Extreme Programming zu einer wesentlichen Maxime erhoben!

Anhang: Elements of Java Style

- Auszüge aus: Vermeulen et al.,
  The Elements of Java Style, Cambridge University Press 2000
- Bezeichnerwahl:
  - Join the vowel generation. ("appendSignature" statt "appdSgtr")
  - Capitalize only the first letter in acronyms. ("loadXmlDocument")
  - Pluralize the names of classes that group static services or constants.
  - Follow the JavaBeans conventions for property access ("get"/"set").
- Programmierung:
  - Follow the "Open/Closed" principle: Software entities should be open for extension but closed for modification.
  - Use assertions to test pre- and postconditions of a method.
  - Exceptions:
    - » Unchecked (runtime) exceptions for serious unexpected errors.
    - » Checked exceptions for errors that may appear in normal program execution.

Zusammenfassung
6.3 Stilfragen der Codierung

- Es gibt (leider) keinen allgemein anerkannten einheitlichen Codingstandard.
- Wichtig ist daher zunächst die Projekt-/Firmen-interne Einigung auf einen Standard!
- Wesentliche Kriterien:
  - Aussagekräftige und kompakte Kommentierung,
  - Namenswahl,
  - Einrückung
  - Größe von Codeblöcken (Methoden)
  - Größe von Modulen (Klassen)
- Java bietet im Internet zugängliche Standards (die weitgehend kompatibel sind)
- Codingstandards verhindern auch „Hacker-Code“

Softwaretechnik WS 02/03

6. Implementierung

6.4. Datenstrukturen in Java

Analyse	Entwurf	Implemen-tierung	Test, Integration	Wartung

Literatur: **einschlägige Java-Bücher, z.B. David Flanagan: Java in a Nutshell**

OO-Design und Nicht-OO-Sprache

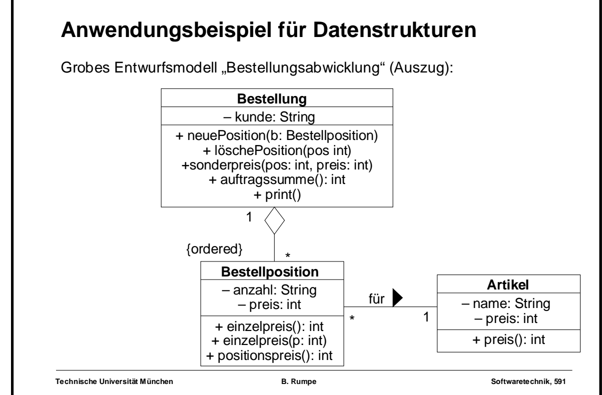
- Elementare Konzepte der Objektorientierung lassen sich teilweise befriedigend in *modularen* Sprachen wiedergeben:
  - Verhalten: Zusammenfassung von Prozeduren in Modulen
  - Zustand: Datengeheimnis von Modulen
  - Identität: Zeigertypen
- *Nichtmodulare* Sprachen (z.B. Original-Pascal) sind dagegen praktisch ungeeignet für OO-Entwurf.
- Fortgeschrittene Konzepte der Objektorientierung lassen sich in jedem Fall nur mit Hilfskonstruktionen darstellen, insbesondere:
  - Vererbung, vor allem von Operationen
  - Polymorphie
- Mögliche Strategien:
  - Im Entwurf Vererbung und Polymorphie entfernen
  - Einsatz von Werkzeugen (Präprozessoren)
  - OO-Erweiterung der Sprache (z.B. Object-COBOL)

Bedeutung von Datenstrukturen

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

Anwendungsbeispiel für Datenstrukturen

Grobes Entwurfsmodell „Bestellungsabwicklung“ (Auszug):



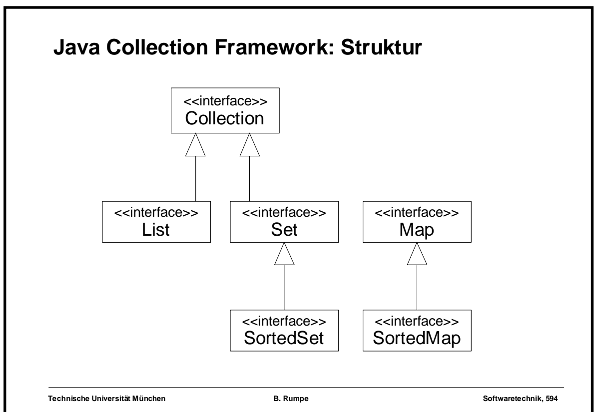
Einfache Realisierung mit Arrays (1)

```
class Bestellung {

  private String kunde;
  private Bestellposition[] liste;
  private int anzahl = 0;

  public Bestellung(String kunde) {
    this.kunde = kunde;
    liste = new Bestellposition[20];
  }

  public void neuePosition (Bestellposition b) {
    liste[anzahl] = b;
    anzahl++;
      // was passiert bei mehr als 20 Positionen ?
  }

  public void loeschePosition (int pos) {
      // geht mit Arrays nicht einfach zu realisieren !
  }
```


Java-2 Collection Framework

- Objektorientierte Datenstrukturbibliothek für Java
  - Die meisten Standard-Datenstrukturen werden abgedeckt
  - Verwendung von Vererbung zur Strukturierung
  - Flexibel auch zur eigenen Erweiterung
- „Geschichte“:
  - Collection Framework basiert auf Java Generic Library (JGL) der Firma ObjectSpace
  - Collection Framework ist erst seit Java 1.2 (Java-2) Standard
    - » java.util.*
  - „Früher“, d.h. bis JDK-Version 1.1:
    - » Java-Datenstrukturen Vector und Hashtable
    - » JGL zusätzlich installierbar

Java Collection Framework: Struktur



Klassifikation von Datenstrukturen

- Collection (Kollektion):
  - Ansammlung von Datenelementen
  - Hinzufügen, Entfernen, Suchen, Durchlaufen
- Set (Menge):
  - Keine Mehrfachvorkommen
  - Reihenfolge des Einfügens spielt keine Rolle
  - SortedSet (geordnete Menge): Ordnung auf den Elementen
- List (Liste):
  - Mehrfachvorkommen werden separat abgelegt
  - Reihenfolge des Einfügens bleibt erhalten
- Map (Abbildung):
  - Zuordnung von Schlüsselwerten auf Eintragswerte
  - Mehrfachvorkommen bei Schlüsseln verboten, bei Einträgen erlaubt
  - SortedMap (geordnete Abbildung): Ordnung auf den Schlüsseln

java.util.Collection (Auszug)

```
public interface Collection {

  public boolean add (Object o);
  public boolean remove (Object o);
  public void clear();
  public boolean isEmpty();
  public boolean contains (Object o);
  public int size();
  ...
}
```

Es gibt zwei Alternativen für allgemeine Datenstrukturen:
- Parametrisierung der Datenstruktur mit einem Datentyp (Templates) wie in C++
- Verwendung der allgemeinen Oberklasse aller Objekte (Object) wie in Java

java.util.List (Auszug)

```
public interface List extends Collection {

  public boolean add (Object o);
  public boolean remove (Object o);
  public void clear();
  public boolean isEmpty();
  public boolean contains (Object o);
  public int size();

  public Object get (int index);
  public Object set (int index, Object element);
  public Object remove (int index);
  public int indexOf (Object o);
  ...
}
```

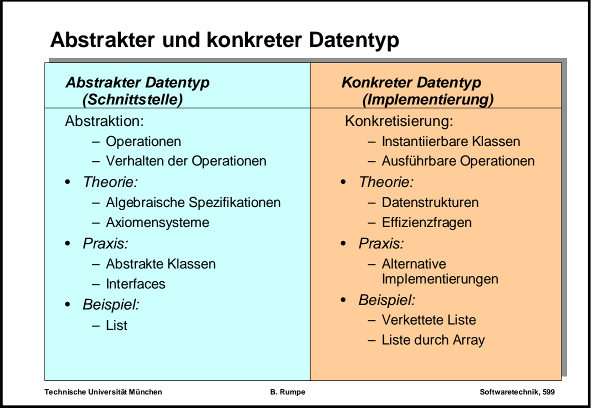

Die Klasse "Object"

- `java.lang.Object`: allgemeine Eigenschaften aller Objekte.
  - Jede Klasse ist Unterklasse von `Object` ("`extends Object`").
  - Diese Vererbung ist *implizit* (d.h. man kann "`extends Object`" weglassen).

```
class Object {
      public boolean equals (Object obj);
      public int hashCode();
      public String toString(); ...
}
```

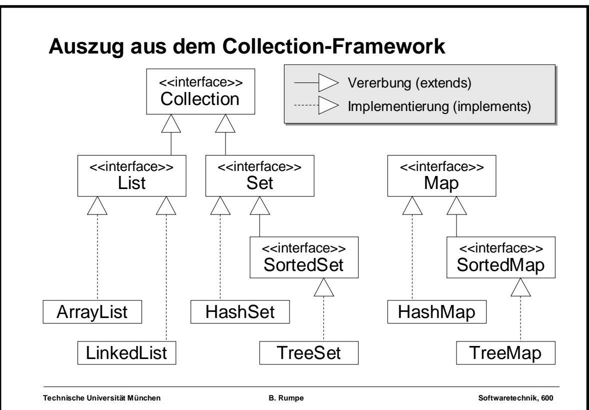
- Jede Klasse kann die Standard-Operationen redefinieren:
  - equals: Objektgleichheit (Standard: Referenzgleichheit)
  - hashCode: Zahlcodierung
  - toString: Textdarstellung, z.B. für print()

Abstrakter und konkreter Datentyp



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

Auszug aus dem Collection-Framework



java.util.ArrayList (Auszug)

```
public class ArrayList implements List {
  public boolean add (Object o);
  public boolean remove (Object o); //exception!
  public void clear();
  public boolean isEmpty();
  public boolean contains (Object o);
  public int size();
  public Object get (int index);
  public Object set (int index, Object element)
  public Object remove (int index);
  public int indexOf (Object o);
  public ArrayList (int initialCapacity);
  public void ensureCapacity (int minCapacity);
  ...
}
```

Implementierung bietet unter Umständen spezifische zusätzliche Operationen an

Anwendungsbeispiel mit ArrayList (2)

```
public void sonderpreis (int pos, int preis) {
  ((Bestellposition)liste.get(pos)).einzelpreis(preis);
}

public int auftragssumme() {
  int s = 0;
  for(int i=0; i<liste.size(); i++)
    s +=
    ((Bestellposition)liste.get(i)).positionspreis();
  return s;
}
```


Anwendungsbeispiel mit ArrayList (1)

```
import java.util.ArrayList;
...
class Bestellung {
  private String kunde;
  private ArrayList liste;

  public Bestellung(String kunde) {
    this.kunde = kunde;
    liste = new ArrayList();
 }

  public void neuePosition (Bestellposition b) {
    liste.add(b);
  }

  public void loeschePosition (int pos) {
    liste.remove(pos);
  }
...
```


Iterator-Konzept

- Aufzählen der in einem "Behälter" befindlichen Elemente
  - Keine Aussage über die Reihenfolge!
  - Interface java.util.Iterator

```
interface Iterator {
      public abstract boolean hasNext();
      public abstract Object next();
      public abstract void remove();
}
```

Verwendungs-beispiel:

```
Iterator i = ...;
while (i.hasNext()) {
      doSomeThing(i.next());
}
```

- Erzeugung eines Iterators für eine beliebige Kollektion (deklariert in java.util.Collection):

```
public Iterator iterator();
```


Anwendungsbeispiel mit ArrayList (falsch!)

```
...
public void sonderpreis (int pos, int preis) {
  liste.get(pos).einzelpreis(preis);
}
...
```

- Compilermeldung:

„Method einzelpreis(int) not found in class java.lang.Object."

?

```
liste. get(pos). einzelpreis(preis);
```

ArrayList — Object — *definiert auf* Bestellposition

Spezialisierung von Object auf Bestellposition?

Anwendungsbeispiel mit Iteratoren

```
import java.util.Iterator;
...
class Bestellung {

  private String kunde;
  private ArrayList liste;

  ...
  public int auftragssumme() {
    Iterator i = liste.iterator();
    int s = 0;
    while (i.hasNext())
      s += ((Bestellposition)i.next()).positionspreis();
    return s;
  }
  ...
}
```

Standardalgorithmen: java.util.Collections

```
public class Collections {
  public static Object max (Collection coll);
  public static Object min (Collection coll);
  public static int binarySearch
      (List list, Object key);
  public static void reverse (List list);
  public static void sort (List list)
  ...
}
```

- Algorithmen arbeiten mit beliebigen Klassen, die das Collection- bzw. List-Interface implementieren.
- Bei manchen Operationen ist Ordnung auf Elementen vorausgesetzt.
- Statische Operationen: Aufruf z.B. `Collections.sort(...)`

Welche Listen-Implementierung?

- Gemessener relativer Aufwand für Operationen auf Listen: (aus Eckel, Thinking in Java, 2nd ed., 2000)

Typ	Lesen	Iteration	Einfügen	Entfernen
array	1430	3850	--	--
ArrayList	3070	12200	500	46850
LinkedList	16320	9110	110	60
Vector	4890	16250	550	46850

- Stärken von ArrayList:
  - wahlfreier Zugriff
- Stärken von LinkedList:
  - Iteration
  - Einfügen und Entfernen irgendwo in der Liste
- Vector generell die langsamste Lösung

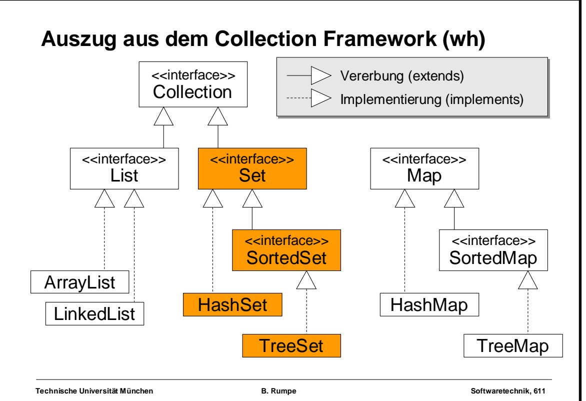
Ordnung auf Elementen: java.lang.Comparable

```
public interface Comparable {
  public int compareTo (Object o);
}
```

Resultat kleiner/gleich/größer 0:
"this" kleiner/gleich/größer als Objekt o

Standarddatentypen (z.B. String) implementieren Comparable

Auszug aus dem Collection Framework (wh)



Selbstdefinierte Collections

- Eigene Implementierungen der Klassen des Collection-Frameworks, z.B.:

```
class MyCollection
  extends AbstractCollection
  implements Collection {
...}
```

- Standard-Gerüste für Implementierung von Kollektionsklassen:
  java.util.AbstractCollection, java.util.AbstractList, ...
- Minimalsignatur, die Collection definieren muss:
  - add()
  - size()
  - iterator()

java.util.Set (Auszug)

```
public interface Set {
  public boolean add (Object o);
  public boolean remove (Object o);
  public void clear();
  public boolean isEmpty();
  public boolean contains (Object o);
  public int size();
  ...
  public boolean equals (Object o);
  public int hashCode();
  ...
  public Iterator iterator();
}
```

Schnittstelle wie bei Collection. Dennoch ist mit "Set" die zusätzliche Semantik verbunden, dass die Reihenfolge und Wiederholung von Einfügeopreationen unerheblich ist.

Anwendungsbeispiel mit HashSet

```
class Warengruppe {
  private String name;
  private String lagerplatz;
  private Set inhalt;
  public Warengruppe
     (String name, String lagerplatz) {
    this.name = name;
    this.lagerplatz = lagerplatz;
    this.inhalt = new HashSet();
  }
  public void add (Artikel a) { inhalt.add(a); }
  public int anzahl() { return inhalt.size(); }
  public String toString() {
    String s = "Warengruppe "+name+"\n";
    Iterator it = inhalt.iterator();
    while (it.hasNext()) {
      s += " "+(Artikel)it.next();
    };
  }
}
```


Wann sind Objekte gleich?

- Vergleich mit Operation `==` :
  - Referenzgleichheit, d.h. physische Identität der Objekte
  - Typischer Fehler: Stringvergleich mit "=="
- Vergleich mit `o.equals()`:
  - deklariert in java.lang.Object
  - redefiniert in vielen Bibliotheksklassen
    - » z.B. java.lang.String
  - für selbstdefinierte Klassen
    - » Standardbedeutung Referenzgleichheit
    - » bei Bedarf selbst redefinieren !
      (Ggf. für kompatible Definition der Operation `o.hashCode()` aus java.lang.Object sorgen)

java.util.SortedSet (Auszug)

```
public interface SortedSet extends Set {
  public boolean add (Object o);
  public boolean remove (Object o);
  public void clear();
  public boolean isEmpty();
  public boolean contains (Object o);
  public int size();
  ...
  public boolean equals (Object o);
  public int hashCode();
  public Iterator iterator();
  ...
  public Object first();
  public Object last(); ...
}
```


java.util.TreeSet

- Modifikation der Klasse Warengruppe:

```
class Warengruppe {
  private Set inhalt;
  public Warengruppe (…) {
    …
    this.inhalt = new TreeSet();
  } …
}
```

- Systemreaktion:

Exception in thread "main" java.lang.ClassCastException: Artikel
at java.util.TreeMap.compare(TreeMap.java, Compiled Code)

- Grund: java.util.TreeSet:

```
public class TreeSet … implements SortedSet … {
 … }
```


Anwendungsbeispiel mit TreeSet

- TreeSet benötigt, dass die Objekte das Interface Comparable anbieten
- Deshalb die Modifikation der Klasse „Artikel“:

```
class Artikel implements Comparable {
...
    public int compareTo (Object o) {
      return name.compareTo(((Artikel)o).name);
    }
}
```

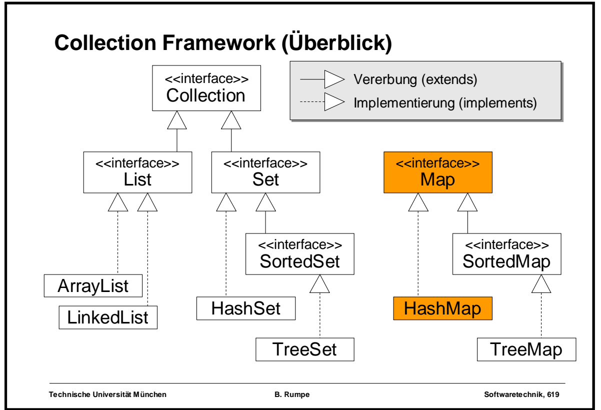

HashSet oder TreeSet?

- Gemessener relativer Aufwand für Operationen auf Mengen:
  (aus Eckel, Thinking in Java, 2nd ed., 2000)

Typ	Einfügen	Enthalten	Iteration
HashSet	36,14	106,5	39,39
TreeSet	150,6	177,4	40,04

- Stärken von HashSet:
  - in allen Fällen schneller !
- Stärken von TreeSet:
  - erlaubt Operationen für sortierte Mengen

Collection Framework (Überblick)



Anwendungsbeispiel mit HashMap

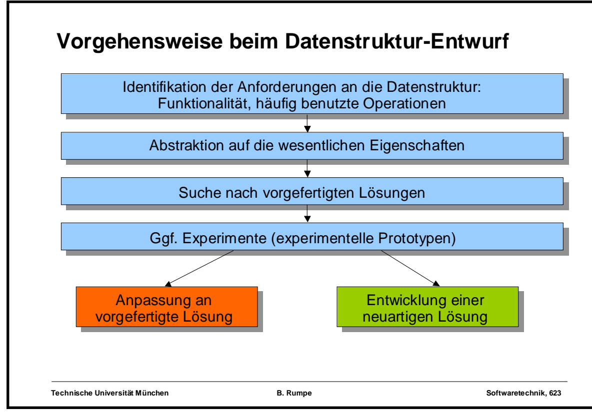
```
class Katalog {
  private String name;
  private Map inhalt;
  public Katalog (String name) {
    this.name = name;
    this.inhalt = new HashMap();
  }
  public void put (String code, Artikel a) {
    inhalt.put(code,a);
  }
  public int anzahl() {
    return inhalt.size();
  }
  public Artikel get (String code) {
    return (Artikel)inhalt.get(code);
  }
  ...
}
```


java.util.Map (Auszug)

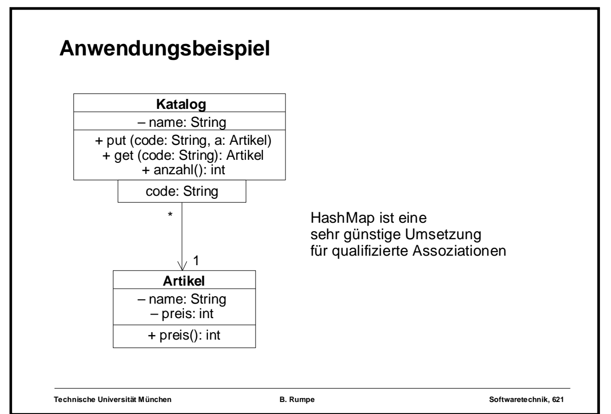
```
public interface Map {
  ...
  public boolean containsKey (Object key);
  public boolean containsValue (Object value);
  public Object get (Object key);
  public Object put (Object key, Object value);
  public Object remove (Object key);
  public int size();
  public Set keySet();
  public Collection values();
  ...
}
```

„Assoziativer Speicher“, „Abbildung“

Vorgehensweise beim Datenstruktur-Entwurf

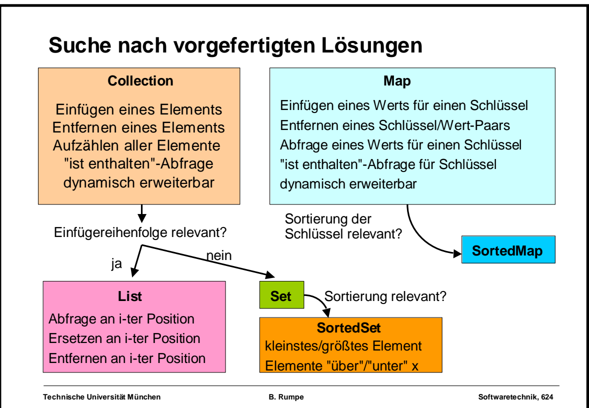


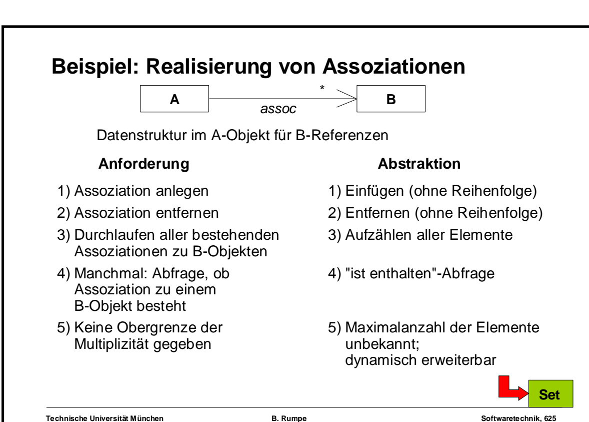
Anwendungsbeispiel



HashMap ist eine sehr günstige Umsetzung für qualifizierte Assoziationen

Suche nach vorgefertigten Lösungen





Zusammenfassung
6.4 Datenstrukturen in Java

- Das Java-Collection-Framework bietet für die wichtigsten Datenstrukturen (Menge, Sequenz, Abbildung) mehrere Implementierungen an.
- Die Auswahl der genutzten Struktur hängt von den Anwendungscharakteristika ab
- Abstrakte Schnittstellen (Map, Set, Collection, List) erlauben
  - den Wechsel der Implementierung
  - Entwicklung eigener Implementierungen
- Collections werden u.a. bei der Umsetzung von Assoziationen, Aggregation und Komposition eingesetzt
- Wesentliche Grundfunktionalität wird angeboten:
  - Vergleiche (equals, == und compareTo)
  - Umwandlung in String (toString)
  - Serialisierung, ...
- Wiederverwendung der effizient realisierten Implementierungen ist geboten.

