
Softwaretechnik

Vorlesung und Übungen

Prof. Dr. Matthias Riebisch
Dr. Guido Gryczan
Eugen Reiswich
Simon Gerlach
Michael König
MIN-Fakultät, FB Informatik
SWK, SWA, RZ

<http://swk-www.informatik.uni-hamburg.de/teaching/swt/>

Stand April 13

Termine im Sommersemester 2013

- Vorlesung I Donnerstag 10:15-11:45 B-201 (Prof. Matthias Riebisch)
- Vorlesung II Fr 14:15-15:45 B-201 (Dr. Guido Gryczan)
Ausnahmen siehe Zeitplan (Stine oder CommSy)
- Übungen
 - Gr.01: Di 14:00-17:45 D-125 (Simon Gerlach)
 - Gr.02: Mi 12:30-15:45 D-220 (Eugen Reiswich)
 - Gr.03: Do 12:30-15:45 F-009 (Michael König)
 - Gr.04: Mi 12:30-15:45 D-010 - evtl. Raumänderung (Michael König)Aktuelle Informationen über CommSy
Übungsteilnahme bei anderen Gruppen nicht möglich, außer wenn mit Übungsleiter anderweitig vereinbart

Abschluss

Modul besteht aus 2 Leistungen:

Vorlesung:

- Note aus mündlicher Prüfung
- Prüfungsinhalte: aus beiden Vorlesungen und Übungen
 - Vorlesung I: Fragen zur Prüfungsvorbereitung
 - Vorlesung II: Prüfungsfragen des Vorjahrs

Übung:

- Schein über Teilnahme
- Schein nicht erteilt, wenn mehr als 1 Übung versäumt

Übung - Ablauf

Donnerstag der Woche vor der Übung:

- Übungsblätter mit Aufgaben im CommSy abholen
- Aufgaben vorbereiten: Vorkenntnisse für Lösung sammeln

In der Übung:

- Aufgaben in Kleingruppe (3-4 Personen) bearbeiten
- Lösung der Kleingruppe vorstellen

Nach der Übung:

- Vergleich mit Musterlösung aus CommSy

Übung - Inhalte

1. Use Case erstellen
 - UML Use Case Diagramm, Use Case Template
2. Aktivitätsdiagramme und Zustandsdiagramme erstellen
3. Mock-Ups erstellen
 - Abbildung von Use-Cases auf Mock-Ups

4. Entwurf
 - Umsetzung Fachliches Modell in Entwurfsmodell, WAM-Ansatz, Klassendiagramme
5. Testfälle erstellen
 - Aus Spezifikation (Use Case Template, Aktivitätsdiagramm, Problembereichsmodell)
 - Testfälle entwickeln und Testabdeckung bewerten
6. Code-Review
 - Quellcode auf Codierstil und Verletzung von Vorgaben bewerten

Materialien für Vorlesung I

- Folien nur für Definitionen, umfangreichen Text und Grafiken
- Fragen zur Prüfungsvorbereitung (je Vorlesung)
- Bereitstellung zum Download im CommSy

Bücher:

- Bernd Oestereich: Analyse und Design mit UML 2.3 - Objektorientierte Softwareentwicklung. Oldenbourg Verlag, 2009
- Martin Fowler: Refactoring - Wie Sie das Design vorhandener Software verbessern. Addison Wesley, 2000.

Gliederung der Vorlesung I – Donnerstags

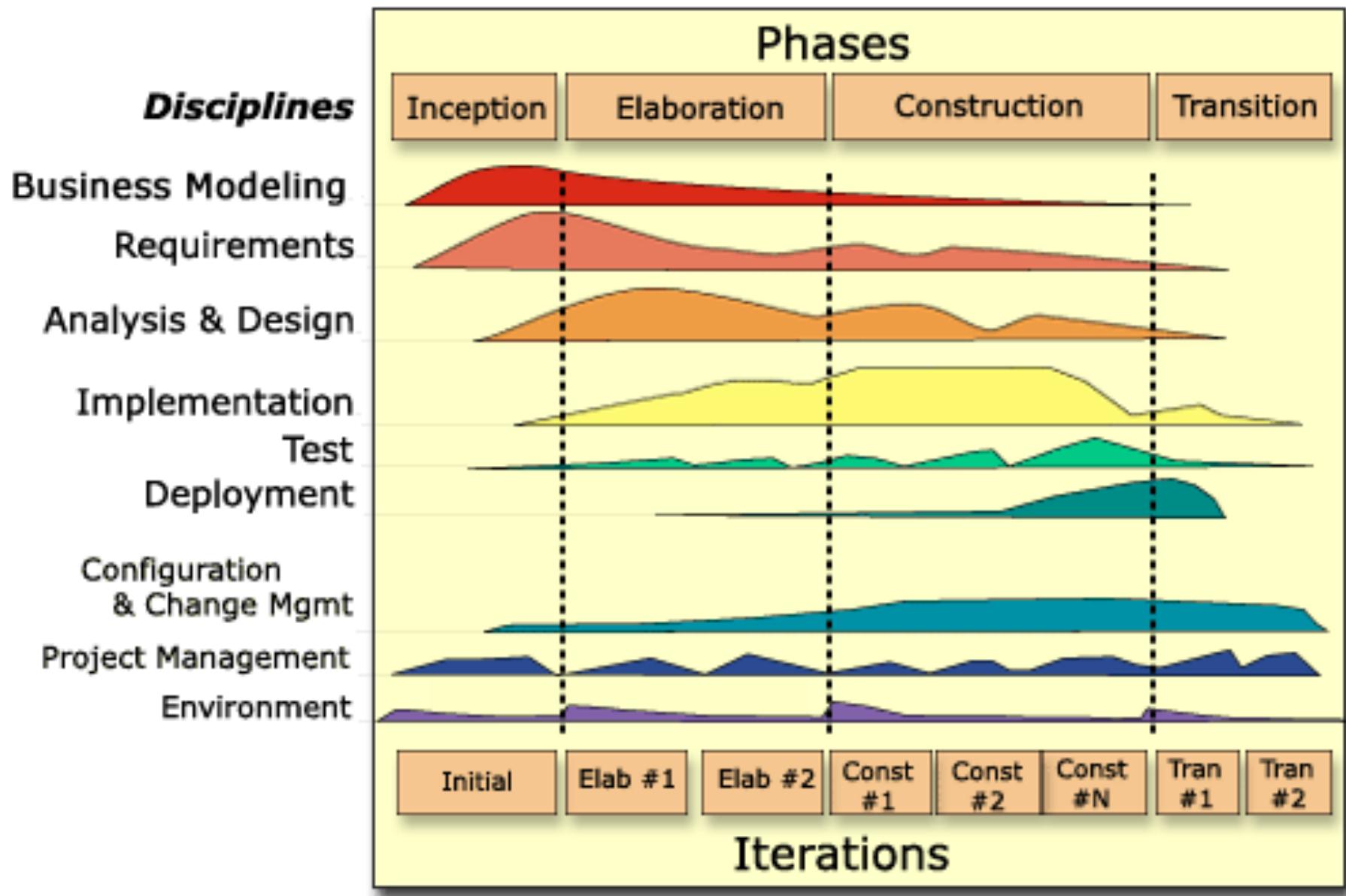
- 0. Organisatorisches
- 1. Anforderungsbeschreibung mit UML
 - 1.1. Die „frühen Phasen“ der Softwareentwicklung
 - 1.2. Use-Case-Modellierung mit UML
 - 1.3. Aktivitätsdiagramm
 - 1.4. Problembereichsmodellierung
 - 1.5. Klassendiagramm
 - 1.6. Zustandsdiagramm
- 2. Entwurf
 - 2.1. Rolle des Entwurfs
 - 2.2. Strukturentwurf
 - 2.3. Zuordnung von Verhalten zu Strukturen
 - 2.4. Kommunikationsdiagramm
 - 2.5. Sequenzdiagramm
- 3. SW-Qualitätsmanagement
 - 3.1. Qualitätsmerkmale und konstruktive Maßnahmen
 - 3.2. Analytische Maßnahmen: Test
 - 3.3. Analytische Maßnahmen: Inspektionen
- 4. Wartung und Reengineering
 - 4.1. Ziele und Begriffe
 - 4.2. Refactoring
 - 4.3. Bad Smells
- 5. Vorgehensmodelle der Softwareentwicklung
 - 5.1. V-Modell
 - 5.2. Agile Vorgehensweisen – SCRUM
 - 5.3. Iteratives Vorgehen - RUP

1. Anforderungsbeschreibung

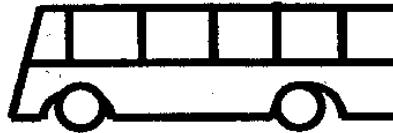
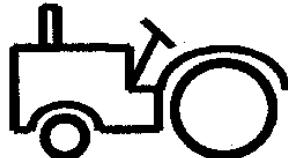
Fragen

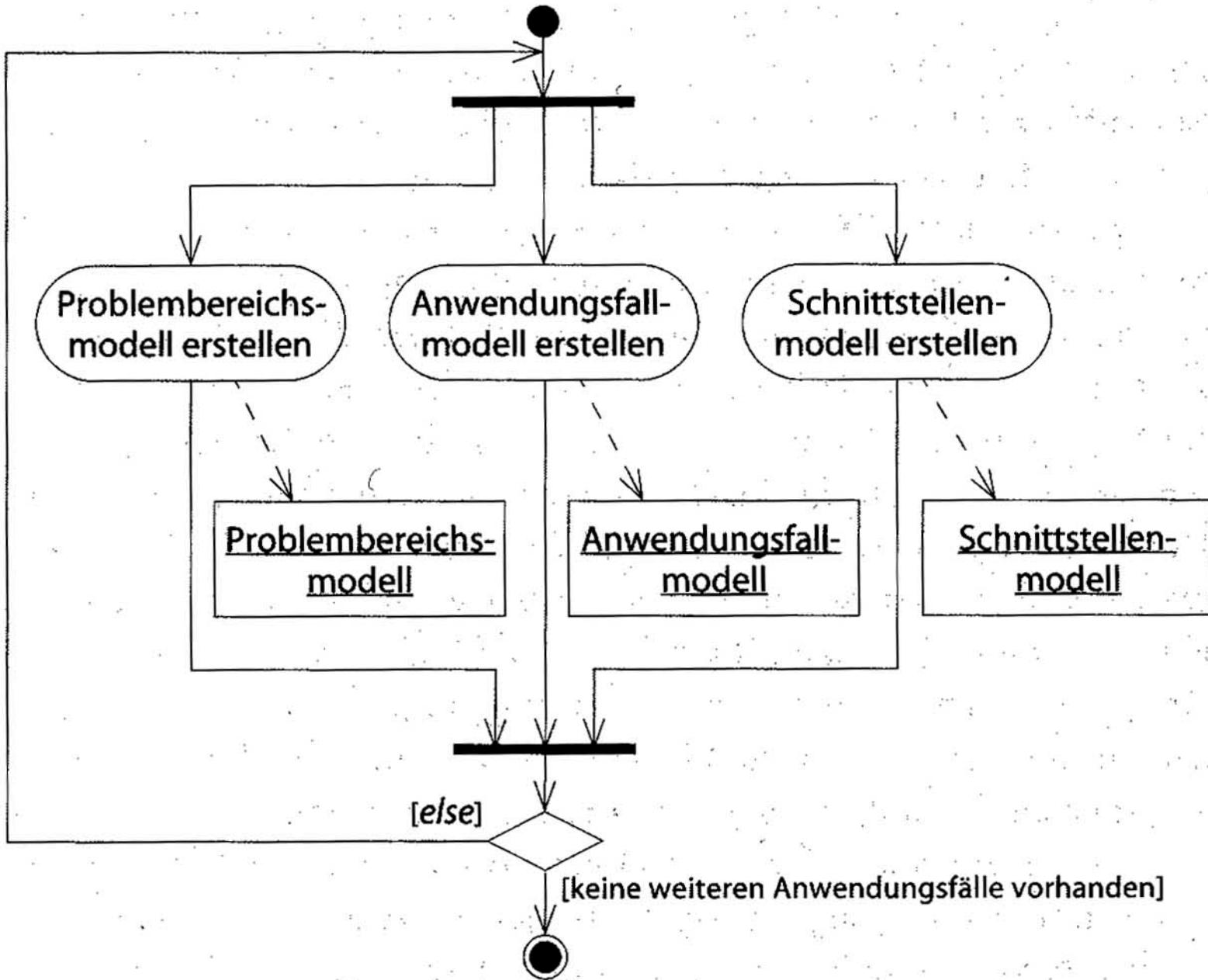
1. Worin besteht die Bedeutung der frühen Phasen der Systementwicklung?
2. Nennen Sie Ziele und Techniken der Anforderungsanalyse!
3. Geben Sie eine Definition für Use Case und Use-Case-Diagramm an!
4. Geben Sie die Elemente des Use-Case-Diagramms an!
5. Geben Sie die Gliederung für Use-Case-Beschreibungen an!
6. Wozu dient ein Paket der UML?
7. Welche Elemente enthält ein Aktivitätsdiagramm?

Iteratives Vorgehen im Unified Process



Anforderungsanalyse

Was der Anwender wollte 	Wie es der Anwender dem Programmierer sagte 	Wie es der Programmierer verstanden hat 
Was der Programmierer bauen wollte 	Was der Programmierer tatsächlich gebaut hat 	Was der Anwender tatsächlich gebraucht hätte 



M:

Spezifikation: Beispiel Volere

PROJECT DRIVERS

1. The Purpose of the Product
2. Client, Customer and other Stakeholders
3. Users of the Product

PROJECT CONSTRAINTS

4. Mandated Constraints
5. Naming Conventions and Definitions
6. Relevant Facts and Assumptions

FUNCTIONAL REQUIREMENTS

7. The Scope of the Work
8. The Scope of the Product
9. Functional and Data Requirements

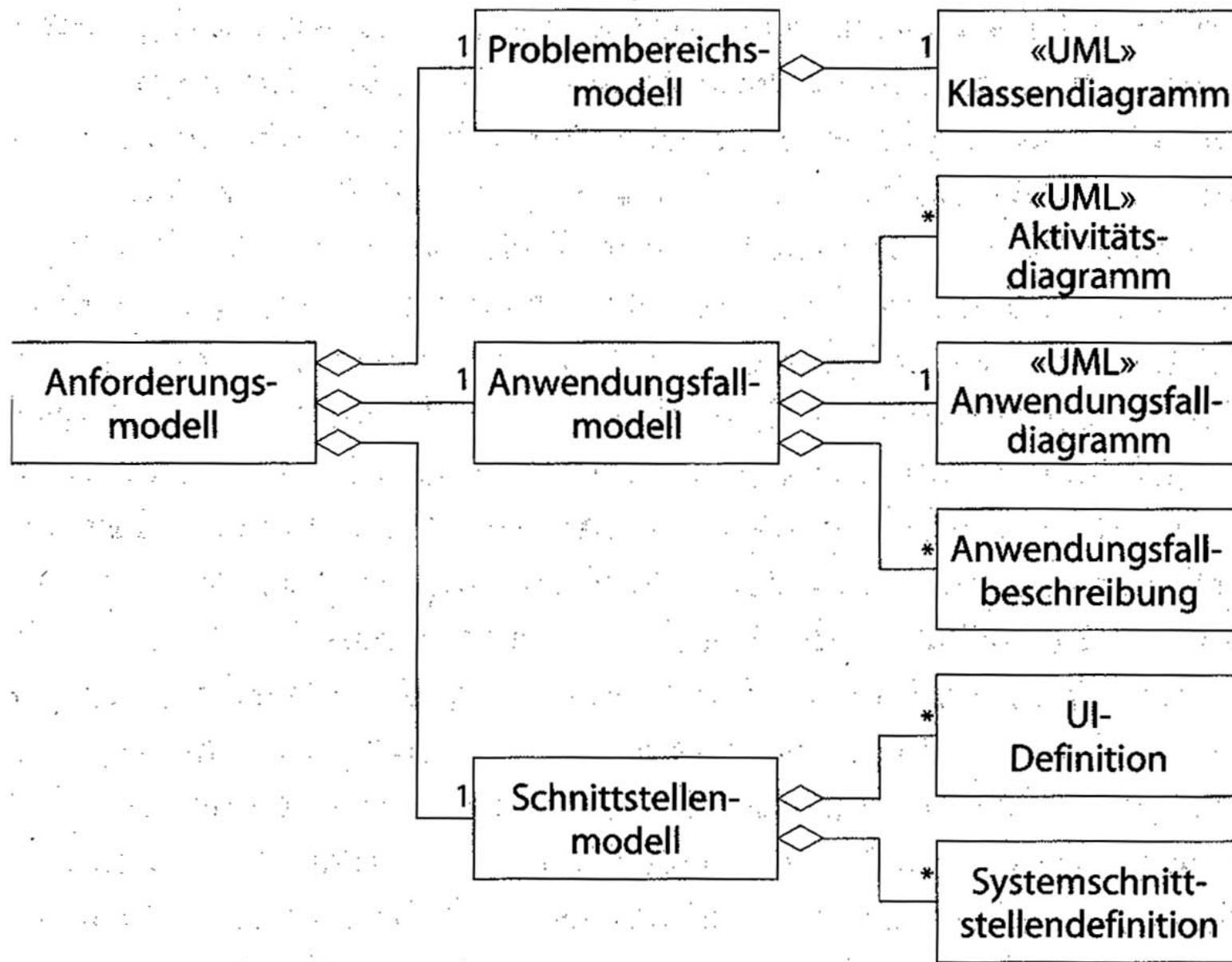
NON-FUNCTIONAL REQUIREMENTS

10. Look and Feel Requirements
11. Usability Requirements
12. Performance Requirements
13. Operational Requirements
14. Maintainability and Portability Requirements
15. Security Requirements
16. Cultural and Political Requirements
17. Legal Requirements

PROJECT ISSUES

18. Open Issues
19. Off-the-Shelf Solutions
20. New Problems
21. Tasks
22. Cutover
23. Risks
24. Costs
25. User Documentation and Training

Teile des Anforderungsmodells



1.2 Use Case und Use-Case-Diagramm

Definition Use Case (Anwendungsfall):

- Ein Use Case beschreibt eine Menge von Aktivitäten eines Systems aus der Sicht seiner Akteure, die für die Akteure zu einem wahrnehmbaren Ergebnis führen. Ein Use Case ist ansonsten eine komplette, unteilbare Beschreibung.

Definition Use-Case-Diagramm (Anwendungsfalldiagramm):

- Ein Use-Case-Diagramm zeigt die Beziehungen zwischen Akteuren und Use Cases.

Identifikation von Use Cases

- Sammeln der Kundenwünsche
- Analysieren von Textdokumenten: Verben mit Tätigkeitsbeschreibung
- "echte" Use Cases finden, verallgemeinern und zusammenfassen
- zunächst recht vollständig, weitere Verfeinerung
- für jeden Use Cases Textbeschreibung angeben
--> Vervollständigung
- Aufnahme in Use-Case-Diagramm

Use-Case-Diagramm: Elemente

- System: Rechteck mit Namen
- Aktor: Strichmännchen oder Rechteck mit <<aktor>>, und Name
- Use Case: Ellipse mit Name
- Kommunikationsbeziehung: einfache Linie
- include-Beziehungen: gestrichelte Linie mit Pfeil; zwischen Use Cases, Teil-von-Beziehung
- extend-Beziehung: kennzeichnet Varianten
- Generalisierungsbeziehung: Verallgemeinerung, ist-ein

Abb. 2-52
Anwendungsfalldiagramm
für CALENDARUM

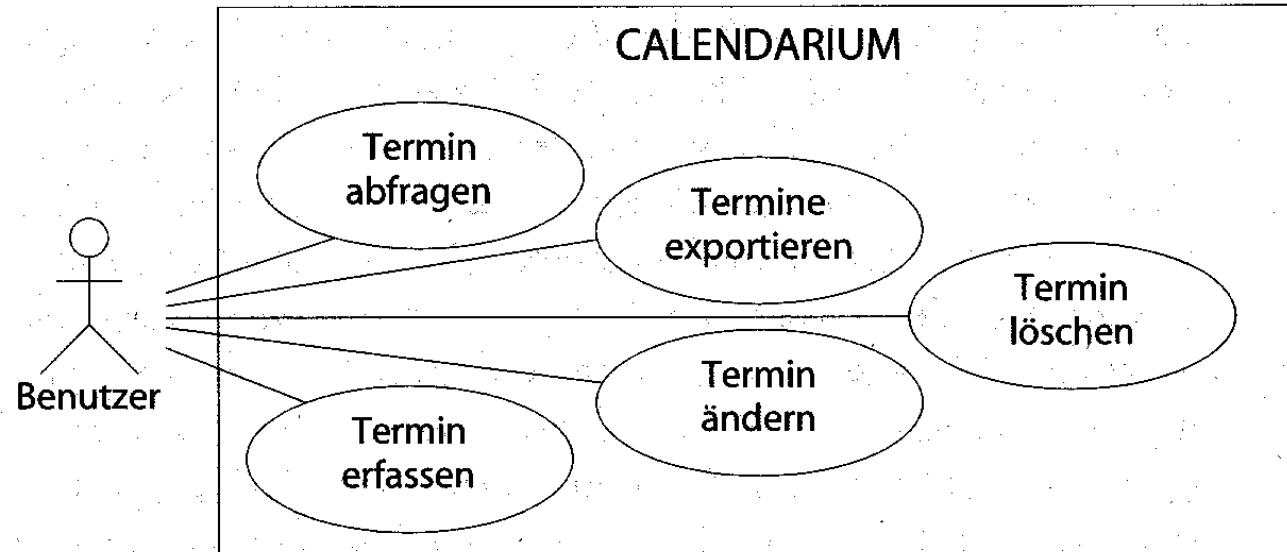
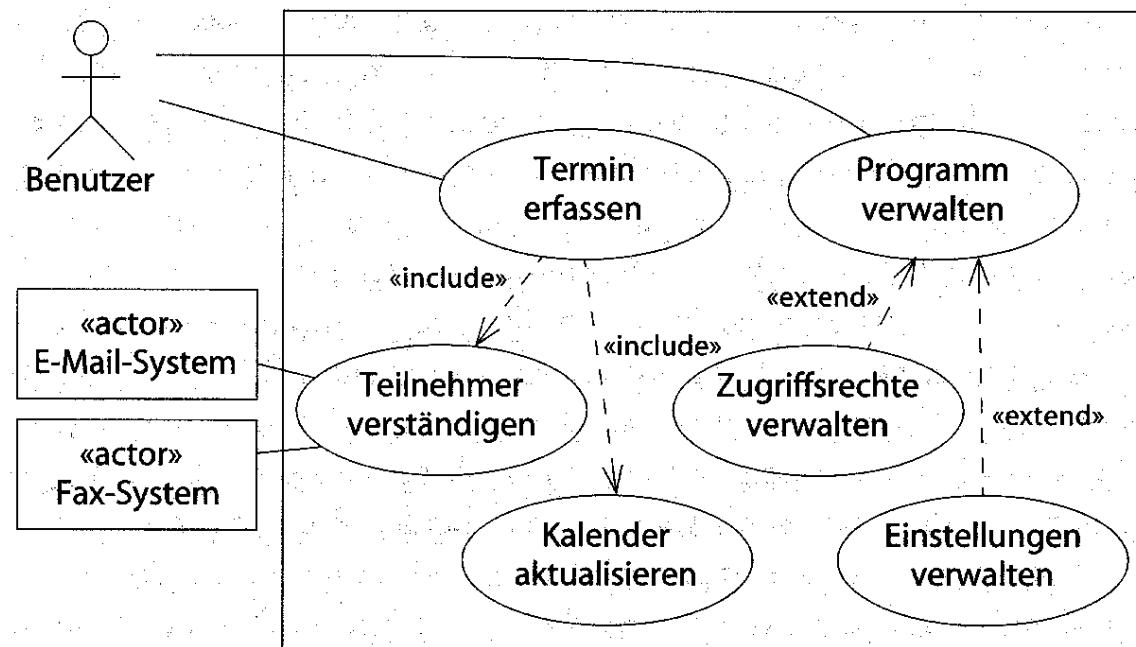


Abb. 2-53
include-Beziehung und
extend-Beziehung zwischen
Anwendungsfällen

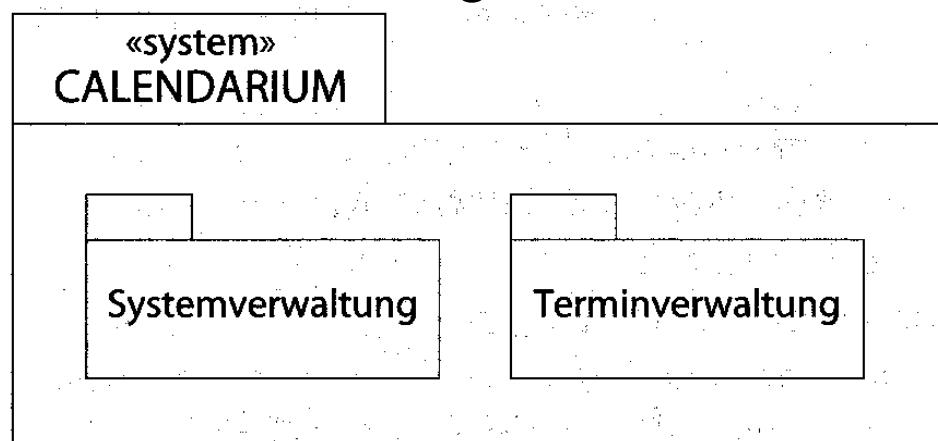


Use-Case-Beschreibung

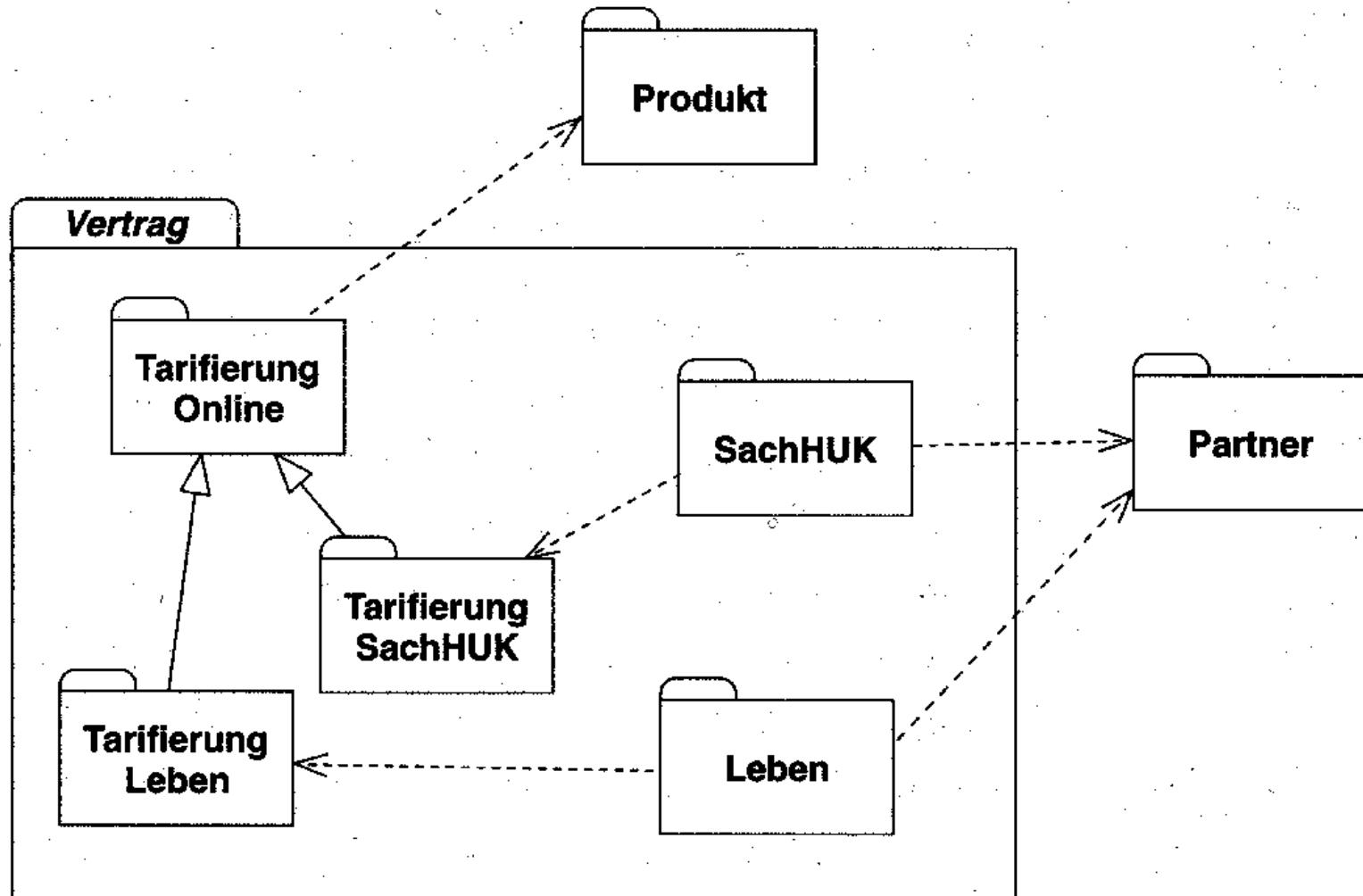
- Anwendungsfall-Nr. Name des Anwendungsfalls
- Akteure
- Vorbedingungen
- Nachbedingungen
- Invarianten
- Qualitätsmerkmale
- Ablaufbeschreibung / Durchführung
- Ausnahmen, Fehlersituation
- Alternativabläufe

Paket in der UML

Definition: Pakete sind Ansammlungen von Modellelementen beliebigen Typs, mit denen das Gesamtmodell in kleinere, überschaubare Einheiten gegliedert wird. Ein Paket definiert einen Namensraum (innerhalb dessen Namen eindeutig sein müssen) und kann (hierarchisch) wiederum Pakete enthalten. Das oberste Paket beinhaltet das ganze System. Jedes Modellelement kann in anderen Paketen referenziert werden, gehört aber zu genau einem (Heimat-)Paket.



Pakete mit Beziehungen



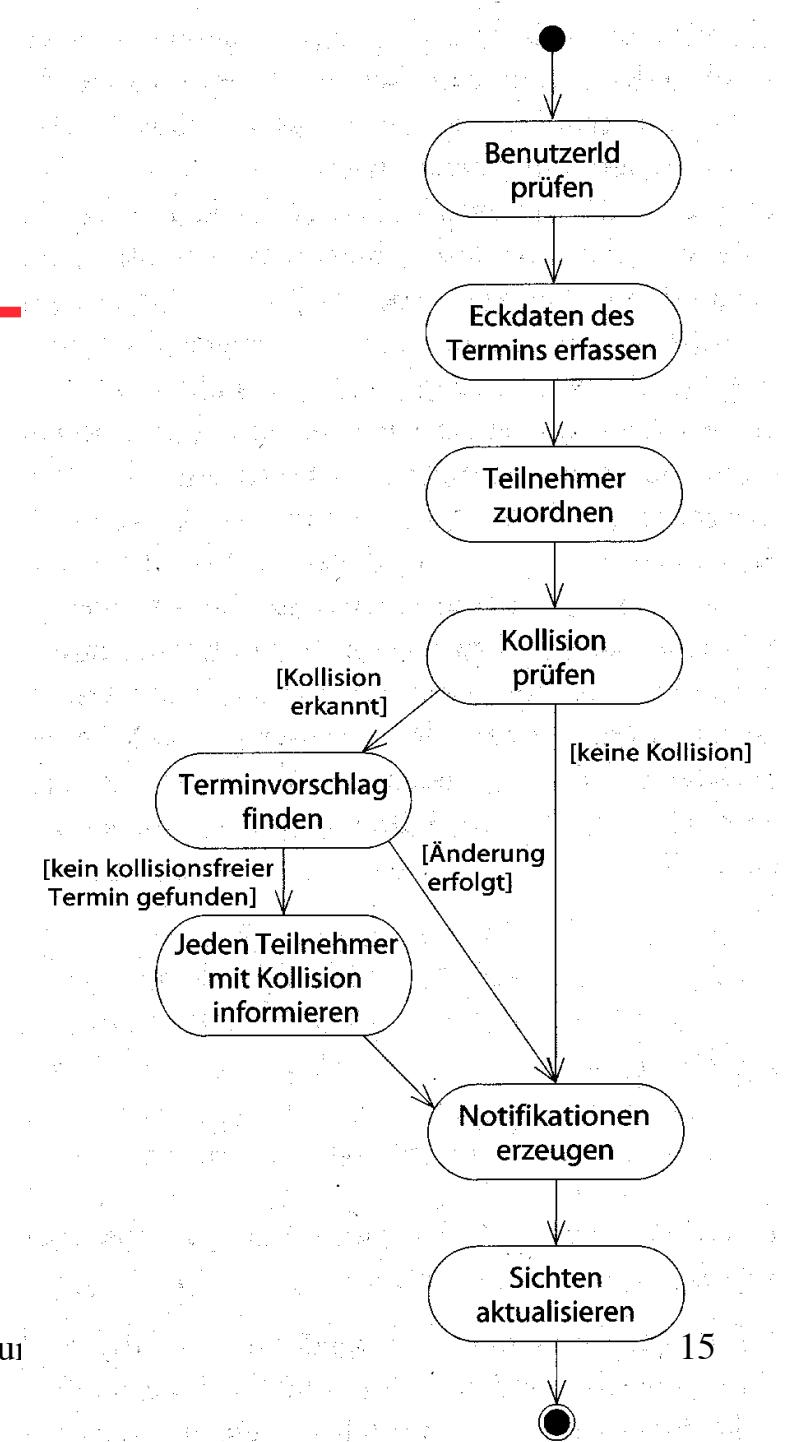
1.3 Aktivitätsdiagramm – Activity Diagram

Definition:

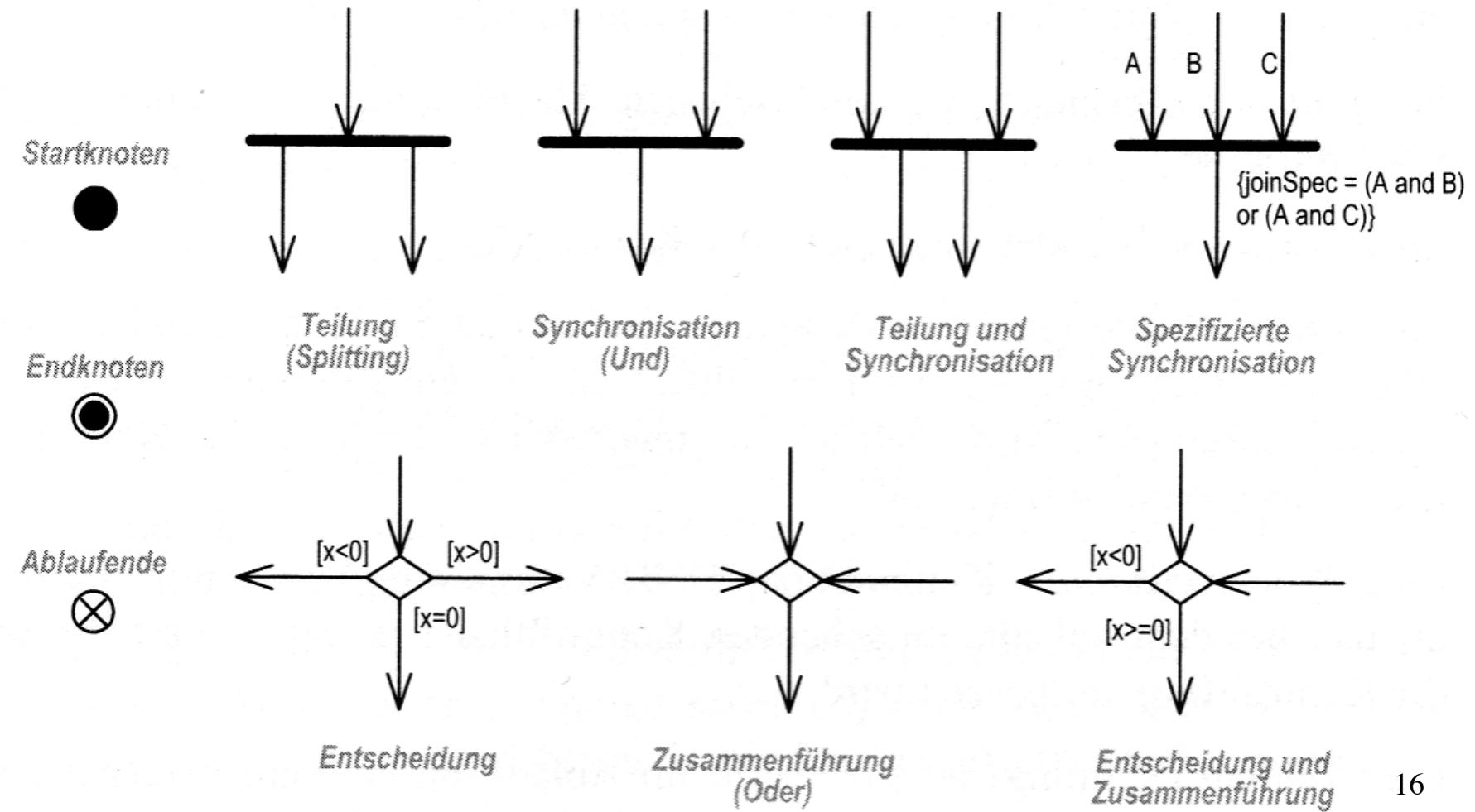
Eine Aktivität beschreibt einen Ablauf und wird definiert durch verschiedene Arten von Knoten, die durch Objekt- und Kontrollflüsse miteinander verbunden sind. Es werden Aktions-, Objekt- und Kontrollknoten unterschieden.

Aktivitätsdiagramm Beispiel

Ablauf des Use Case
„Termin erfassen“ aus dem
CALENDARIUM-Beispiel als
Aktivitätsdiagramm – 1. Version



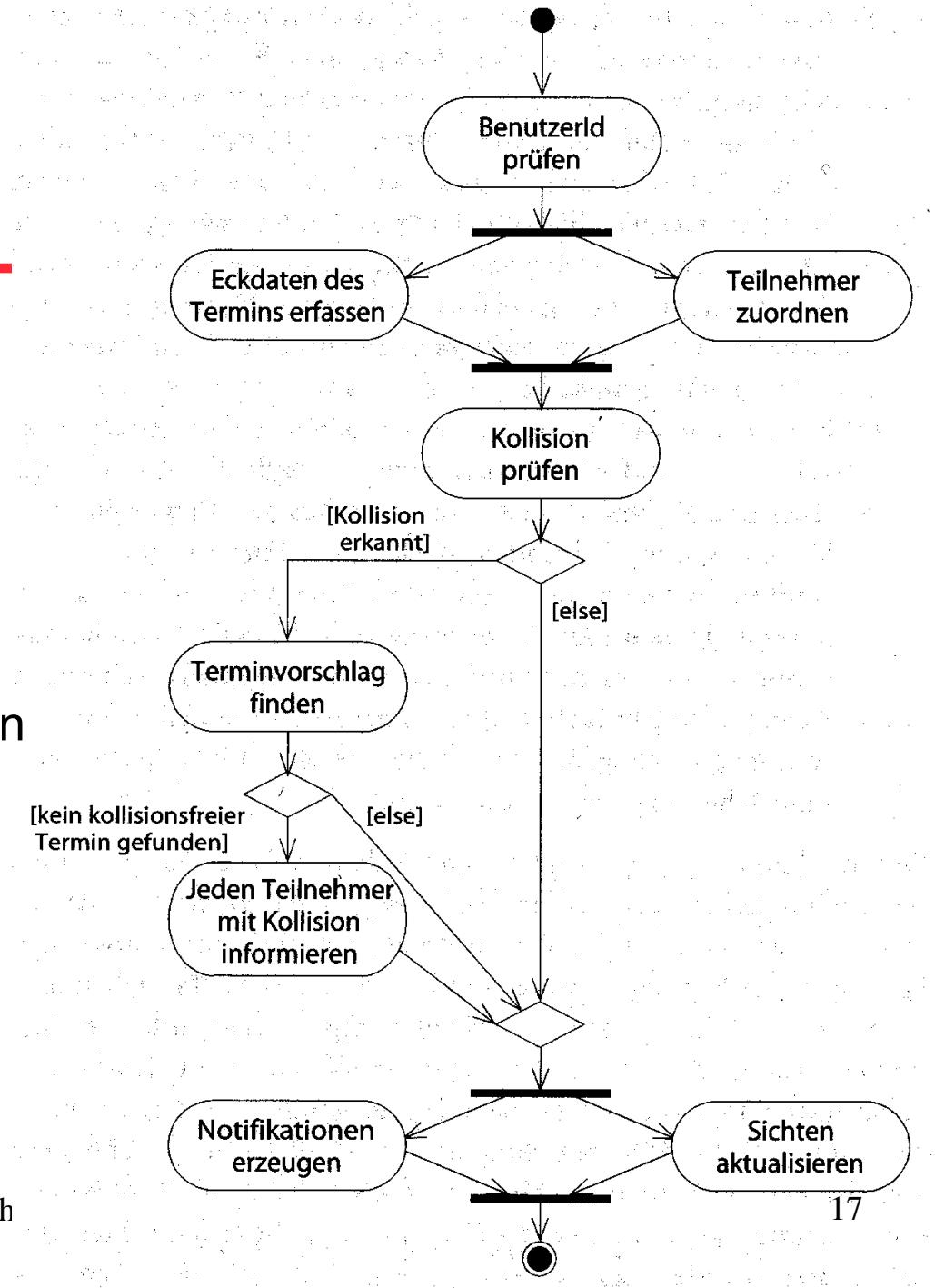
Kontrollknoten: Steuerfluss



Steuerfluss Beispiel

Aktivitätsdiagramm für
„Termin erfassen“ Version 2:

Steuerfluss mit Entscheidungen,
sowie Synchronisation zur
Darstellung von Nebenläufigkeiten

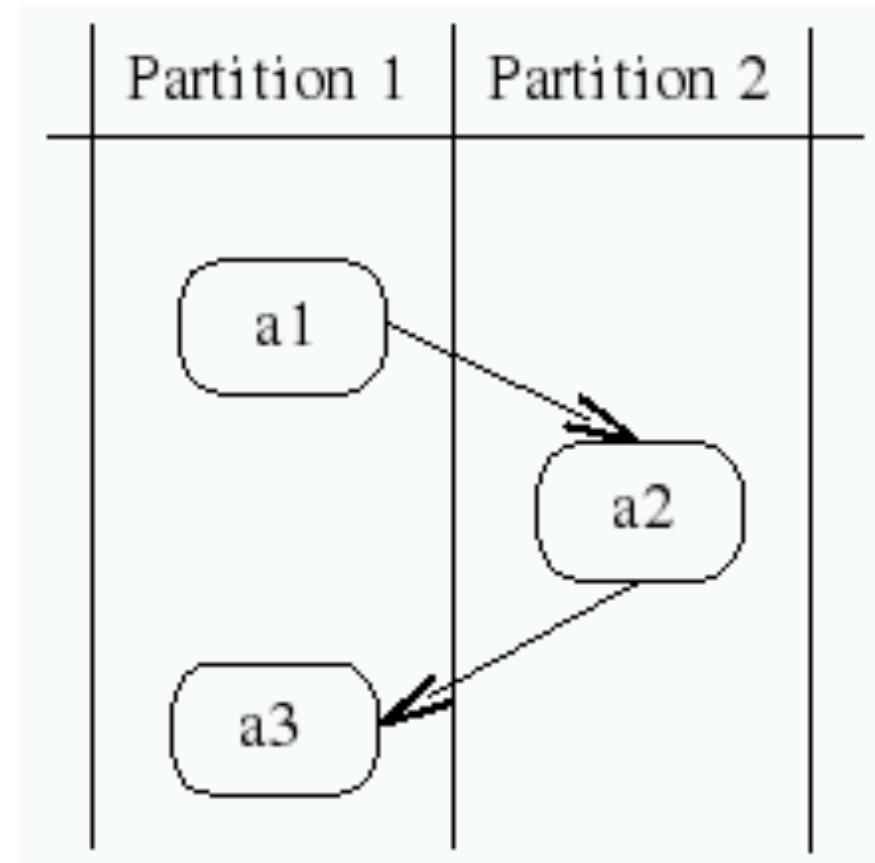


Verantwortlichkeitsbereiche: Partitionen

Definition:

Eine Partition

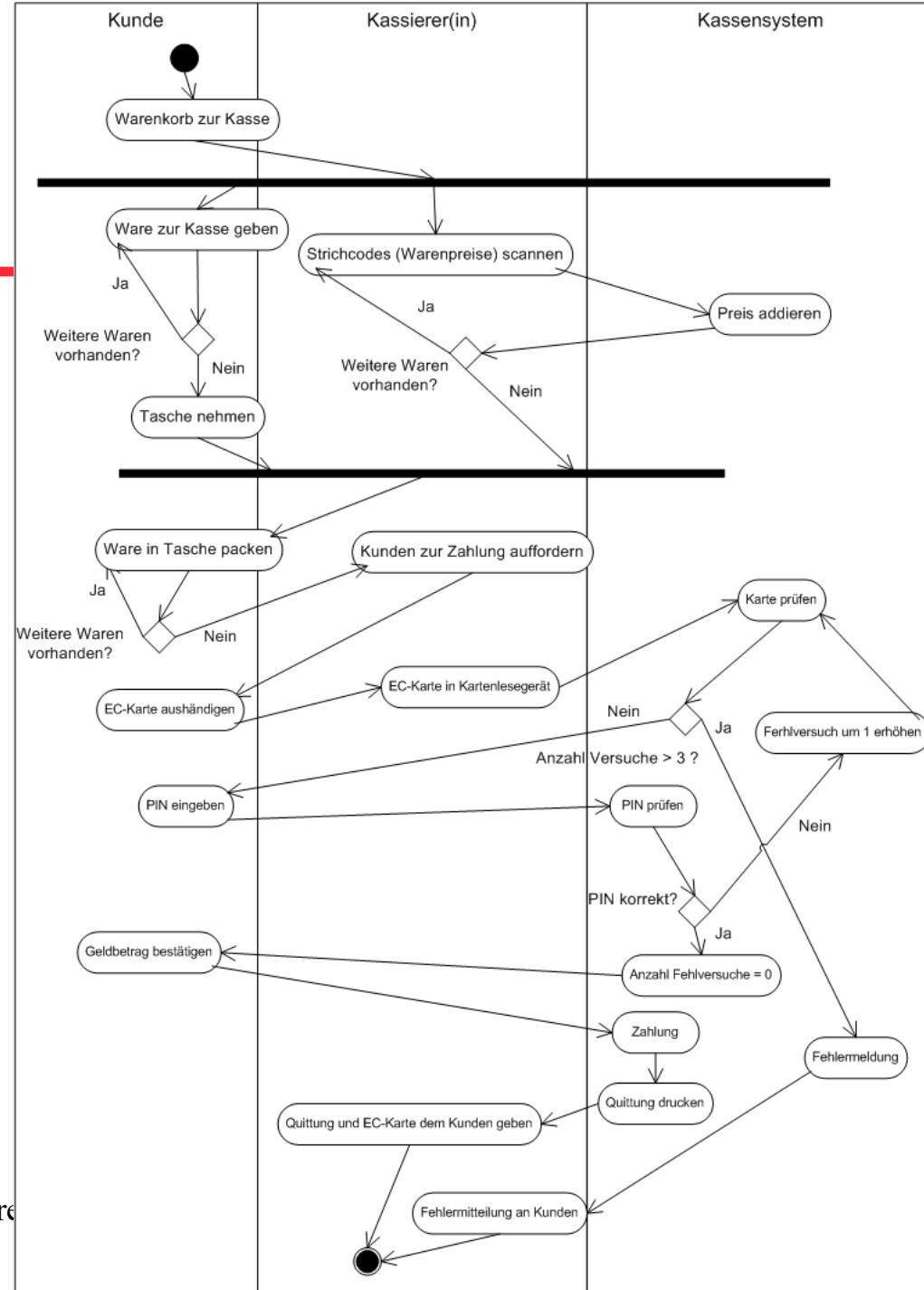
(Verantwortlichkeits- oder Funktionsbereich) beschreibt innerhalb des Aktivitätsmodells, wer oder was für einen Knoten verantwortlich ist oder welche gemeinsame Eigenschaft Knoten kennzeichnet.



Beispiel Kasse

Matthias Riebisch

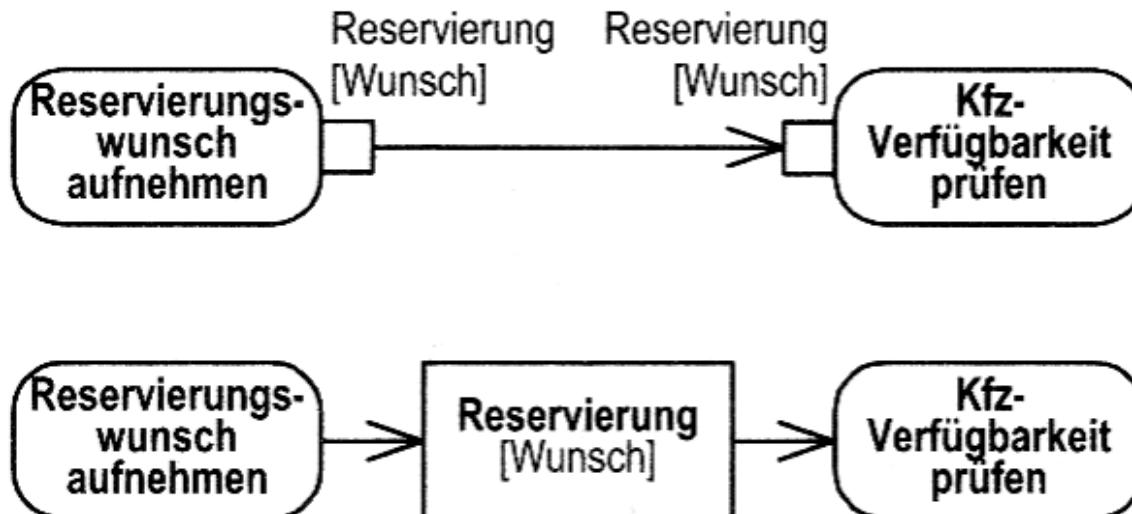
Software



Datenfluss: Objektknoten und Objektfluss

Ein Objektknoten gibt an, dass ein Objekt oder eine Menge davon existiert. Objektknoten können als ein- oder ausgehende Parameter in Aktivitäten verwendet werden.

Ein Objektfluss ist wie ein Kontrollfluss, bei dem jedoch Objekte transportiert werden



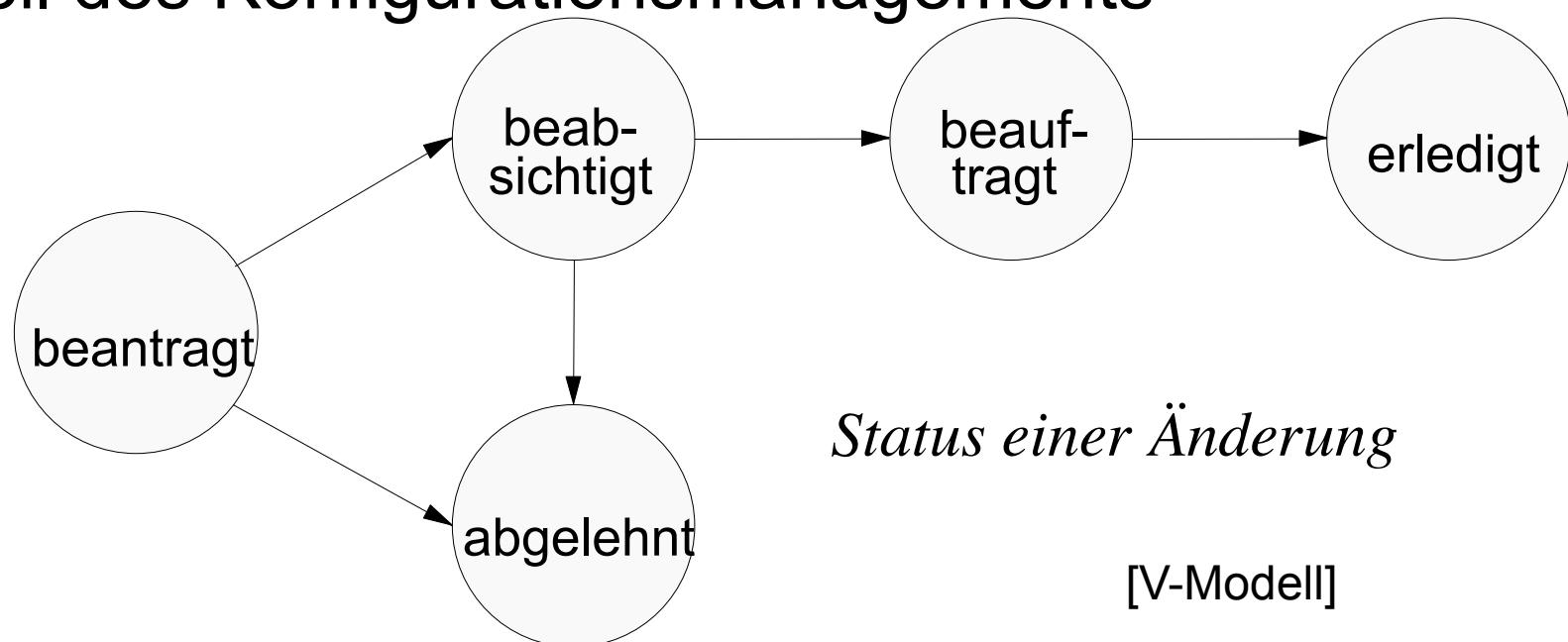
1. Anforderungsbeschreibung

Fragen #2

8. Wozu ist Änderungsmanagement erforderlich?
9. Nennen Sie Kriterien für die Prüfung einer Anforderungsbeschreibung!
10. Erläutern Sie die Unterscheidung zwischen Problembereichsmodell und Lösung!
11. Warum sollte eine Anforderungsbeschreibung keine Aussagen zur technischen Realisierung enthalten?
12. Welche Beziehungen werden im Klassendiagramm dargestellt?
13. Was ist eine Assoziation (Aggregation, Komposition)?