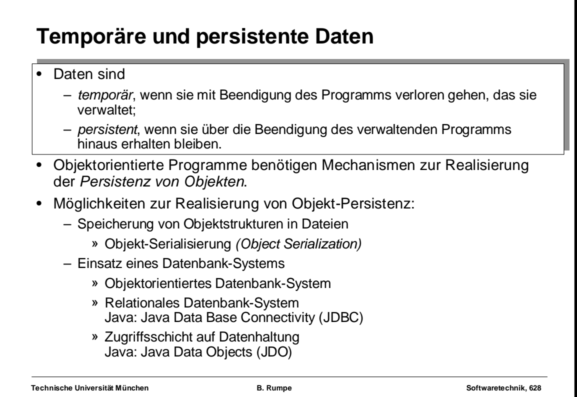
Softwaretechnik WS 02/03

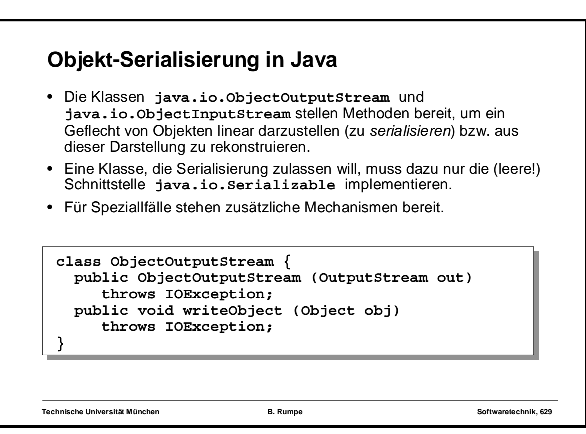
6. Implementierung

6.5. Persistenz und Datenbank-Anbindung

Analyse | Entwurf | Implementierung | Test, Integration | Wartung

Literatur: **Balzert LE 31 und LE 24-26 (Grundlagen)**
Ambler Kap. 10





Objekt-Serialisierung: Abspeichern

```
import java.io.*;

class XClass implements Serializable {
  private int x;
  public XClass (int x) {
    this.x = x;
  }
}

...
XClass xobj;
...
FileOutputStream fs = new FileOutputStream("Xfile.dat");
ObjectOutputStream os = new ObjectOutputStream(fs);
os.writeObject(xobj);
...
```

Objekt-Serialisierung: Einlesen

```
import java.io.*;

class XClass implements Serializable {
  private int x;
  public XClass (int x) {
    this.x = x;
  }
}

...
XClass xobj;
...
FileInputStream fs = new FileInputStream("Xfile.dat");
ObjectInputStream os = new ObjectInputStream(fs);
xobj = (XClass) os.readObject();
```


Einsatz von Datenbanksystemen

- Persistente Daten
  - Dateien in Austauschformaten
  - Programmiersprachenspezifische Mechanismen
  - Datenbanksysteme
    - » relational
    - » objektorientiert
    - » andere, z.B. hierarchisch, netzorientiert
- Anwendungsentwurf
  - entkoppeln von Wahl des Persistenzmechanismus
- Altsysteme
  - meist relationale oder hierarchische Datenbank
  - Anbindung an moderne Anwendungsprogramme

- Wegen XML erhalten hierarchische Systeme eine Wiederbelebung

Vergleich RDBS - ODBS

relationales DBS	objektorientiertes DBS
Wertidentität Schlüssel und Fremdschlüssel	Objektidentität
elementare Attributtypen	komplexe Attributtypen
sichtbare Attribute	Kapselung (nicht für lesende Zugriffe)
DB-Sprache (SQL) grundsätzlich separat von Programmiersprache	Enge Integration zwischen DB-Sprache und Progr.spr.
Serverorientiert	Clientorientiert
Persistente Daten	Persistente und transiente Objekte

Objektidentität

- Nachteile der Wertidentifikation von Objekten (Tupeln):
  - Schlüsselattribute können Anwendungssemantik tragen
    - » z.B. 'Kurzname' in 'Firma': Namenswechsel einer Firma?
  - Wertgleichheit (der Attribute) und Identität nicht unterscheidbar
    - » z.B. grafische Objekte in einem Zeichenprogramm:
      - Objekt 1: Kreis, Farbe blau, Koordinaten (1,1)
      - Objekt 2: Kreis, Farbe rot, Koordinaten (2,1)
      - Ändere Farbe von Objekt 2 in blau
      - Verschiebe Objekt 2 um 1 nach links
      - Verschiedene Objekte mit gleichen Attributwerten
- Vorteile einer reinen Objektidentifikation:
  - "Opaker" Typ verhindert versehentliche oder fahrlässige Manipulation
  - "Smart Pointers" (Objekt-Caching)
- Nachteile der Objektidentifikation:
  - zusätzliches Element, weniger effizient
  - zusätzlicher Aufwand, um nach Wertattributen zu suchen, da nicht Schlüssel

Objektorientiertes Design und RDBS

- Objektorientiertes Design (OOD)
  - wegen Flexibilität und Zukunftssicherheit
- Relationales Datenbanksystem (RDBS)
  - wegen vorhandener Altsysteme
  - zur Vermeidung von Lizenzkosten und Schulungsaufwand

- Abbildung von OOD auf RDBS:
  - Abbildung von Klassen auf Tabellen
  - Behandlung von Attributen mit komplexen Typen
  - Abbildung von Assoziationen und Aggregationen
  - Abbildung von Vererbung
  - Softwarehilfsmittel: "Objektrelationale Middleware"

Abbildung von Klassen auf Tabellen

- In der Regel wird jede Klasse auf eine Tabelle abgebildet.
- Objektidentität wird durch eine zusätzliche Spalte aller Tabellen *(surrogate)* realisiert.

Person
Name Adresse [0..1]

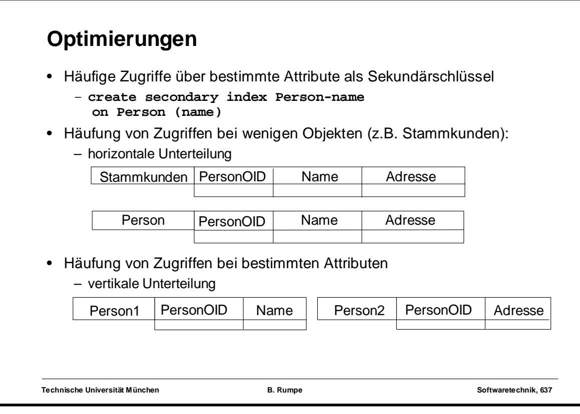
↓

Person	PersonOID	Name	Adresse

Primärschlüssel: PersonOID

↓

```
create table Person
      (PersonOID ID          not null,
       name      char(40)    not null,
       adresse   char(60)
       primary key (PersonOID))
```



Attribute mit komplexen Typen

- Beispiel Attribut 'Adresse' von Klasse 'Firma' (in ODL):

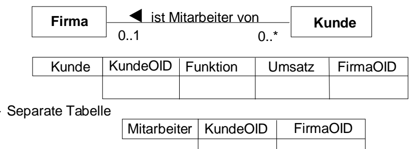
```
attribute struct AdresseT
  <String Straße, Integer HausNr,
  Integer PLZ, String Ort> Adresse;
```

- Neue Attributnamen:
  - `String AdresseStraße`
  - `Integer AdresseHausNr`
  - `Integer AdressePLZ`
  - `String AdresseOrt`
- Alternative: eigene Tabelle 'Adressen'

- Eigene Tabellen nötig für Listentypen (ohne feste Grenzen)

Abbildung von Assoziationen/Aggregationen

- 1:1- und 1:m-Assoziationen:
  - OID-Fremdschlüsselattribut in der 'm-Klasse'

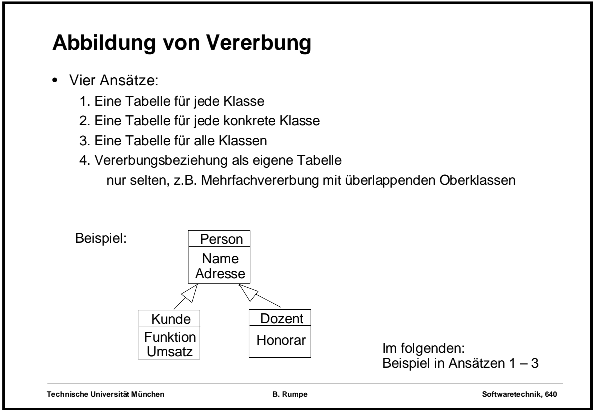


Kunde	KundeOID	Funktion	Umsatz	FirmaOID

  - Separate Tabelle

Mitarbeiter	KundeOID	FirmaOID

- m:n-Assoziationen:
  - eigene Tabelle
- Behandlung von Aggregationen analog, evtl. mit Nutzung spezieller SQL-Konstrukte



Beispiel: Abbildung von Vererbung

- Ansatz 1: Eine Tabelle für jede Klasse

Person	PersonOID	Name	Adresse
	11 22	Huber Schmidt	München Dresden

Kunde	PersonOID	Funktion	Umsatz
	11	Entwickler	500

Dozent	PersonOID	Honorar
	22	2000

  - Einfach und systematisch
  - Gut geeignet für Mehrfachvererbung (aus disjunkten Klassen) und überlappende bzw. wechselnde Rollen
  - Nachteile: Viele Tabellen, komplexe Zugriffe

Beispiel: Abbildung von Vererbung

- Ansatz 2: Eine Tabelle für jede konkrete Klasse (Oberklassenattribute in Unterklassen aufnehmen)

Kunde	PersonOID	Name	Adresse	Funktion	Umsatz
	11	Huber	München	Entwickler	500

Dozent	PersonOID	Name	Adresse	Honorar
	22	Schmidt	Dresden	2000

  - Einfacher Zugriff auf Attribute
  - Besonders günstig wenn es viele abstrakte Klassen gibt
  - Nachteile:
    - Änderungen von Oberklassenattributen betreffen mehrere Tabellen
    - z.B. Name als secondary index für alle Personen nicht möglich
    - Ungünstig für überlappende/wechselnde Rollen

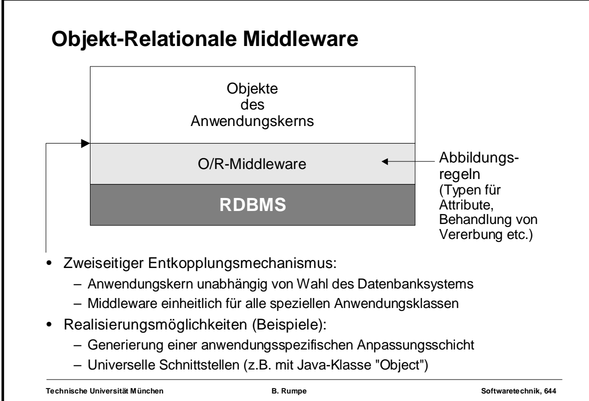
Beispiel: Abbildung von Vererbung

- Ansatz 3: Eine Tabelle für alle Klassen
  (Unterklassenattribute in Oberklassen aufnehmen)

Person	PersonOID	Art	Name	Adresse	Funktion	Umsatz	Honorar
	11	Kunde	Huber	München	Entwickler	500	**null**
	22	Dozent	Schmidt	Dresden	**null**	**null**	2000

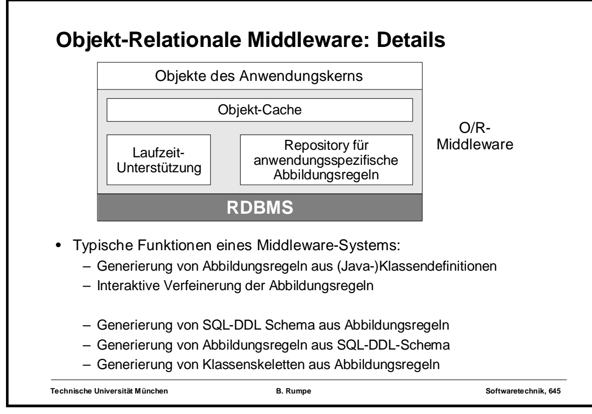
  - Schlecht strukturiert (keine "3. Normalform")
  - Erkennung der Klasse durch nicht gesetzte Werte (**null**-Einträge)
  - Wenig speichereffizient (**null**-Einträge)
  - Nur geeignet bei wenigen Unterklassen mit wenigen Attributen
  - Gut geeignet für überlappende/wechselnde Rollen

Objekt-Relationale Middleware



- Zweiseitiger Entkopplungsmechanismus:
  - Anwendungskern unabhängig von Wahl des Datenbanksystems
  - Middleware einheitlich für alle speziellen Anwendungsklassen
- Realisierungsmöglichkeiten (Beispiele):
  - Generierung einer anwendungsspezifischen Anpassungsschicht
  - Universelle Schnittstellen (z.B. mit Java-Klasse "Object")

Objekt-Relationale Middleware: Details



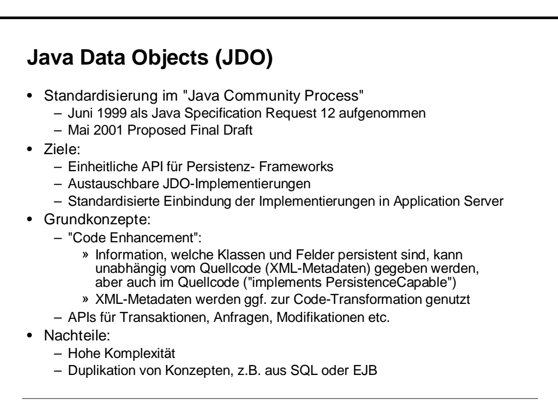
- Typische Funktionen eines Middleware-Systems:
  - Generierung von Abbildungsregeln aus (Java-)Klassendefinitionen
  - Interaktive Verfeinerung der Abbildungsregeln

  - Generierung von SQL-DDL Schema aus Abbildungsregeln
  - Generierung von Abbildungsregeln aus SQL-DDL-Schema
  - Generierung von Klassenskeletten aus Abbildungsregeln

Implementierungsalternativen

- Objektrelationale Middleware
  - leistungsfähige und flexible Lösung
  - teuer, komplex, Performanceverlust
  - oft proprietär (Standardisierung für Java: Java Data Objects JDO)
  - oft Teil der Infrastruktur (z.B. Application Server, Java: EJB)
- Standardisierte DBMS-Schnittstellen
  - z.B. SQL-Einbindung in Programmiersprache (Java: JDBC)
  - Festlegung auf relationale Datenbanken mit passender Software
- Proprietäre DBMS-Schnittstellen
  - unflexibelste Lösung
  - Unter Umständen wegen Performance-Gewinnen sinnvoll
- Professioneller Softwareentwurf:
  - Abwägen zwischen Alternativen
  - Evtl. Entscheidung mit Experimenten absichern
    (experimentelles Prototyping)

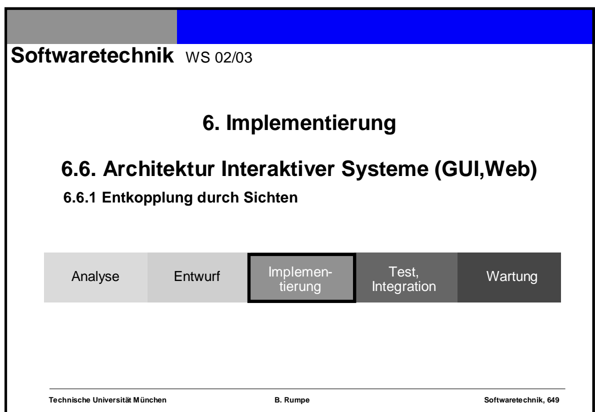
Java Data Objects (JDO)

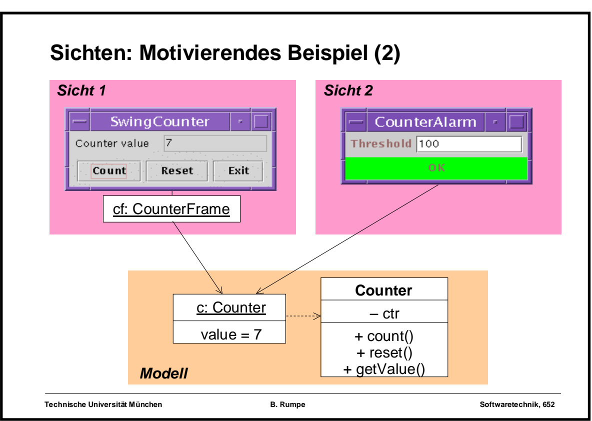


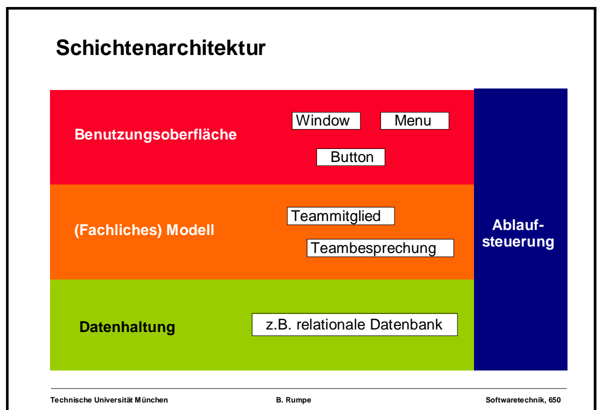
- Standardisierung im "Java Community Process"
  - Juni 1999 als Java Specification Request 12 aufgenommen
  - Mai 2001 Proposed Final Draft
- Ziele:
  - Einheitliche API für Persistenz- Frameworks
  - Austauschbare JDO-Implementierungen
  - Standardisierte Einbindung der Implementierungen in Application Server
- Grundkonzepte:
  - "Code Enhancement":
    - » Information, welche Klassen und Felder persistent sind, kann unabhängig vom Quellcode (XML-Metadaten) gegeben werden, aber auch im Quellcode ("implements PersistenceCapable")
    - » XML-Metadaten werden ggf. zur Code-Transformation genutzt
  - APIs für Transaktionen, Anfragen, Modifikationen etc.
- Nachteile:
  - Hohe Komplexität
  - Duplikation von Konzepten, z.B. aus SQL oder EJB

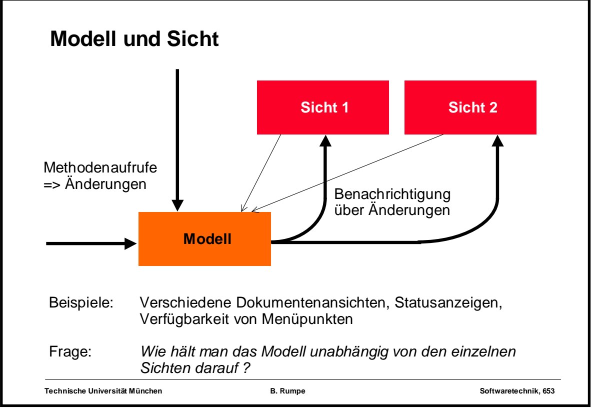
Zusammenfassung:
6.5. Persistenz und Datenbank-Anbindung

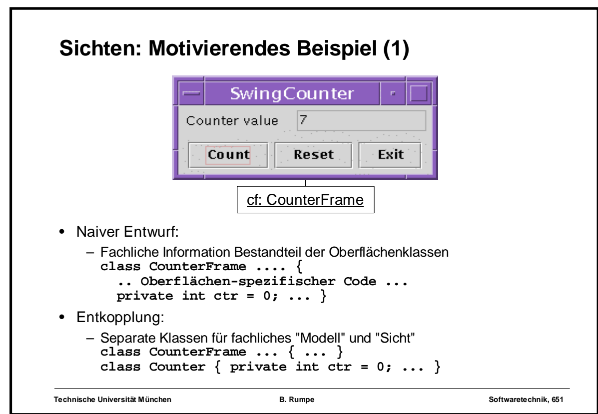
- Speicherformen:
  - Datenbank
  - Datei mit eigenem Format, Java-Serialisierung oder XML-Dialekt
- OO Daten werden heute zumeist in relationale Datenbanken gespeichert
  - Problem der Umsetzung von OO in relationale Modelle:
    - » Generierung von Abbildungen
    - » Manuelle, optimierende Definition
    - » Objekt-relationale Middleware
- Trade-Off zwischen
  - Effizienz (Durchsatz, Reaktionszeit),
  - Qualität der Datenspeicherung (Transaktionen, Ausfallsicherheit, ...) und
  - Aufwand zur Implementierung

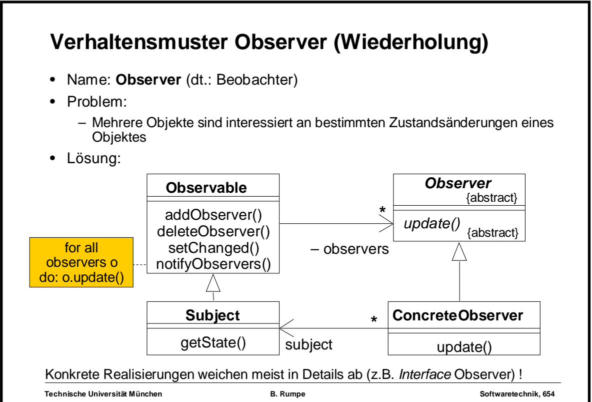

Softwaretechnik WS 02/03
6. Implementierung
6.6. Architektur Interaktiver Systeme (GUI,Web)
6.6.1 Entkopplung durch Sichten
Analyse
Entwurf
Implemen-tierung
Test, Integration
Wartung
Technische Universität München
B. Rumpe
Softwaretechnik, 649