Änderungsmanagement

- Änderung (von Anforderungen) koordinieren
- Vor ungeplanten Änderungen schützen
- Teil des Konfigurationsmanagements



Prüfkriterien

Anforderungsbeschreibung

Eindeutigkeit - nur eine einzige Interpretation möglich?

Vollständigkeit - alle notwendigen Funktionen berücksichtigt?

Überprüfbarkeit - Erfüllung der Anforderungen überprüfbar?

Widerspruchsfreiheit - Konflikte zwischen den Anforderungen?

Verständlichkeit - für alle Beteiligten?

Herkunft - Herkunft/Begründung der Anforderung klar beschrieben?

Flexibilität und Abhängigkeiten - ohne Auswirkung auf andere Anforderungen änderbar?

Rückverfolgbarkeit - eindeutig zu identifizieren?

Abstrahierbarkeit - Ist die Anforderung implementierungsunabhängig?

Klassifizierbarkeit bezüglich Bedeutung - Risiken bezüglich Realisierbarkeit; Stabilität über gesamten Lebenszyklus?

Angemessenheit - Wünsche und Bedürfnisse des Benutzers abgedeckt?.

1.4 Problembereichsmodellierung

Konzeptionellen Beschreibung („was“) statt Darstellung der Realisierung („wie“):

Struktur: Klassendiagramm (Komponentendiagramm)

Verhalten: Zustands-, Kommunikations-, Aktivitätsdiagramm

Elemente der Strukturmodelle:

Klasse, Objekt, Attribut, Methode, Paket, (Komponente)

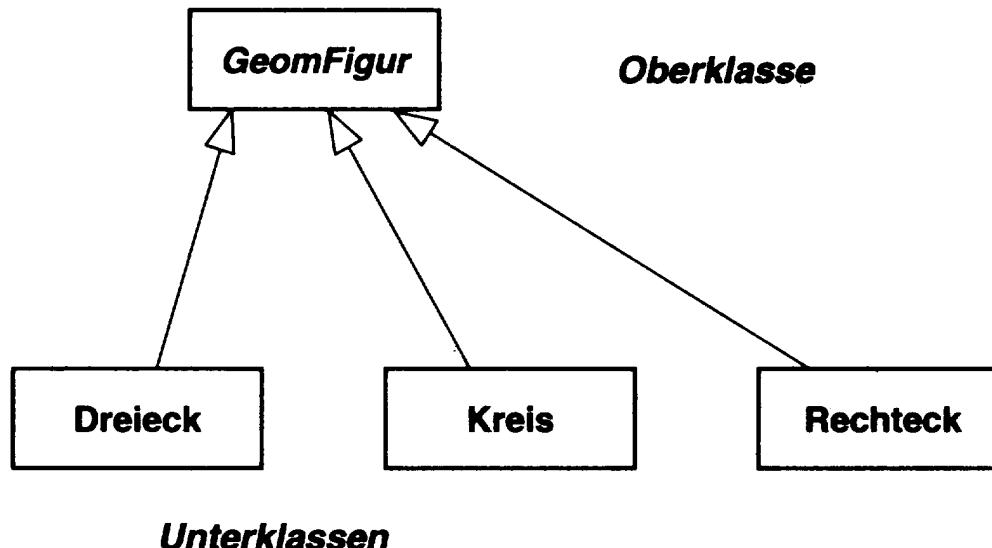
Beziehungen:

Generalisierung (Vererbung), Assoziation, Rolle,
Aggregation, Komposition, Abhängigkeit

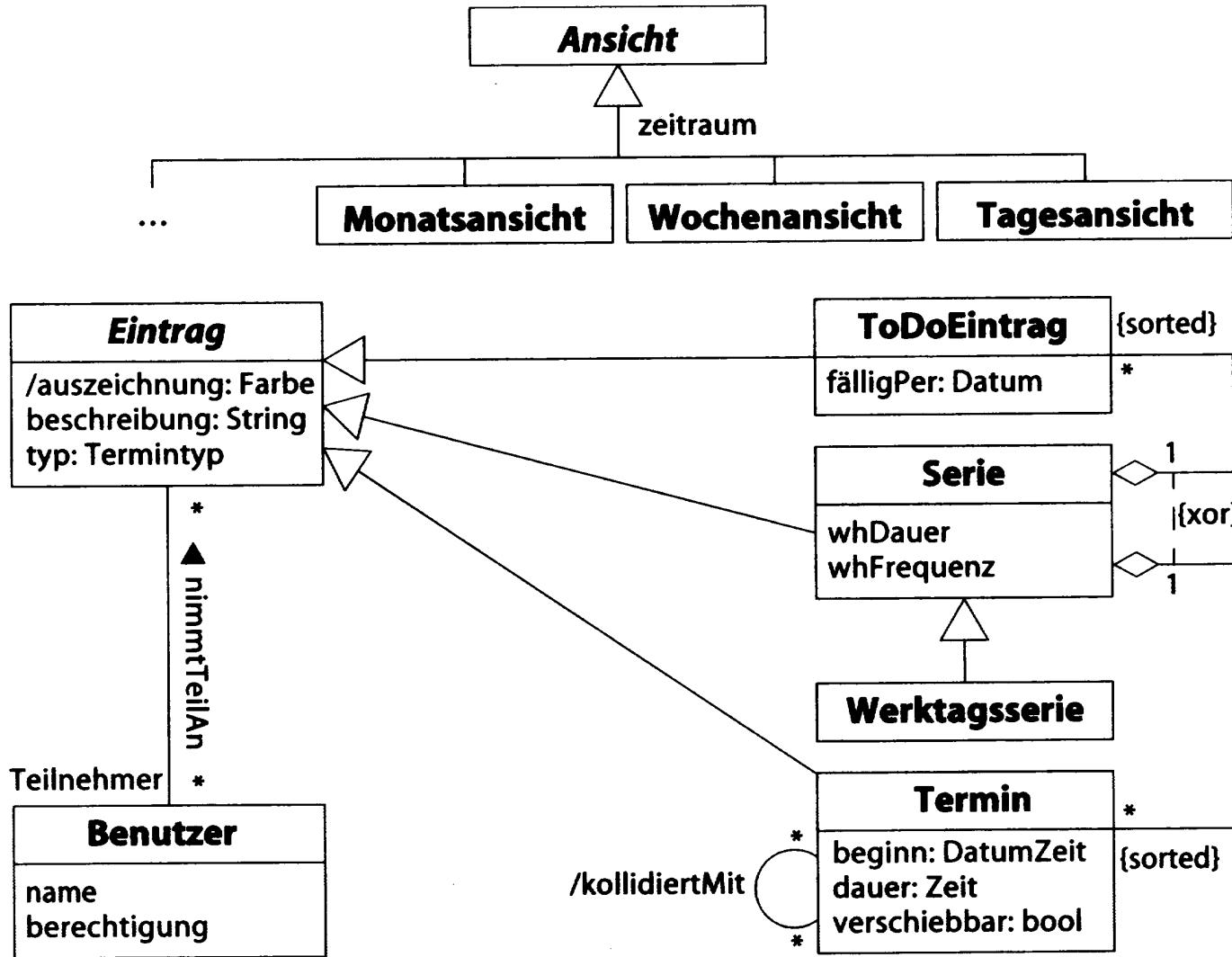
Generalisierung

Generalisierung als Beziehung zwischen Klassen wird wie Taxonomie und Vererbung genutzt, um Eigenschaften zusammenzufassen und zu strukturieren. Dadurch wird erreicht, dass das Beschreiben bzw. Ändern einer Beschreibung der Eigenschaften weniger Aufwand erfordert, weil Redundanzen verringert werden.

Durch Vererbung werden Eigenschaften auf untergeordnete Klassen übertragen, dabei können sie von diesen ergänzt und verändert werden (Spezialisierung).

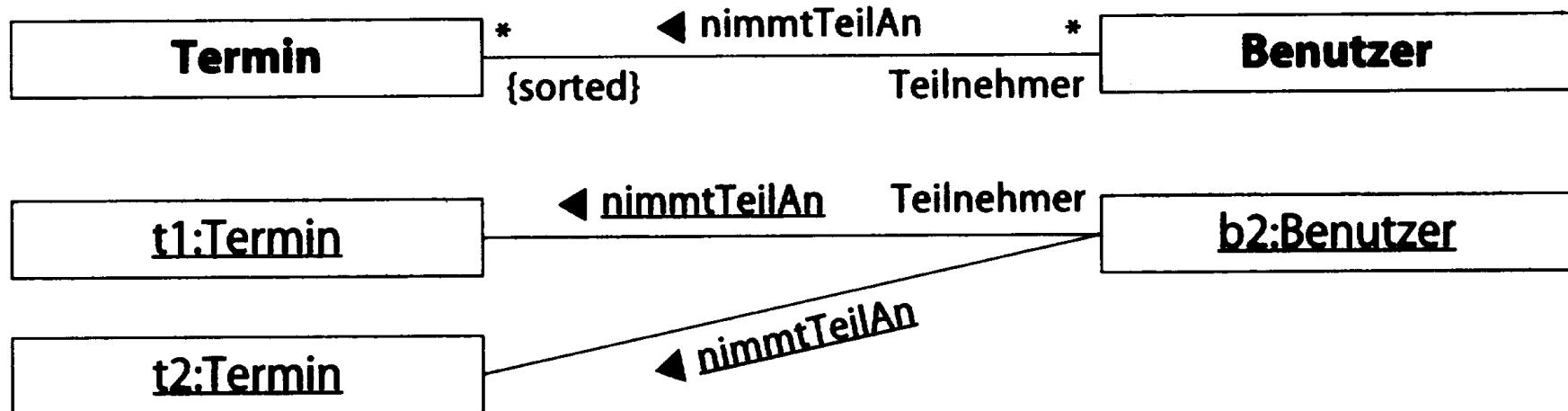


Generalisierung am Beispiel



Abstraktion: Beziehungen zwischen Klassen

Verallgemeinert: Klassenbeziehung



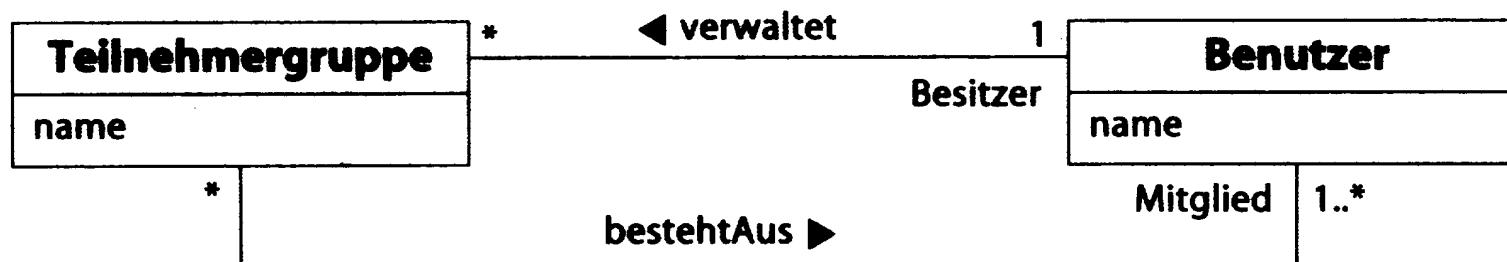
Beziehungen zwischen Objekten

Assoziation

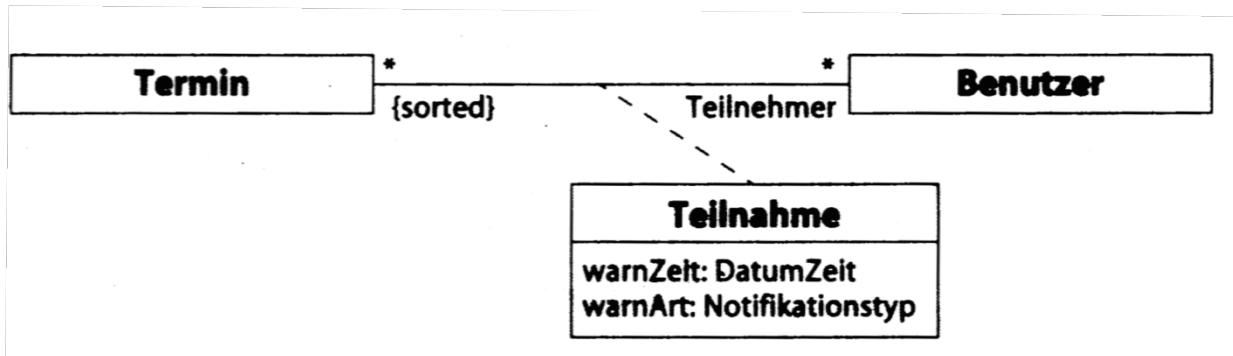
Die Assoziation stellt eine statische Beziehung zwischen Objekten dar.

Sie wird beschrieben durch:

- Name (Semantik) und Richtung
- Vielfachheit: Anzahl der Partnerobjekte
- Evtl. Rolle beteiligter Objekte

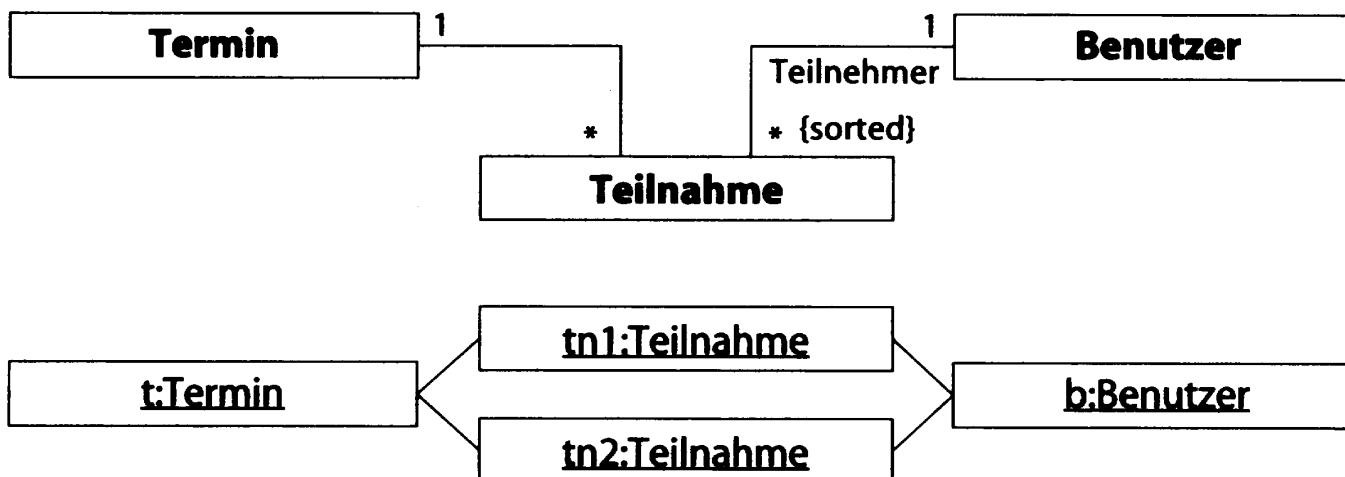


Assoziation: weitere Informationen



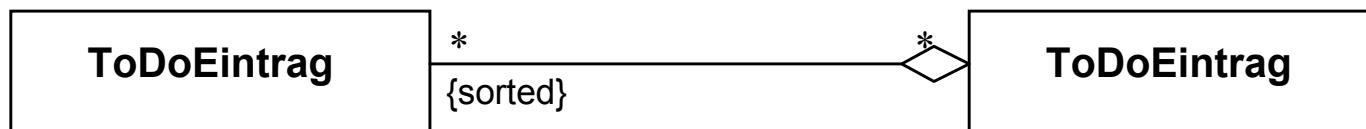
Assoziationsklasse beschreibt Assoziation genauer

Auflösung der Assoziationsklasse, dadurch „legale“ Beziehungen:



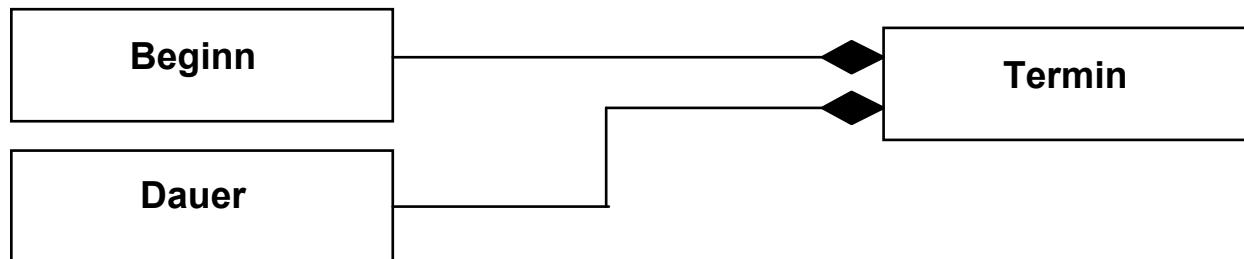
Aggregation und Komposition

Die Aggregation (part-of, Teil-Ganzes-Beziehung) stellt eine besondere Assoziation dar, bei der ein zusammengesetztes Objekt in Beziehung zu seinen Teilen dargestellt wird. Es handelt sich um eine asymmetrische Beziehung zwischen nicht gleichwertigen Partnern.



Die Komposition stellt eine spezielle Form der Aggregation dar.

- Ein Teil darf nur zu einem Ganzen gehören
- Ein Teil existiert nur, solange sein Ganzes existiert



1. Anforderungsbeschreibung

Fragen #3

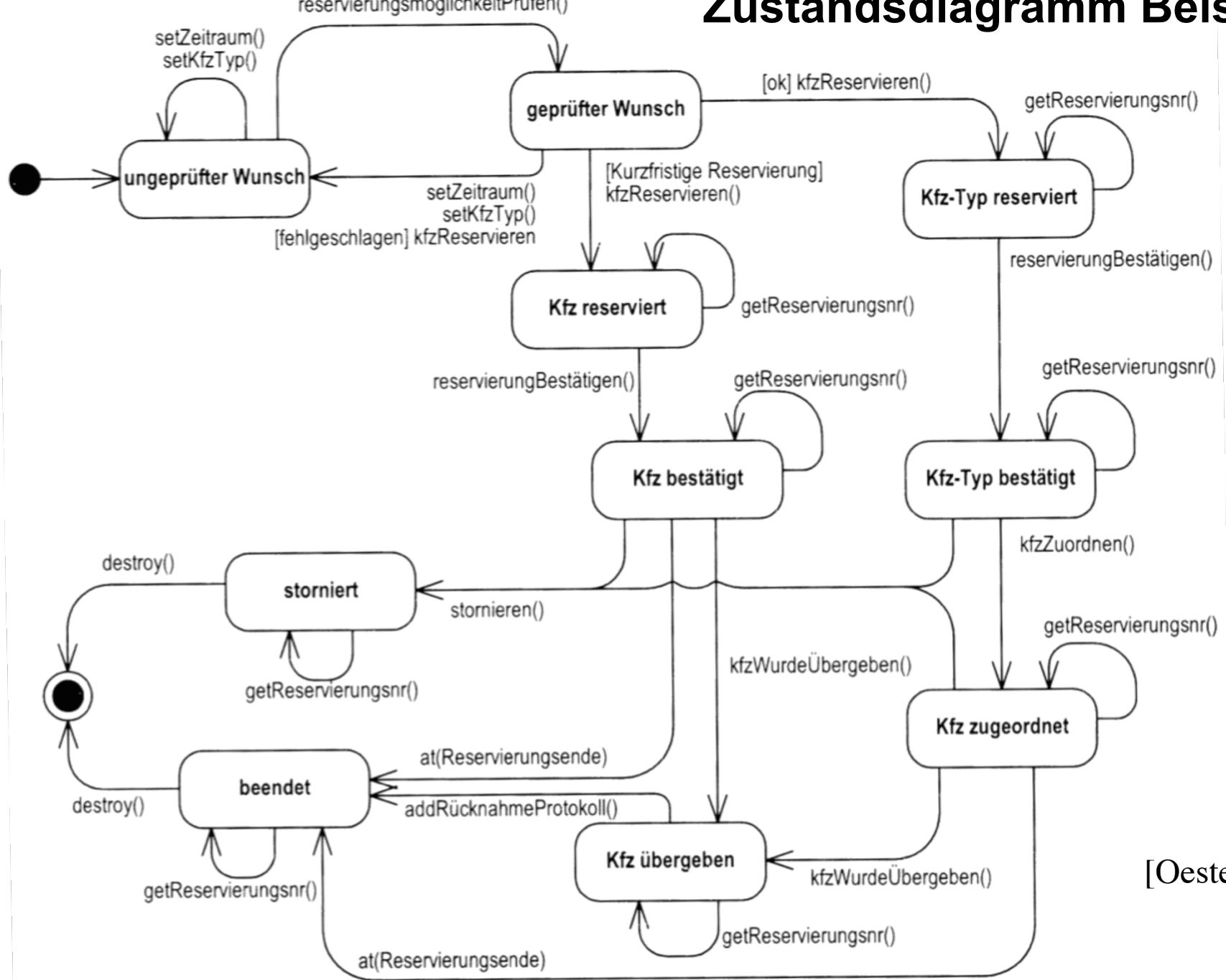
14. Was ist ein Zustandsdiagramm? (Definition)
15. Welche Elemente werden im Zustandsdiagramm der UML dargestellt?
16. Für die Darstellung welcher Aspekte sollte ein Zustandsdiagramm verwendet werden im Unterschied zu Aktivitäts-, Sequenz- und Kommunikationsdiagramm?
17. Wozu werden Unterzustände benötigt?

1.6 Zustandsdiagramm – State (Transition) Diagram

Beschreibt eine Folge von Zuständen, die ein Objekt im Laufe seines Lebenszyklus einnehmen kann, sowie Stimuli und davon bewirkte Zustandsänderungen

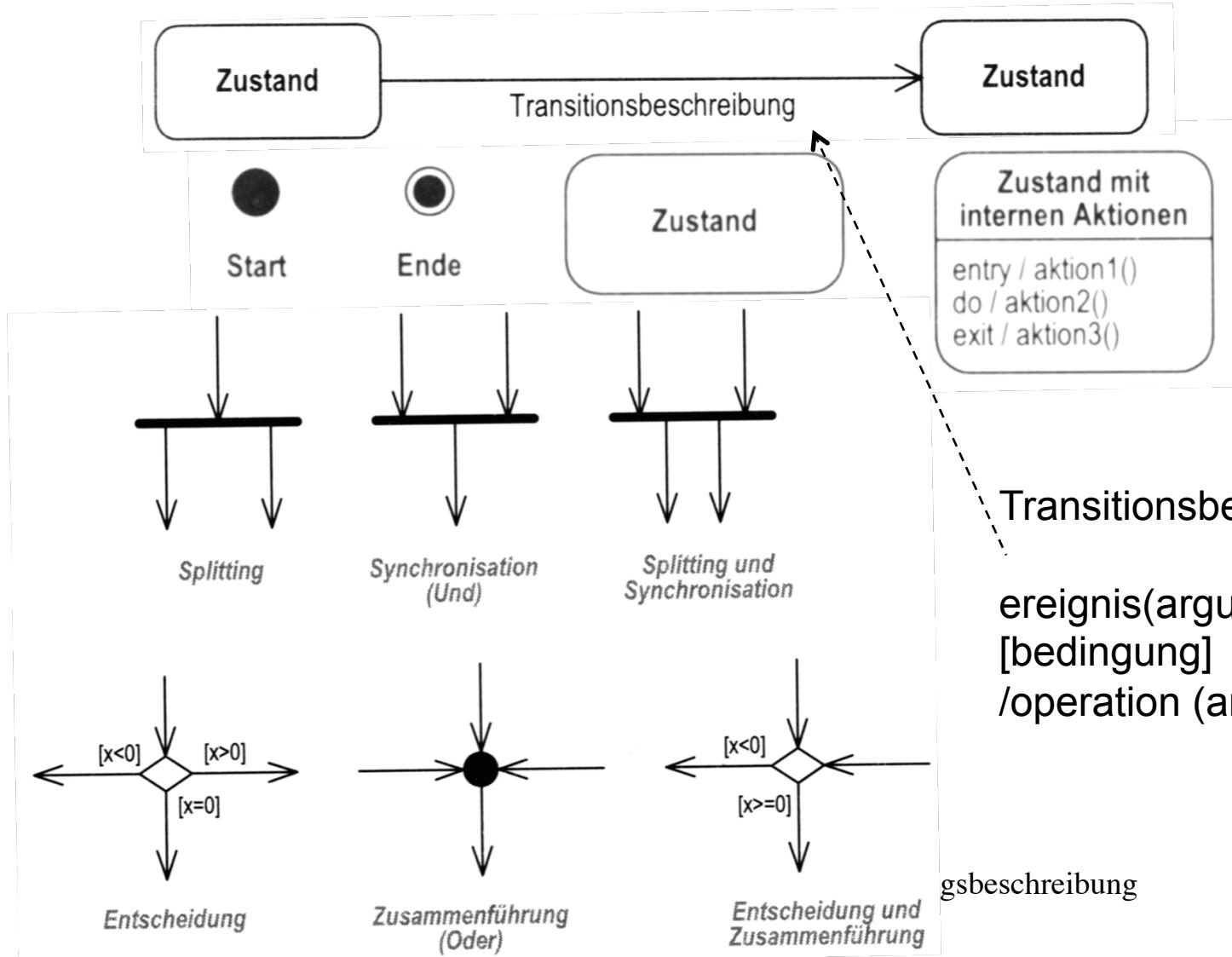
- Hypothetische Maschine – endlicher Automat
 - Endliche Menge von Zuständen
 - Endliche Menge von Ereignissen
 - Zustandsübergänge
 - Anfangszustand
 - Menge von Endzuständen

Zustandsdiagramm Beispiel



[Oestreich]

Zustandsdiagramm Elemente

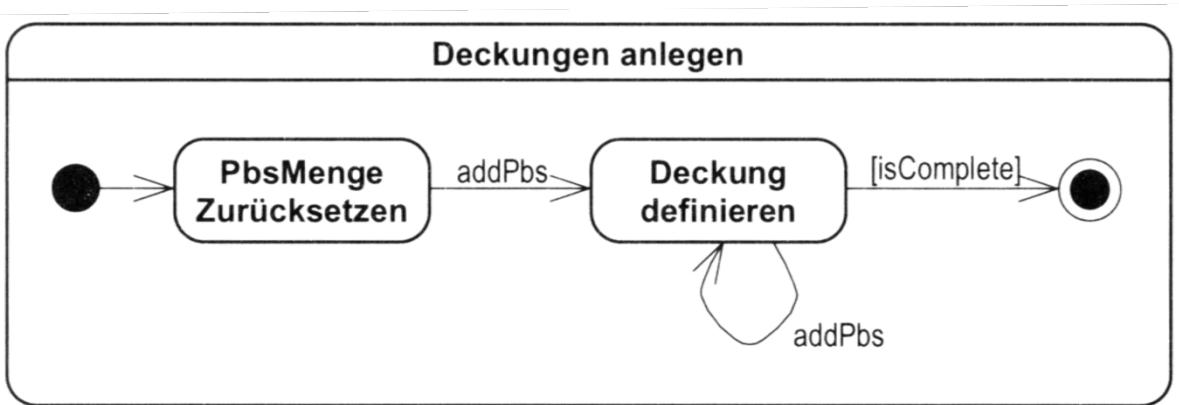


Transitionsbeschreibung:

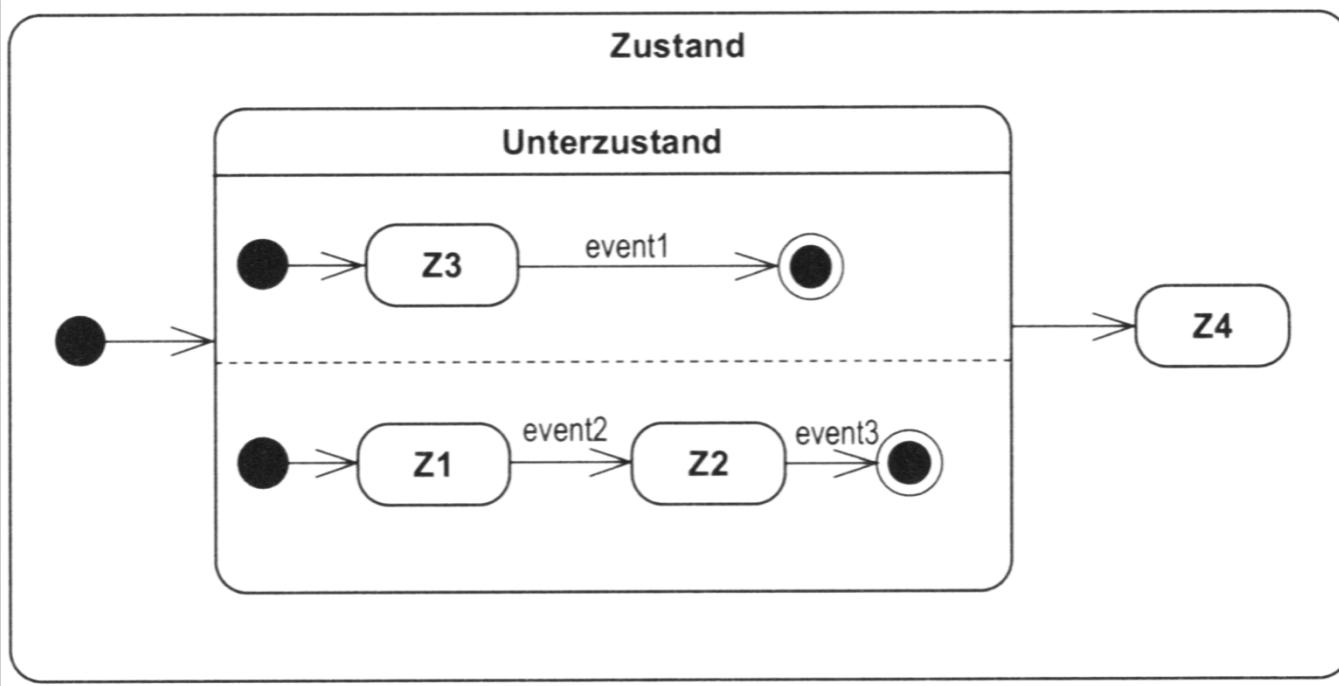
ereignis(argumente)
[bedingung]
/operation (argumente)

[Oestreich]

Hierarchische Gliederung - Unterzustände



Sequentielle Unterzustände am Beispiel

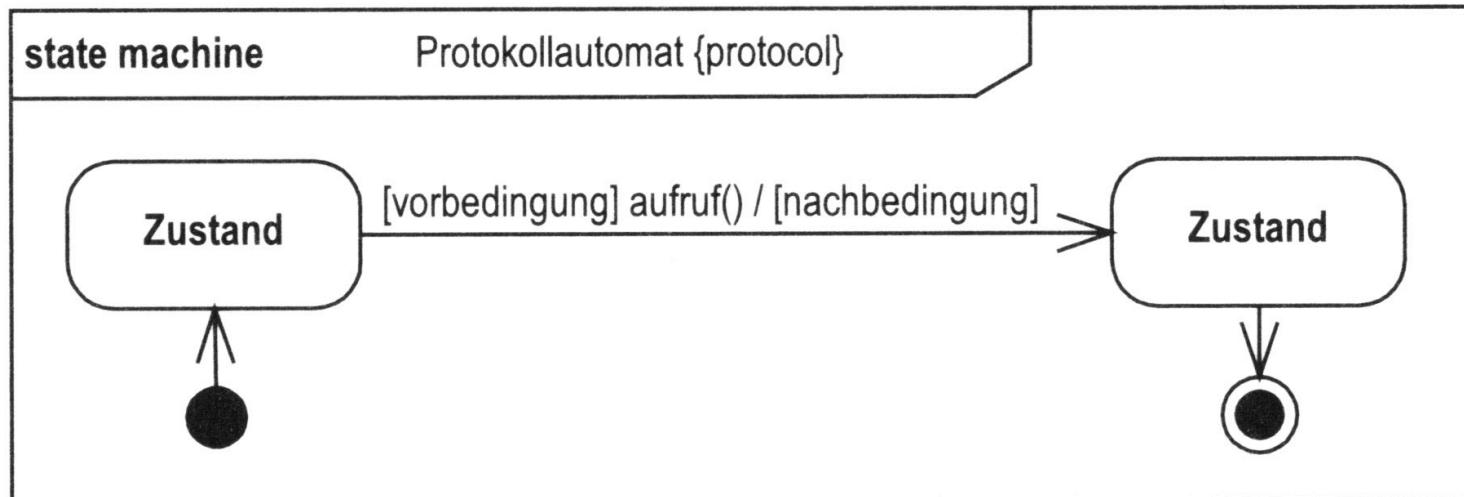


Parallele Unterzustände

[Oestereich]

Protokoll-Zustandsautomat - Protocol State Machine

Spezielle Form des Zustandsdiagramms, beschreibt lediglich die möglichen und verarbeitbaren Ereignisse ohne weitergehendes Verhalten



[Oestereich]

2. Entwurf

Fragen #1

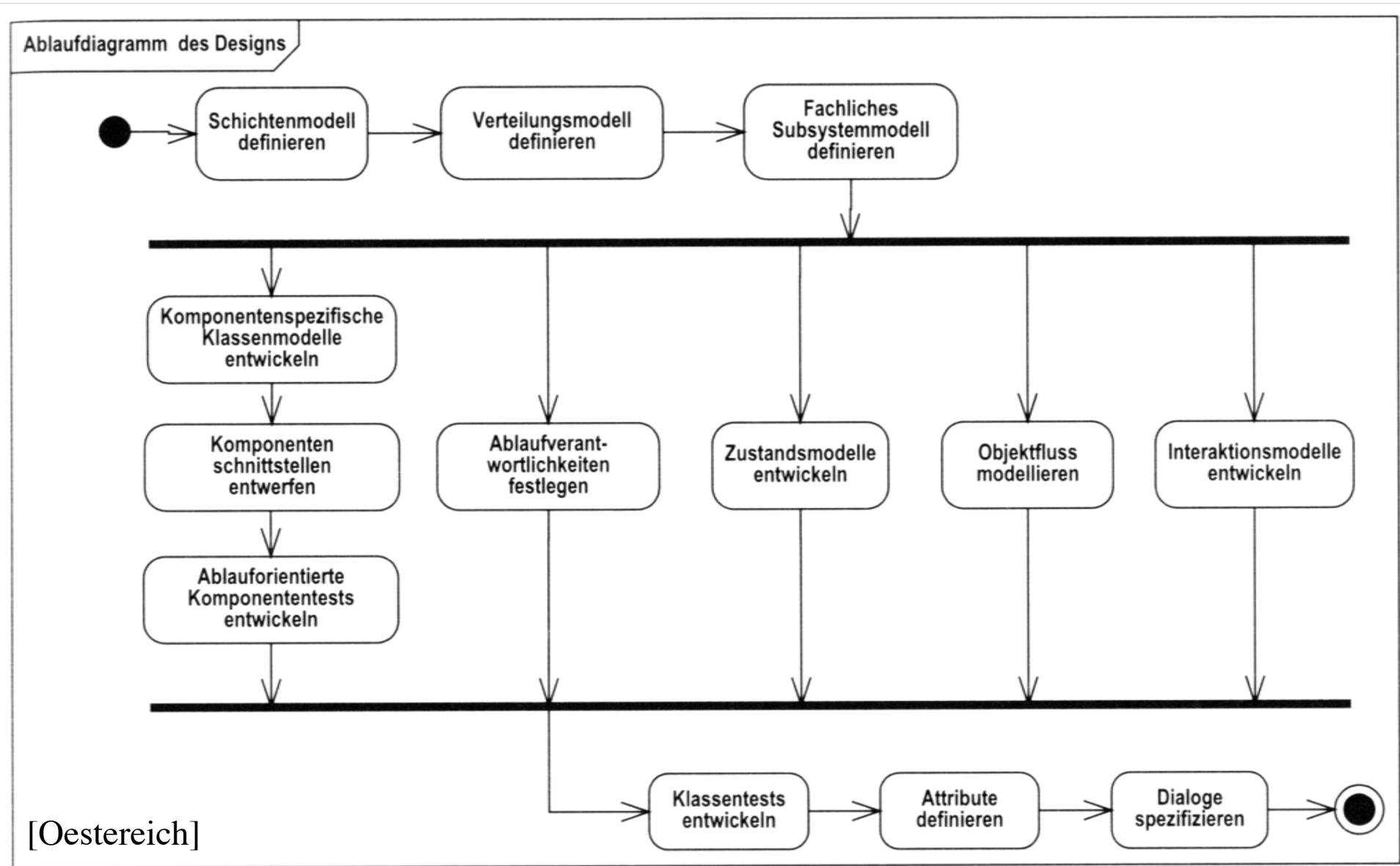
1. Nennen Sie Ziele, Techniken und Ergebnis der Phase Entwurf!
2. Nennen und erläutern Sie die drei Klassen, die nach der Jacobson-Methode beim Entwurf für jede fachliche Klasse eingeführt werden!
3. Wodurch unterscheiden sich die Realisierungen von Assoziationen mit Vielfachheit 1 und Vielfachheit * ?

Aufgaben Entwurf

Überführen der Anforderungen in technische Lösung

- Erfüllung funktionaler Anforderungen
 - Workflows und Aktionen, Datenstrukturen, GUI
- Erfüllung von Randbedingungen
 - Plattform, Schnittstellen zu anderen System
- Erfüllung nicht-funktionaler Anforderungen
 - Zeitverhalten
 - Skalierbarkeit
 - Robustheit
 - Änderbarkeit
 - Sicherheit

Ablauf Entwurf - Überblick



Ziele Entwurf und Architekturentwicklung

- Effiziente Entwicklung
 - Iterativ, inkrementelle Entwicklung zulassen
 - Grundlage der Projektplanung und Management: Organisation, aktive Führung, Einblick, Verhandlungsbasis
 - unabhängige, verteilte Implementierung
- Risiken minimieren
 - Reihenfolge nach Risiken
 - Randbedingungen früh berücksichtigen
 - Anforderungserfüllung früh prüfen
- Verständnis schaffen
 - Kommunikation zwischen Stakeholdern: Forderungen, Sichten
 - Basis für Schulungen
 - Leitbild und Referenz: Dokumentation aus verschiedenen Perspektiven
- Kernwissen des Systems konservieren
 - Übertragbares Modell: für ähnliche Systeme, Wiederverwendung
 - KnowHow, geistiges Eigentum
 - Verbesserung ermöglichen

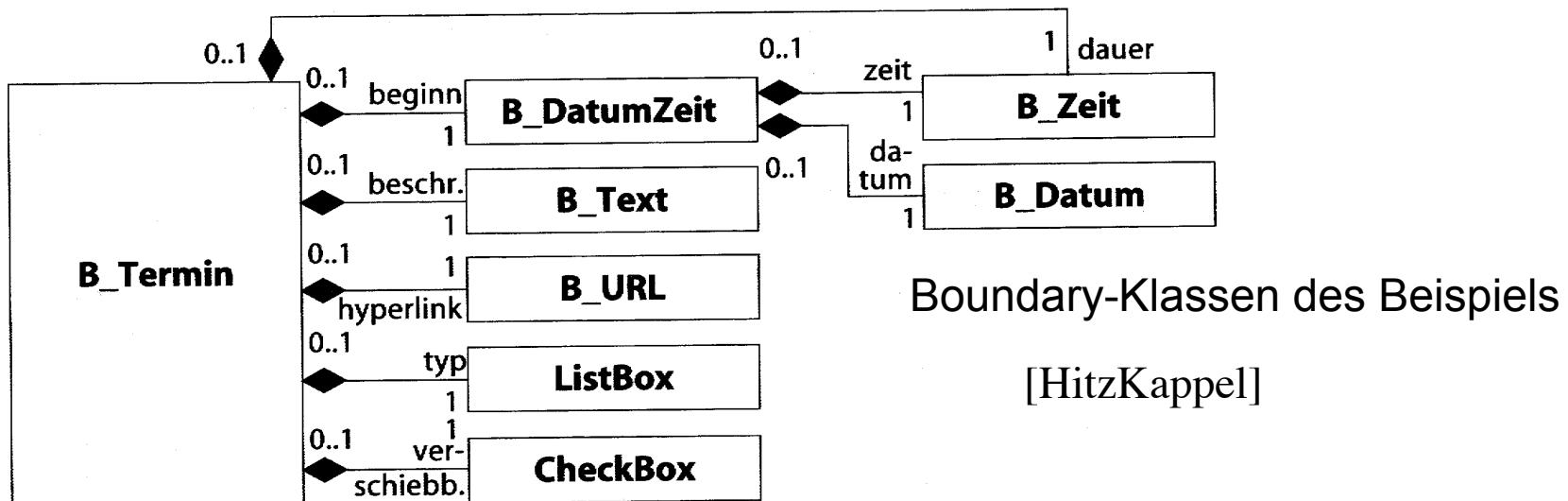
Übergang Problemreichsmodell → Entwurf

- Entwurf nach Jacobson: Systemfunktionalität nach Dimensionen organisieren

Information: Informationsobjekte (entity object)

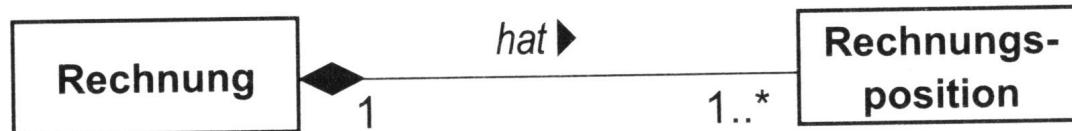
Darstellung: Schnittstellenobjekte (boundary object)

Verhalten: Steuerungsobjekte (control object)



Realisierung von Vielfachheiten

- 1 - Attribut mit Referenz (Typ des verbundenen Objekts)
- * - Attribut mit Liste von Referenzen (Collection-Typ)



Rechnung
kunde
positionen

Rechnungsposition
rechnung
artikel
preis

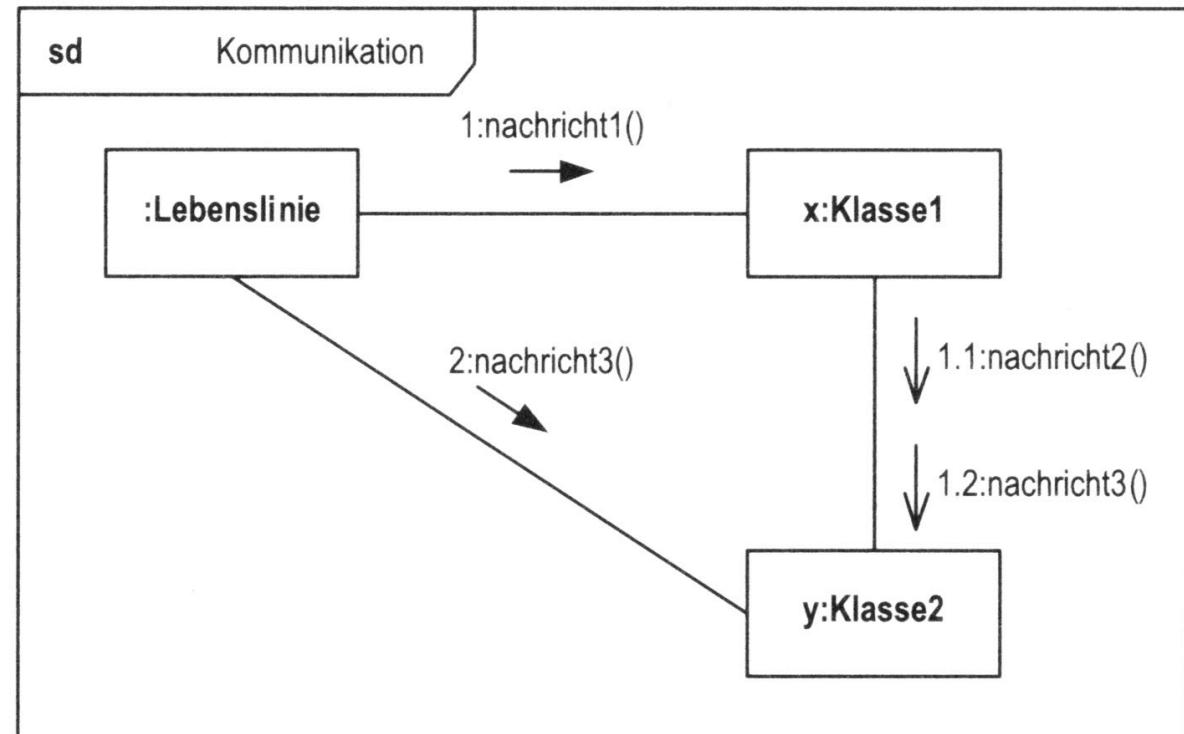
2. Entwurf

Fragen #2

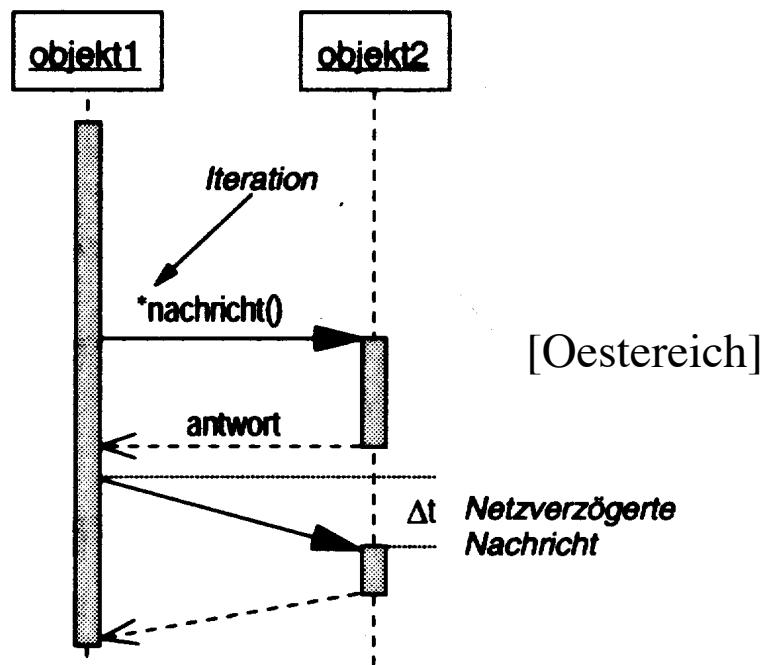
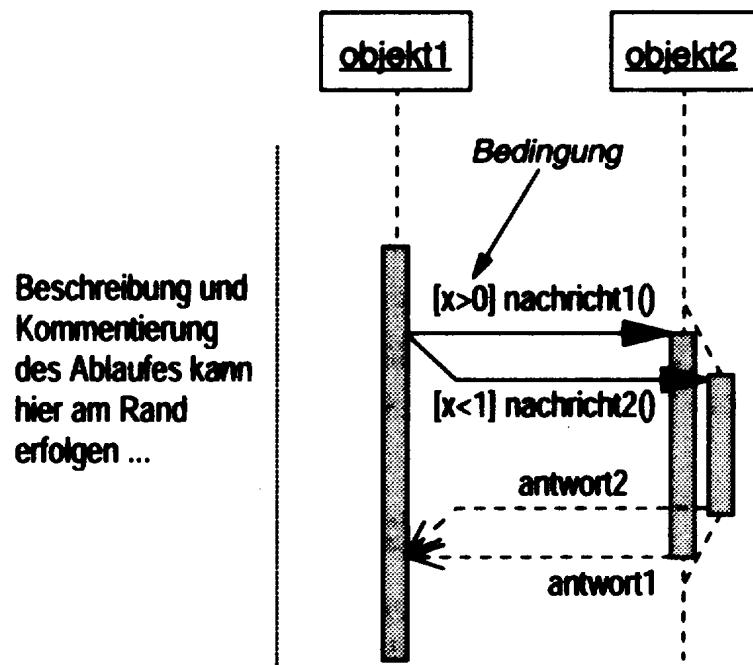
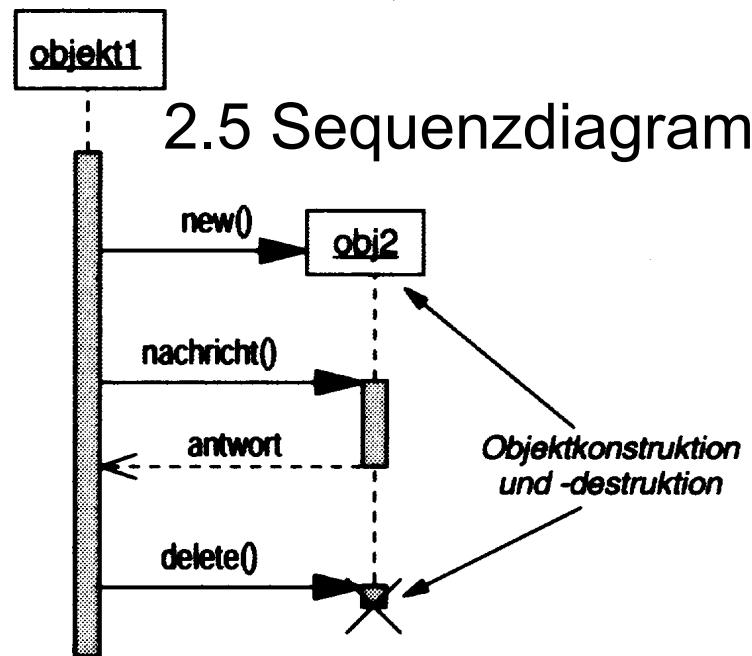
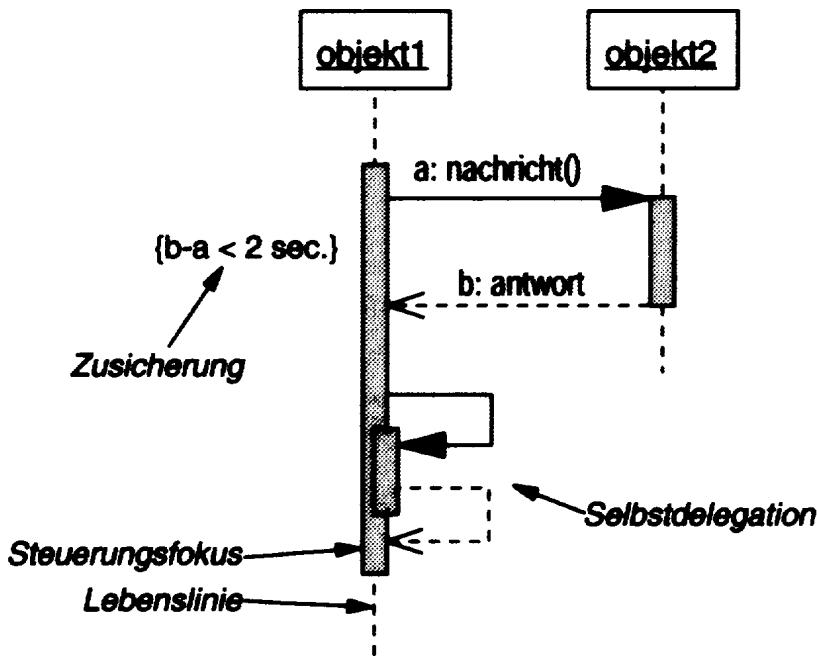
4. Wie wird im Sequenzdiagramm die Zeitachse dargestellt?
5. Welche Möglichkeiten der hierarchischen Verfeinerung sind im Sequenzdiagramm vorgesehen?
6. Welche Informationen eines Sequenzdiagramms werden in einem Klassendiagramm ebenfalls sichtbar, und wie?
7. Stellen Sie Sequenz- und Kommunikationsdiagramm gegenüber bezüglich Darstellung von Beziehungen und Zeitachse sowie Anwendungsbereich!

2.4 Kommunikationsdiagramm

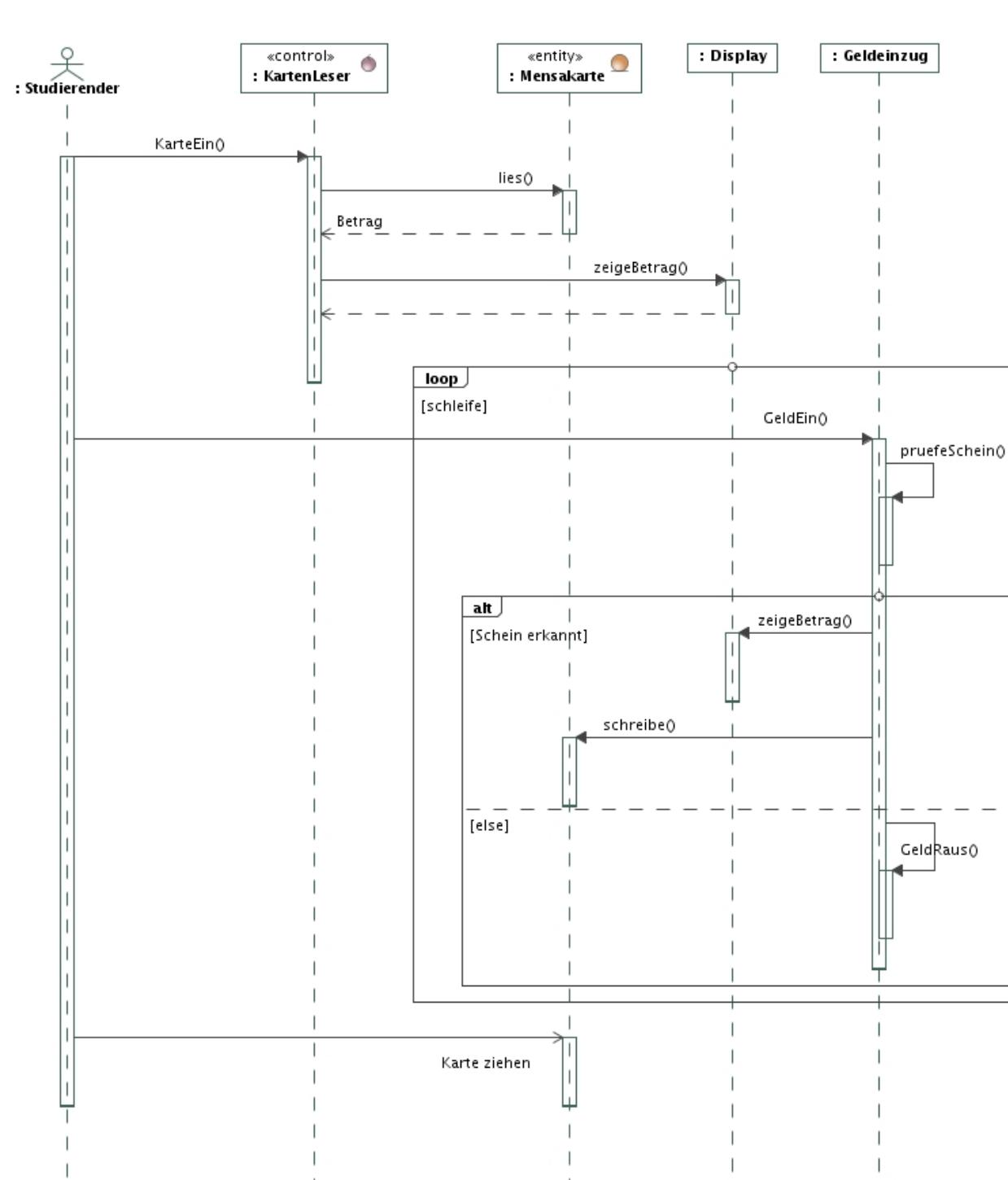
- Reihenfolge durch Zahlen
- Focus: Objektbeziehungen



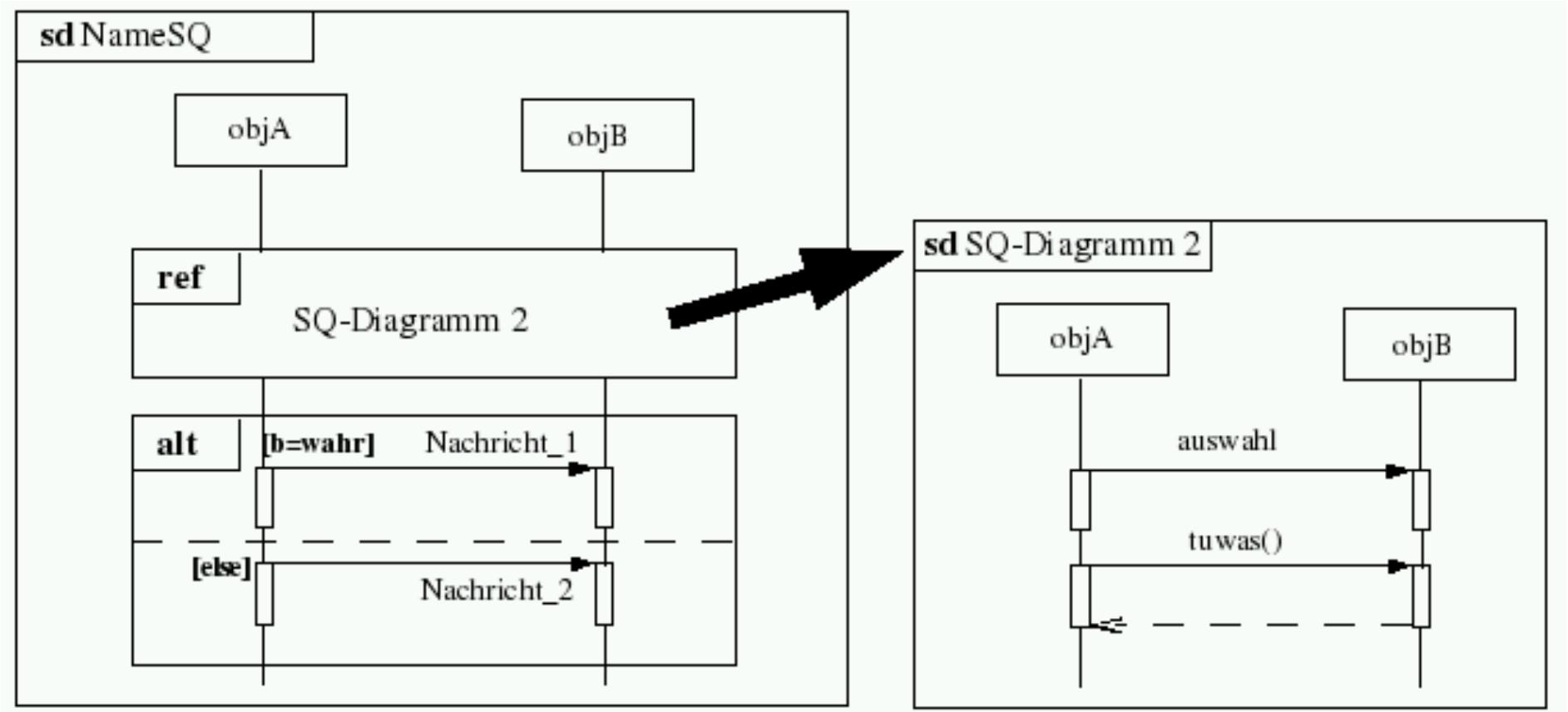
[Oestereich]



Sequenzdiagramm mit Schleife und Alternative



Sequenzdiagramm mit Verfeinerung



2. Entwurf

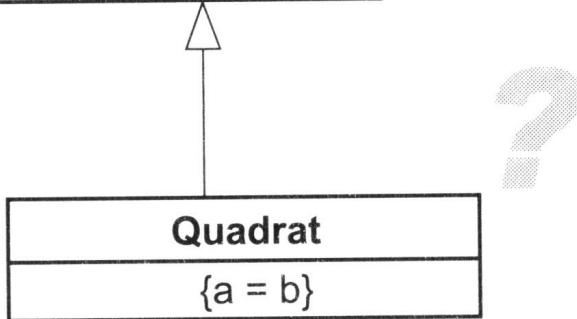
Fragen #3

8. Nennen Sie Nachteile der Vererbung als Mittel der Realisierung!
9. Warum sollte Delegation gegenüber Vererbung bevorzugt werden?
10. Worin liegen die Nachteile der Benutzung von Referenzen als Parameter?
11. Erläutern Sie Möglichkeiten der Entkopplung zwischen Komponenten!

Spezialisierung – Versuch 1

Quadrat als
spezielles Rechteck
durch Vererbung abgeleitet:

Rechteck
a { $a > 0$ }
b { $b > 0$ }
setA(neuA)
setB(neuB)
anzeigen()
entfernen()



Zusicherungs-Verantwortlichkeitsprinzip:
Eine Unterklasse sollte keine
Zusicherungen auf
Eigenschaften einer Oberklasse
machen

[Oestereich]

Spezialisierung – Versuch 2

Umgekehrte Vererbungsbeziehung:

Rechteck ist ein Quadrat, jedoch mit einer zweiten Kante b

→ möglich,
aber unverständliche Zuweisungen erlaubt:

```
class Rechteck extends Quadrat { ... }
Rechteck r;
Quadrat q;
...
q = r; // Zuweisung zulässig, da typkompatibel,
       // jedoch nicht sinnvoll
```

Spezialisierung – Versuch 3

Vererbung nicht verwendet
Quadrat als Spezialfall
in Attribut modelliert

Rechteck
a {a > 0}
b {b > 0}
setA(neuA)
setB(neuB)
anzeigen()
entfernen()
istQuadrat():Boolean

[Oestreich]

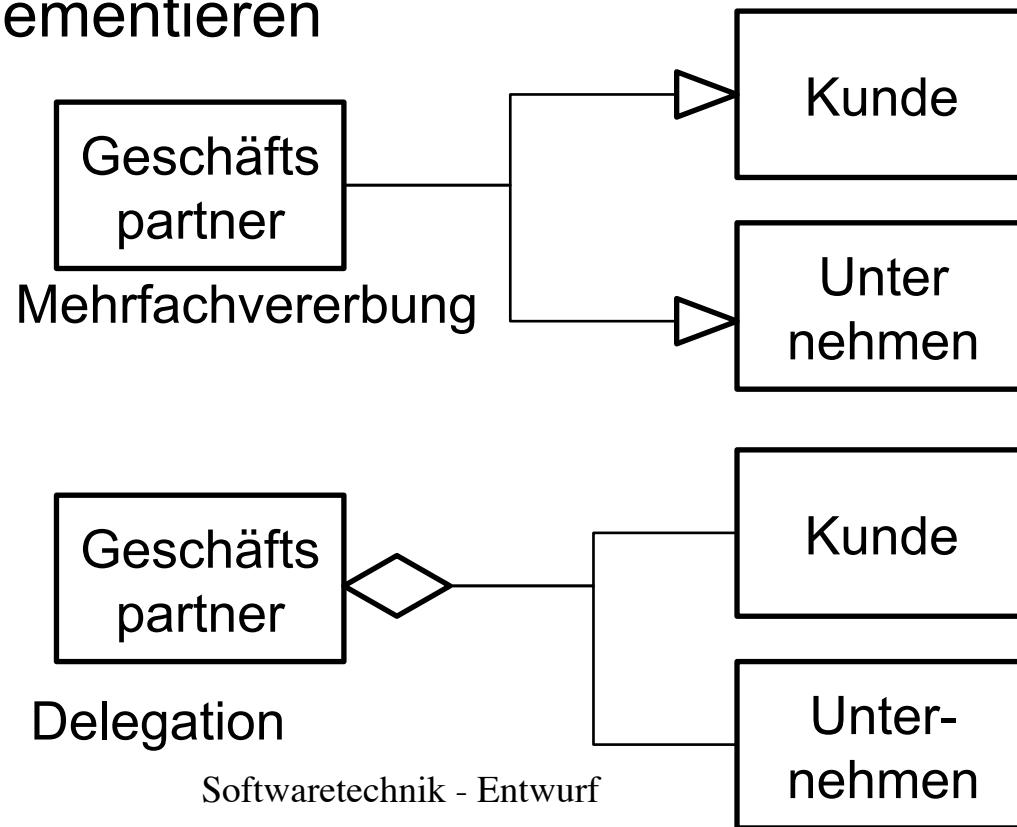
Matthias Riebisch

Softwaretechnik - Entwurf

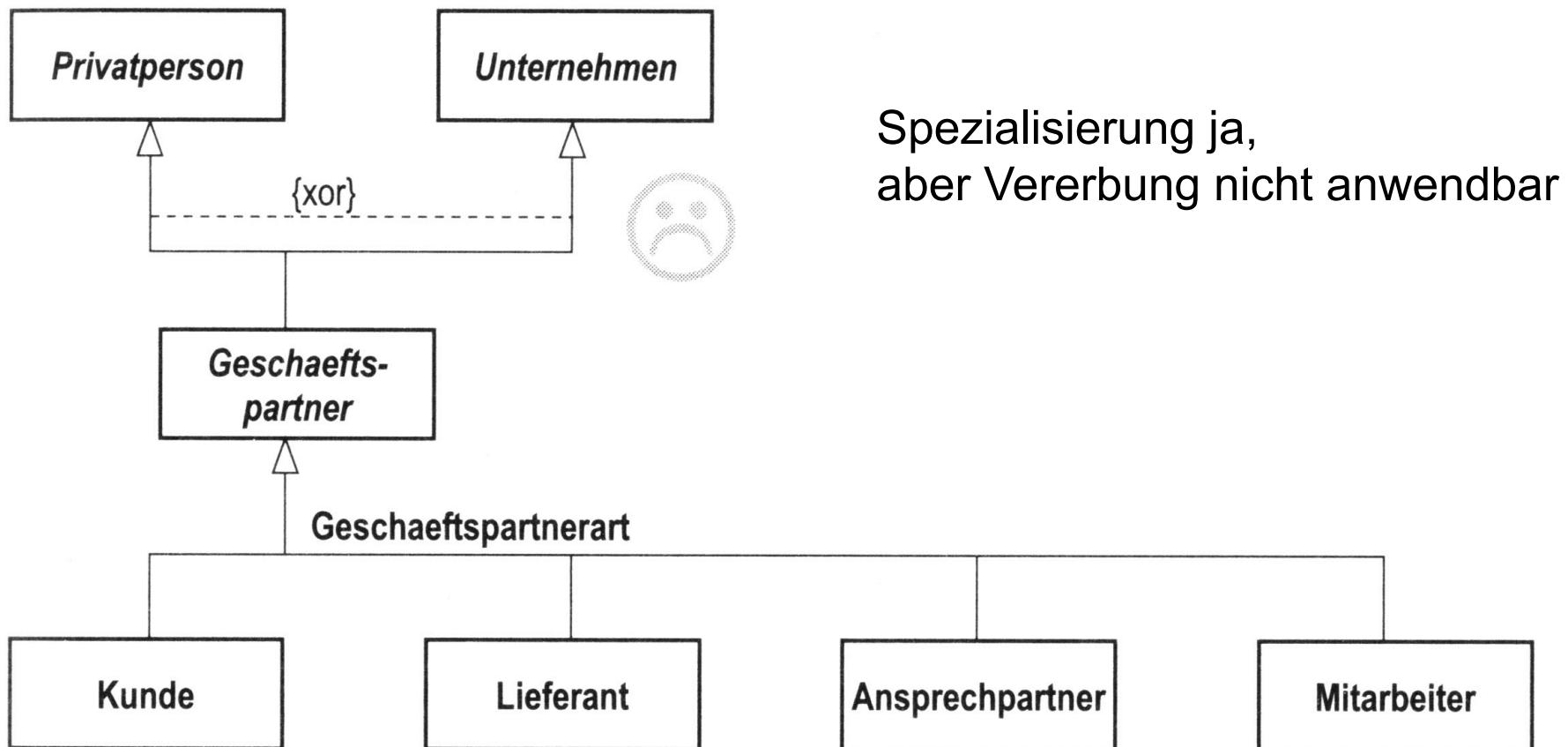
15

Delegation statt Vererbung

Delegation:
bestimmte Eigenschaften gezielt weitergeben
Aber: separat implementieren

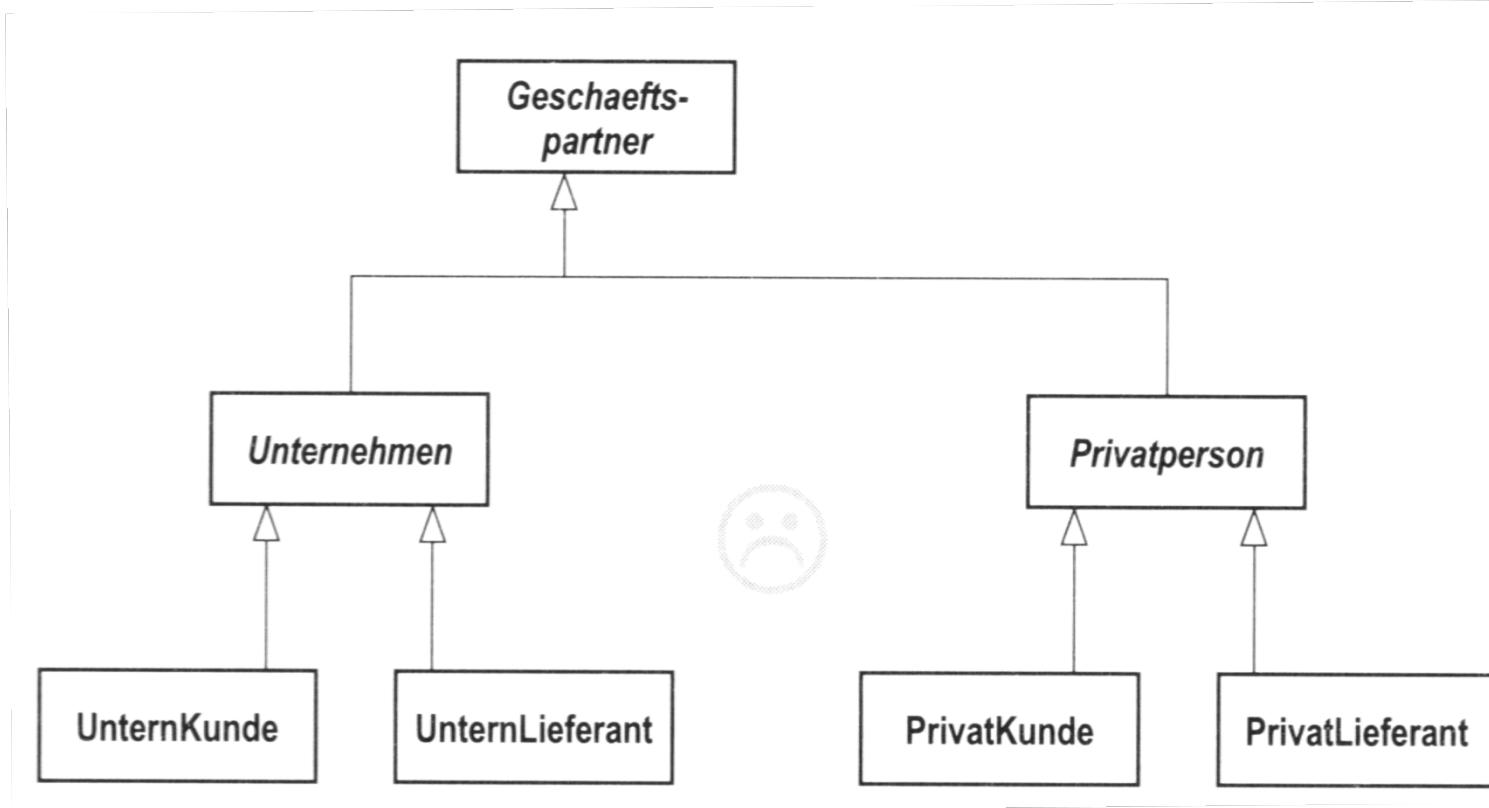


Spezialisierung wechseln: Rollen



[Oestereich]

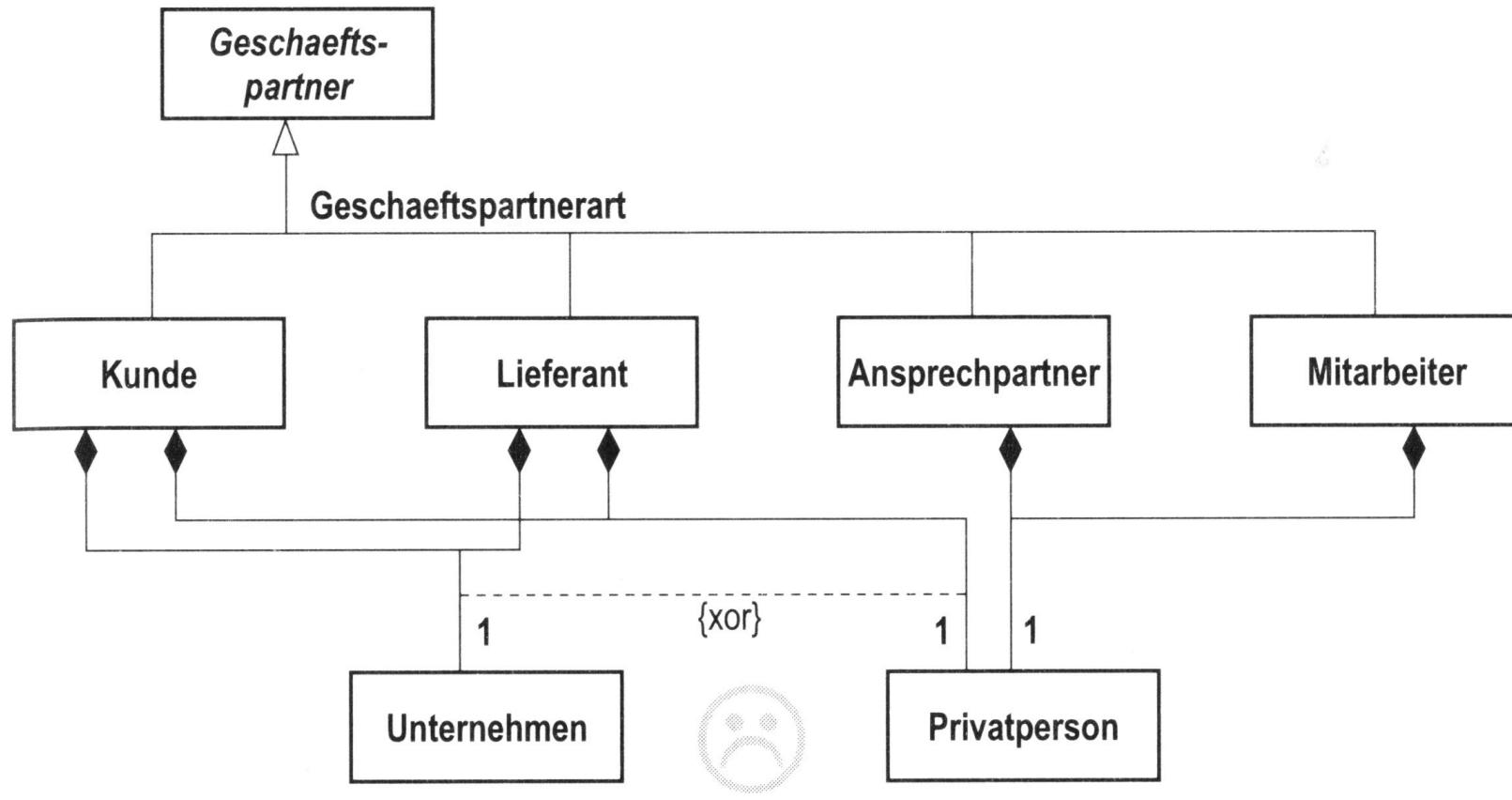
Vererbung – Versuch 2



[Oestereich]

Kombinatorische Explosion

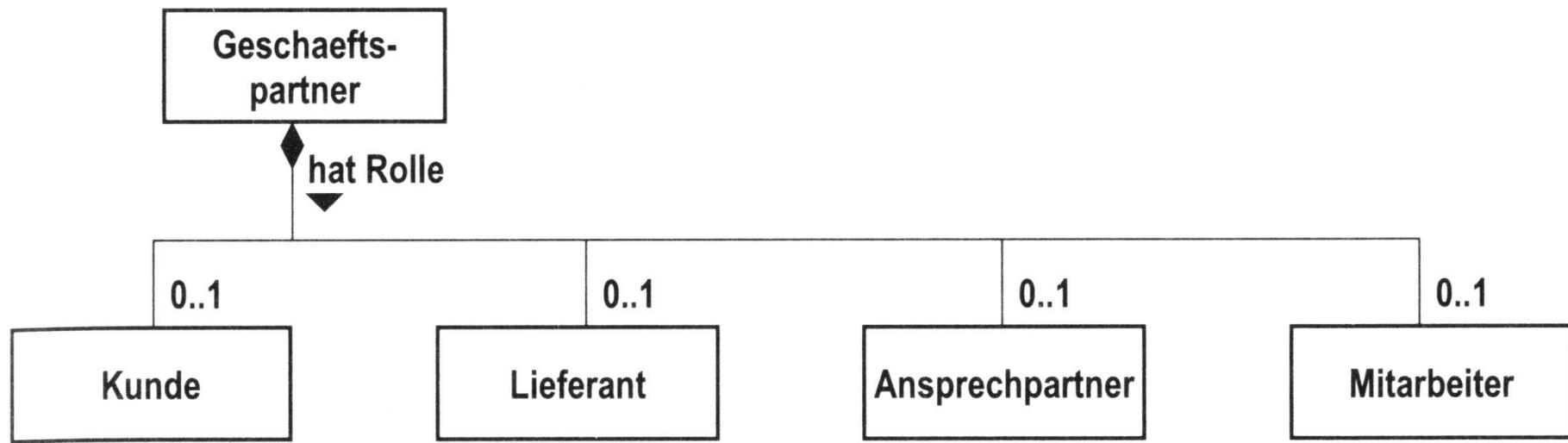
Komposition - Versuch 3



[Oestreich]

Bedingungen notwendig

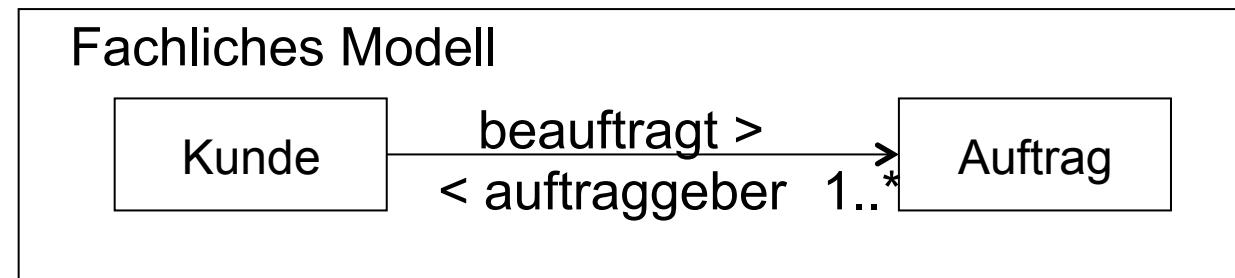
Rollen – Versuch 4



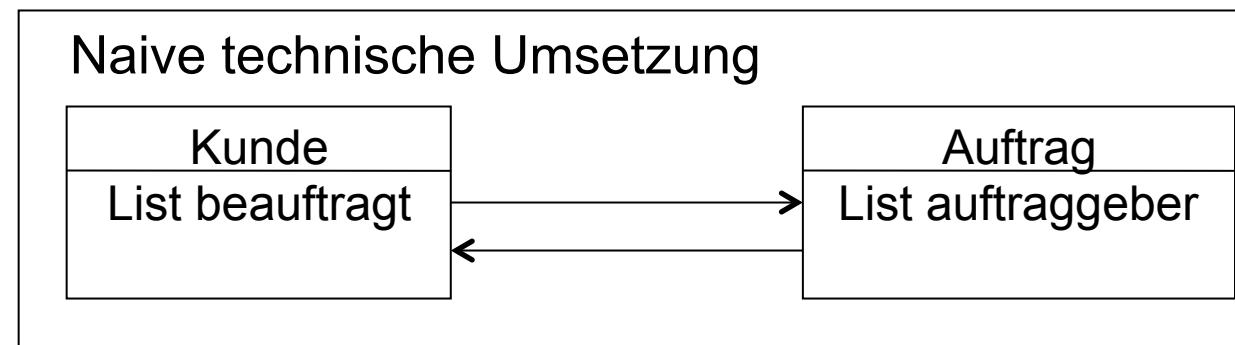
[Oestreich]

Rolle = Sichtweise

Enge Kopplung durch Objekt-Referenzen



Wechselseitige
Beziehungen



Wechselseitige
Objektreferenzen

Abhängigkeiten!!