Sichten: Motivierendes Beispiel (2)
Sicht 1
Sicht 2
SwingCounter
Counter value
7
Count
Reset
Exit
CounterAlarm
Threshold
100
cf: CounterFrame
c: Counter
value = 7
Counter
– ctr
+ count()
+ reset()
+ getValue()
Modell
Technische Universität München
B. Rumpe
Softwaretechnik, 652


Schichtenarchitektur
Benutzungsoberfläche
Window
Menu
Button
(Fachliches) Modell
Teammitglied
Teambesprechung
Datenhaltung
z.B. relationale Datenbank
Ablauf-steuerung
Technische Universität München
B. Rumpe
Softwaretechnik, 650


Modell und Sicht
Sicht 1
Sicht 2
Methodenaufrufe
=> Änderungen
Benachrichtigung
über Änderungen
Modell
Beispiele: Verschiedene Dokumentenansichten, Statusanzeigen, Verfügbarkeit von Menüpunkten
Frage: Wie hält man das Modell unabhängig von den einzelnen Sichten darauf ?
Technische Universität München
B. Rumpe
Softwaretechnik, 653

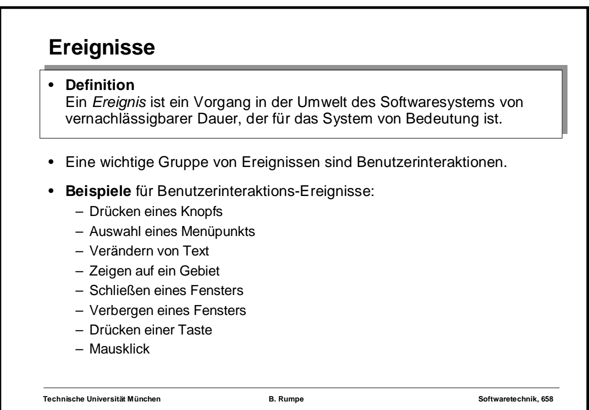

Sichten: Motivierendes Beispiel (1)
SwingCounter
Counter value
7
Count
Reset
Exit
cf: CounterFrame
• Naiver Entwurf:
– Fachliche Information Bestandteil der Oberflächenklassen
class CounterFrame .... {
.. Oberflächen-spezifischer Code ...
private int ctr = 0; ... }
• Entkopplung:
– Separate Klassen für fachliches "Modell" und "Sicht"
class CounterFrame ... { ... }
class Counter { private int ctr = 0; ... }
Technische Universität München
B. Rumpe
Softwaretechnik, 651

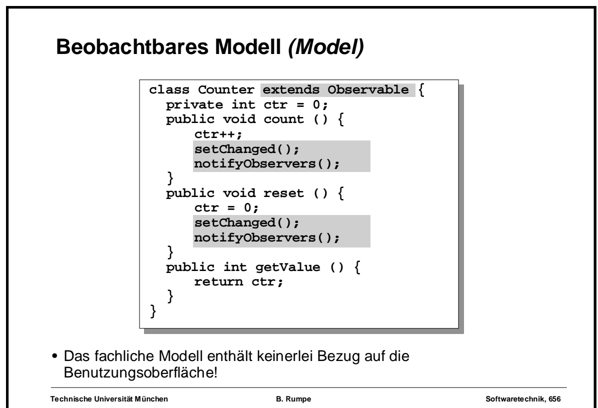

Verhaltensmuster Observer (Wiederholung)
• Name: Observer (dt.: Beobachter)
• Problem:
– Mehrere Objekte sind interessiert an bestimmten Zustandsänderungen eines Objektes
• Lösung:
Observable
addObserver()
deleteObserver()
setChanged()
notifyObservers()
Observer
{abstract}
update()
{abstract}
– observers
for all observers o do: o.update()
Subject
getState()
subject
ConcreteObserver
update()
Konkrete Realisierungen weichen meist in Details ab (z.B. Interface Observer) !
Technische Universität München
B. Rumpe
Softwaretechnik, 654

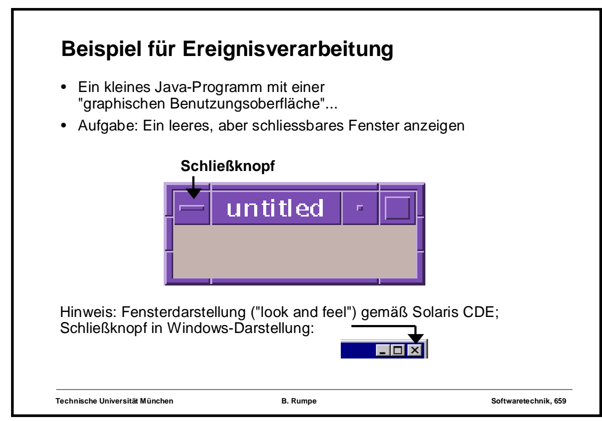
Ein Zähler (Beispiel fachliches Modell)

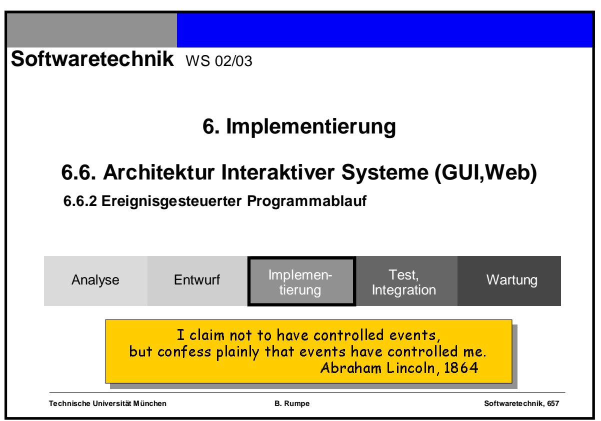
```
class Counter                  {
  private int ctr = 0;
  public void count () {
      ctr++;


  }
  public void reset () {
      ctr = 0;


  }
  public int getValue () {
      return ctr;
  }
}
```

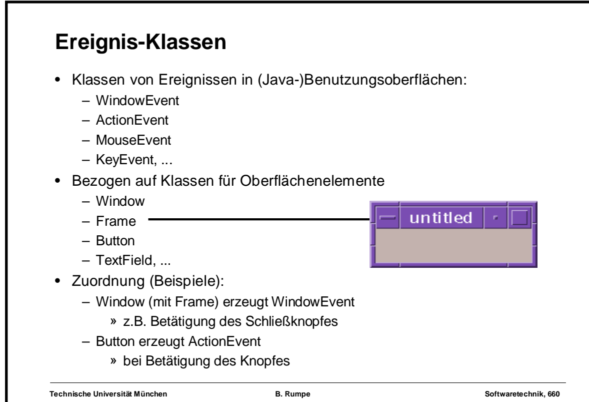
Technische Universität München B. Rumpe Softwaretechnik, 655







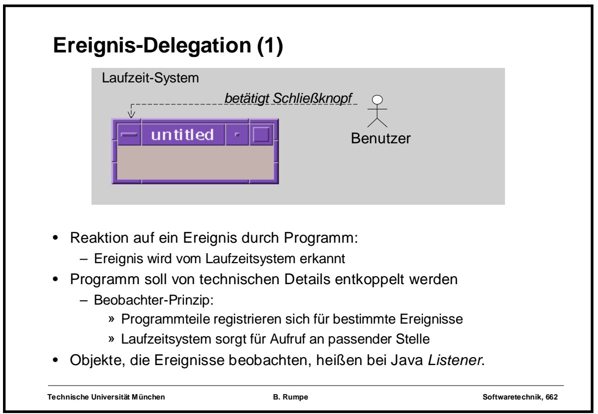




Hauptprogramm für Fensteranzeige

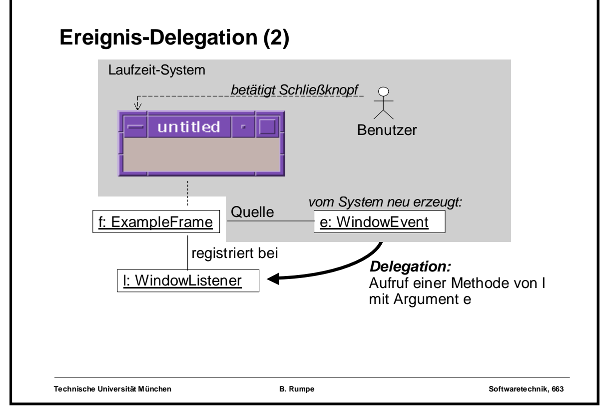
```
// eigener Code
import java.awt.*;

class ExampleFrame extends Frame {

  public ExampleFrame () {
     setTitle("untitled");
     setSize(150, 50);
     setVisible(true);
   }
}

class GUI1 {
  public static void main (String[] argv) {
    ExampleFrame f = new ExampleFrame();
  }
}
```


Ereignis-Delegation (1)



- Reaktion auf ein Ereignis durch Programm:
  - Ereignis wird vom Laufzeitsystem erkannt
- Programm soll von technischen Details entkoppelt werden
  - Beobachter-Prinzip:
    - » Programmteile registrieren sich für bestimmte Ereignisse
    - » Laufzeitsystem sorgt für Aufruf an passender Stelle
- Objekte, die Ereignisse beobachten, heißen bei Java *Listener.*

Ereignis-Delegation (2)



Registrierung für Listener

- In java.awt.Frame (ererbt von java.awt.Window):

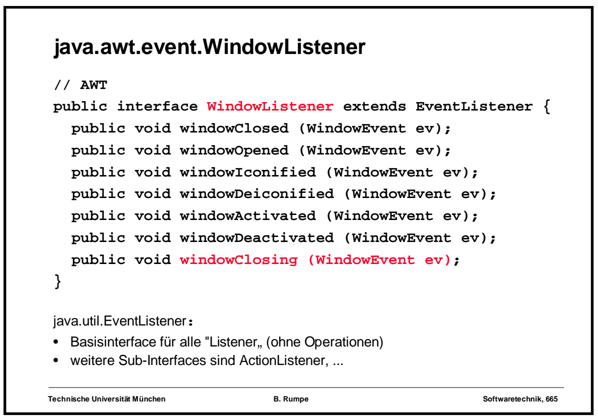
```
public class Frame ... {              // AWT
  public void addWindowListener
     (WindowListener l)
}
```

- java.awt.event.WindowListener ist eine Schnittstelle:

```
public interface WindowListener {    // AWT
  ... Methoden zur Ereignisbehandlung
}
```

- Vergleich mit Observer-Muster:
  - Frame bietet einen "Observable"-Mechanismus
  - Window-Listener ist eine "Observer"-Schnittstelle

java.awt.event.WindowListener



```
// AWT
public interface WindowListener extends EventListener {
  public void windowClosed (WindowEvent ev);
  public void windowOpened (WindowEvent ev);
  public void windowIconified (WindowEvent ev);
  public void windowDeiconified (WindowEvent ev);
  public void windowActivated (WindowEvent ev);
  public void windowDeactivated (WindowEvent ev);
  public void windowClosing (WindowEvent ev);
}
```

java.util.EventListener:

- Basisinterface für alle "Listener„ (ohne Operationen)
- weitere Sub-Interfaces sind ActionListener, ...

java.awt.event.WindowEvent

```
// AWT
public class WindowEvent extends AWTEvent {
  ...
  // Konstruktor, wird vom System aufgerufen
  public WindowEvent (Window source, int id);

  // Abfragemöglichkeiten
  public Window getWindow();
  ...
}
```

WindowListener für Ereignis "Schließen"

```
// eigener Code
import java.awt.*;
import java.awt.event.*;
class WindowCloser implements WindowListener {

  public void windowClosed (WindowEvent ev) {}
  public void windowOpened (WindowEvent ev) {}
  public void windowIconified (WindowEvent ev) {}
  public void windowDeiconified (WindowEvent ev) {}
  public void windowActivated (WindowEvent ev) {}
  public void windowDeactivated (WindowEvent ev) {}

  public void windowClosing(WindowEvent event) {
    System.exit(0);
  }
}
```

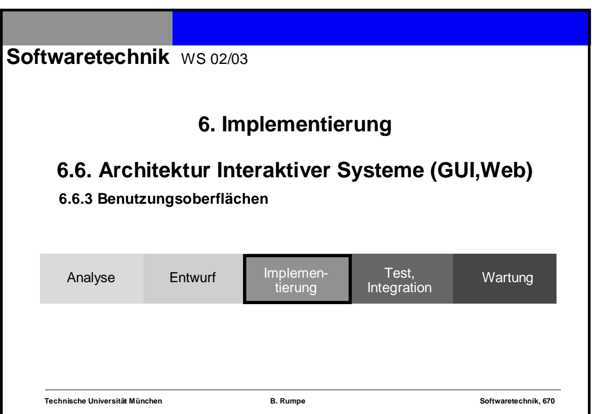

Erweiterung Programm für schließbares Fenster

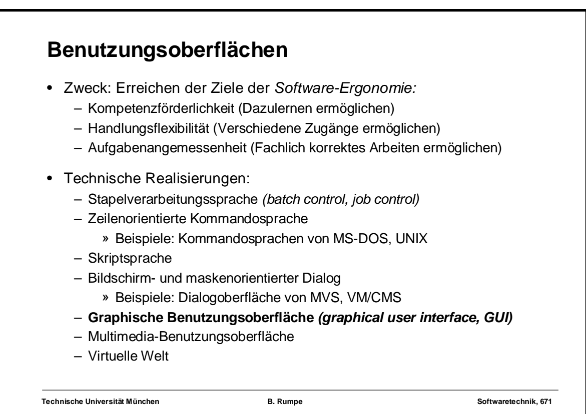
```
// eigener Code
import java.awt.*;
import java.awt.event.*;

class ExampleFrame extends Frame {

  public ExampleFrame () {
      setTitle("untitled");
      setSize(150, 50);
      addWindowListener(new WindowCloser());
      setVisible(true);
    }
}
...
```

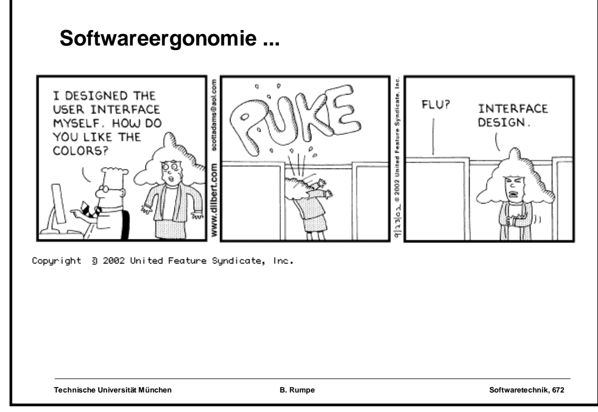
Vereinfachungen, die den Programmierkomfort erhöhen:
- **java.awt.WindowAdapter bietet bereits leere Implementierungen für WindowListener**
- **WindowCloser kann als innere oder sogar anonyme Klasse des ExampleFrame definiert werden.**

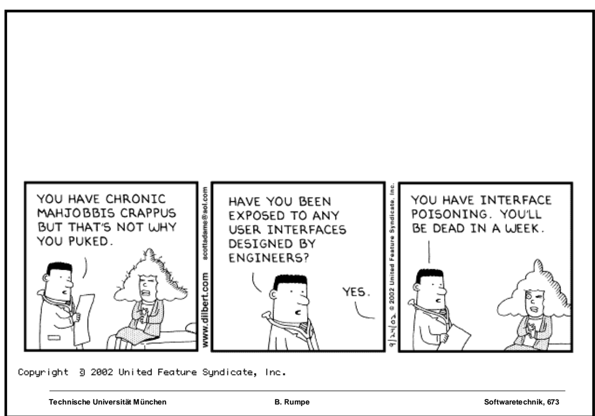
Ergebnis: Vereinfachung des Programmcodes

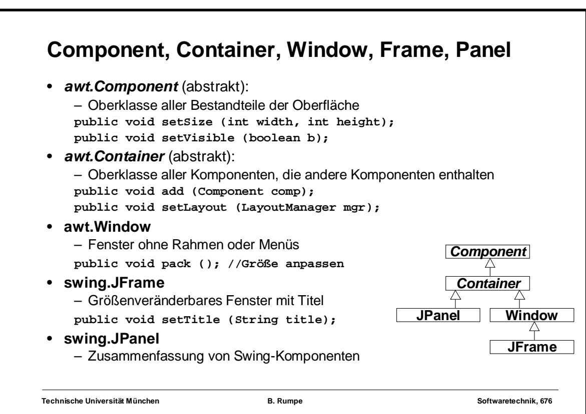
```
import java.awt.*;
import java.awt.event.*;

class ExampleFrame extends Frame {

  public ExampleFrame () {
      setTitle("untitled");
      setSize(150, 50);
      addWindowListener(new WindowAdapter() {
       public void windowClosing(WindowEvent event) {
         System.exit(0);
        }});
      setVisible(true);
    }
}

class GUI5 {
  public static void main (String[] argv) {
      ExampleFrame f = new ExampleFrame();
    }
}
```

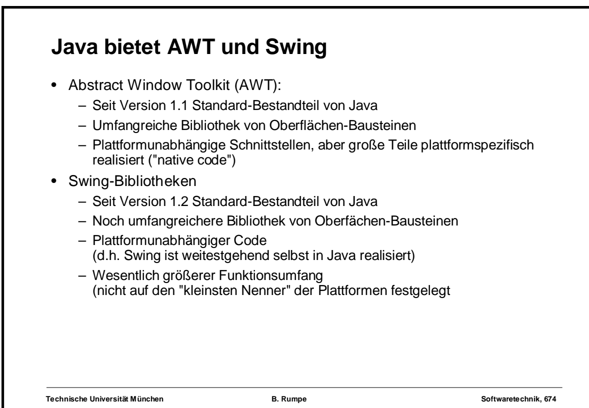



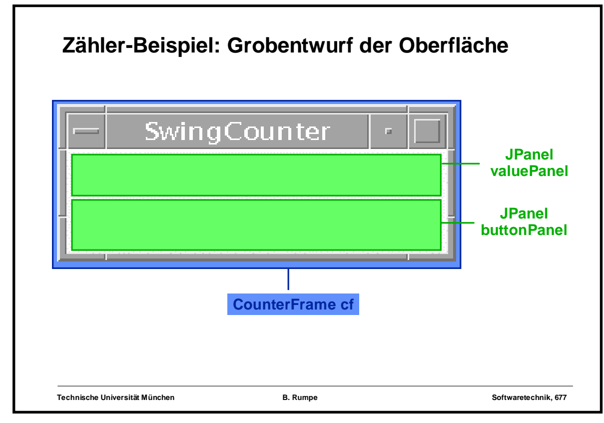












Bibliotheken von AWT und Swing

- Wichtigste AWT-Pakete:
  - **java.awt**: u.a. Grafik, Oberflächenkomponenten, Layout-Manager
  - **java.awt.event**: Ereignisbehandlung
  - Andere Pakete für weitere Spezialzwecke
- Wichtigstes Swing-Paket:
  - **javax.swing**: Oberflächenkomponenten
  - Andere Pakete für Spezialzwecke
- Viele AWT-Klassen werden auch in Swing verwendet!
- Standard-Vorspann:

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
```

- (Naiver) Unterschied zwischen AWT- und Swing-Komponenten:
  - AWT: Button, Frame, Menu, ...
  - Swing: JButton, JFrame, JMenu, ...

Die Sicht *(View)*: Gliederung, 1. Versuch

```
class CounterFrame extends JFrame {
  JPanel valuePanel = new JPanel();

  JPanel buttonPanel = new JPanel();



  public CounterFrame (Counter c) {
    setTitle("SwingCounter");


    ... valuePanel zu this hinzufügen



    ... buttonPanel zu this hinzufügen
    pack();
    setVisible(true);
  }
}
```

Hinzufügen von Komponenten zu JFrames

- Ein JFrame ist ein "Container", d.h. dient zur Aufnahme weiterer Elemente.
- Ein JFrame ist intern in verschiedene "Scheiben" *(panes)* organisiert. Die wichtigste ist die *content pane.*

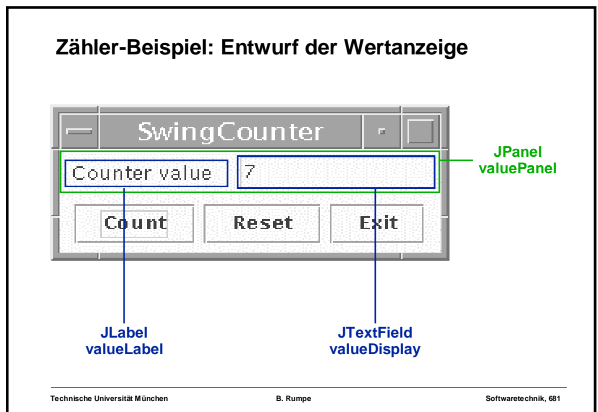
content pane
menu bar
layered pane
glass pane

- In JFrame ist definiert:
  `Container getContentPane();`

Die Sicht *(View):* Gliederung, 2. Versuch

```
class CounterFrame extends JFrame {
  JPanel valuePanel = new JPanel();