Problemlösung:

gemeinsame Komponente (kleine Systeme)

lose Kopplung: Übergabe von Werten statt Referenzen

3. SW-Qualitätsmanagement

Fragen #1

1. Nennen Sie die Untermerkmale des Qualitätsmerkmals Wartbarkeit (Effizienz, Zuverlässigkeit, Übertragbarkeit) ?
2. Worin besteht der Zusammenhang zwischen Prozess- und Produktqualität?
3. Erläutern Sie den Zusammenhang zwischen Fehlerverweilzeit und Fehlerbehebungskosten!
4. Warum sind Inspektionen zusätzlich zu Tests sinnvoll?
5. Vergleichen Sie Inspektionen und Tests bezüglich Einsatzmöglichkeiten (Phasen, Produkte) und Art von gefundenen Fehlern und Mängeln!
6. Wieso hat der unterschiedliche Einsatzzeitpunkt von Inspektionen und Tests Einfluss auf die Produktivität?

3.1 Software-Qualitätsmerkmale

ISO/IEC 25000: Produktmerkmale

Äußere Qualitätsmerkmale
ISO/IEC 25020, 25023

Funktionalität

Angemessenheit
Richtigkeit
Interoperabilität
Sicherheit
Ordnungsmäßigkeit

Benutzbarkeit

Verständlichkeit
Erlernbarkeit
Bedienbarkeit
Attraktivität
Konformität (Ben.)

Zuverlässigkeit

Reife
Fehlertoleranz
Wiederherstellbarkeit
Konformität (Zuverl.)

Effizienz

Zeitverhalten
Verbrauchsverhalten
Konformität (Effiz.)

Innere Qualitätsmerkmale
ISO/IEC 25020, 25022

Wartbarkeit

Analysierbarkeit
Änderbarkeit
Stabilität
Testbarkeit
Konformität (Wartbark.)

Übertragbarkeit

Anpassbarkeit
Installierbarkeit
Koexistenz
Austauschbarkeit
Konformität (Übertragb.)

Funktionalität

Funktionalität: Inwieweit besitzt die Software die geforderten Funktionen

- Angemessenheit: Eignung von Funktionen für spezifizierte Aufgaben
- Richtigkeit: Liefern der richtigen oder vereinbarten Ergebnisse oder Wirkungen
- Interoperabilität: Fähigkeit, mit vorgegebenen Systemen zusammenzuwirken
- Sicherheit: Fähigkeit, unberechtigten Zugriff, sowohl versehentlich als auch vorsätzlich, auf Programme und Daten zu verhindern
- Ordnungsmäßigkeit: anwendungsspezifische Normen oder Vereinbarungen oder gesetzliche Bestimmungen erfüllen

Zuverlässigkeit

Zuverlässigkeit: ein bestimmtes Leistungsniveau unter bestimmten Bedingungen über einen bestimmten Zeitraum aufrechterhalten

- Reife: Geringe Versagenshäufigkeit durch Fehlerzustände
- Fehlertoleranz: spezifiziertes Leistungsniveau bei Software-Fehlern oder Nicht-Einhaltung ihrer spezifizierten Schnittstelle bewahren
- Wiederherstellbarkeit: bei Versagen Leistungsniveau wiederherzustellen
- Konformität: Grad, in dem die Software Normen oder Vereinbarungen zur Zuverlässigkeit erfüllt

Benutzbarkeit

Benutzbarkeit: Aufwand zur Benutzung

- Verständlichkeit: Aufwand für Verständnis von Konzept und Anwendung
- Erlernbarkeit: Aufwand für Erlernen der Anwendung
- Bedienbarkeit: Aufwand für Bedienen der Anwendung
- Attraktivität: Anziehungskraft gegenüber dem Benutzer
- Konformität: Grad, in dem die Software Normen oder Vereinbarungen zur Benutzbarkeit erfüllt

Effizienz

- Effizienz: Verhältnis zwischen Leistungsniveau der Software und eingesetzten Betriebsmitteln
- Zeitverhalten: Antwort- und Verarbeitungszeiten sowie Durchsatz bei der Funktionsausführung
- Verbrauchsverhalten: Anzahl und Dauer der benötigten Betriebsmittel bei der Erfüllung der Funktionen. Ressourcenverbrauch, wie CPU-Zeit, Festplattenzugriffe
- Konformität: Grad, in dem die Software Normen oder Vereinbarungen zur Effizienz erfüllt.

Wartbarkeit

- Wartbarkeit: Aufwand für Durchführung vorgegebener Änderungen an der Software
- Analysierbarkeit: Aufwand für Diagnose von Mängeln oder Ursachen von Versagen
- Änderbarkeit: Aufwand zur Ausführung von Veränderungen
- Stabilität: Wahrscheinlichkeit des Auftretens unerwarteter Wirkungen von Änderungen.
- Testbarkeit: Aufwand zur Prüfung der geänderten Software
- Konformität: Grad, in dem die Software Normen oder Vereinbarungen zur Änderbarkeit erfüllt.

Übertragbarkeit

- Übertragbarkeit: Aufwand für Übertragung der Software in eine andere Umgebung
- Anpassbarkeit: Aufwand zur Anpassung an verschiedene Umgebungen anzupassen.
- Installierbarkeit: Aufwand zum Installieren in festgelegter Umgebung
- Koexistenz: Fähigkeit der Arbeit neben einer anderen Systemen mit ähnlichen oder gleichen Funktionen
- Austauschbarkeit: Möglichkeit der Verwendung der Software in anderer Umgebung jener Software zu verwenden, sowie der dafür notwendige Aufwand.
- Konformität: Grad, in dem die Software Normen oder Vereinbarungen zur Übertragbarkeit erfüllt

Merkmale der Prozessqualität

Prozess-Reifegrad
Reifegradmodell
Terminplanung
Termineinhaltung
Güte der Abschätzung
Budgetplanung
Budgeteinhaltung
Güte der Abschätzung
Produktivität
Ergebnis ./. Aufwand
Aufgaben-Koordination
Anteil Leerlaufzeiten
Grad Paralleltätigkeiten

Organisation und Kommunikation
Regelungsgrad
Informationsweg-Länge
Spezialisierungsgrad
Kommunikationsaufwand
Erfahrungsmanagement
Einarbeitungsaufwand
Fehlervermeidung / Anteil
Rework
Optimierungsgeschwindigkeit
Kundenzufriedenheit
Mitarbeiterzufriedenheit

Fehlerfortpflanzung

- Fehler-Erkennung häufig erst an Folgewirkung
- Rückverfolgung und Ursachenerkennung bei komplexen Systemen schwierig
 - Deutlich aufwendiger als Behebung
- Fehler pflanzen sich in Folgeschritten fort:
 - Entwicklungsschritte Vorläuferprodukte zu wiederholen – „Rework“
 - Je später erkannt, um so umfangreicher Rework
- Fehlerverweilzeit senken = Produktivität erhöhen

Konstruktive Maßnahmen

- Technisch-konstruktive Maßnahmen
 - Methoden- und Werkzeuganwendung (Softwareengineering)
 - Programmiersprachen
- Organisatorische Maßnahmen
 - Projektmanagement (z. B. Pläne und Koordinierung)
 - Konfigurationsmanagement (z. B. Schutz vor Veränderungen und Inkonsistenzen)
 - Vorgehensmodell
- Psychologisch orientierte Maßnahmen
 - Schulungen (z. B. zu Qualitätsmaßnahmen und Zielen)
 - Motivationsfördernde Maßnahmen (z. B. Qualitätszirkel)
 - Kommunikationsverbessernde Maßnahmen

3.2 Analytische Maßnahmen

- Qualität prüfen und bewerten

Ziele:

- Kunde: erfüllt Produkt meine Anforderungen?
- Projektleiter: Wie ist Qualität der Zwischen- und Endprodukte?
- Entwickler: Habe ich gute Arbeit geleistet?

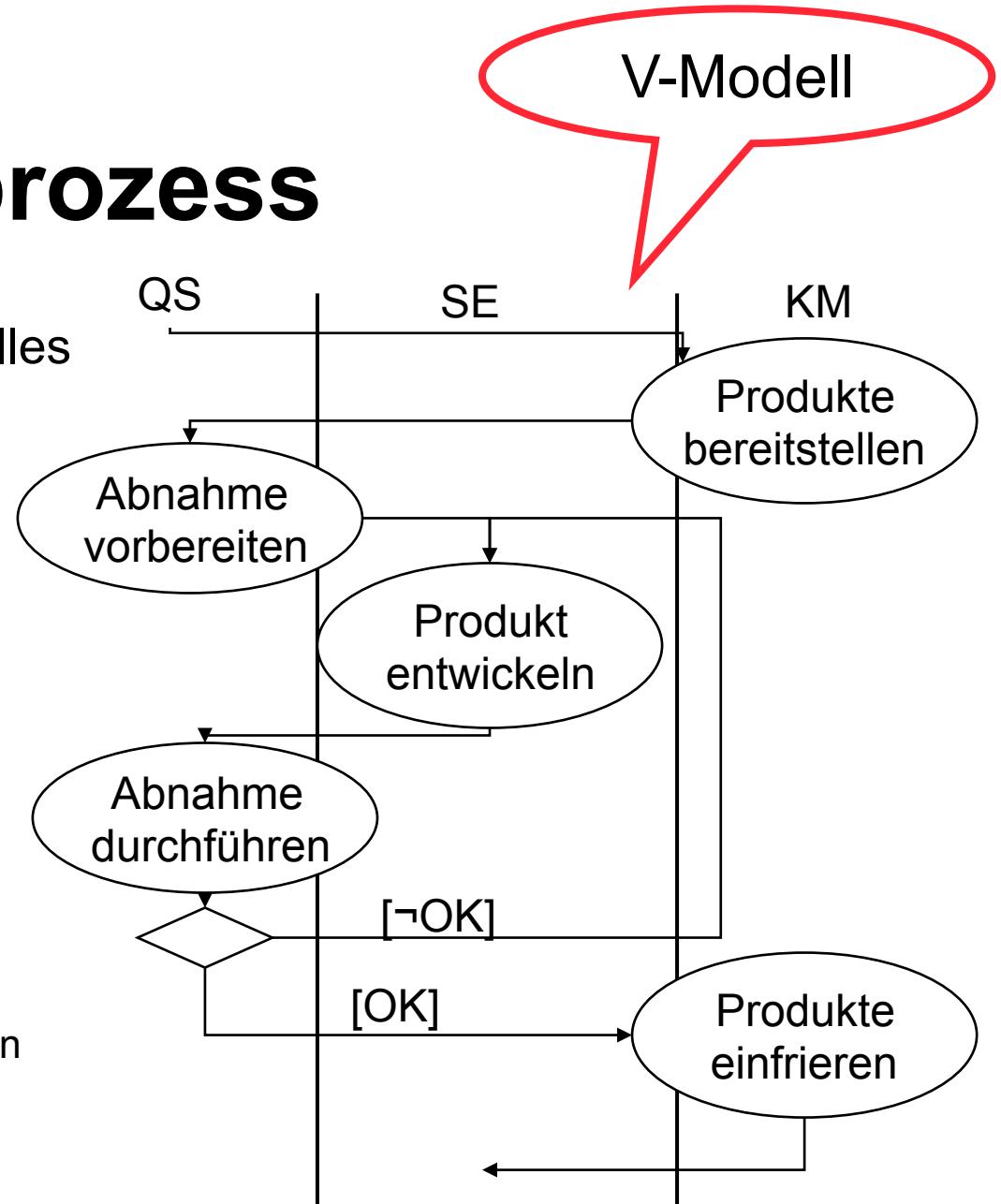
Verifikation: Übereinstimmung der Ergebnisse einer Phase mit der vorigen

Validation: Übereinstimmung der Ergebnisse mit Anforderungen

- Frühzeitig analysieren: Risiken und Rework-Aufwand reduzieren

Einbindung in Entwicklungsprozess

- Möglichkeiten für inkrementelles Vorgehen nutzen
- Abnahmekriterien, Testszenarien / -fälle vor Entwicklung bereitstellen
- Wartbarkeit, Erweiterbarkeit, Flexibilität usw. als explizite Ziele aufnehmen
- Zugehörige Dokumente einbeziehen
- Schrittweise „Reifegrad“ erhöhen
 - Dokumentationsbasis verbessern
 - Architekturqualität verbessern
 - Modell-Qualität verbessern



Statische und dynamische Maßnahmen - Einsatzbereich

	Statische Maßnahmen: Review	Dynamische Maßnahmen: Test
Ablauf	Prüfobjekt liegt vor, wird begutachtet	Prüfobjekt wird ausgeführt, Verhalten beobachtet
Produkte	Alle Arten	Nur ausführbare Produkte (Code)
Phasen	Frühzeitig möglich	Erst nach teilweiser Implementierung
Fehler, Mängel	Auch innere Q- Merkmale: Wartbarkeit etc.	Nur Fehlverhalten (äußere Q-Merkmale)

Statische und dynamische Maßnahmen – gefundene Fehler

Review in Entwicklungsphase	gefundene Fehler
Review der funktionalen Spezifikation	2 ... 5 %
Review des Grobentwurfes	10 ... 16 %
Review des Feinentwurfs	17 ... 22 %
Review des Codes	20 ... 32 %
Modultest	12 ... 17 %
Funktionstest	10 ... 14 %
Komponententest	8 ... 14 %
Systemtest	0,5 ... 7 %

3. SW-Qualitätsmanagement

Fragen #2

7. Stellen Sie die Testmethoden gegenüber!
8. Nennen Sie Testabdeckungsmaße!
9. Nennen Sie die Gemeinsamkeiten und Unterschiede von Black- und White-Box-Tests!
10. Geben Sie ein Beispiel für die Kombination von Black- und White-Box-Tests auf verschiedenen Test-Ebenen an!
11. Was sind Äquivalenzklassen?
12. Was sind Grenzwertanalysen bei der Aufstellung von Testfällen?
13. Erläutern Sie Zustands-, Entscheidungs- und Use-Case-basiertes Testen!
14. Was sind die Besonderheiten beim Testen objektorientierter Software?

Testumgebung und Testdurchführung

Test im eigentlichen Sinne: Ausführung mit Testdaten

- Testrahmen erforderlich
 - Treiber (driver) zum Aufrufen des Testobjektes und zur Übergabe von Eingabedaten und Resultaten
 - Stellvertreter (stubs) zur Simulation der externen Funktionen
- Tests müssen
 - wiederholbar sein (z.B. für Regressionstests)
 - nachvollziehbar sein
 - dokumentiert werden
 - d.h. geplant und kontrolliert werden

Testmethoden

- Beobachten des Verhaltens bei Ausführung
- Stichprobenverfahren, kein Nachweis der Korrektheit
- Viele mögliche Kombinationen → viele Testfälle → Extrem hoher Aufwand
 - Verringerung der Anzahl der Testfälle
 - Durchführung von Positiv-, Negativ- und Lasttests
- Abdeckungsmaße zur Bewertung der Tests

Testabdeckungsmaße

- Anweisungsabdeckung
Jede Anweisung im Testobjekt mindestens einmal ausgeführt
Maß (C0) = Anzahl der ausgeführten Anweisungen / Anzahl aller Anweisungen → nicht ausreichend
- Zweigabdeckung
Jede Bedingung, die zur Änderung des Ablaufs führt, soll mindestens einmal den Wert TRUE und einmal den Wert FALSE liefern. → Mindestanforderung
Maß (C1) = Anzahl der ausgeführten Zweige / Anzahl aller Zweige
- Pfadabdeckung
Alle möglichen unterschiedlichen Abläufe des Testobjektes sollen in der Testphase einmal ausgeführt werden. → nicht realistisch

Gegenüberstellung Testtechniken

	Black Box „Funktionaler Test“	White Box „Strukturtest“
Ablauf	Testgegenstand mit Testdaten beschicken, Verhalten beobachten, Resultate mit Erwartungen vergleichen	
Quelle der Testdaten	Spezifikation, Schnittstellenbeschreibung	Struktur, Steuerflußanalyse
Art gefundener Fehler	Falsche Funktionalität, auch Entwurfsfehler, daten-abhängige Fehler	Falsche Funktionalität, auch verborgene Funktionalität (Viren, Trojaner)
Anwendung in Phasen	Subsystem-, Integrations-, Abnahmetest	„Entwicklertest“ Klassen-, Modultest

Testebenen

- Modultest/Klassentest - Testen im Kleinen
 - Verhalten von Funktionen, Programmteilen, Modulen
- Integrationstest - Testen im Großen
 - Montage der fertig getesteten Module
 - Test des Zusammenspiels der Module (Schnittstellentest)
- Systemtest - Test des kompletten Systems
 - Außenverhalten des Gesamtsystems
 - Laufzeit- und Stresstests
 - Bewertung der Benutzungsfreundlichkeit

Integration der Testtechniken

1) funktionsorientierter Klassentest

- Instrumentierung für Zweigüberdeckung
- Äquivalenzklassen-Tests nach Spezifikation
- Kontrolle der erreichten Zweigüberdeckung
typische Zweigüberdeckung 70-80 %

2) strukturorientierter Klassentest

- Testfälle für noch nicht durchlaufene Zweige
- ➔ garantierte Erfüllung der Minimalbedingungen

Äquivalenzklassen

- Partitionierung des Eingangsraumes einer Methode durch Bildung von Äquivalenzklassen für die Eingangswerte.
 - Werte aus einer Äquivalenzklasse
 - verursachen identisches funktionales Verhalten
 - testen identische spezifizierte Funktionen
- Verringerte Anzahl der Testfälle, gleiche Testabdeckung

Regeln zur Bildung der Äquivalenzklassen:

- Positiv-Klassen: Testfälle aus Kombination möglichst vieler Testdaten aus gültigen Äquivalenzklassen
- Negativ-Klassen: Testfälle aus
 - einem Testdatum einer ungültigen Äquivalenzklassen
 - weiteren Testdaten ausschließlich aus gültigen Äquivalenzklassen

Grenzwertanalyse

- Fehlerwahrscheinlichkeit höher an den Grenzen einer Äquivalenzklasse als innen
- Testen an Grenzen erfolgreicher
- Testfälle für Grenzen gültiger Äquivalenzklasse, ungültiger Äquivalenzklassen

Beispiel

Strafen bei
Geschwindigkeitsüberschreitung Innerorts:

bis 10 km/h: 15 €, 0 Monate, 0 Punkte
11 bis 15 km/h: 25 €, 0 Monate, 0 Punkte
16 bis 20 km/h: 35 €, 0 Monate, 0 Punkte
21 bis 25 km/h: 80 €, 0 Monate, 1 Punkt
26 bis 30 km/h: 100 €, 0 Monate, 3 Punkte
31 bis 40 km/h: 160 €, 1 Monat, 3 Punkte
41 bis 50 km/h: 200 €, 1 Monat, 4 Punkte
51 bis 60 km/h: 280 €, 2 Monate, 4 Punkte
61 bis 70 km/h: 480 €, 3 Monate, 4 Punkte
über 70 km/h: 680 €, 3 Monate, 4 Punkte

Erstellte Testfälle

v = -1 (Negativ-Test)
v = 1, v= 9 (nahe Grenze)
v = 11, v= 15 (nahe Grenze)
v = 16, v= 19 (nahe Grenze)
v = 21, v= 25 (nahe Grenze)
v = 26, v= 30 (nahe Grenze)
v = 31, v= 39 (nahe Grenze)
v = 41, v= 49 (nahe Grenze)
v = 51, v= 59 (nahe Grenze)
v = 61, v= 69 (nahe Grenze)
v = 16, v= 99 (nahe Grenze)

Entscheidungstabellen- basierter Test

Entscheidungstabelle für:

- Aufstellung der möglichen Kombinationen
- Prüfung der Vollständigkeit
- Abgrenzung von Äquivalenzklassen

vier Teilbereiche:

- Bedingungen: mögliche Zustände
- mögliche Aktionen
- Regeln: mögliche Bedingungskombinationen
- Aktionsanzeiger: Zuordnung der Aktivitäten zu Bedingungskombinationen

Bedingungen	Regeln
Aktionen	Aktionsanzeiger

Entscheidungstabelle – Vorgehen

Aufstellung:

1. Bedingungen festlegen
2. Aktionen angeben
3. Regeln und Aktionsanzeiger setzen
4. Konsolidierung der Entscheidungstabelle
5. Prüfung auf Widerspruchsfreiheit und Vollständigkeit

Entscheidungstabelle - Beispiel

Bereichsleiter Schmid möchte eine Mitarbeiterin im Krankenhaus besuchen. Er informiert sich telefonisch an der Information über die Besuchsmöglichkeiten und erhält folgende Antwort:

Die Patientin kann ohne Einschränkungen innerhalb der Besuchszeit besucht werden, sofern keine ansteckende Krankheit vorliegt und sie kein Fieber hat. Außerhalb der Besuchszeit ist in diesem Fall eine Schwester als Begleitung erforderlich. Falls die Patientin eine ansteckende Krankheit hat, werden Besuche ganz abgelehnt. Wenn die Krankheit nicht ansteckend ist, die Patientin aber Fieber hat, darf der Besuch innerhalb der Besuchszeit maximal 30 Minuten betragen, außerhalb der Besuchszeit dürfen Patienten mit Fieber nicht besucht werden.

Bedingungen:

- Patientin hat ansteckende Krankheit (j/n)
- Besuch innerhalb Besuchszeit (j/n)
- Patientin hat Fieber (j/n)

Aktionen:

- Besuchszeit maximal 30 Minuten
- Besuch ablehnen
- Besuch mit Begleitung einer Schwester
- Normalbesuch in Besuchszeit

Bedingungen	1	2	3	4	5	6	7	8
Ansteckende Krankheit (j/n)	j	j	j	j	n	n	n	n
innerhalb Besuchszeit (j/n)	j	j	n	n	j	j	n	n
Fieber (j/n)	j	n	j	n	j	n	j	n
Aktionen					x			
Besuchszeit maximal 30 Minuten					x			
Besuch ablehnen	x	x	x	x			x	
Mit Begleitung einer Schwester								x
Normalbesuch in Besuchszeit						x		

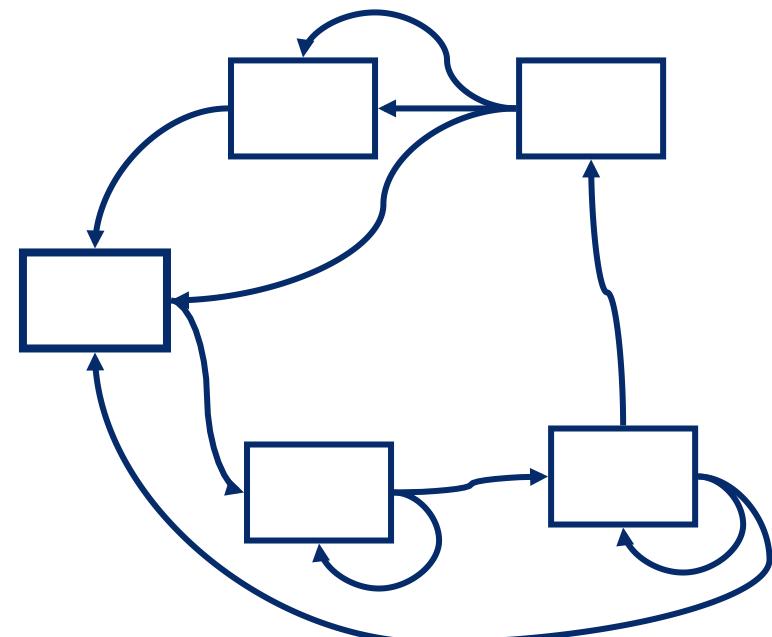
Zustandsbasierter Test

Anwendungsgebiet:

Test von Methodensequenzen, die als Zustandsautomat spezifiziert werden.

Abdeckungsmaße:

- typische Folgen von Zuständen
- alle Zustände (Minimum)
- alle Zustandsübergänge einmal durchlaufen (besser)
- typische Folgen von Zustandsübergängen
- alle Ereignisse (noch besser)
- ungültige Zustandsübergänge



Use-Case-basierter Test

- Interaktionen zwischen Benutzer und System führt zu Reaktionen
- Vorbedingungen für Eintreten des Use Case
- Nachbedingungen nach Ende des Use Case (soweit sichtbar)
- Mehrere Szenarien: Normalfall, Ausnahmefall, Fehlerfall
 - Für Prüfung von Abläufen (z.B. Workflows)
 - Für Prüfung des Zusammenspiels von Komponenten
 - Meist in Kombination mit anderen Methoden

Testen objektorientierter Software

- Kapselung von Objekten behindert das Testen
 - Fehlerbehandlung
 - gegenüber Benutzer sinnvoll
 - gegenüber Test-Software nicht sinnvoll, da
 - Fehler maskiert werden
 - Testbarkeit reduziert wird
- Fehlerwirkung forcieren
- Zusicherungen erhöhen Test- und Beobachtbarkeit



Ziel: Kein zusätzlicher Code für Fehlerbehandlung



Klassentest

- Testen der Methoden eines Objekts, welches Instanz der zu testenden Klassen ist.
- Im Vergleich zu Modulen aus imperativen Programmierparadigmen zeigen Methoden
 - eine einfache Kontrollstruktur.
 - eine starke Verflechtung über die Attribute der Klasse.

Eine Methode sollte

- eine gewisse Mindestkomplexität aufweisen.
- keine zu große Abhängigkeit von anderen Teilen des Objektes oder zu anderen Objekten besitzen.

Voraussetzung für den Test einer einzelnen Methode

Die Methode sollte

- eine gewisse Mindestkomplexität aufweisen.
- keine zu große Abhängigkeit von anderen Teilen des Objektes oder zu anderen Objekten besitzen.
- Ziel des OO-Testens ist nicht das Testen der Methoden an sich, sondern das Testen von deren Zusammenspiel über die Attribute einer Klasse.

Zusammenfassung

Testen objektorientierter Software

- Gleiche Testtechniken wie bei imperativen Programmiersprachen
 - funktionaler Test
 - kontrollflußorientierter Test
- Unterschiedliche Vorgehensweise beim OO-Testen
- Integration der Testtechniken zur simultanen Testdurchführung

3. SW-Qualitätsmanagement

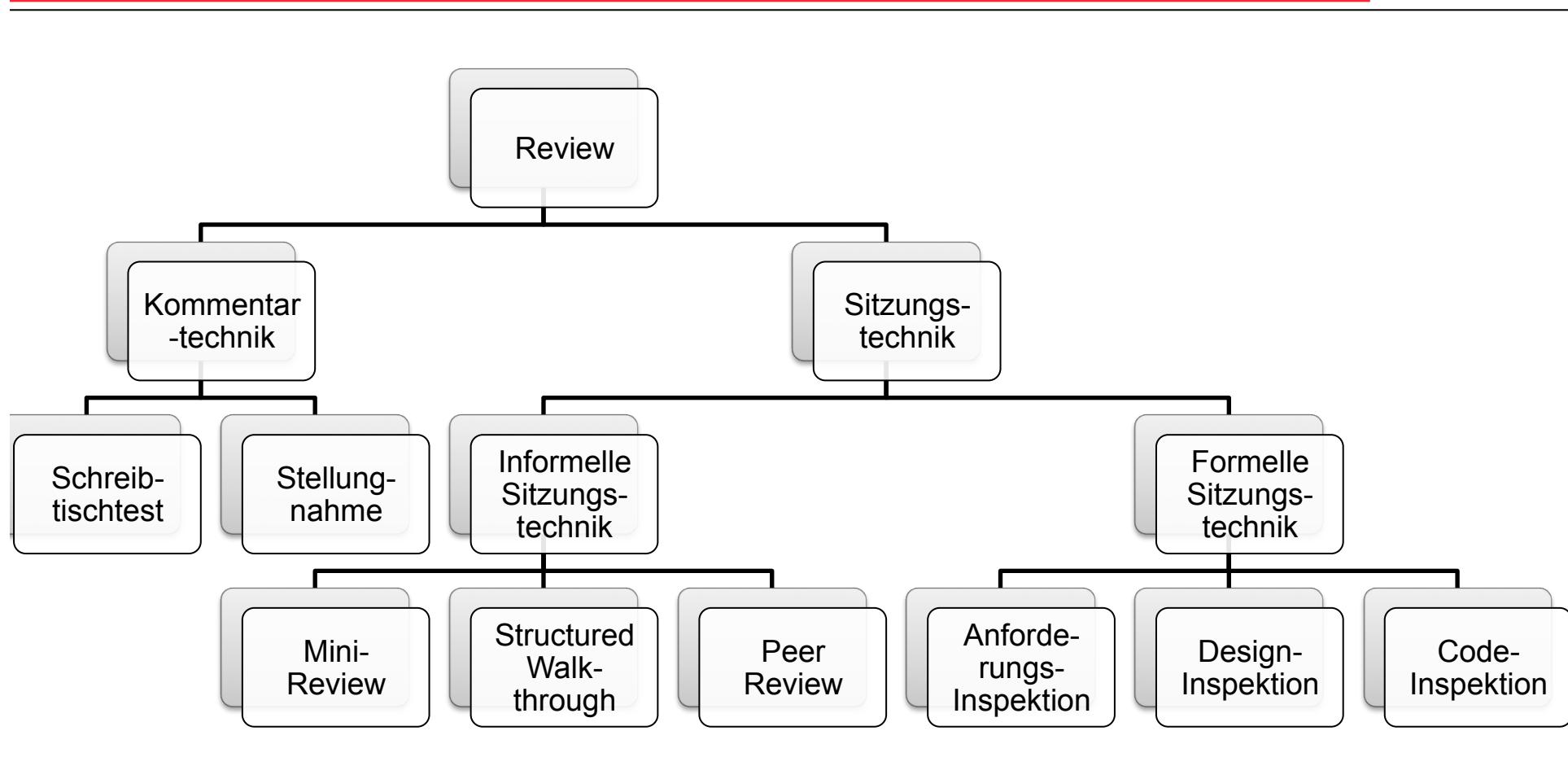
Fragen #3

15. Stellen Sie Inspektion und Walkthrough gegenüber!
16. Nennen Sie die Quelle der Kriterien für eine Code-Inspektion (einen Code-Review)!
17. Geben Sie Kriterien für eine Code-Inspektion (einen Code-Review) bezüglich Wartbarkeit an!
18. Geben Sie Beispiele für sinnvolle Zeitpunkte für Reviews an, und erläutern Sie diese!
19. Geben Sie den Ablauf einer Inspektion an!
20. Nennen Sie die Rollen der Beteiligten an einer Code-Inspektion (einem Code-Review)!
21. Nennen Sie die mit Reviews und Inspektionen verfolgten Ziele!

3.3 Analytische statische Maßnahmen

- Audit
 - Prüfung der Einhaltung von Vorgaben
 - Prüfung der Sinnhaftigkeit, Wirksamkeit
- Review, Inspektion
 - Formaler Analyseprozeß vor Gutachtern
- Walkthrough
 - Durchspielen von Abläufen
- Korrektheitsbeweiser
 - Gegenüber formaler Spezifikation
- Symbolische Programmausführung

Statische Maßnahmen: Review-Techniken



Statische Maßnahme: Review, Inspektion

Insbesondere Peer Review:

- Gemeinsame Analyse mit Fragen und Erläuterung durch Autor
- Fremde Fragestellungen: neue Erkenntnisse
- Kognitive Fähigkeiten vereinigt - 4 Augen
- Anerkennung der Ergebnisse und Verbesserungshinweise
- Effektiver als Test: mehr Fehler pro Zeiteinheit
- Motivation für Dokumentation und Programmierstil

Jedoch: Erfolg personen- und klimaabhängig

Vorteile

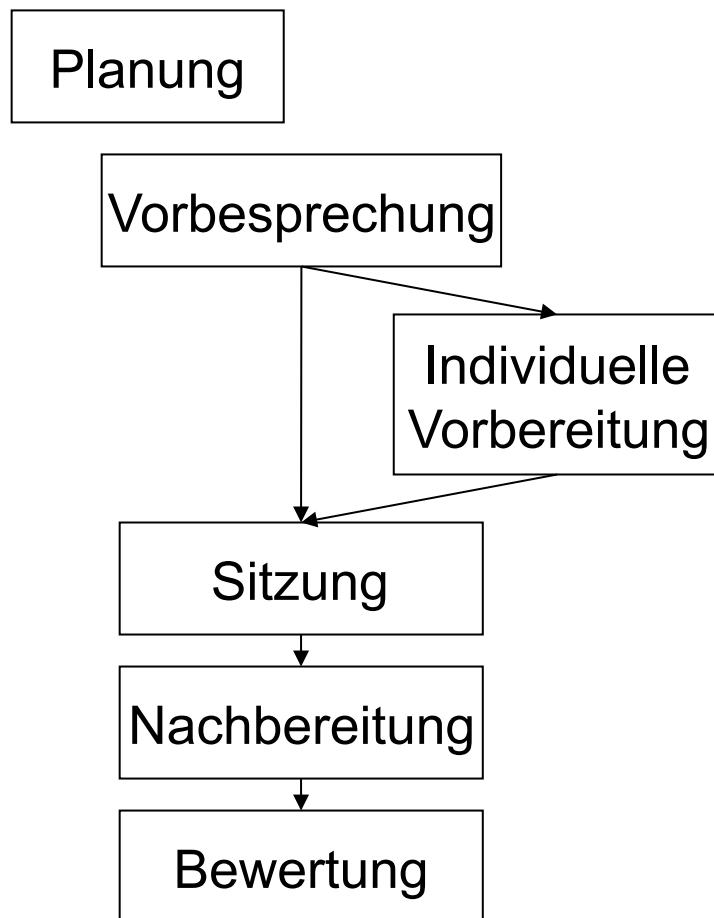
- ca. 80% der Softwarefehler entdeckt
- Mängel entdeckt: Wartbarkeit, Tricks
- einfach durchzuführen, keine spezielle Software benötigt
- Collective Code Ownership statt Einzelkämpfertum
- Unterstützt praktischen Wissenstransfer zwischen erfahrenen und unerfahrenen Mitarbeitern
 - Entwurfsmethoden
 - Defensiver Programmierstil
- Qualitätskriterien schon während des Implementierungsprozesses überprüft

Zeitpunkte Inspektion

- vor Freigabe
- sobald eine gewisse Qualität garantiert werden muss
 - vor dem Besuch des Kunden
 - bevor der Autor in Urlaub geht
- auf Wunsch eines Mitarbeiters, falls selbst unsicher über Qualität seines Codes
- nach ca. 2, höchstens 4 Implementierungswochen (bei Anfänger öfters, bei Profis seltener)

Zeitpunkt der Inspektion frühzeitig bekannt geben, um Vorbereitung zu ermöglichen

Inspektion - Ablauf



Vorbereitung:

- Verstehen, Formale Fehler anhand Checkliste sammeln

Sitzung:

- Mängel nur entdecken und sammeln, nicht diskutieren
- Dauer max. 2 h
- Konstruktives Klima

Nachbereitung:

- Autor behebt Mängel

Bewertung:

- Prüfung der Behebung

Review-Kriterien

- Zielorientiertes Qualitätsmanagement: Anforderungen der Stakeholder als Quelle
- Auftraggeber, Kunden, Benutzer:
 - äußere Q-Merkmale, wie Funktionalität, Benutzbarkeit, Effizienz
- Entwickler, Management des Auftragnehmers:
 - Prozessmerkmale wie Effizienz, Fehlervermeidung
 - innere Q-Merkmale wie Wartbarkeit, Verständlichkeit
- Konzentration auf wenige Kriterien für effiziente Reviews
 - Priorisierung der Kriterien wichtig
 - Erweiterung nach Auftreten von Problemen

Checkliste und Protokoll

Checkliste

- Guter Codierstil mit Facetten:
 - Do's und Don'ts der Sprache: Fehler und Mängel
 - Lesbarkeit
 - Namenskonventionen (allgemein, sprach-, projektspezifisch)
 - Defensiver Programmierstil
 - Verständlichkeit: Kompliziertes vermeiden
- Portabilität, Nebenläufigkeit, Fehlerbehandlung

Protokoll als Liste gefundener Fehler und Mängel:

- Ort
- Art, Erläuterung (keine Behebung)
- Schwere: Severe - Intermediate - Cosmetic

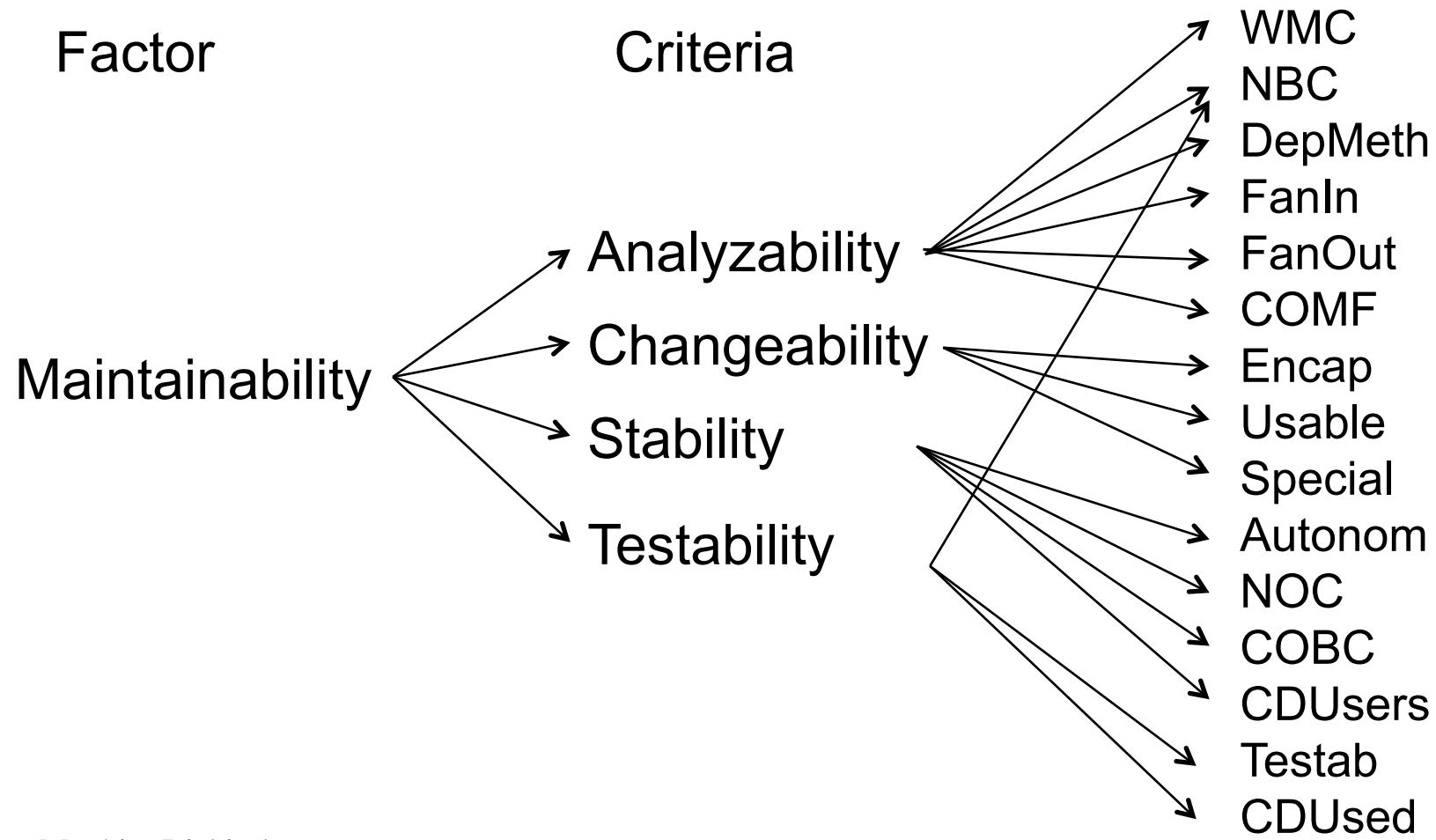
Werkzeugunterstützung: Findbugs, Checkstyle, PMD

- Werkzeuge zur maschinelle Prüfung von Stil-Richtlinien
 - Sun-Richtlinien für Java
- Possible bugs - empty try/catch/finally/switch statements
- Dead code - unused local variables, parameters and private methods
- Suboptimal code - wasteful String/StringBuffer usage
- Overcomplicated expressions - unnecessary if statements, for loops that could be while loops
- Duplicate code - copied/pasted code means copied/pasted bugs

Factor-Criteria-Metrics FCM

- Quality Factors: benutzerorientierte Qualitätsmerkmale der Software.
- Quality Criteria:
 - Teilmmerkmale, die Qualitätsmerkmale verfeinern
 - oft Einfluss auf mehrere Qualitätsmerkmale
 - hilfreich bei Priorisierung
- Quality Metrics: Qualitätsindikatoren / Qualitätsmetriken, die Teilmmerkmale meß- und bewertbar machen
 - inkl. Meßmethode

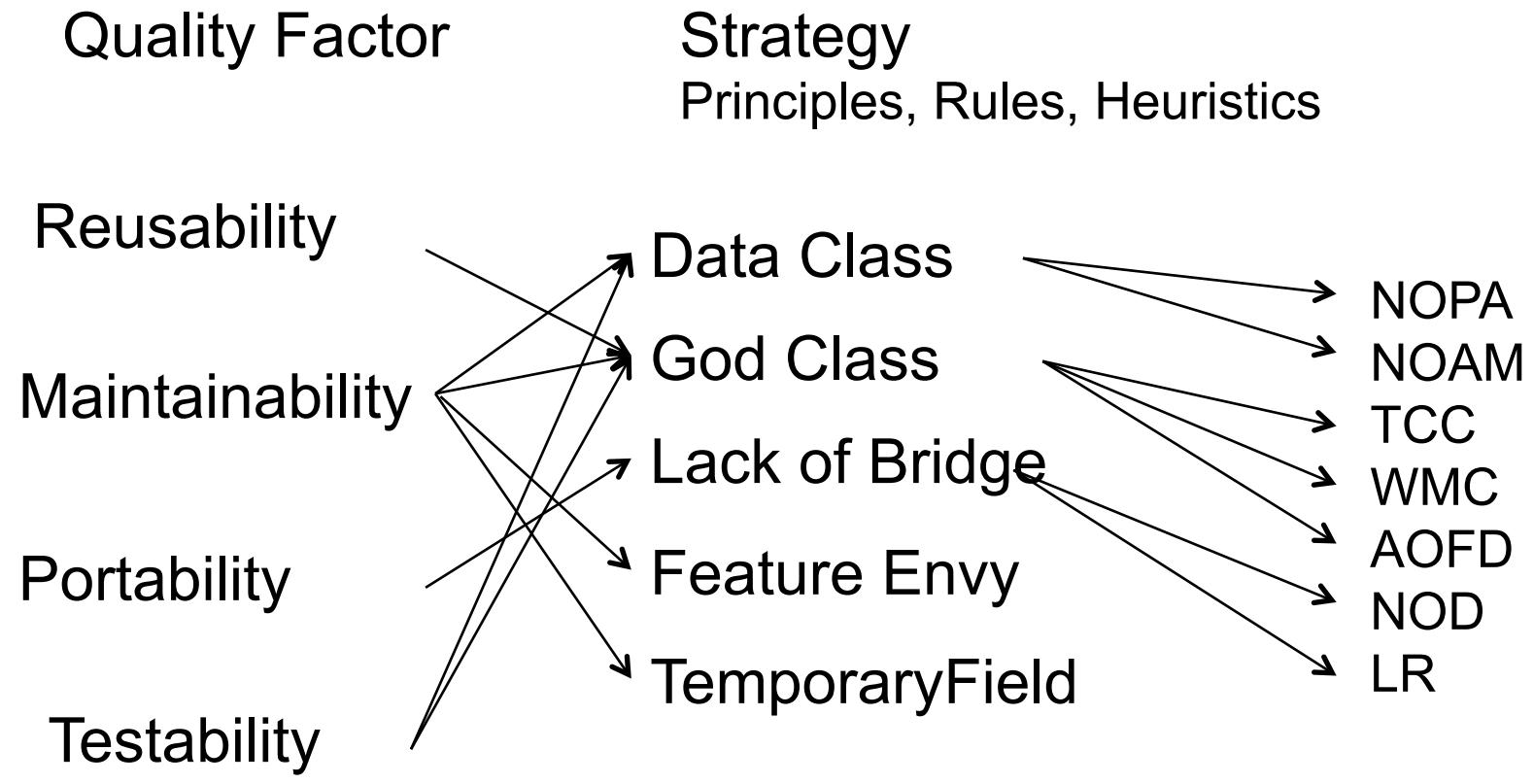
Factor-Criteria-Metrics Beispiel



Factor-Strategy-Modell

- Quality Factors: Qualitätsmerkmale wie bereits behandelt
- Strategy:
 - Erkennungsstrategie, zugeordnet zu Factor
 - Aufdeckung eines Entwurfsproblems
 - Ansätze für Verbesserung
 - Kriterien für Inspektion
 - Kriterien für Refactoring: Bad Smells

Factor-Strategy-Model Beispiel



[MarinescuRatui2004]

Beispiele für Styleguides und Checklisten

Sprachspezifisch:

- Java Styleguide von Sun, bei Oracle: <http://www.oracle.com/technetwork/java/javase/documentation/codeconvtoc-136057.html>

Vorgaben für Namen

- Styleguide der Fa. Geosoft <http://geosoft.no/development/javastyle.html>

Styleguide für die Verwendung von Bibliotheken, APIs:

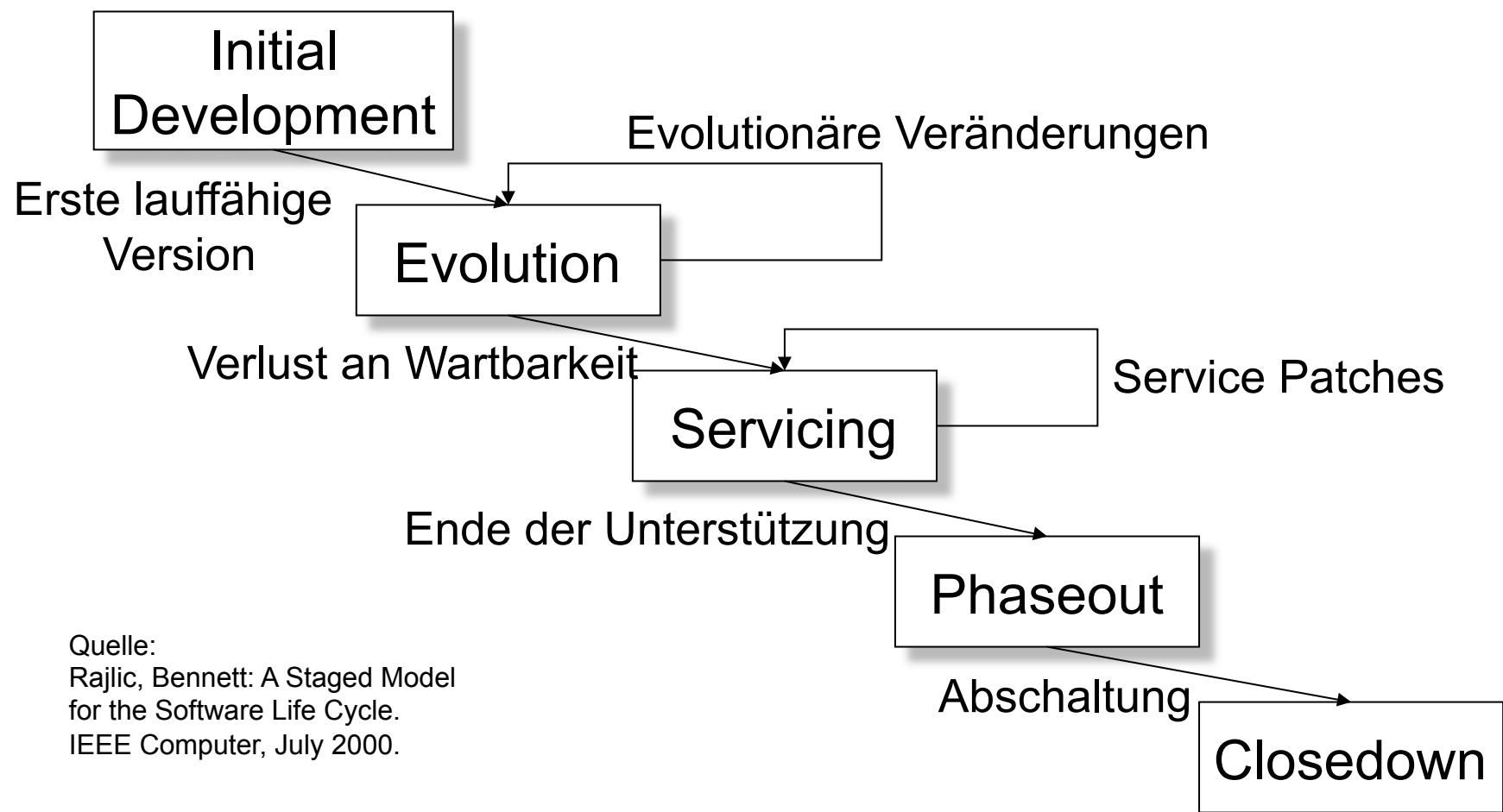
- Android Styleguide <https://sites.google.com/a/android.comopensource/submit-patches/code-style-guide>
- Verwendung von GIT <http://www.jnode.org/book/export/html/3> (ab Git Etiquette bis Configuration Files)

4. Wartung und Reengineering

Fragen #1

1. Worin liegt die wirtschaftliche Bedeutung des Reengineering geschäftskritischer Softwaresysteme?
2. Welche Rolle spielen Änderungen für die Anwendbarkeit eines Softwaresystems?
3. Nennen Sie die Risiken der Ablösung eines existierenden Softwaresystems durch eine Neuentwicklung!
4. Erläutern Sie das Staged Model der Zustände eines Softwaresystems!
5. Erläutern Sie die Begriffe Reengineering, Reverse Engineering und Refactoring!

Staged Model



Quelle:

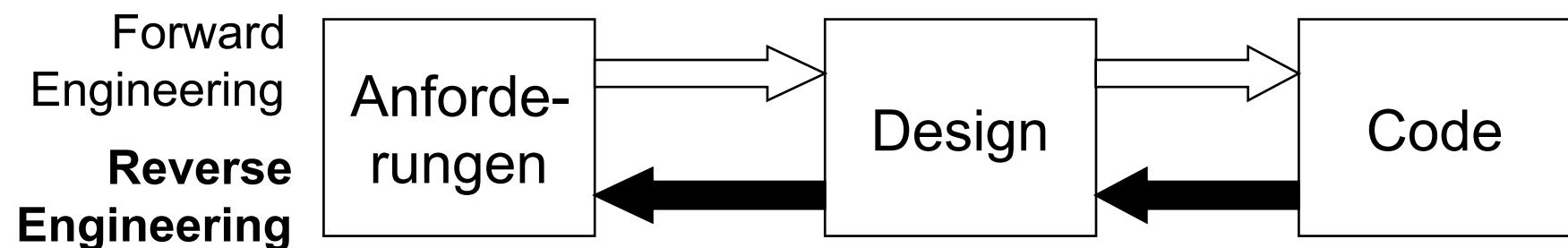
Rajlic, Bennett: A Staged Model
for the Software Life Cycle.
IEEE Computer, July 2000.