  JPanel buttonPanel = new JPanel();



  public CounterFrame (Counter c) {
    setTitle("SwingCounter");


    getContentPane().add(valuePanel);


    getContentPane().add(buttonPanel);
    pack();
    setVisible(true);
  }
}
```

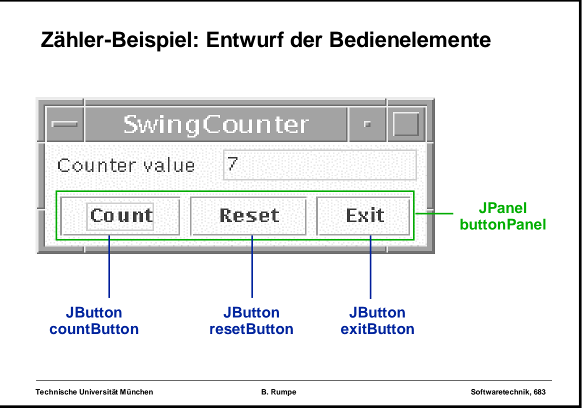

Zähler-Beispiel: Entwurf der Wertanzeige



Die Sicht *(View):* Elemente der Wertanzeige

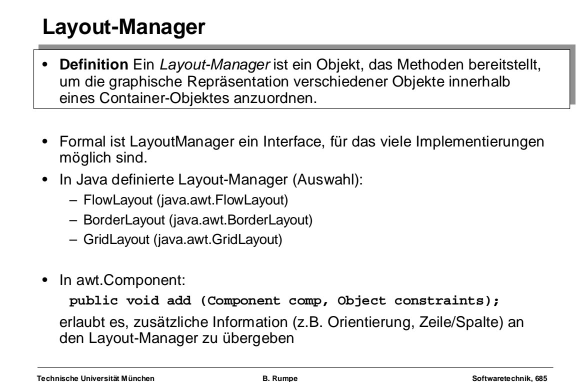
```
class CounterFrame extends JFrame {
  JPanel valuePanel = new JPanel();
  JTextField valueDisplay = new JTextField(10);
  JPanel buttonPanel = new JPanel();



  public CounterFrame (Counter c) {
    setTitle("SwingCounter");
    valuePanel.add(new JLabel("Counter value"));
    valuePanel.add(valueDisplay);
    valueDisplay.setEditable(false);
    getContentPane().add(valuePanel);


    getContentPane().add(buttonPanel);
    pack();
    setVisible(true);
  }
}
```

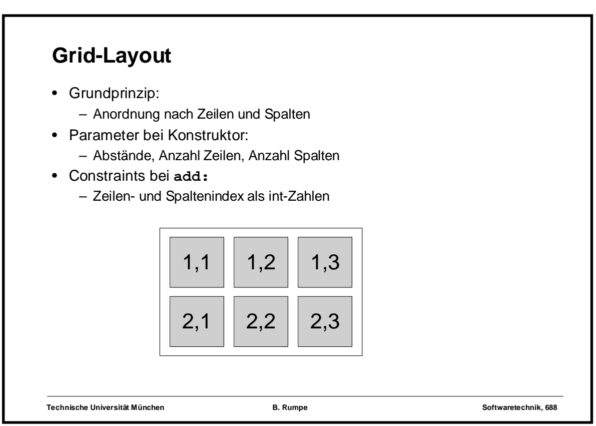

Zähler-Beispiel: Entwurf der Bedienelemente

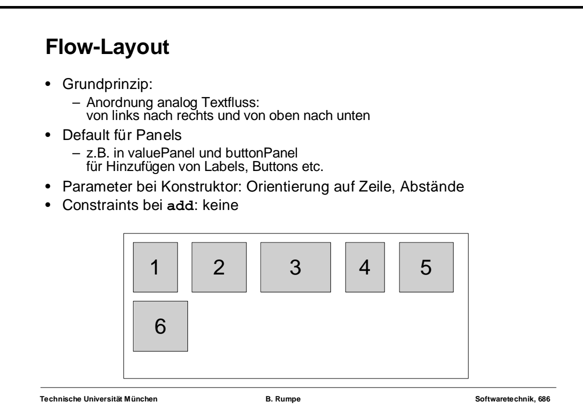


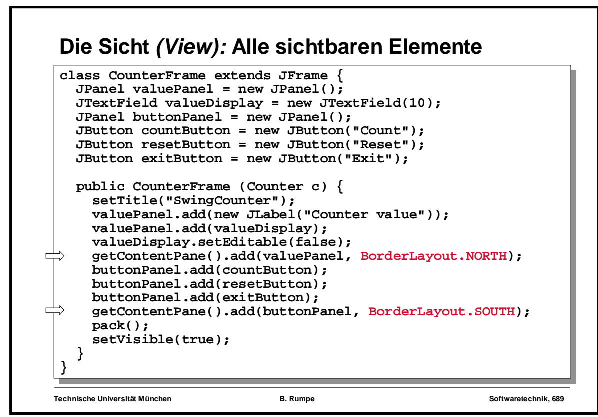
Die Sicht *(View):* Bedienelemente

```
class CounterFrame extends JFrame {
  JPanel valuePanel = new JPanel();
  JTextField valueDisplay = new JTextField(10);
  JPanel buttonPanel = new JPanel();
  JButton countButton = new JButton("Count");
  JButton resetButton = new JButton("Reset");
  JButton exitButton = new JButton("Exit");

  public CounterFrame (Counter c) {
    setTitle("SwingCounter");
    valuePanel.add(new JLabel("Counter value"));
    valuePanel.add(valueDisplay);
    valueDisplay.setEditable(false);
    getContentPane().add(valuePanel);
    buttonPanel.add(countButton);
    buttonPanel.add(resetButton);
    buttonPanel.add(exitButton);
    getContentPane().add(buttonPanel);
    pack();
    setVisible(true);
  }
}
```









Border-Layout

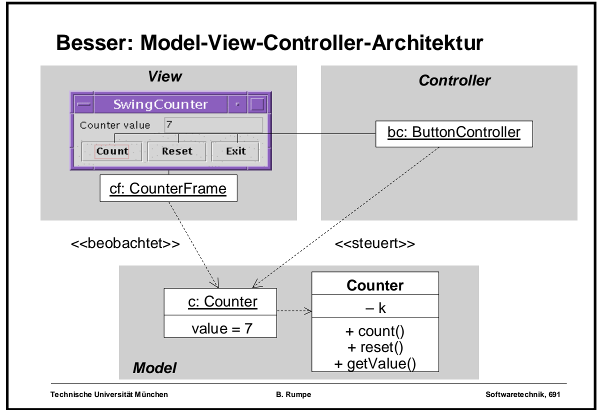
- Grundprinzip:
  - Orientierung nach den Seiten (N, S, W, O) bzw. Mitte (center)
- Default für Window, Frame
  - z.B. in CounterFrame für Hinzufügen von valuePanel, buttonPanel
- Parameter bei Konstruktor: Keine
- Constraints bei **add**:
  - **BorderLayout.NORTH, SOUTH, WEST, EAST, CENTER**

Oberer ("Nord")-Bereich (z.B. valuePanel)
Unterer ("Süd")-Bereich (z.B. buttonPanel)

Zähler-Beispiel: Anbindung über Model/View

```
class CounterFrame extends JFrame
  implements Observer {
  ...
  JTextField valueDisplay = new JTextField(10);
  ...

  public CounterFrame (Counter c) {
    ...
    valuePanel.add(valueDisplay);
    valueDisplay.setEditable(false);
    valueDisplay.setText(String.valueOf(c.getValue()));
    ...
    c.addObserver(this);
    pack();
    setVisible(true);
  }

  public void update (Observable o, Object arg) {
    Counter c = (Counter) o;
    valueDisplay.setText(String.valueOf(c.getValue()));
  }
}
```

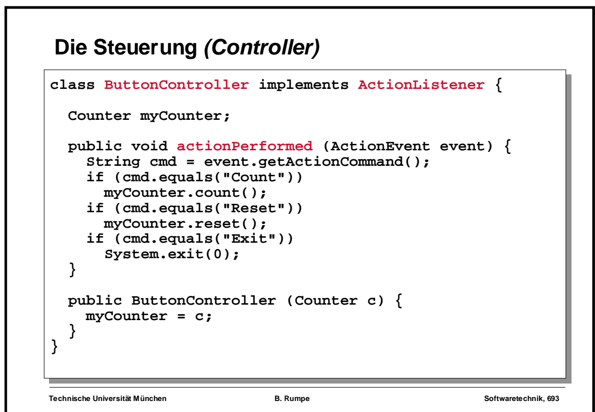
Besser: Model-View-Controller-Architektur



Model-View-Controller-Architektur

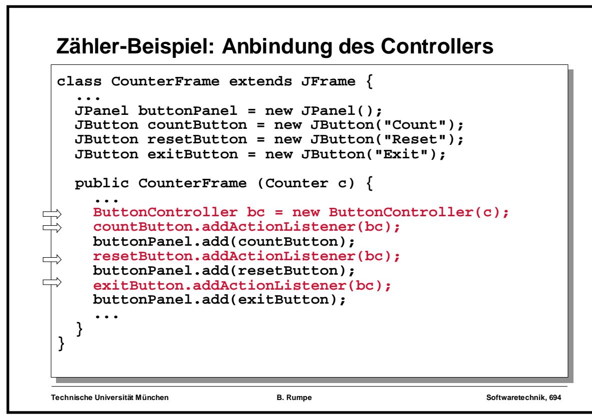
- Model:
  - Fachliches Modell, weitestgehend unabhängig von Oberfläche
  - Beobachtbar (Observable)
- View:
  - Repräsentation auf Benutzungsoberfläche
  - Beobachter des Modells
  - Erfragt beim "update" ggf. notwendige Daten beim Modell
- Controller:
  - Modifiziert Werte im Modell
  - Ist an bestimmte Elemente der "View" (z.B. Buttons) gekoppelt
  - Reagiert auf Ereignisse und setzt sie um in Methodenaufrufe

Die Steuerung *(Controller)*



```
class ButtonController implements ActionListener {

  Counter myCounter;

  public void actionPerformed (ActionEvent event) {
    String cmd = event.getActionCommand();
    if (cmd.equals("Count"))
      myCounter.count();
    if (cmd.equals("Reset"))
      myCounter.reset();
    if (cmd.equals("Exit"))
      System.exit(0);
  }

  public ButtonController (Counter c) {
    myCounter = c;
  }
}
```

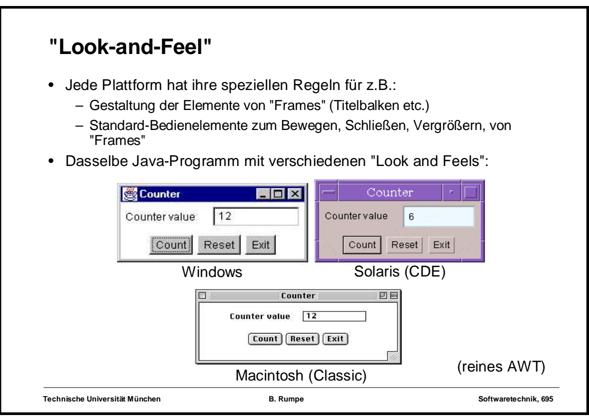

Zähler-Beispiel: Anbindung des Controllers



```
class CounterFrame extends JFrame {
  ...
  JPanel buttonPanel = new JPanel();
  JButton countButton = new JButton("Count");
  JButton resetButton = new JButton("Reset");
  JButton exitButton = new JButton("Exit");

  public CounterFrame (Counter c) {
    ...
    ButtonController bc = new ButtonController(c);
    countButton.addActionListener(bc);
    buttonPanel.add(countButton);
    resetButton.addActionListener(bc);
    buttonPanel.add(resetButton);
    exitButton.addActionListener(bc);
    buttonPanel.add(exitButton);
    ...
  }
}
```

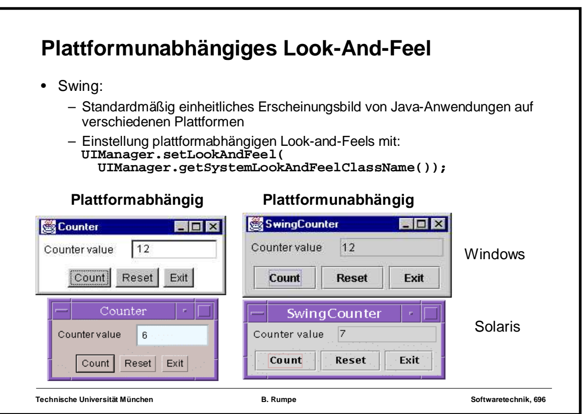

"Look-and-Feel"

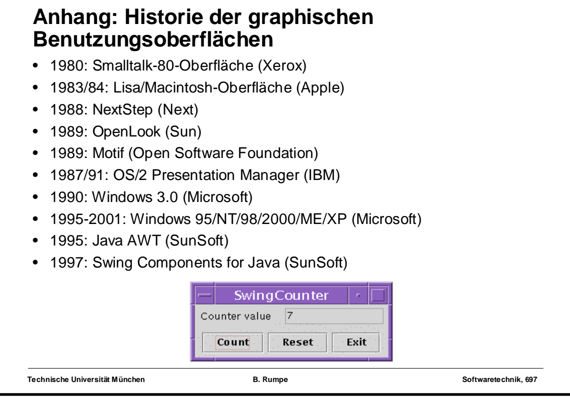
- Jede Plattform hat ihre speziellen Regeln für z.B.:
  - Gestaltung der Elemente von "Frames" (Titelbalken etc.)
  - Standard-Bedienelemente zum Bewegen, Schließen, Vergrößern, von "Frames"
- Dasselbe Java-Programm mit verschiedenen "Look and Feels":



Plattformunabhängiges Look-And-Feel

- Swing:
  - Standardmäßig einheitliches Erscheinungsbild von Java-Anwendungen auf verschiedenen Plattformen
  - Einstellung plattformabhängigen Look-and-Feels mit:
    `UIManager.setLookAndFeel(`
    `UIManager.getSystemLookAndFeelClassName());`





Anhang: Historie der graphischen Benutzungsoberflächen

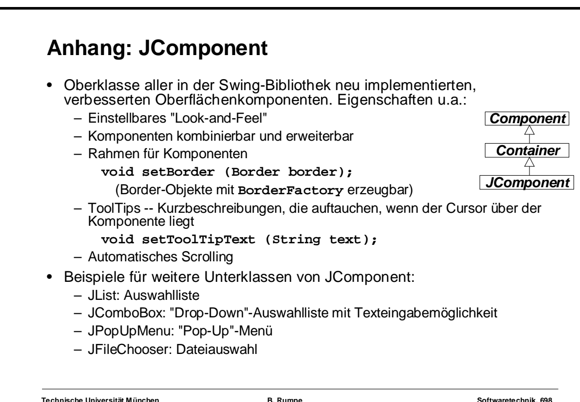
- 1980: Smalltalk-80-Oberfläche (Xerox)
- 1983/84: Lisa/Macintosh-Oberfläche (Apple)
- 1988: NextStep (Next)
- 1989: OpenLook (Sun)
- 1989: Motif (Open Software Foundation)
- 1987/91: OS/2 Presentation Manager (IBM)
- 1990: Windows 3.0 (Microsoft)
- 1995-2001: Windows 95/NT/98/2000/ME/XP (Microsoft)
- 1995: Java AWT (SunSoft)
- 1997: Swing Components for Java (SunSoft)

Softwaretechnik WS 02/03

6. Implementierung

6.6. Architektur Interaktiver Systeme (GUI,Web)

6.6.4 Web-Architekturen



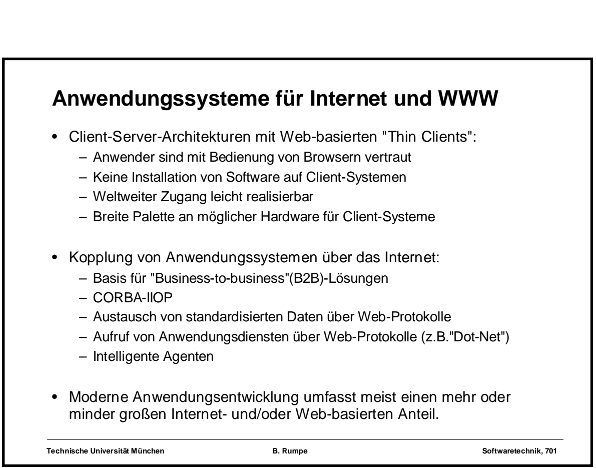
Anhang: JComponent

- Oberklasse aller in der Swing-Bibliothek neu implementierten, verbesserten Oberflächenkomponenten. Eigenschaften u.a.:
  - Einstellbares "Look-and-Feel"
  - Komponenten kombinierbar und erweiterbar
  - Rahmen für Komponenten

    `void setBorder (Border border);`

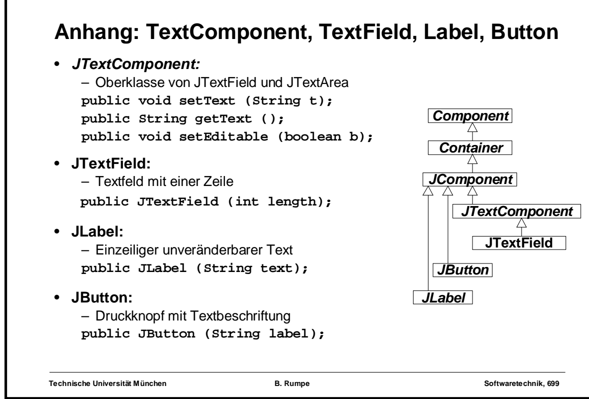
    (Border-Objekte mit `BorderFactory` erzeugbar)
  - ToolTips -- Kurzbeschreibungen, die auftauchen, wenn der Cursor über der Komponente liegt

    `void setToolTipText (String text);`
  - Automatisches Scrolling
- Beispiele für weitere Unterklassen von JComponent:
  - JList: Auswahlliste
  - JComboBox: "Drop-Down"-Auswahlliste mit Texteingabemöglichkeit
  - JPopUpMenu: "Pop-Up"-Menü
  - JFileChooser: Dateiauswahl



Anwendungssysteme für Internet und WWW

- Client-Server-Architekturen mit Web-basierten "Thin Clients":
  - Anwender sind mit Bedienung von Browsern vertraut
  - Keine Installation von Software auf Client-Systemen
  - Weltweiter Zugang leicht realisierbar
  - Breite Palette an möglicher Hardware für Client-Systeme
- Kopplung von Anwendungssystemen über das Internet:
  - Basis für "Business-to-business"(B2B)-Lösungen
  - CORBA-IIOP
  - Austausch von standardisierten Daten über Web-Protokolle
  - Aufruf von Anwendungsdiensten über Web-Protokolle (z.B."Dot-Net")
  - Intelligente Agenten
- Moderne Anwendungsentwicklung umfasst meist einen mehr oder minder großen Internet- und/oder Web-basierten Anteil.



Anhang: TextComponent, TextField, Label, Button

- ***JTextComponent:***
  - Oberklasse von JTextField und JTextArea

```
public void setText (String t);
public String getText ();
public void setEditable (boolean b);
```

- **JTextField:**
  - Textfeld mit einer Zeile

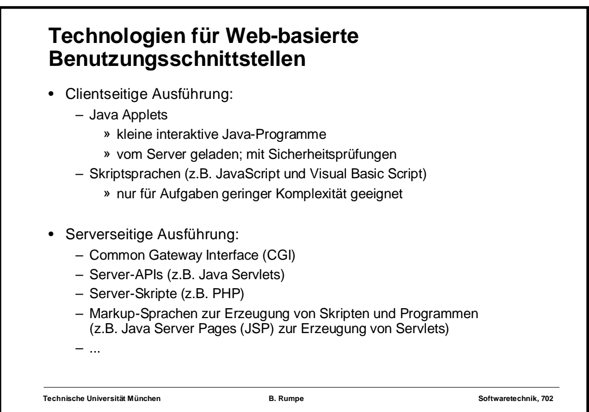
```
public JTextField (int length);
```

- **JLabel:**
  - Einzeiliger unveränderbarer Text

```
public JLabel (String text);
```

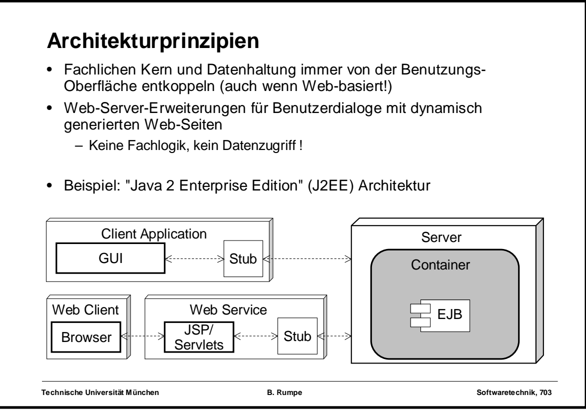
- **JButton:**
  - Druckknopf mit Textbeschriftung

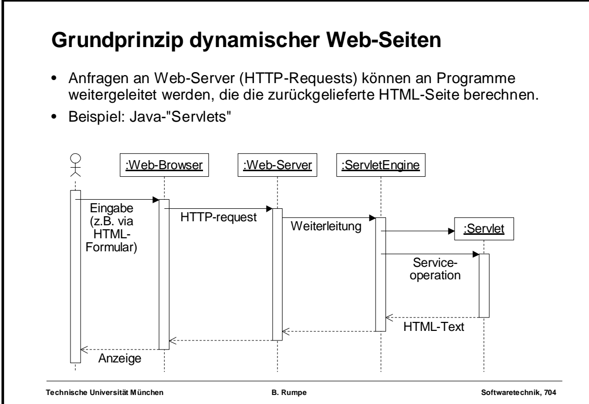
```
public JButton (String label);
```

Technologien für Web-basierte Benutzungsschnittstellen

- Clientseitige Ausführung:
  - Java Applets
    - » kleine interaktive Java-Programme
    - » vom Server geladen; mit Sicherheitsprüfungen
  - Skriptsprachen (z.B. JavaScript und Visual Basic Script)
    - » nur für Aufgaben geringer Komplexität geeignet
- Serverseitige Ausführung:
  - Common Gateway Interface (CGI)
  - Server-APIs (z.B. Java Servlets)
  - Server-Skripte (z.B. PHP)
  - Markup-Sprachen zur Erzeugung von Skripten und Programmen (z.B. Java Server Pages (JSP) zur Erzeugung von Servlets)
  - ...



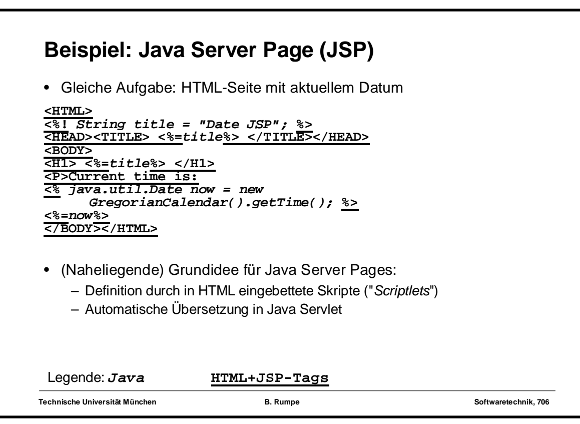


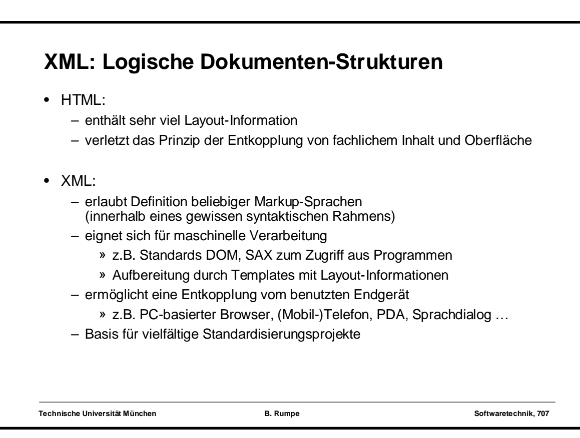
Beispiel: Einfaches Java Servlet

- Aufgabe: HTML-Seite mit aktuellem Datum

```
public class DateServlet extends HttpServlet {
  public void doGet (HttpServletRequest request,
                     HttpServletResponse response)
               throws ServletException, IOException {
    String title = "Date Servlet Page";
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    out.println("<HTML><HEAD><TITLE>");
    out.println(title);
    out.println("</TITLE></HEAD><BODY>");
    out.println("<H1>" + title + "</H1>");
    out.print("<P>Current time is: ");
    out.println(new
      java.util.GregorianCalendar().getTime());
    out.println("</BODY></HTML>");
    out.close();
  }
}
```

Legende: *Java* HTML





XML: Beispiel

- Logische Kerninformation eines Briefs:

```
<Brief>
  <Adresse>
    <Name>Meier</Name>
    <Strasse>Duererstrasse 26</Strasse>
    <PLZ>01207</PLZ>
    <Ort>Dresden</Ort>
  </Adresse>
  <Betreff>Ihr Schreiben vom 10.10. d.J.</Betreff>
  <Anrede>Sehr geehrte Damen und Herren</Anrede>
  <Text>blabla blabla</Text>
</Brief>
```

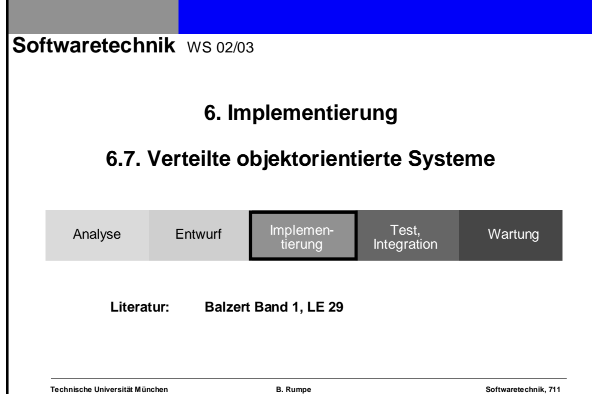
- Basis: Dokumentenstrukturdefinition
  (Document Type Definition (DTD) bzw. (neuer) XML-Schema)

Bedeutung von Web-Technologien

- Aktuelle "Visionen" der Industrie (z.B. Microsoft "Dot-Net"):
  - Anwendungsbausteine (Web-Services) über Internet bereitgestellt
  - Universelle XML-basierte Formate für
    - » Beschreibung von Dienst-Angeboten (Web Service Description Language WSDL)
    - » Verzeichnisse von Anbietern (Universal Data Dictionary Interface UDDI)
    - » Aufruf von Diensten (Simple Object Access Protocol SOAP)
- Mögliche Chancen:
  - Weltweite komponentenbasierte, föderierte Anwendungssysteme
- Probleme:
  - Sicherheitsfragen
  - Grobgranularität aus Performance-Gründen
  - Herstellerunabhängige Standardisierung von Komponenten?
  - Modellierung und Entwurf Web-basierter Systeme

Zusammenfassung: 6.6. Architektur Interaktiver Systeme

- 3-Schichtenarchitektur hat sich durchgesetzt:
  - GUI, Applikationskern, persistente Datenhaltung
- Model-View-Controller (oder vereinfacht Model-View) trennt Datenmodell, Kontrolle und Darstellung
- Framework-Unterstützung in Java durch AWT und Swing gegeben. Dabei kommen Entwurfsmuster, wie Observable oder Komponenten zum Einsatz
- Ereignissteuerung liegt im Framework. Durch „Callbacks“ an die „Listener“ werden Ereignisse im Applikationskern verarbeitet



Offene verteilte Systeme

- Situation: Heterogene, vernetzte Computersysteme:
  - heterogene Hardware, Betriebssysteme, Programmiersprachen, Formate, Netztechnologien
  - Anwendungen im Normalfall verteilt
    - » Client/Server-Architekturen
- Ziele:
  - Portabilität (Objektcode, Quellcode, Entwurf)
  - Interoperabilität
  - Lokations- und Netzwerk-Transparenz
- Hilfsmittel:
  - Standards für “offene Systeme”
  - Eindeutige Definition von Schnittstellen
  - Flexible Erweiterung um neue Schnittstellen

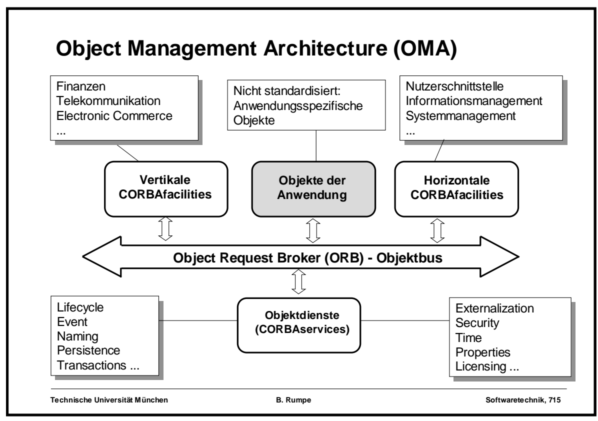
Realisierung von Client/Server-Systemen

- Sockets ("Peer-to-Peer"-Kommunikation über den TCP/IP Stack)
  - Niedriges Abstraktionsniveau
  - Plattform-Transparenz (z.B. Java Package `java.net`)
- Remote Procedure Call (RPC)
  - Höheres Abstraktionsniveau als Sockets (Prozedurebene)
  - Einbettung in prozedurale Programmiersprachen (z.B. C)
  - Realisierung von synchronen entfernten Prozeduraufrufen
- "Middleware" für verteilte Objekte (CORBA, Microsoft DCOM)
  - Hohes Abstraktionsniveau (Objektebene)
  - Transparenz von Ort und Programmiersprache
  - Bei CORBA: auch Plattform-Transparenz
- Java Remote Methode Invocation (RMI)
  - Hohes Abstraktionsniveau (Objektebene)
  - Java-spezifischer Mechanismus (Java Package `java.rmi`)
  - Ab Java-Version 1.3 "RMI-IIOP": Kompatibel zu CORBA (Java Package `javax.rmi`)
  - Ort- und Plattform-Transparenz

Standards der Object Management Group

- Object Management Group (OMG):
  - Zusammenschluss von Anbietern und Anwendern objektorientierter Softwaretechnologie
  - Gegründet 1989, mehr als 600 Mitglieder
  - Standards und Spezifikationen, aber mit Verpflichtung zur kommerziellen Implementierung
  - “Middleware” für verteilte objektorientierte Anwendungen (CORBA)
  - Interoperabilität verschiedenener CORBA-Implementierungen ist Realität
- Arten von Standards
  - “Offizielle” Gremien, z.B. ISO, ITU (relativ schwerfällig)
  - Firmen-“Standards”, z.B. Microsoft OLE/DCOM (proprietär)
  - Industriekonsortien, z.B. OMG (Mittelweg)
- CORBA-Standard der OMG wird ISO-Standard
  - ISO 14750 (IDL), ISO 19500-2 (Interoperability)

Object Management Architecture (OMA)

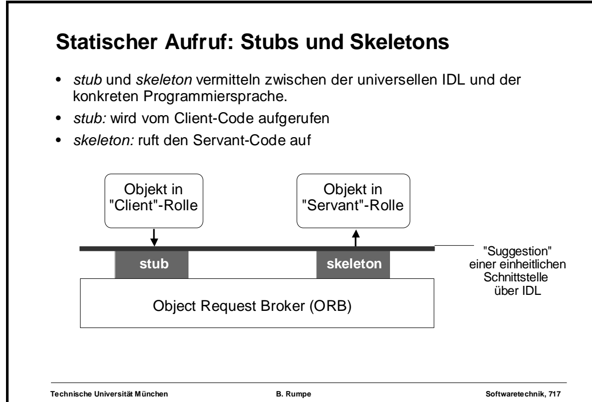


Dienstanforderungen (object requests)

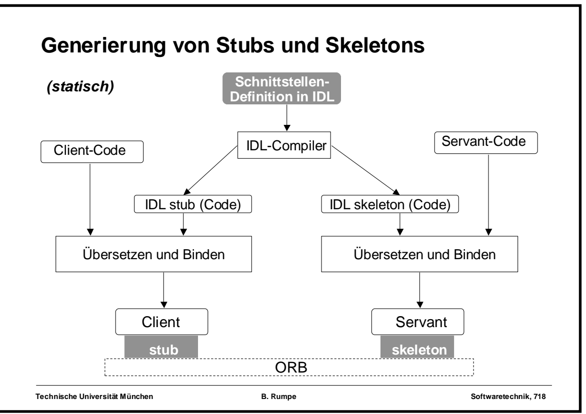
- *Client*-Objekte senden Anforderungen *(requests)* an abstrakte Referenzen, die über dann an Objektimplementierungen *(servants)* zugestellt werden.
- Bestandteile einer Anforderung:
  - Angabe des angesprochenen Server-Objekts (Objekt-Referenz)
  - Name der angeforderten Operation
  - Kommunikationsart
    - » synchron, asynchron oder abgesetzt synchron
  - Ein- und Ausgabeparameter
  - (optional) Ausnahmebedingungen
  - (optional) Kontextangaben (z.B. Ort des Clients)
- Schnittstellen:
  - Beschreibung von Anforderungstypen
  - Zusammenfassung in Schnittstellen
  - Interface Definition Language (IDL)

Statischer Aufruf: Stubs und Skeletons

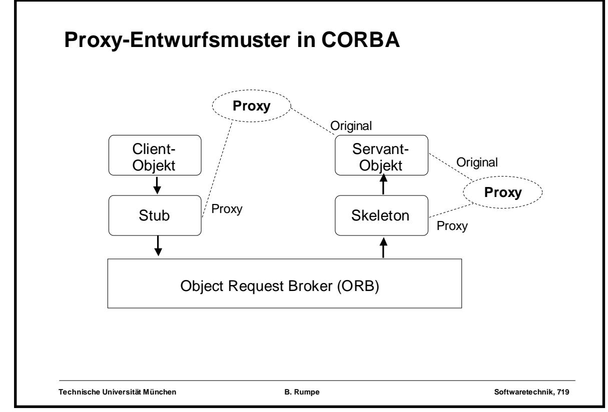
- *stub* und *skeleton* vermitteln zwischen der universellen IDL und der konkreten Programmiersprache.
- *stub:* wird vom Client-Code aufgerufen
- *skeleton:* ruft den Servant-Code auf

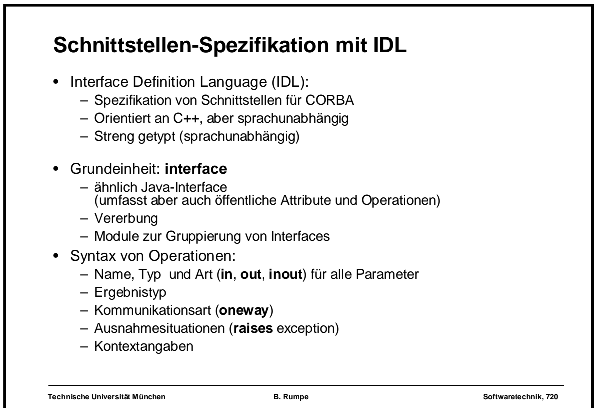


Generierung von Stubs und Skeletons



Proxy-Entwurfsmuster in CORBA





Schnittstellen-Spezifikation mit IDL

- Interface Definition Language (IDL):
  - Spezifikation von Schnittstellen für CORBA
  - Orientiert an C++, aber sprachunabhängig
  - Streng getypt (sprachunabhängig)
- Grundeinheit: **interface**
  - ähnlich Java-Interface (umfasst aber auch öffentliche Attribute und Operationen)
  - Vererbung
  - Module zur Gruppierung von Interfaces
- Syntax von Operationen:
  - Name, Typ und Art (**in**, **out**, **inout**) für alle Parameter
  - Ergebnistyp
  - Kommunikationsart (**oneway**)
  - Ausnahmesituationen (**raises** exception)
  - Kontextangaben

IDL: Datentypen

- Einfache Datentypen
  - Ganzzahltypen
    - » **short, long, long long**
    - » **signed** und **unsigned**
  - Gleitpunkttypen
    - » **float, double, long double, fixed**
  - **char, wchar, boolean, octet**
  - **Any**
  - CORBA Objektreferenz
  - Value-Typ (Objekte als Werte)
- Strukturierte Datentypen
  - Verbunde (struct)
  - Vereinigung
  - Aufzählung
  - Sequenzen
  - Strings
  - Felder

IDL: Beispiel

```
module BeispielModul {
  struct Adresse {
    string strasse;
    string plz;
    string ort;};

  interface Person {
    attribute string name;
    readonly attribute string vorname;
    readonly attribute long geburtsjahr;
    attribute Adresse anschrift;

    typedef sequence <Person> PersonListe;
    attribute PersonListe Bekannte;

    long berechneLebensjahre  (in long aktJahr);
    void berechneLebensjahreP
      (in long aktJahr, out long lebensJahre);
  }; // Interface
}; // Modul
```

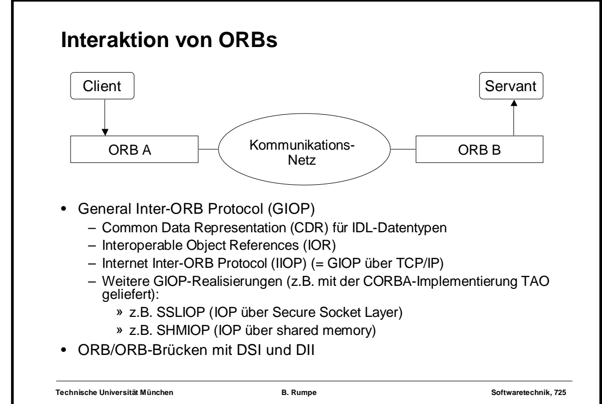

Abbildung IDL-Programmiersprache

- Abbildung von Schnittstellen und Operationen
  - C:
    - » Keine direkte Abbildung von Schnittstellen (Schnittstellenname wird in Funktionsname übernommen, Parameter vom Schnittstellentyp ist erster Parameter der Funktionen einer Schnittstelle)
    - » Operationen werden Funktionen
  - C++:
    - » Module werden auf C++ Namespaces abgebildet
    - » Schnittstellen auf Klassen
    - » Operationen auf Methoden
  - Java:
    - » Module werden auf Java Packages abgebildet
    - » Schnittstellen auf öffentliche Java Schnittstellen sowie Helper- und Holder Klassen in Java
    - » Operationen auf Methoden

Statische und Dynamische Schnittstellen

- Statische Aufrufschnittstelle *(Static Invocation Interface, SII)*
  - Festlegung der Operation (und Typinformation) zur Übersetzungszeit (aber dynamische Bindung an Objektinstanzen)
  - Generierung von *stubs*
- Dynamische Aufrufschnittstelle*(Dynamic Invocation Interface, DII)*:
  - Festlegung der Operation und Typinformation zur Laufzeit
  - Flexibler in der Kommunikationsart (auch "abgesetzt synchron")
- Statische Skeleton-Schnittstelle *(Static Skeleton Interface, SSI)*
  - Festlegung der Operation (und Typinformation) zur Übersetzungszeit
  - Generierung von *skeletons*
- Dynamische Skeleton-Schnittstelle *(Dynamic Skeleton Interface, DSI)*
  - Bereitstellung eines flexiblen Skeletons
  - Analyse von Operation, Parametern, Typinformation zur Laufzeit
- Unabhängige Wahl: SII/DII (Client), SSI/DSI (Servant)

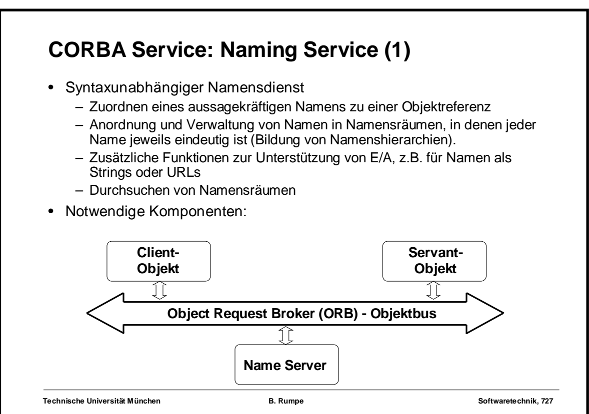
Interaktion von ORBs

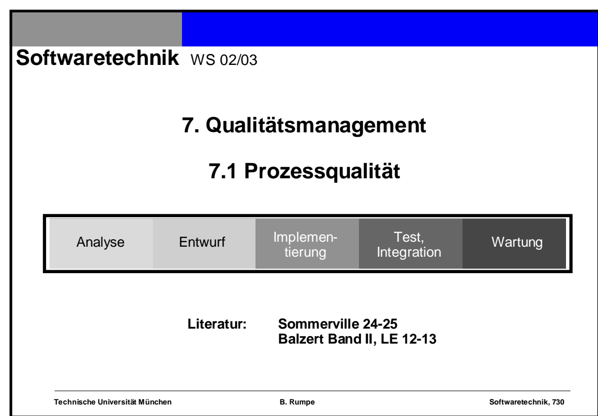


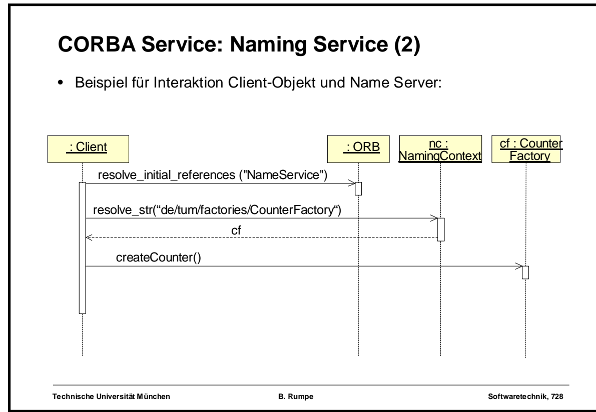
- General Inter-ORB Protocol (GIOP)
  - Common Data Representation (CDR) für IDL-Datentypen
  - Interoperable Object References (IOR)
  - Internet Inter-ORB Protocol (IIOP) (= GIOP über TCP/IP)
  - Weitere GIOP-Realisierungen (z.B. mit der CORBA-Implementierung TAO geliefert):
    - » z.B. SSLIOP (IOP über Secure Socket Layer)
    - » z.B. SHMIOP (IOP über shared memory)
- ORB/ORB-Brücken mit DSI und DII

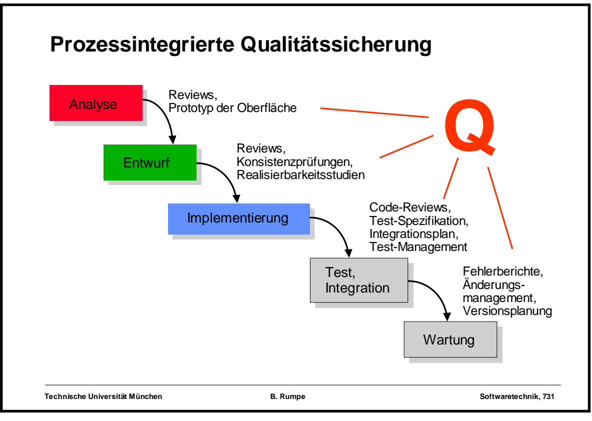
Spezielle Objektdienste (CORBAservices)

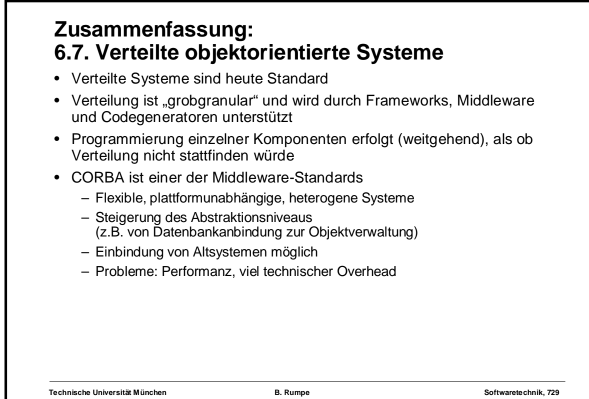
- *CORBAservices* vervollständigen die Funktion des ORB
- Beispiele:
  - Namensdienst *(naming service)*
  - Such- und Vermittlungsdienst *(object trading service)*
  - Objektverwaltung *(life cycle service)*
  - Persistenzdienst *(persistent object service)*
  - Transaktionsdienst *(transaction service)*
  - Abfragedienst *(query service)*
  - Lizenzdienst *(licensing service)*
  - Eigenschaftsverwaltung *(property service)*
  - Zeitdienst *(time service)*
  - Sicherheitsdienst *(security service)*

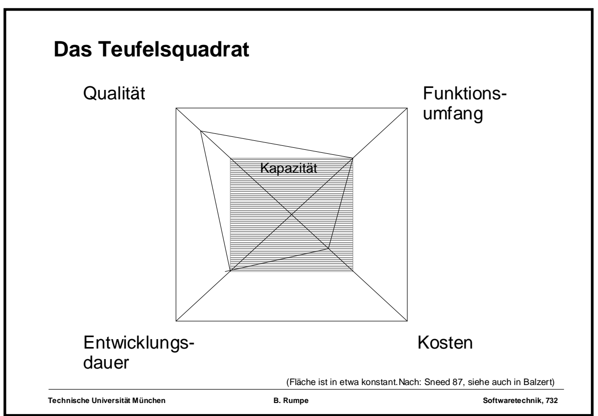

CORBA Service: Naming Service (1)
• Syntaxunabhängiger Namensdienst
– Zuordnen eines aussagekräftigen Namens zu einer Objektreferenz
– Anordnung und Verwaltung von Namen in Namensräumen, in denen jeder Name jeweils eindeutig ist (Bildung von Namenshierarchien).
– Zusätzliche Funktionen zur Unterstützung von E/A, z.B. für Namen als Strings oder URLs
– Durchsuchen von Namensräumen
• Notwendige Komponenten:
Client-Objekt
Servant-Objekt
Object Request Broker (ORB) - Objektbus
Name Server
Technische Universität München
B. Rumpe
Softwaretechnik, 727


Softwaretechnik WS 02/03
7. Qualitätsmanagement
7.1 Prozessqualität
Analyse
Entwurf
Implementierung
Test, Integration
Wartung
Literatur:
Sommerville 24-25
Balzert Band II, LE 12-13
Technische Universität München
B. Rumpe
Softwaretechnik, 730


CORBA Service: Naming Service (2)
• Beispiel für Interaktion Client-Objekt und Name Server:
: Client
: ORB
nc : NamingContext
cf : Counter Factory
resolve_initial_references ("NameService")
resolve_str("de/tum/factories/CounterFactory")
cf
createCounter()
Technische Universität München
B. Rumpe
Softwaretechnik, 728


Prozessintegrierte Qualitätssicherung
Analyse
Reviews, Prototyp der Oberfläche
Entwurf
Reviews, Konsistenzprüfungen, Realisierbarkeitsstudien
Implementierung
Code-Reviews, Test-Spezifikation, Integrationsplan, Test-Management
Test, Integration
Fehlerberichte, Änderungs-management, Versionsplanung
Wartung
Q
Technische Universität München
B. Rumpe
Softwaretechnik, 731


Zusammenfassung:
6.7. Verteilte objektorientierte Systeme
• Verteilte Systeme sind heute Standard
• Verteilung ist „grobgranular" und wird durch Frameworks, Middleware und Codegeneratoren unterstützt
• Programmierung einzelner Komponenten erfolgt (weitgehend), als ob Verteilung nicht stattfinden würde
• CORBA ist einer der Middleware-Standards
– Flexible, plattformunabhängige, heterogene Systeme
– Steigerung des Abstraktionsniveaus (z.B. von Datenbankanbindung zur Objektverwaltung)
– Einbindung von Altsystemen möglich
– Probleme: Performanz, viel technischer Overhead
Technische Universität München
B. Rumpe
Softwaretechnik, 729


Das Teufelsquadrat
Qualität
Funktions-umfang
Kapazität
Entwicklungs-dauer
Kosten
(Fläche ist in etwa konstant. Nach: Sneed 87, siehe auch in Balzert)
Technische Universität München
B. Rumpe
Softwaretechnik, 732

Qualitätssicherung für Analyse und Entwurf

- Hohe Bedeutung früher Phasen für Produktqualität !
  - Deshalb: Alle Dokumente frühzeitig überprüfen ("Validation").
- Techniken:
  - Anforderungskatalog-Begutachtung
    - » Echte Benutzer einbeziehen
    - » Anforderungskatalog auf Vollständigkeit und Korrektheit prüfen
  - Use-Case-Szenarien
    - » Echte Benutzer einbeziehen
    - » "Funktionsfähigkeit" der abstrakten Modelle demonstrieren
  - Prototyping
    - » Prototyp auf der Basis der Analyse/Entwurfs-Dokumente
    - » Echte Benutzer einbeziehen
    - » Vorgezogener Akzeptanztest
  - Abgleich des Entwurfs mit Use-Cases/Anforderungskatalog
    - » Erste verifizierende Tätigkeiten

Begutachtung (Review)

- Produkt wird von Expertengremium begutachtet
  - Anwendbar auf fast alle Phasen der Entwicklung
- Was wird begutachtet?
  - Genau definiertes Dokument (Art, Status, Prozesseinbindung)
  - Teil der Gesamtplanung des Projekts (Termin)
- Wer begutachtet?
  - Teammitglieder (Peer-Review)
  - Externe Spezialisten
  - Echte Benutzer
  - Moderator, Manager
- Wie läuft die Begutachtung ab?
  - Einladung
  - Vorbereitung, Sammlung von Kommentaren
  - Begutachtungssitzung(en), Protokolle
  - Auswertung und Konsequenzen (Wiederholung, Statusänderung)

Regeln für wirkungsvolle Begutachtung

- "Checklisten" für die Gutachter vorbereiten
  - z.B. "Enthält das Dokument alle laut Firmenstandard vorgesehenen Inormationen?" - "Gibt es für jede Klasse eine informelle Beschreibung?" - "Sind Kardinalitäten im Klassendiagramm vollständig und korrekt?" - "Sind alle Klassen kohärent?" - ...
- Das Dokument wird begutachtet, nicht die Autoren !
- Richtige Vorkenntnisse der Gutachter sind wesentlich.
- Die Rolle des Moderator ist anspruchsvoll:
  - Vermittlung in persönlichen Konflikten und Gruppenkonflikten
  - Fachlicher Gesamtüberblick
- Das Ziel ist Einigkeit, nicht ein Abstimmungsergebnis !
- Ergebnisse von Begutachtungen müssen Auswirkungen haben.
- Wesentliche Beiträge von Gutachtern müssen Anerkennung finden.
- Begutachtung ist auch anwendbar auf Programm-Code (Code-Inspektion).

Produktqualität und Prozessqualität

- Software:
  - Kaum Qualitätsmängel durch Massenproduktion
  - Qualitätsmängel im Herstellungsprozess begründet
- Qualitätsmanagement:
  - Organisatorische Maßnahmen zur Prüfung und Verbesserung der Prozessqualität
  - Beachtung von internen Kunden-/Lieferantenbeziehungen
- **Qualität des Entwicklungsprozesses!**
- Ansätze:
  - ISO 9000
  - Total Quality Management (TQM)
    - » Kaizen: Ständige Verbesserung
  - Capability Maturity Model (CMM)
  - Software Process Improvement and Capability Determination (SPICE)
  - *Business (Process) Engineering*

ISO 9000

- Internationales Normenwerk
  - ISO 9000: Allgemeiner Leitfaden
    - » ISO 9000-3: Leitfaden zur Anwendung von ISO 9001 auf Software
  - ISO 9001: Modelle zur Darlegung der Qualitätssicherung insbesondere in Entwurf und Entwicklung
- Allgemeingültige Forderungen an die Organisation eines Unternehmens:
  - Regelung von Zuständigkeiten
  - Erstellung und korrekte Verwaltung wesentlicher Dokumente
  - Existenz eines unabhängigen Qualitätssicherungs-Systems
- Zertifizierung:
  - durch unabhängige (zertifizierte) Prüforganisationen
  - Audit, Prüfsiegel
- ISO 9000 prüft nicht die Qualität des Produkts, sondern die Qualität der Organisation !

Capability Maturity Model (CMM)

- Einstufungsverfahren für Reifegrad der Organisation
  - entwickelt von der Carnegie-Mellon University

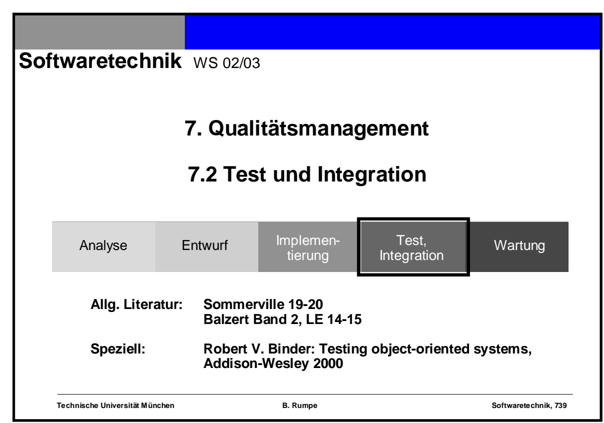
Stufe 5: **Optimierend** – Kontrolle über Prozessqualität

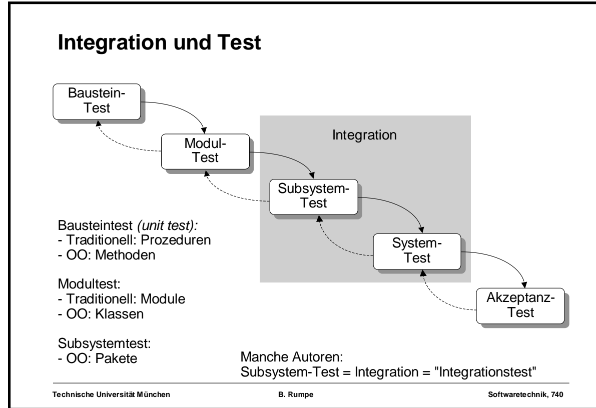
Stufe 4: **Gesteuert** – Kontrolle über Produktqualität

Stufe 3: **Definiert** – Kosten und Termine zuverlässig, aber Qualität schwankend

Stufe 2: **Wiederholbar** – Terminkontrolle, aber Kosten und Qualität schwanken

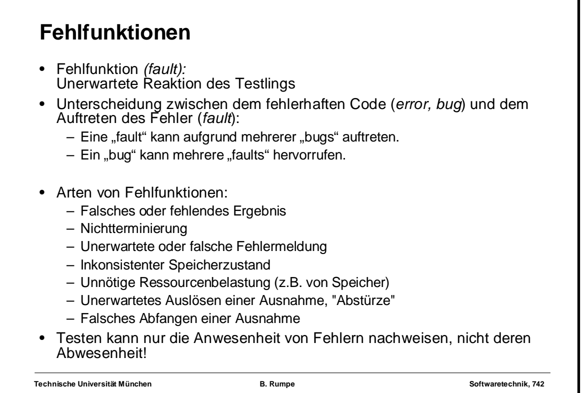
Stufe1: **Initial** – Kosten, Zeit, Qualität unvorhersehbar

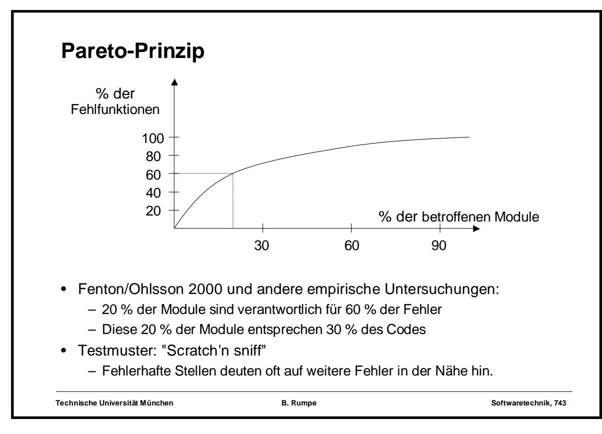




Systematisches Testen: Begriffe

- *Testgegenstand (Testling, Prüfling):*
  Programm bzw. Komponente, die zu testen ist
- *Testfall:*
  Satz von Eingabedaten, der die (vollständige) Ausführung des Testgegenstands bewirkt
- *Testdaten:*
  Daten, die für den Testfall benötigt werden
- *Testtreiber:*
  Rahmenprogramm für Ausführung von Tests
- *Attrappe (Stub, Dummy):*
  Ersatz für ein (noch) nicht vorhandenes Unterprogramm
- *Regressionstest:*
  Nachweis, dass eine geänderte Version des Testgegenstands früher durchgeführte Tests erneut besteht
- *Alphatest:* Test experimenteller Vorversion durch Benutzer
- *Betatest:* Test funktional vollständiger Software durch Benutzer





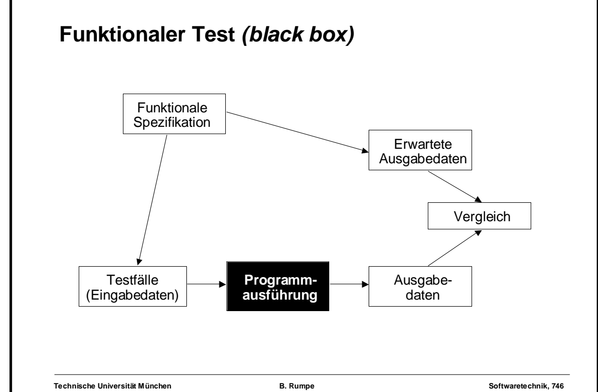
Testplanung

- Testen ist aufwändig, deshalb ist gute Planung nötig!
- Testplanung sollte bereits mit der Anforderungsanalyse beginnen und im Entwurf verfeinert werden (V-Modell, Test-First-Ansatz)!
- Typische Bestandteile einer *Test-Spezifikation:*
  - Phasenmodell des Testprozesses
  - Zusammenhang zur Anforderungsspezifikation
    - » z.B. dort festgelegte Qualitätsziele
  - Zu testende Produkte
  - Zeitplan für die Tests
  - Abhängigkeiten der Testphasen
  - Aufzeichnung der Testergebnisse
  - Hardware- und Softwareanforderungen
  - Einschränkungen und Probleme

Klassifikation von Testverfahren

- Funktionaler Test *(black box test)*
  (Testfallauswahl beruht auf Spezifikation)
  - Äquivalenzklassentest, Grenzwerte, Test spezieller Werte
  - Zustandstest auf Spezifikationsbasis
- Strukturtest *(white box test, glass box test)*
  (Testfallauswahl beruht auf Programmstruktur)
  - Kontrollflussorientierter Test
    - » Anweisungsüberdeckung, Zweigüberdeckung, Pfadüberdeckung
  - Datenflussorientierter Test
    - » *Defs/Uses* -Verfahren
  - Zustandstest auf Codebasis
- Weitere:
  - Zufallstest ("*monkey test*")
  - Stresstest
  - Paralleltest

Funktionaler Test *(black box)*



Äquivalenzklassen

- Äquivalenzklasse:
  - Teilmenge der möglichen Datenwerte der Eingabeparameter
  - Annahme: Programm reagiert für alle Werte aus der Äquivalenzklasse prinzipiell gleich
  - Test je eines Repräsentanten jeder Äquivalenzklasse
- Finden von Äquivalenzklassen:
  - Zulässige und unzulässige Teilbereiche der Datenwerte
  - Unterteilung der zulässigen Bereiche nach verschiedenen Ausgabewerten
  - Ggf. Verwendung von Äquivalenzklassen der Ausgabewerte

Äquivalenzklassen: Beispiel

```
int search (int[] a, int k)
```

pre: a.length > 0

post: (result >= 0 and a[result] == k) or
(result == –1 and (not exists i . i >= 0 and i < a.length and a[i] == k))

- Äquivalenzklassen:
  - Klasse 1A: a.length == 0
  - Klasse 1B: a.length > 0
  - Klasse 2A: k in a
  - Klasse 2B: k nicht in a
- Testfälle:
  - a == [], k == 17, result == undef. (Klassen 1A, 2B)
  - a == [11, 17, 45], k == 17, result == 1 (Klassen 1B, 2A)
  - a == [11, 23, 45], k == 17, result == -1 (Klassen 1B, 2B)

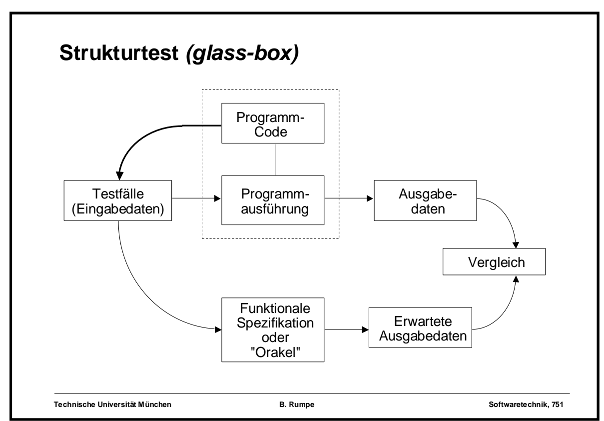
Grenzwertanalyse und Test spezieller Werte

- Grenzwerte: Randfälle der Spezifikation
  - Werte, bei denen "gerade noch" ein gleichartiges Ergebnis zum Nachbarwert erzielt wird, bzw. "gerade schon" ein andersartiges
  - Beispiele:
    - » Ränder von Zahl-Intervallen
    - » Schwellenwerte
- Spezielle Werte:
  - Zahlenwert 0
  - Leere Felder, Sequenzen und Zeichenreihen
  - Einelementige Felder, Sequenzen und Zeichenreihen
  - Null-Referenzen
  - Sonderzeichen (Steuerzeichen, Tabulator)

Grenzwertanalyse: Beispiel

- Neue Klassen:
  - Klasse 3A: Element am linken Rand
  - Klasse 3B: Element am rechten Rand
  - Klasse 3C: Element an keinem Rand
  - Klasse 4A: a einelementig
  - Klasse 4B: a nicht einelementig
  - Klasse 5A: k ist 0
  - Klasse 5B: k ist nicht 0
- Bisherige Testfälle:
  - a == [], k == 17, result == undef. (Klassen 1A, 2B)
  - a == [11, 23, 45], k == 17, result == 1 (Klassen 1B, 2A, 3C, 4B, 5B)
  - a == [11, 17, 45], k == 17, result == -1 (Klassen 1B, 2B, 3C, 4B, 5B)
- Neue Testfälle:
  - a == [17], k == 17, result == 0 (Klassen 1B, 2B, 3A+B, 4A, 5B)
  - a == [11, 23, 0], k == 0, result == 2 (Klassen 1B, 2B, 3B, 4B, 5A)
  - a == [17, 23, 34], k == 17, result == 0 (Klassen 1B, 2B, 3A, 4B, 5B)
  - ...

Strukturtest *(glass-box)*



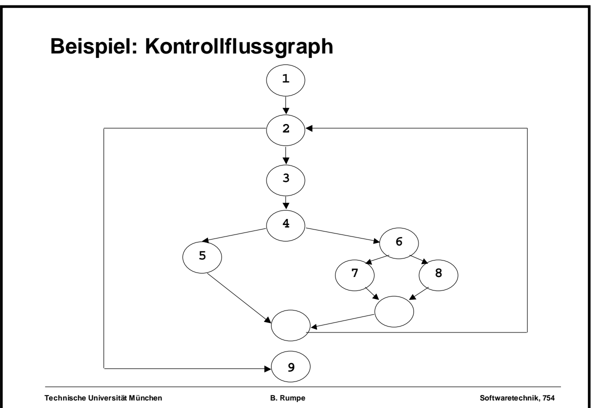
Überdeckungsgrad *(coverage)*

- Maß für die Vollständigkeit eines Tests
- Welcher Anteil des Programmtexts wurde ausgetestet?
- **Messung** der Überdeckung:
  - *Instrumentierung* des Programmcodes
  - Ausgabe statistischer Informationen
- **Planung** der Überdeckung:
  - gezielte Anlage der Tests auf volle Überdeckung
- Überdeckungsarten:
  - *Anweisungsüberdeckung:* Anteil ausgeführter Anweisungen
  - *Zweigüberdeckung:* Anteil ausgeführter Anweisungen und Verzweigungen
  - *Pfadüberdeckung:* Anteil ausgeführter Programmablaufpfade
- Hilfsmittel:
  - *Kontrollflussgraph*

Beispielprogramm: Binäre Suche

```
public static final int NOTFOUND = -1;

// Binaere Suche auf Array a.
// Annahme: a ist sortiert
// Ergebnis ist NOTFOUND, wenn k nicht in A enthalten ist.
// Ergebnis ist i falls a[i] gleich k ist.
public static int binSearch (int[] a, int k) {
    int ug = 0, og = a.length-1, m, pos = NOTFOUND;//(1)
    while (ug <= og && pos == NOTFOUND) {          //(2)
        m = (ug + og) / 2;                         //(3)
        if (a[m] == k)                             //(4)
            pos = m;                               //(5)
        else
            if (a[m] < k)                          //(6)
                ug = m + 1;                        //(7)
            else
                og = m - 1;                        //(8)
    };
    return pos;                                    //(9)
}
```


Beispiel: Kontrollflussgraph



Anweisungsüberdeckender Test

- Überdeckung aller Anweisungen: C_0*-Test*
  - Jede Anweisung (numerierte Zeile) des Programms wird mindestens einmal ausgeführt.
- Beispiel:

  a = {11, 22, 33, 44, 55}, k = 33

  überdeckt Anweisungen: 1, 2, 3, 4, 5, 9

  a = {11, 22, 33, 44, 55}, k = 15

  überdeckt Anweisungen: 1, 2, 3, 4, 6, 7, 8, 9

  Beide Testfälle zusammen erreichen volle Anweisungsüberdeckung.
- Messen der Anweisungsüberdeckung:
  - "Instrumentieren" des Codes (durch Werkzeuge)
  - Einfügen von Zählanweisungen bei jeder Anweisung

Zweigüberdeckender Test

- Überdeckung aller Zweige: C_1*-Test*
  - Bei jeder Fallunterscheidung (einschließlich Schleifen) werden beide Zweige ausgeführt (Bedingung=true und Bedingung=false).
  - Zweigtest zwingt auch zur Untersuchung "leerer" Alternativen:

```
if (x >= 1)
      y = true; // kein else-Fall
```

- Beispiel:
  - Die beiden Testfälle der letzten Folie überdecken alle Zweige.
- Messung/Instrumentierung:
  - Fallunterscheidung:

```
if (...) { bT[i]++; ... } else { bF[i]++; ... }
```

  - While-Schleife:

```
while (...) { bT[i]++; ... } bF[i]++;
```

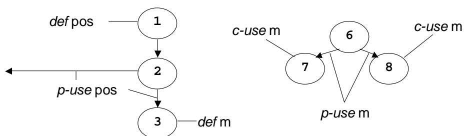
Pfadüberdeckung

- *Pfad* =
  Sequenz von Knoten im Kontrollflussgraphen,
  so dass in der Sequenz aufeinanderfolgende Knoten
  im Graphen direkt miteinander verbunden sind.

  Anfang = Startknoten, Ende = Endknoten.
- Theoretisch optimales Testverfahren
- Explosion der Zahl von möglichen Pfaden, deshalb
  nicht praxisrelevant. (Schleifen haben unendliche Pfad-Anzahlen)
- Praktikablere Varianten, z.B. *bounded-interior*-Pfadtest:
  Alle Schleifenrümpfe höchstens n-mal (z.B. einmal) wiederholen

Datenflussorientierte Testkriterien

- Information über Variablenbenutzung:
  - *def:* Festlegung des Variablenwerts
  - *c-use:* Verwendung des Variablenwerts zur Berechnung
  - *p-use:* Verwendung des Variablenwerts in Steuerbedingung



- Beispiel(!) für ein Kriterium: *all p-uses/all c-uses*
  - Für jeden Knoten mit '*def* x' wird ein x-definitionsfreier Pfad zu jedem '*p-use* x' und zu jedem '*c-use* x' durchlaufen.
  - Etwas leistungsfähiger als Zweigabdeckung.

Testen objektorientierter Programme

- Klassische Verfahren anwendbar für einzelne Methoden
  - aber: Methoden vergleichsweise kurz und einfach
- Komplexität liegt zum großen Teil in der ***Kooperation.***
- Probleme mit Vererbung: Tests der Oberklassen müssen u.U. für alle Unterklassen wiederholt werden:
  - Beeinflussung von Variablen der Oberklasse
  - Redefinition von Methoden der Oberklasse
- Spezielle Techniken für objektorientiertes Testen:
  - Zustandstests
  - Sequenztests

Zustandstests

- Spezifikationsbezogen ("black box"):
  - Verwendung von Zustandsdiagrammen (z.B. UML) aus Analyse und Entwurf
  - Abdeckungskriterien:
    - » alle Zustände
    - » alle Übergänge
    - » für jeden Übergang alle Folgeübergänge der Länge n
  - Praxistauglich
- Programmbezogen ("glass box"):
  - Automatische Berechnung eines Zustandsdiagramms
  - Zustand = Abstraktion einer Klasse zulässiger Attributwerte
  - Bestimmung der Übergänge erfordert symbolische Auswertung von Methoden
    - » problematisch bei Schleifen und Rekursion
  - Im Forschungsstadium

Wohin mit dem Test-Code?

- Zur Durchführung von Tests entsteht zusätzlicher Code:
  - Testtreiber
  - Testattrappen (Stubs)
  - Testsuiten
- Testcode muss aufbewahrt, Tests nach allen größeren Änderungen wiederholt werden
- Alternativen:
  - Testcode als Bestandteil der Klassen
    - » einfach zu verwalten, vergrößert Code
  - "Spiegelbildliche" Hierarchie von Testklassen
    - » Redundanzproblem
    - » Erleichtert Verwendung von Test-Frameworks (z.B. *JUnit*)
  - Testskripten in eigenen Sprachen
    - » klassischer Ansatz, keine Vererbung von Testfällen möglich
  - Test-Unterklassen
    - » problematisch wegen Mehrfachvererbung

Inkrementelles Testen

- Programmierer testen meist nicht gerne.
  - Testen ist zerstörerische Tätigkeit.
  - Zu spätes Testen führt zu komplizierten Fehlersuchen
- Inkrementelles Testen / Test-First-Ansatz:
  Tests entstehen parallel zum Code (oder sogar ***vor*** dem Code)
- Programmierer schreiben Tests für praktischen Nutzen:
  - Klare Spezifikation von Schnittstellen
  - Beschreibung kritischer Ausführungsbedingungen
  - Dokumentation von Fehlerbeseitigung
  - Festhalten von notwendigem Verhalten vor größerem Umbau ("Refaktorisierung")
- Tests archiviert und automatisch ausführbar
- Weitere Information:
  - K. Beck, E. Gamma: Programmers love writing tests (*JUnit*-Web site)
  - K. Beck: Extreme programming explained, Addison-Wesley 2000

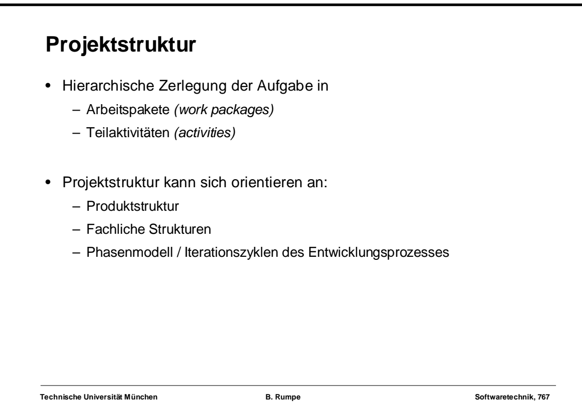
Zusammenfassung:
7. Qualitätsmanagement

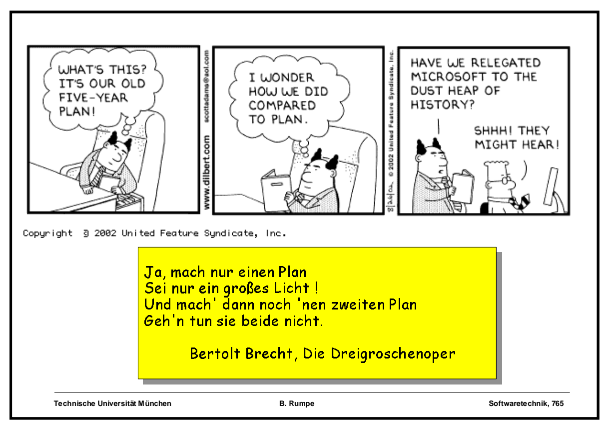
- Qualitätsmanagement beginnt bereits mit den frühen Aktivitäten und begleitet das gesamte Projekt
- Qualitätsmanagement beinhaltet primär
  - Reviews, Inspektionen
  - Tests
- aber auch:
  - Bildung von Prototypen,
  - Einbindung von Kunden, intensive Kommunikation, gutes Arbeitsumfeld, etc.
- Testen ist eine sehr komplexe Tätigkeit, die sehr viel Erfahrung fordert
  - Tests sollten gut geplant und wenn möglich bereits frühzeitig begonnen werden
- Reviews erfordern gute Soft-Skills von allen Beteiligten und sind nur konstruktiv

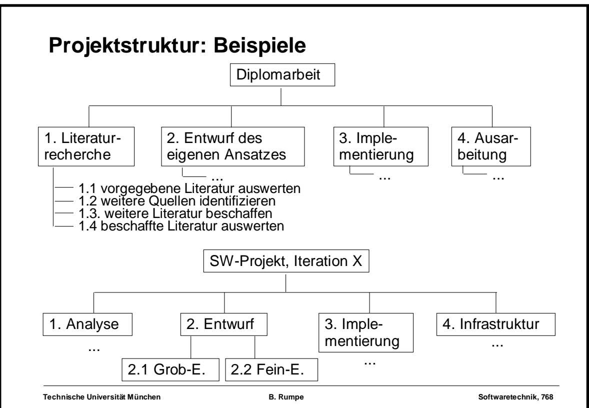
Aufgaben eines Projektmanagers

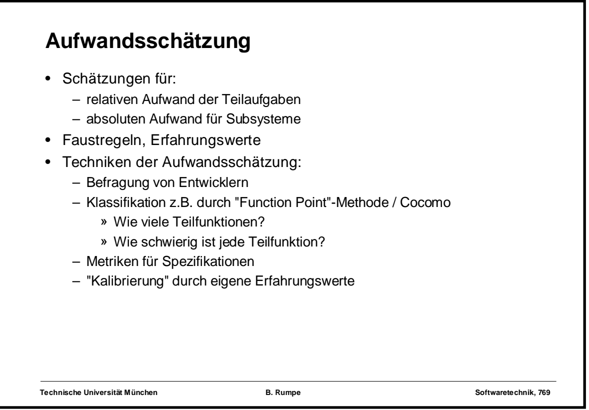
- Schätzen, kalkulieren, planen
- Verträge aushandeln
- Teams zusammenstellen und betreuen
- Beobachten, berichten, abstimmen
- Infrastruktur bereitstellen (Werkzeuge, Räume, Möbel)
- Arbeit organisieren und koordinieren
- Änderungen von Zeitplan, Anforderungen oder Budget systematisch behandeln
- Der Projektmanager agiert als Schnittstelle zur Umwelt

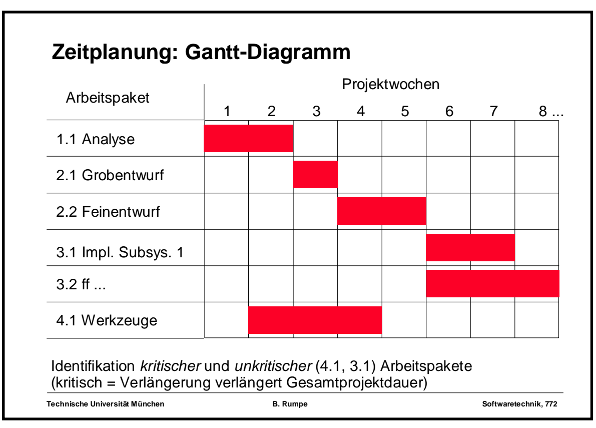
Der (die) Projektmanager(in) ist nicht ein autokratischer "Chef" des Projekts, sondern erbringt eine wesentliche *Dienstleistung* für das Funktionieren des Projekts !

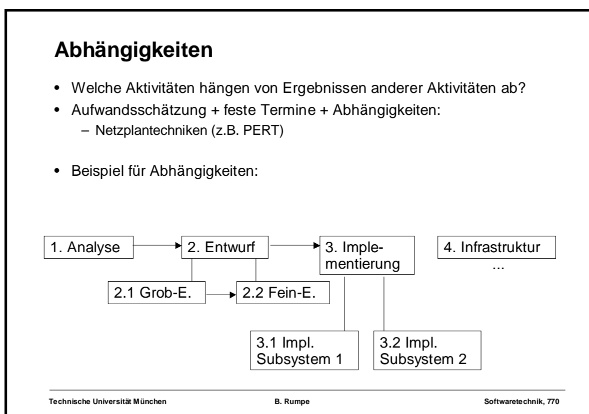


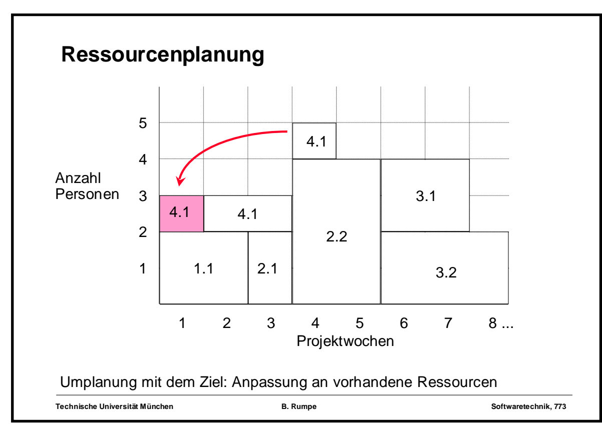


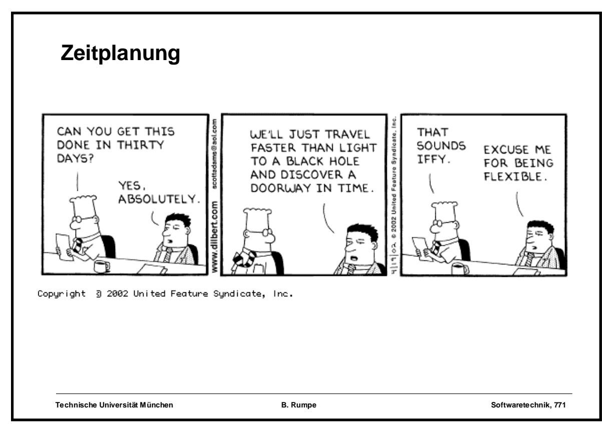


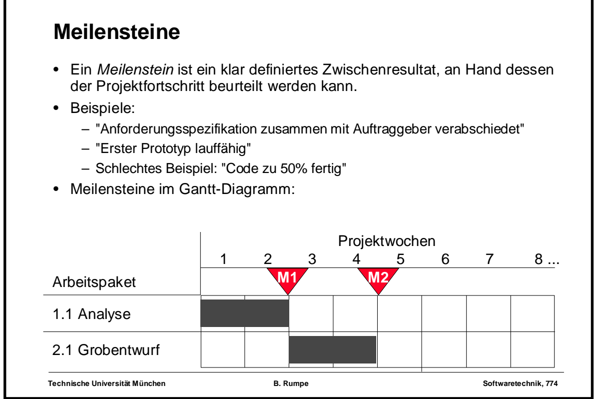

Aufwandsschätzung
Schätzungen für:
relativen Aufwand der Teilaufgaben
absoluten Aufwand für Subsysteme
Faustregeln, Erfahrungswerte
Techniken der Aufwandsschätzung:
Befragung von Entwicklern
Klassifikation z.B. durch "Function Point"-Methode / Cocomo
Wie viele Teilfunktionen?
Wie schwierig ist jede Teilfunktion?
Metriken für Spezifikationen
"Kalibrierung" durch eigene Erfahrungswerte
Technische Universität München
B. Rumpe
Softwaretechnik, 769

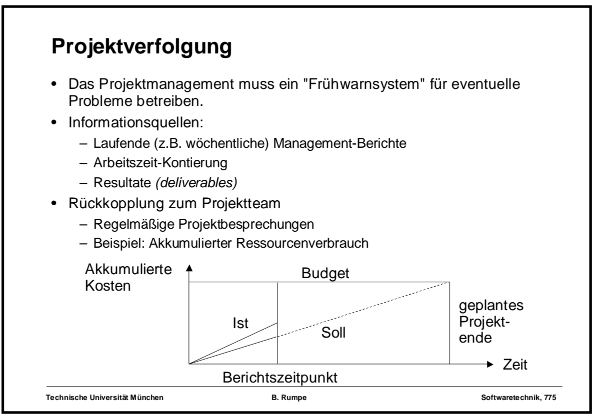

Zeitplanung: Gantt-Diagramm
Arbeitspaket
Projektwochen
1 2 3 4 5 6 7 8 ...
1.1 Analyse
2.1 Grobentwurf
2.2 Feinentwurf
3.1 Impl. Subsys. 1
3.2 ff ...
4.1 Werkzeuge
Identifikation kritischer und unkritischer (4.1, 3.1) Arbeitspakete
(kritisch = Verlängerung verlängert Gesamtprojektdauer)
Technische Universität München
B. Rumpe
Softwaretechnik, 772

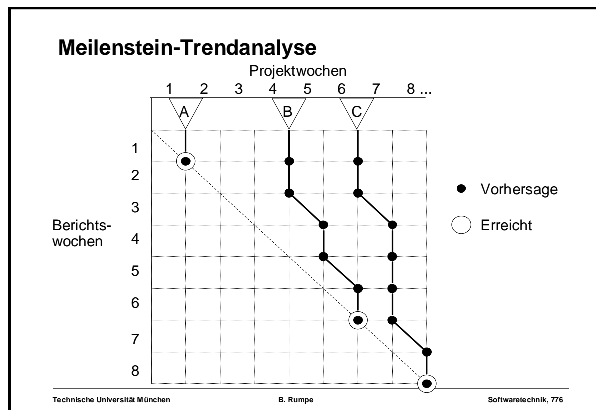

Abhängigkeiten
Welche Aktivitäten hängen von Ergebnissen anderer Aktivitäten ab?
Aufwandsschätzung + feste Termine + Abhängigkeiten:
Netzplantechniken (z.B. PERT)
Beispiel für Abhängigkeiten:
1. Analyse
2. Entwurf
3. Imple-mentierung
4. Infrastruktur
...
2.1 Grob-E.
2.2 Fein-E.
3.1 Impl. Subsystem 1
3.2 Impl. Subsystem 2
Technische Universität München
B. Rumpe
Softwaretechnik, 770

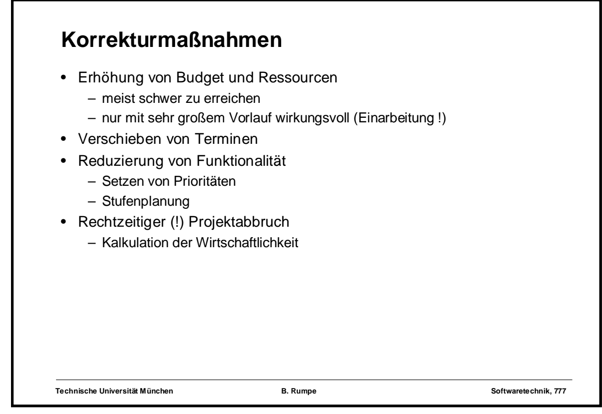

Ressourcenplanung
Anzahl Personen
5
4
3
2
1
4.1
4.1
4.1
2.2
3.1
1.1
2.1
3.2
1 2 3 4 5 6 7 8 ...
Projektwochen
Umplanung mit dem Ziel: Anpassung an vorhandene Ressourcen
Technische Universität München
B. Rumpe
Softwaretechnik, 773

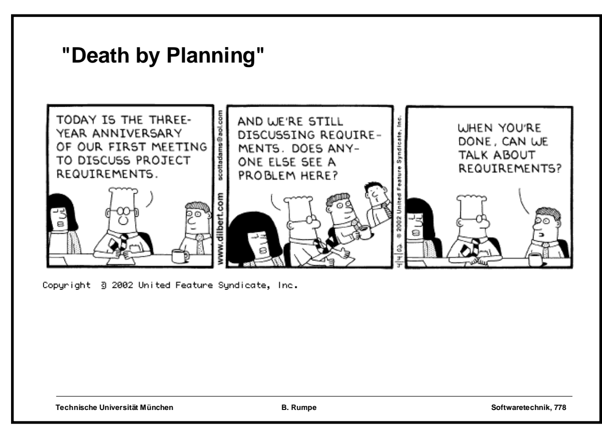

Zeitplanung
CAN YOU GET THIS DONE IN THIRTY DAYS?
YES, ABSOLUTELY.
WE'LL JUST TRAVEL FASTER THAN LIGHT TO A BLACK HOLE AND DISCOVER A DOORWAY IN TIME.
THAT SOUNDS IFFY.
EXCUSE ME FOR BEING FLEXIBLE.
www.dilbert.com
Copyright © 2002 United Feature Syndicate, Inc.
Technische Universität München
B. Rumpe
Softwaretechnik, 771

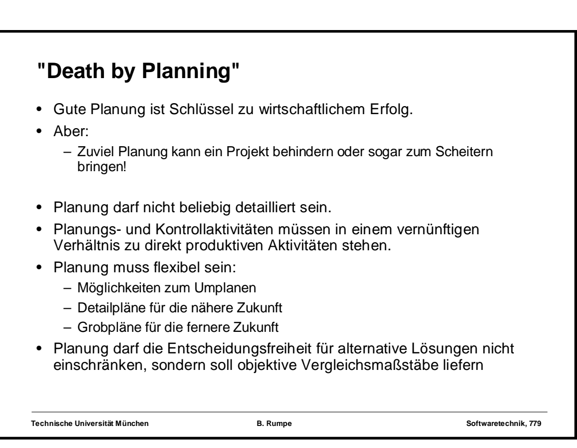

Meilensteine
Ein Meilenstein ist ein klar definiertes Zwischenresultat, an Hand dessen der Projektfortschritt beurteilt werden kann.
Beispiele:
"Anforderungsspezifikation zusammen mit Auftraggeber verabschiedet"
"Erster Prototyp lauffähig"
Schlechtes Beispiel: "Code zu 50% fertig"
Meilensteine im Gantt-Diagramm:
Projektwochen
1 2 3 4 5 6 7 8 ...
M1
M2
Arbeitspaket
1.1 Analyse
2.1 Grobentwurf
Technische Universität München
B. Rumpe
Softwaretechnik, 774

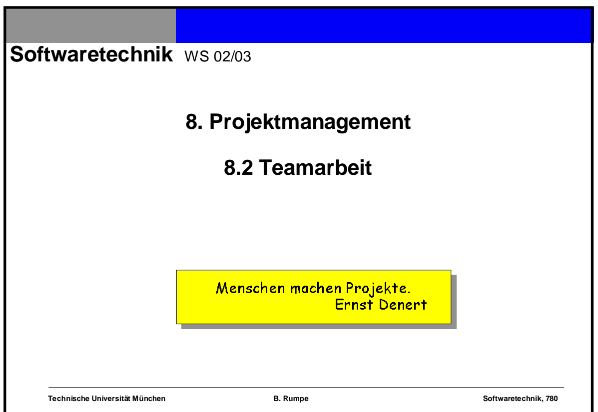

Projektverfolgung
• Das Projektmanagement muss ein "Frühwarnsystem" für eventuelle Probleme betreiben.
• Informationsquellen:
– Laufende (z.B. wöchentliche) Management-Berichte
– Arbeitszeit-Kontierung
– Resultate (deliverables)
• Rückkopplung zum Projektteam
– Regelmäßige Projektbesprechungen
– Beispiel: Akkumulierter Ressourcenverbrauch
Akkumulierte Kosten
Budget
Ist
Soll
geplantes Projekt-ende
Zeit
Berichtszeitpunkt
Technische Universität München
B. Rumpe
Softwaretechnik, 775

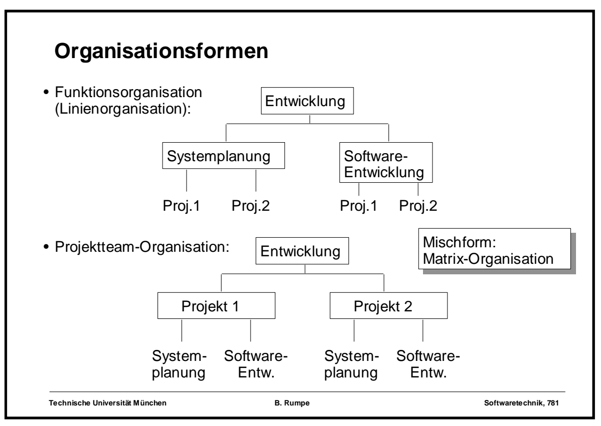

Meilenstein-Trendanalyse
Projektwochen
1 2 3 4 5 6 7 8 ...
A
B
C
Berichts-wochen
1
2
3
4
5
6
7
8
Vorhersage
Erreicht
Technische Universität München
B. Rumpe
Softwaretechnik, 776


Korrekturmaßnahmen
• Erhöhung von Budget und Ressourcen
– meist schwer zu erreichen
– nur mit sehr großem Vorlauf wirkungsvoll (Einarbeitung !)
• Verschieben von Terminen
• Reduzierung von Funktionalität
– Setzen von Prioritäten
– Stufenplanung
• Rechtzeitiger (!) Projektabbruch
– Kalkulation der Wirtschaftlichkeit
Technische Universität München
B. Rumpe
Softwaretechnik, 777

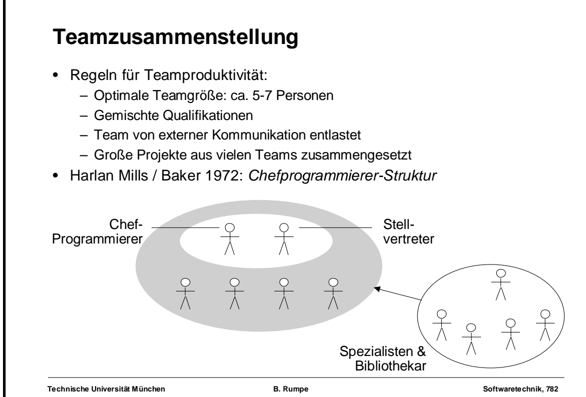

"Death by Planning"
TODAY IS THE THREE-YEAR ANNIVERSARY OF OUR FIRST MEETING TO DISCUSS PROJECT REQUIREMENTS.
AND WE'RE STILL DISCUSSING REQUIREMENTS. DOES ANYONE ELSE SEE A PROBLEM HERE?
WHEN YOU'RE DONE, CAN WE TALK ABOUT REQUIREMENTS?
www.dilbert.com
Copyright © 2002 United Feature Syndicate, Inc.
Technische Universität München
B. Rumpe
Softwaretechnik, 778


"Death by Planning"
• Gute Planung ist Schlüssel zu wirtschaftlichem Erfolg.
• Aber:
– Zuviel Planung kann ein Projekt behindern oder sogar zum Scheitern bringen!
• Planung darf nicht beliebig detailliert sein.
• Planungs- und Kontrollaktivitäten müssen in einem vernünftigen Verhältnis zu direkt produktiven Aktivitäten stehen.
• Planung muss flexibel sein:
– Möglichkeiten zum Umplanen
– Detailpläne für die nähere Zukunft
– Grobpläne für die fernere Zukunft
• Planung darf die Entscheidungsfreiheit für alternative Lösungen nicht einschränken, sondern soll objektive Vergleichsmaßstäbe liefern
Technische Universität München
B. Rumpe
Softwaretechnik, 779

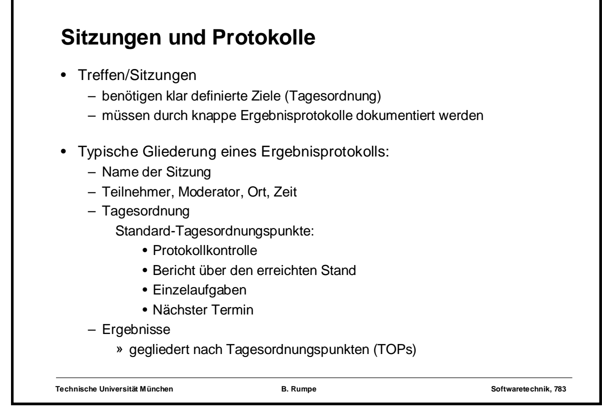

Softwaretechnik WS 02/03
8. Projektmanagement
8.2 Teamarbeit
Menschen machen Projekte.
Ernst Denert
Technische Universität München
B. Rumpe
Softwaretechnik, 780



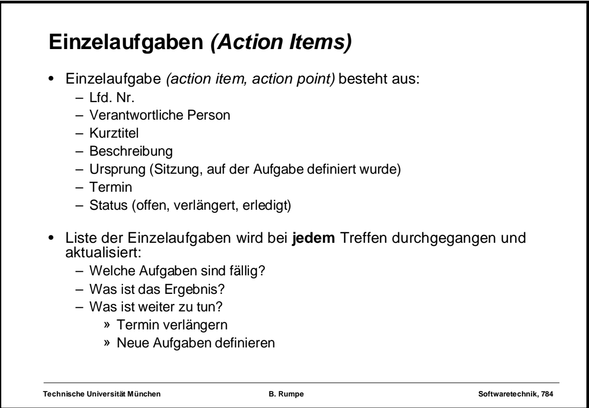
Organisationsformen
• Funktionsorganisation (Linienorganisation):
Entwicklung
Systemplanung
Software-Entwicklung
Proj.1
Proj.2
Proj.1
Proj.2
• Projektteam-Organisation:
Entwicklung
Mischform: Matrix-Organisation
Projekt 1
Projekt 2
System-planung
Software-Entw.
System-planung
Software-Entw.
Technische Universität München
B. Rumpe
Softwaretechnik, 781



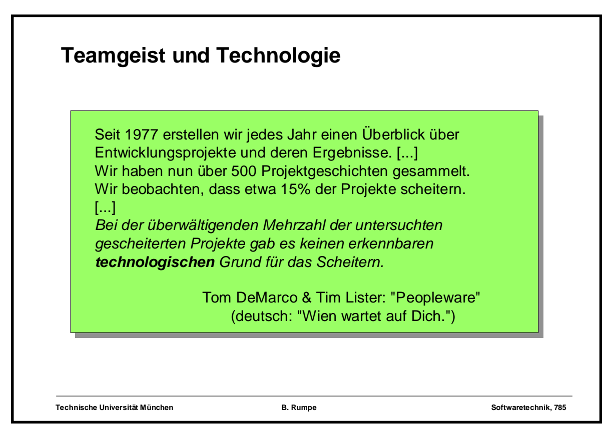
Teamzusammenstellung
• Regeln für Teamproduktivität:
– Optimale Teamgröße: ca. 5-7 Personen
– Gemischte Qualifikationen
– Team von externer Kommunikation entlastet
– Große Projekte aus vielen Teams zusammengesetzt
• Harlan Mills / Baker 1972: Chefprogrammierer-Struktur
Chef-Programmierer
Stell-vertreter
Spezialisten & Bibliothekar
Technische Universität München
B. Rumpe
Softwaretechnik, 782



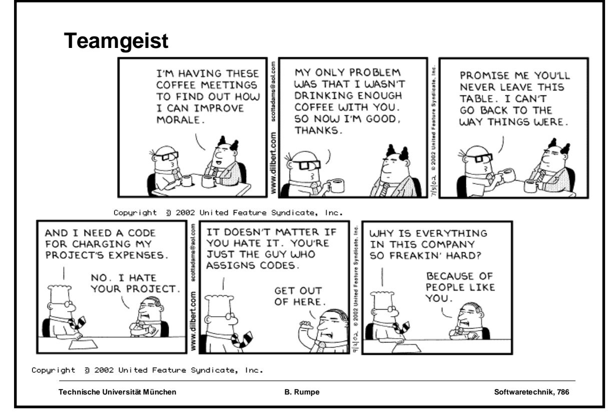
Sitzungen und Protokolle
• Treffen/Sitzungen
– benötigen klar definierte Ziele (Tagesordnung)
– müssen durch knappe Ergebnisprotokolle dokumentiert werden
• Typische Gliederung eines Ergebnisprotokolls:
– Name der Sitzung
– Teilnehmer, Moderator, Ort, Zeit
– Tagesordnung
Standard-Tagesordnungspunkte:
• Protokollkontrolle
• Bericht über den erreichten Stand
• Einzelaufgaben
• Nächster Termin
– Ergebnisse
» gegliedert nach Tagesordnungspunkten (TOPs)
Technische Universität München
B. Rumpe
Softwaretechnik, 783



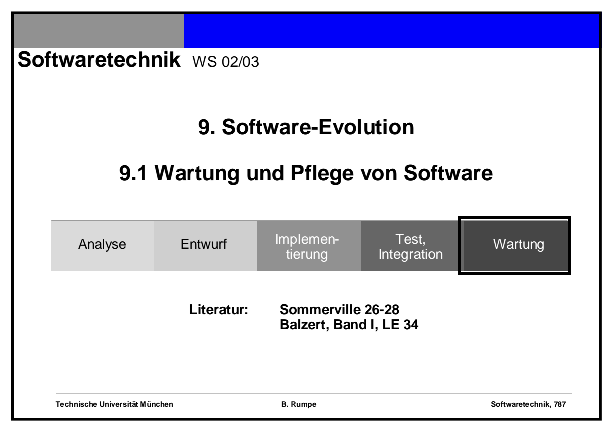
Einzelaufgaben (Action Items)
• Einzelaufgabe (action item, action point) besteht aus:
– Lfd. Nr.
– Verantwortliche Person
– Kurztitel
– Beschreibung
– Ursprung (Sitzung, auf der Aufgabe definiert wurde)
– Termin
– Status (offen, verlängert, erledigt)
• Liste der Einzelaufgaben wird bei jedem Treffen durchgegangen und aktualisiert:
– Welche Aufgaben sind fällig?
– Was ist das Ergebnis?
– Was ist weiter zu tun?
» Termin verlängern
» Neue Aufgaben definieren
Technische Universität München
B. Rumpe
Softwaretechnik, 784



Teamgeist und Technologie
Seit 1977 erstellen wir jedes Jahr einen Überblick über Entwicklungsprojekte und deren Ergebnisse. [...]
Wir haben nun über 500 Projektgeschichten gesammelt.
Wir beobachten, dass etwa 15% der Projekte scheitern.
[...]
Bei der überwältigenden Mehrzahl der untersuchten gescheiterten Projekte gab es keinen erkennbaren technologischen Grund für das Scheitern.
Tom DeMarco & Tim Lister: "Peopleware"
(deutsch: "Wien wartet auf Dich.")
Technische Universität München
B. Rumpe
Softwaretechnik, 785

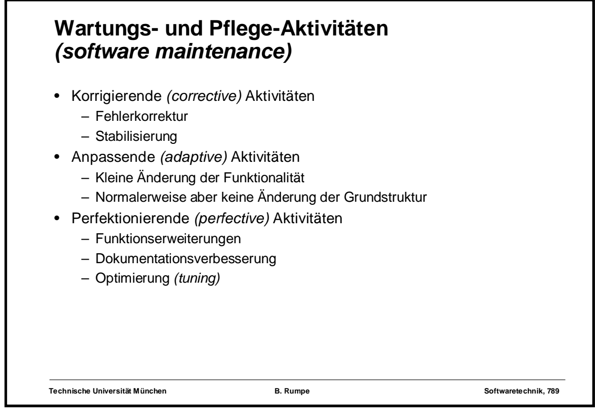


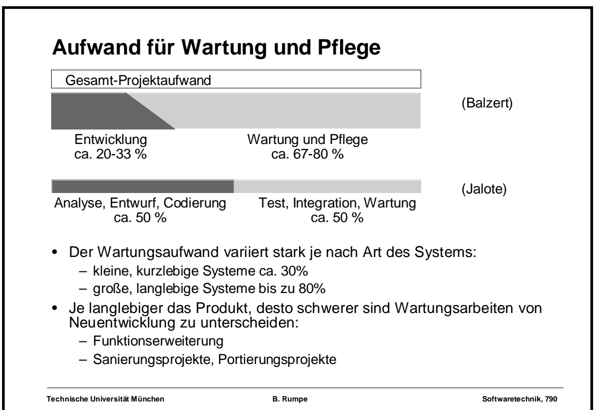
Teamgeist
I'M HAVING THESE COFFEE MEETINGS TO FIND OUT HOW I CAN IMPROVE MORALE.
MY ONLY PROBLEM WAS THAT I WASN'T DRINKING ENOUGH COFFEE WITH YOU. SO NOW I'M GOOD. THANKS.
PROMISE ME YOU'LL NEVER LEAVE THIS TABLE. I CAN'T GO BACK TO THE WAY THINGS WERE.
Copyright © 2002 United Feature Syndicate, Inc.
AND I NEED A CODE FOR CHARGING MY PROJECT'S EXPENSES.
NO. I HATE YOUR PROJECT.
IT DOESN'T MATTER IF YOU HATE IT. YOU'RE JUST THE GUY WHO ASSIGNS CODES.
GET OUT OF HERE.
WHY IS EVERYTHING IN THIS COMPANY SO FREAKIN' HARD?
BECAUSE OF PEOPLE LIKE YOU.
Copyright © 2002 United Feature Syndicate, Inc.
Technische Universität München
B. Rumpe
Softwaretechnik, 786

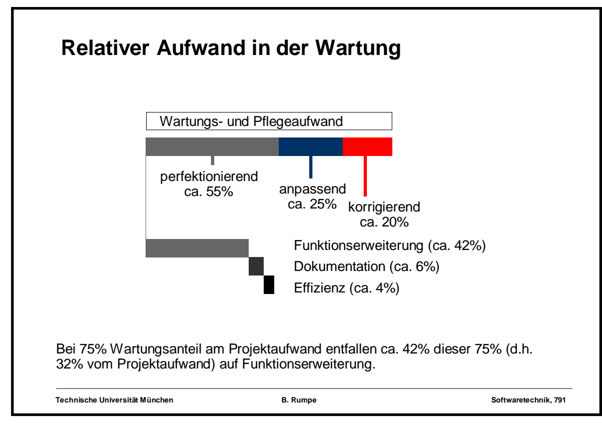


Notwendigkeit von Wartung und Pflege

- Nicht entdeckte Fehler (Restfehler)
  - *Restfehler* sind Fehler im Produkt nach Produktfreigabe
  - Durchschnittlich:
    - » 7 Restfehler auf 1000 Anweisungen (1977)
    - » 0,2 Restfehler auf 1000 Anweisungen (1990)
  - Auf 10 Fehler, die vor der Freigabe (z.B. durch Test) gefunden werden, kommt 1 Restfehler
  - Ungefähr 10facher Aufwand zur Behebung von Restfehlern (im Einsatz) im Vergleich zu Fehlerbehebung vor der Freigabe
- Neue Benutzeranforderungen
- Neue Hardware- und Software-Plattformen
- Unbefriedigende Zeit- und/oder Speichereffizienz



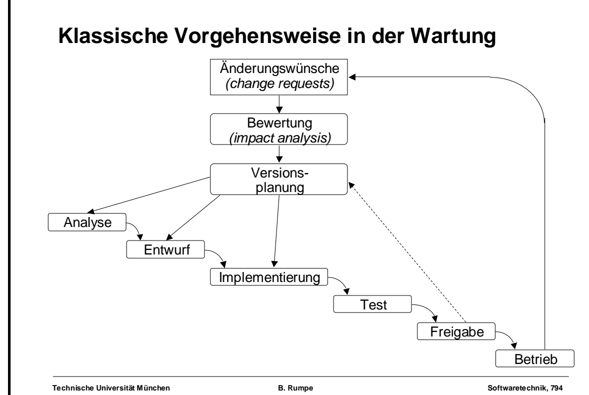




Typische Probleme in der Wartungsphase

- Einsatz wenig erfahrenen Personals
- Zu wartende Programme sind alt („Erosion“):
  - fehlendes Hintergrundwissen
  - schlechtere Struktur
  - oft stärker optimiert (knappe Ressourcen)
- Fehlerbehebung führt oft neue Fehler ein.
- Fehlerbehebung verschlechtert Programmstruktur.
- Zusammenhang zwischen Programm und Dokumentation geht verloren.

Klassische Vorgehensweise in der Wartung



Einflussfaktoren auf die Wartungskosten

- Nicht-technische Faktoren
  - Anwendungsbereich
  - Personelle Stabilität im Entwicklungsteam
  - Alter des Systems
  - Äußere Einflussfaktoren
  - Evolution der Hardware- und Software-Plattform
- Technische Faktoren
  - Niedrige Kopplung zwischen Modulen
  - Programmiersprache, Werkzeuge
  - Aufwand für Anforderungsvalidation
  - Aufwand für Programmtests
  - Qualität der Dokumentation
  - Konfigurationsverwaltung
  - Verwaltung von Änderungswünschen

Versionen und Konfigurationen

- Große Programmsysteme bestehen aus vielen Komponenten:
  - Programm-Module, Klassen
  - Subsysteme
  - Notwendige Dateien (z.B. Grafiken und andere Mediendaten)
  - Dokumente
    - » Spezifikationen
    - » Entwickler-Dokumentation
    - » Benutzer-Dokumentation
- Programmsysteme existieren in mehreren Versionen:
  - Versionen für verschiedene Plattformen (z.B. Windows, Mac, Unix)
  - Versionen für verschiedene Länder
  - Versionen für verschiedene Kunden
  - Freigabeversionen (Releases)
  - Testversionen (Alpha- und Beta-Test, Interne Zwischenversionen)

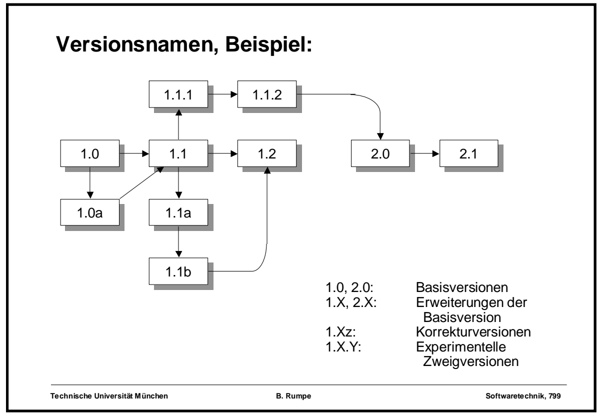
Identifizieren und Aufbewahren

- Jedes Zwischen- und Teilprodukt muss eindeutig identifizierbar sein:
  - Typ des Produkts (z.B. "Spezifikation", "Java-Programmcode")
  - Name des Produkts (z.B. "Anforderungsspezifikation", "package GUI")
  - Versionsnummer
- Weitere relevante Information:
  - Original-Autor
  - Autor der aktuellen Version
  - Erstellungsdatum
  - Datum der letzten Änderung
  - Größe
- Aufbewahrung aller Versionen aller Zwischenprodukte!
  - Werkzeuge zur Versionsverwaltung (z.B. CVS)
    - » Unterstützung von Entwicklerteams: Ein/Aus-"checken"
    - » Speichereffizienz z.B. durch Speichern von Differenzen

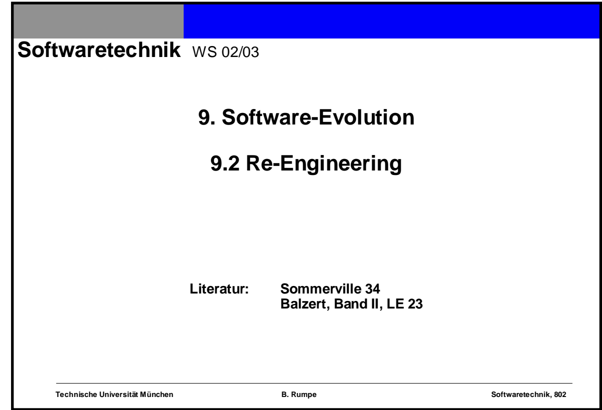
Konfigurationsmanagement

- Jede Systemkonfiguration (komplettes Produkt) muss eine eindeutige Versionsnummer tragen (Systemversion).
- Zuordnung Systemversionen – Versionen aller Komponenten:
  - Angabe der zugehörigen Systemversion in allen Dokumenten und Code-Dateien, oder
  - Zuordnungstabelle bzw. Datenbank
- Verzweigung der Versionsentwicklung:
  - z.B. Ländervarianten, experimentelle Versionen
  - Generell möglichst zu vermeiden !
    - » z.B. bei Ländervarianten: Kernversion und flexible Mechanismen zur "Lokalisierung"
  - Bei echter Verzweigung: Konsolidierungsproblem

Versionsnamen, Beispiel:



1.0, 2.0: Basisversionen
1.X, 2.X: Erweiterungen der Basisversion
1.Xz: Korrekturversionen
1.X.Y: Experimentelle Zweigversionen



Versionsplanung

- Beginnt bereits in der Anforderungsanalyse
  - Anforderungsspezifikation für neue Versionen jeweils anpassen!
- Begleitet die Produktevolution mit zeitlichem Vorlauf
- Ist eng mit Fragen des Marketing verknüpft:
  - Unternehmensstrategie
  - Marktentwicklung
  - Kundenerwartungen
  - Mitbewerber-Beobachtung
  - Standards (auch *de facto*)
  - Marktnischen

Sanierung von Altsystemen

- Immer öfter werden Systeme nicht neu erstellt, sondern aus Altsystemen entwickelt, die sie ablösen.
- Altsysteme *(legacy systems):*
  - Wesentliche Funktion für ein Unternehmen
  - Wissen und Verfahren, die nicht oder schlecht dokumentiert sind
  - Häufig für Großrechner *(mainframes)*
  - Oft Sammlung nur schlecht integrierter Programme
  - Oft unsystematische und redundante Datenhaltung
  - Wartung kaum möglich
- Sanierung:
  - Übertragung des "versteckten" Wissens in moderne Umgebung
  - Verbesserung der Wartbarkeit
  - Portierung auf neue Plattformen (z.B. PCs, Workstations)

Empirische Erfahrungen zur Wartung

- Komponenten mit hohen Fehlerzahlen im Test werden auch viele Fehler nach der Freigabe verursachen.
- Änderungen in der Wartung verursachen mehr Fehler als "normale" Softwareentwicklung
  - Regressionstests nach jeder Änderung !
- Als "einfach" eingeschätzte Änderungen sind oft fehlerbehaftet
  - "Titanic"-Effekt (Gerald Weinberg)

Quelle für diese und andere nützliche "Alltagsweisheiten":

Alan Davis, 201 Principles of Program Development, McGraw-Hill 1995

Re-Engineering und Reverse Engineering

Vorwärtsentwicklung ***(forward engineering):***

Spezifikation → Entwurf, Implementierung → System, Dokumentation

Rückwärtsentwicklung ***(reverse engineering):***

System → Analyse, Beschreibung → Spezifikation, Dokumentation

Sanierung ***(re-engineering):***

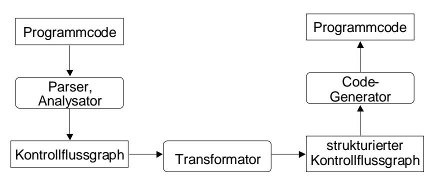
Alt-System → Transformation → Neu-System

Varianten des Re-Engineering

- Übertragung in andere Programmiersprache oder Umgebung
  - automatisch
  - manuelle Ergänzungen
- Kontrollstrukturverbesserung
  - automatische Strukturierung
  - manuelle/halbautomatische Strukturierung
- Datenstrukturverbesserung
  - manuelle Entwurfsarbeit
  - Werkzeughilfe
- Architekturveränderung
  - Gründliche Analyse (Reverse Engineering)
  - Teilweiser Neuentwurf

Automatische Re-Strukturierung

- Vom "Spaghetti-Code" zum strukturierten Programm
- Kombinierbar mit Umsetzung in andere Sprachen



- Probleme:
  - Verlust von Kommentaren
  - Abkopplung von der Dokumentation
  - Oft entstehen tief verschachtelte Strukturen

Daten-Restrukturierung, Anwendungsbeispiel

Heterogene Datenhaltung:

Programm 1, Programm 2, Programm 3, Datei 1, Datei 2

Homogene Datenhaltung:

Programm 1, Programm 2, Programm 3, Datenbank, Datenmodell, beschreibt

Datenhaltungs-Probleme bei Altsystemen

- Probleme innerhalb eines Programms:
  - Namensgebung
  - Feldlängen, Formate
    - » z.B. Jahr-2000-Problem: '01' = 1901 oder '01' = 2001 ?
  - Fest kodierte Konstante
- Probleme mehrerer kooperierender Programme:
  - Inkonsistente Datensatz-Organisation
  - Inkonsistente Einheiten (z.B. kp - N, fl.oz. - l)
  - Inkonsistente Formate (z.B. TIFF - GIF)
  - Inkonsistente Default-Werte
  - Inkonsistente Konventionen (z.B. DM - DEM)
  - Inkonsistente Integriltätsbedingungen
  - Inkonsistente Ausnahmebehandlung

Informationsquellen im Reverse Engineering

- Dokumentation
  - Oft fehlerhaft oder nicht existent
- Code des Altsystems ("glass box")
  - Unter Umständen sehr schwer verständlich!
- Experimente mit dem System ("black box")
  - Ermöglicht im Prinzip zweifelsfreie Untersuchung der Funktion
- Erfahrungen der Benutzer
  - Befragungen der Benutzer ähnlich wie bei Anforderungsanalyse
- Erfahrungen des Wartungspersonals
  - Hinweise auf besonders sanierenswerte Teile des Systems
- Neue Analyse des Problembereichs
  - Forward- statt Reverse-Engineering

Automatisierung von Reverse Engineering

- Halbautomatisch:
  - Analyse von Altsystem-Code ("Extractor")
    - » einschließlich Datenflussanalyse, wie z.B. in optimierenden Compilern
  - Ablage in Repository
  - Flexible Abfragemöglichkeiten ("Query")
    - » Aufrufstruktur
    - » Import-Beziehungen
  - Visualisierung komplexer Information ("Viewer")
- Automatisch
  - z.B. Versuche zur automatischen Synthese von Klassen aus prozeduralen Programmen
    - » "Cluster-Analyse"
  - Meist Zusatzinformation nötig
  - Aktuelles Forschungsgebiet

Probleme im automat. Reverse Engineering

- Code-Analyse
  - Vielfalt von Sprachen, Dialekten, Varianten
  - Schlechte Verfügbarkeit von Grammatiken
- Repository/Abfrage
  - Datenexplosion, wenn jede Verwendung einer Variablen festgehalten wird
  - Statische Information u.U. nicht ausreichend
- Visualisierung
  - Standard-Notationen, z.B. UML
    - » Sehr detaillierte Diagramme schwer verständlich
    - » "Antimuster" (z.B. "Blob") gut erkennbar
  - Statistische Aussagen, z.B. Histogramme
    - » Vielzahl von Informationen schwer zu korrelieren
- Aktueller Gesamteindruck:
  - Abstraktionsprinzipien noch wenig beherrscht