Begriff: Software-Wartung

- ANSI/IEEE Standard 729-1983:
 - “Modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a changed environment.”
- Üblicher Sprachgebrauch:
 - Änderungen am System nach dessen Auslieferung.
 - Schließt Anpassungen an neue Anforderungen ein

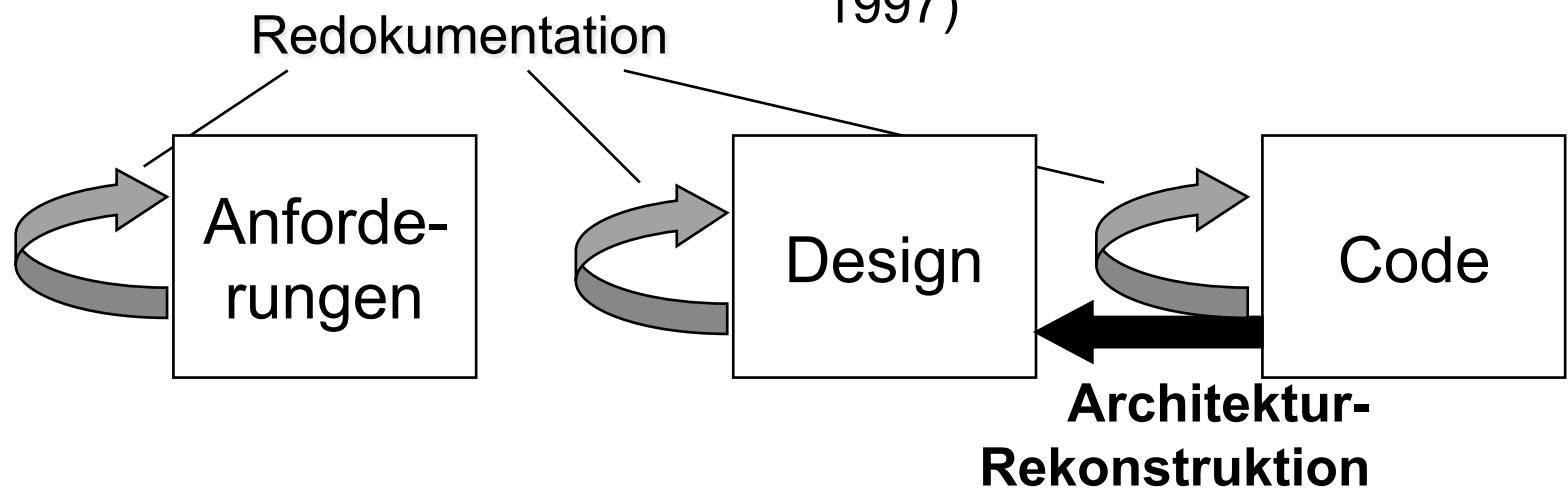
Begriff: Reverse Engineering

- Identifikation der Systemkomponenten und deren Beziehungen. Ziel ist die Beschreibungen des Systems in einer anderen Form oder auf höherem Abstraktionsniveau.
(Chikofsky und Cross II. 1990)
- Aktivitäten zum Erhöhen des Verständnisses und Verbesserung von Wartbarkeit, Wiederverwendbarkeit, nach Inbetriebnahme eines Programmes
(Arnold 1993)



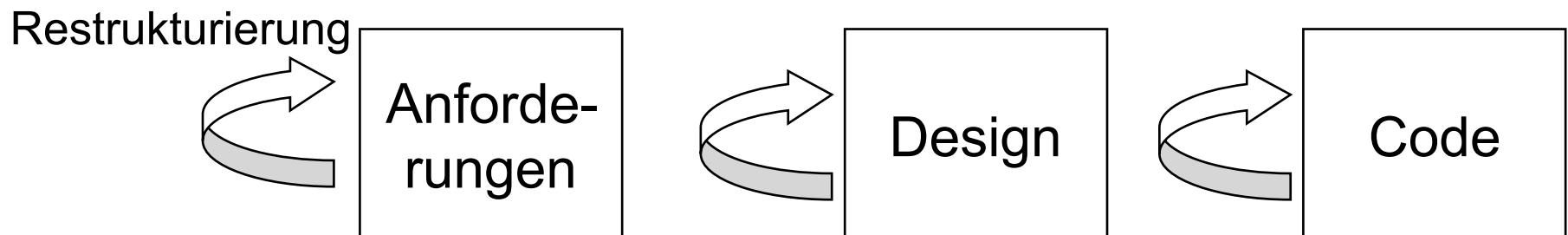
Begriff: Recovery, Redokumentation

- Architektur-Rekonstruktion (Architectural Recovery, Design Recovery): Reverse Engineering mit dem Ziel, eine Beschreibung der Architektur des Systems zu erstellen (Müller 1997, Koschke 2003)
- Redokumentation: Erzeugung einer semantisch äquivalenten Repräsentation des betrachteten Objekts (Programm oder Spezifikation) auf demselben Abstraktionsniveau (Müller 1997)



Begriff: Restrukturierung, Refactoring

- Transformation einer Repräsentation in eine andere auf derselben Abstraktionsebene, ohne Änderung der Funktionalität des Systems. (Chikofsky und Cross II. 1990)

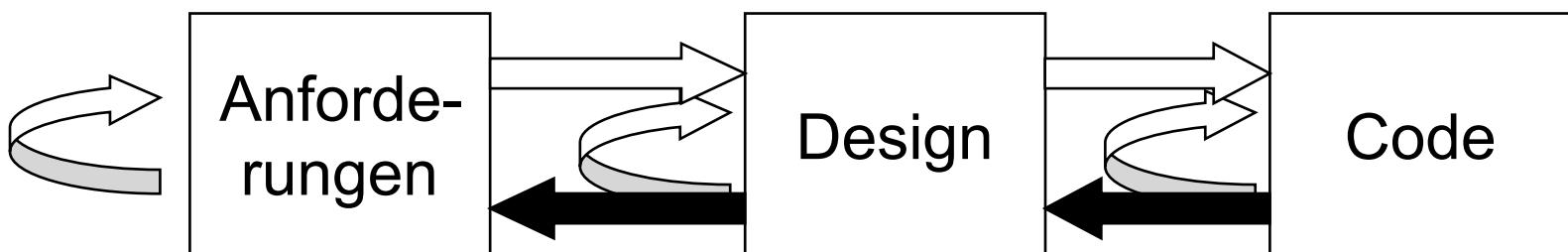


Begriff: Reengineering

(Renovation, Reclamation)

- Untersuchung (Reverse Engineering) und Änderung des Systems, um es in neuer Form zu implementieren.
(Chikofsky und Cross II. 1990)
 - Keine Änderung der Funktionalität
- Erweitertes Reengineering: analysieren / restrukturieren, um dann Funktionalität zu ändern

Forward & Reverse Engineering & Restrukturierung



4. Wartung und Reengineering

Fragen #2

5. Nennen Sie die Definition von Refactoring!
6. Durch welche Maßnahmen wird beim Refactoring das Fehlerrisiko gering gehalten?
7. Warum sollten Refactoring und Erweiterungen klar getrennt werden?
8. Welche Rolle spielen Bad Smells beim Refactoring?
9. Ordnen Sie die Bad Smells *Feature Envy*, *Message Chain* und *Shotgun Surgery* Qualitätsmerkmalen zu!

4.2 Refactoring

Refactoring - Definition

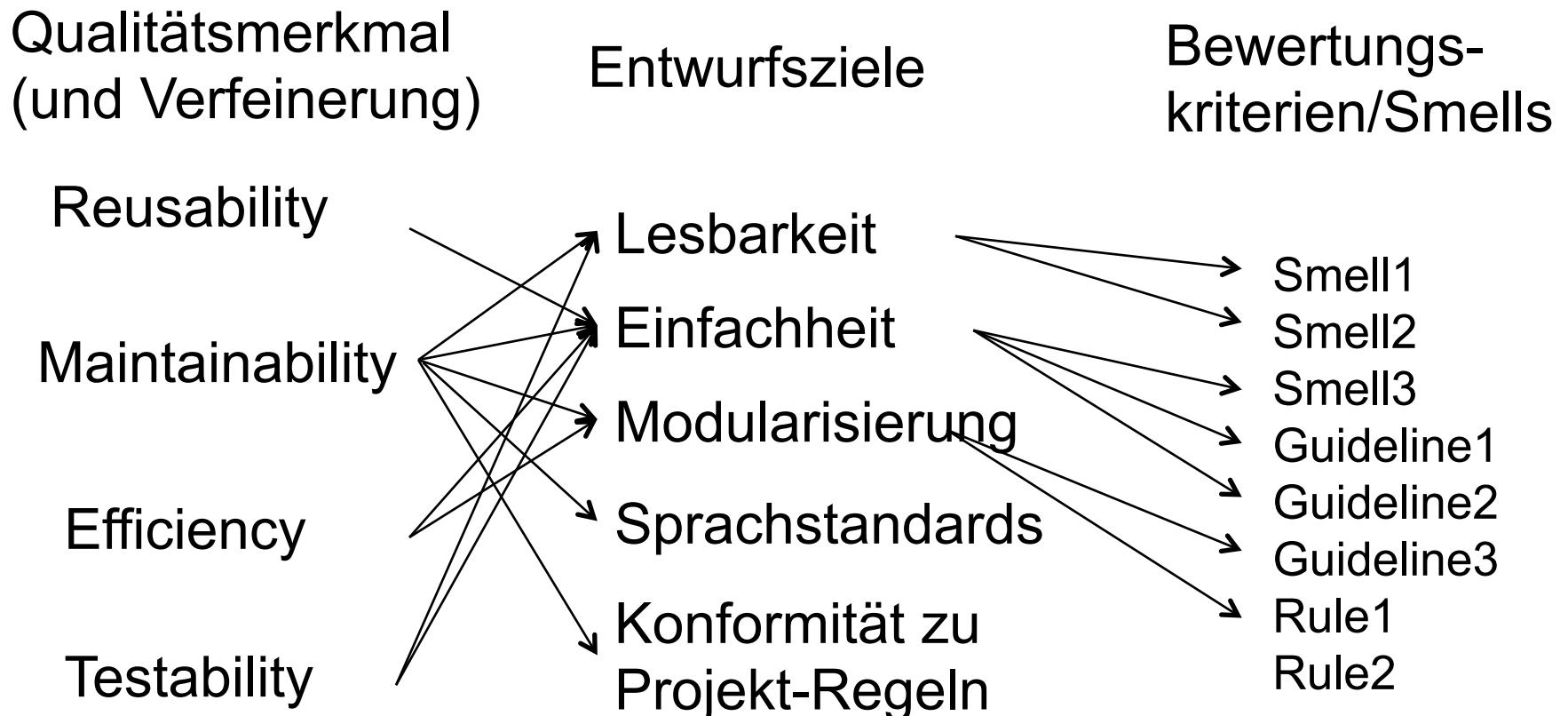
- A change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behaviour

Martin Fowler: Refactoring: Improving the Design of Existing Code, Addison Wesley 1999.

Inhalte online verfügbar: <http://www.refactoring.com>

Katalog auf Deutsch: <http://www.tutego.de/java/refactoring/catalog/>

Zuordnung von Qualitätsmerkmalen zu Bewertungskriterien und Bad Smells



Bad Smells: The Bloaters

Something that has grown so large that it cannot be effectively handled

- Long Method
- Large Class
- Primitive Obsession
- Long Parameter List
- Data Clumps

Qualitätsmerkmale:

- Wartbarkeit
 - Lesbarkeit
 - Einfachheit

Mäntylä, M. V. and Lassenius, C. "Subjective Evaluation of Software Evolvability Using Code Smells: An Empirical Study". Journal of Empirical Software Engineering, vol. 11, no. 3, 2006, pp. 395-431.
http://www.soberit.hut.fi/~mmantyla/ESE_2006.pdf,
<http://www.soberit.hut.fi/mmantyla/badcodessmellstaxonomy.htm>

Bad Smells: The Object-Orientation Abusers

Solution does not fully exploit the possibilities of object-oriented design

- Switch Statements
- Temporary Field
 - Member variables used only occasionally
- Refused Bequest
 - Subclass does not use much behaviour of super class
- Alternative Classes with Different Interfaces

Qualitätsmerkmale:

- Wartbarkeit
 - Verständlichkeit
 - Einfachheit
 - Erweiterbarkeit
 - Konformität

Bad Smells: The Change Preventers

Solution hinders changing or further developing the software by tangling and scattering

- Parallel Inheritance Hierarchies
- Divergent Change
 - a single class to be modified by many different changes
- Shotgun Surgery
 - need to modify many classes when making a single change

Qualitätsmerkmale:

- Wartbarkeit
 - Erweiterbarkeit
 - Einfachheit
 - Verständlichkeit

Bad Smells: The Dispensables

Something unnecessary that should be removed from the source code

- Lazy Class
- Data Class
- Duplicated Code
- Dead Code
- Speculative Generality

Qualitätsmerkmale:

- Wartbarkeit
 - Einfachheit
 - Verständlichkeit
 - Erweiterbarkeit
 - Testbarkeit

Bad Smells: The Couplers

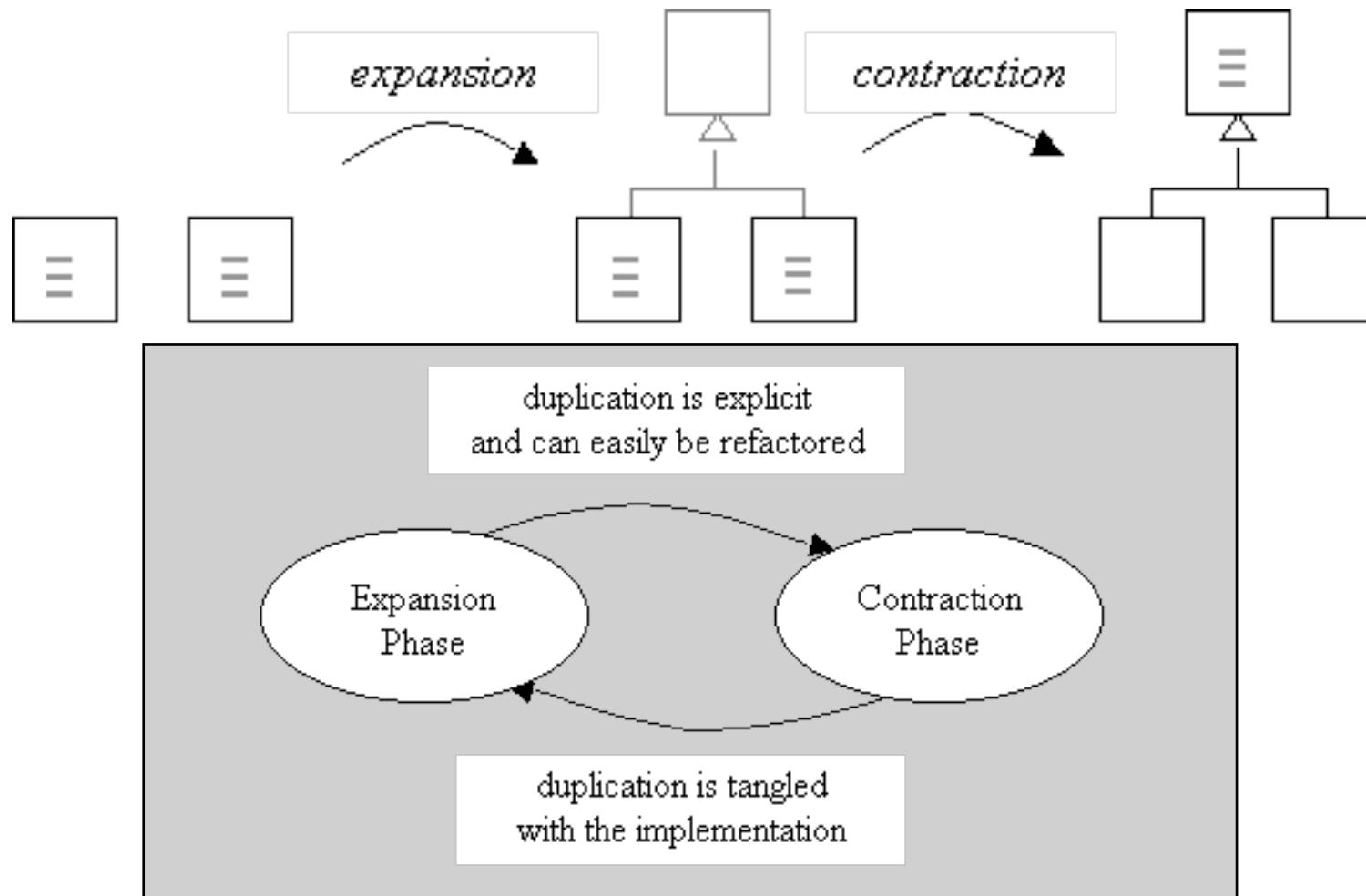
Smells representing inappropriate coupling

- Feature Envy
 - a method is more heavily coupled to other classes than the one that it is in
- Inappropriate Intimacy
- Message Chains
 - A.getB().getC().getD().getTheNeededData()
- Middle Man
 - trying to avoid high coupling with constant delegation

Qualitätsmerkmale:

- Wartbarkeit
 - Verständlichkeit
 - Einfachheit
 - Erweiterbarkeit
 - Testbarkeit

Ablauf Refactoring



Refactoring-Katalog: <http://www.refactoring.be/thumbnails.html>

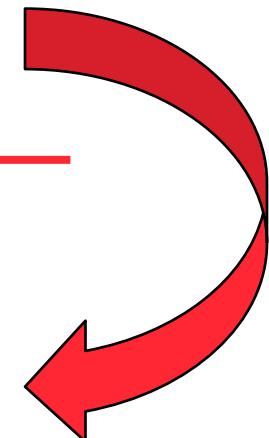
- Pull Up Feature
- Push Down Feature
- Encapsulate Feature
- **Inline Feature**
- Collapse Hierarchy
- Extract Hierarchy
- Introduce Indirection
- Inline Indirection
- Introduce Duplication
- Extract Variation
- Duplicate Feature
- Consolidate Abstraction
- Consolidate Interface
- Eliminate Duplication By Inheritance
- Eliminate Duplication By Composition
- Replace Singletons With Singleton
- Enable Substitution With Interfaces
- Generalize Behavior With Inheritance
- Specialize Behavior With Inheritance
- Split Implicit Layer
- Encapsulate Multiplicity
- Hide Implementation With Interface
- Trade Variation For Duplication
- Trade Duplication For Variation
- Replace Implementation Inheritance With Composition
- Replace Specializing Composition With Inheritance
- Separate Interface From Implementation
- Separate Module Dependencies With Adapter
- Hide Subsystem Complexity With Facade
- Enable Configurable Behavior With Plugin
- Enable Component Subcomponent Substitution
- Replace Template With Strategy
- Replace Concrete Interfacing Class With Explicit Interface
- Replace Concrete Template Class With Abstract Template

Methode extrahieren (Extract Method)

```
void printOwing() {  
    printBanner();  
    //print details  
    System.out.println ("name: " + _name);  
    System.out.println ("amount" + getOutstanding());  
}  
  
void printOwing() {  
    printBanner();  
    printDetails(getOutstanding());  
}  
  
void printDetails (double outstanding) {  
    System.out.println ("name: " + _name);  
    System.out.println ("amount" + outstanding);  
}
```

Smell: Codefragment kann zusammengefasst werden.
Aktion: Setze die Fragmente in eine Methode, deren Namen den Zweck kennzeichnet.

Übersetzung: <http://www.tutego.de/>



Replace Temp with Query

```
double basePrice = _quantity * _itemPrice;  
if (basePrice > 1000)  
    return basePrice * 0.95;  
else  
    return basePrice * 0.98;
```

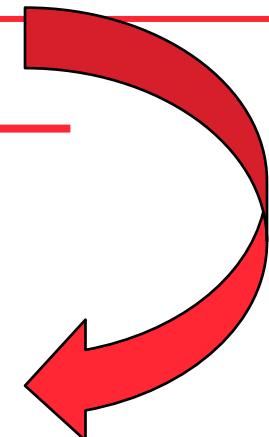
```
if (basePrice() > 1000)  
    return basePrice() * 0.95;  
else  
    return basePrice() * 0.98; ...  
double basePrice() {  
    return _quantity * _itemPrice;  
}
```

Smell:

temporary variable used to hold the result of an expression.

Action:

Extract the expression into a method. Replace all references to the temp with the expression. The new method can then be used in other methods.



Side Effects! See Book p.120

Quelle: <http://www.refactoring.com/>

Replace Method with Method Object

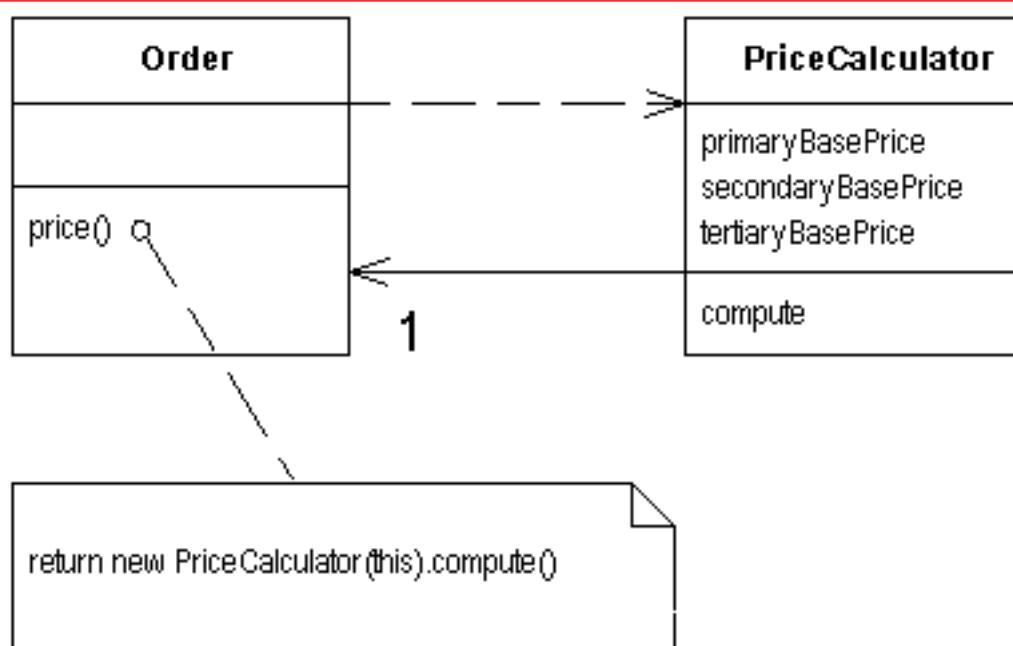
```
class Order...    double price() {  
    double primaryBasePrice;  
    double secondaryBasePrice;  
    double tertiaryBasePrice;  
    // long computation;  
    ...        }
```

Smell:

long method uses local variables, Extract Method not applicable.

Action:

Turn the method into its own object so that all the local variables become fields on that object. You can then decompose the method into other methods on the same object.



Comments: See Book p.135

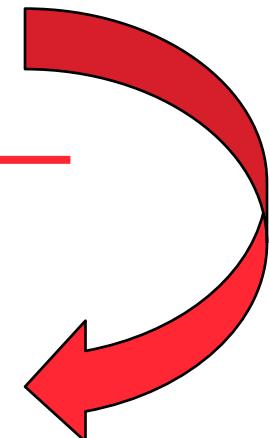
Quelle: <http://www.refactoring.com/>

Zerlege Bedingung (Decompose Conditional)

Smell: komplizierter Ausdruck
in einer Fallunterscheidung.
Aktion: Fasse die Bedingung
in einer Methode zusammen.

```
if (date.before(SUMMER_START) || date.after(SUMMER_END))  
    charge = quantity * _winterRate + _winterServiceCharge;  
else charge = quantity * _summerRate;
```

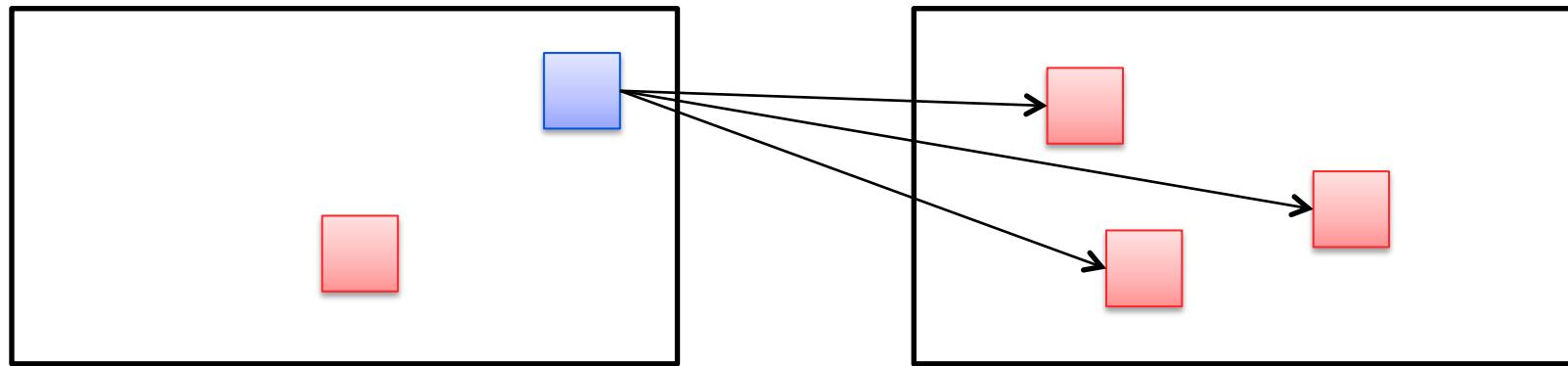
```
if (notSummer(date))  
    charge = winterCharge(quantity);  
else charge = summerCharge(quantity);
```



Übersetzung: <http://www.tutego.de/>

Smells im Kontext

Beispiel Neid (Feature Envy)

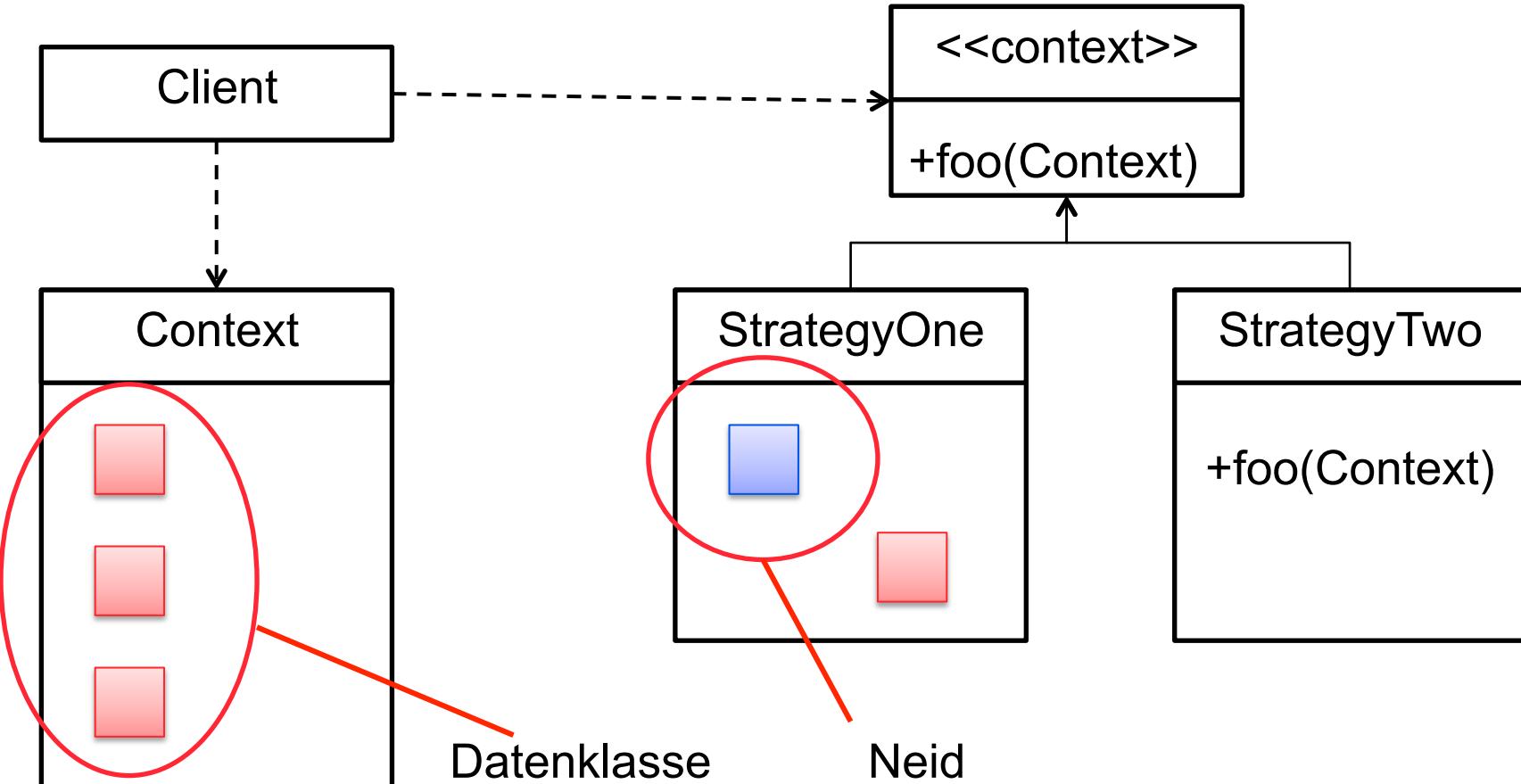


Kennzeichen Neid:

- **Einige** (3) fremde Attribute
- Verhältnis 0,25 → **unterhalb einem Drittel**
- Daten gehören zu **Einigen** (1) Klassen

[SpeicherJanke2010]

Strategy Pattern impliziert Neid



[SpeicherJanke2010]

Matthias Riebisch

Softwaretechnik - Wartung & Reengineering

23

Embedded DSLs

Architekturstile:

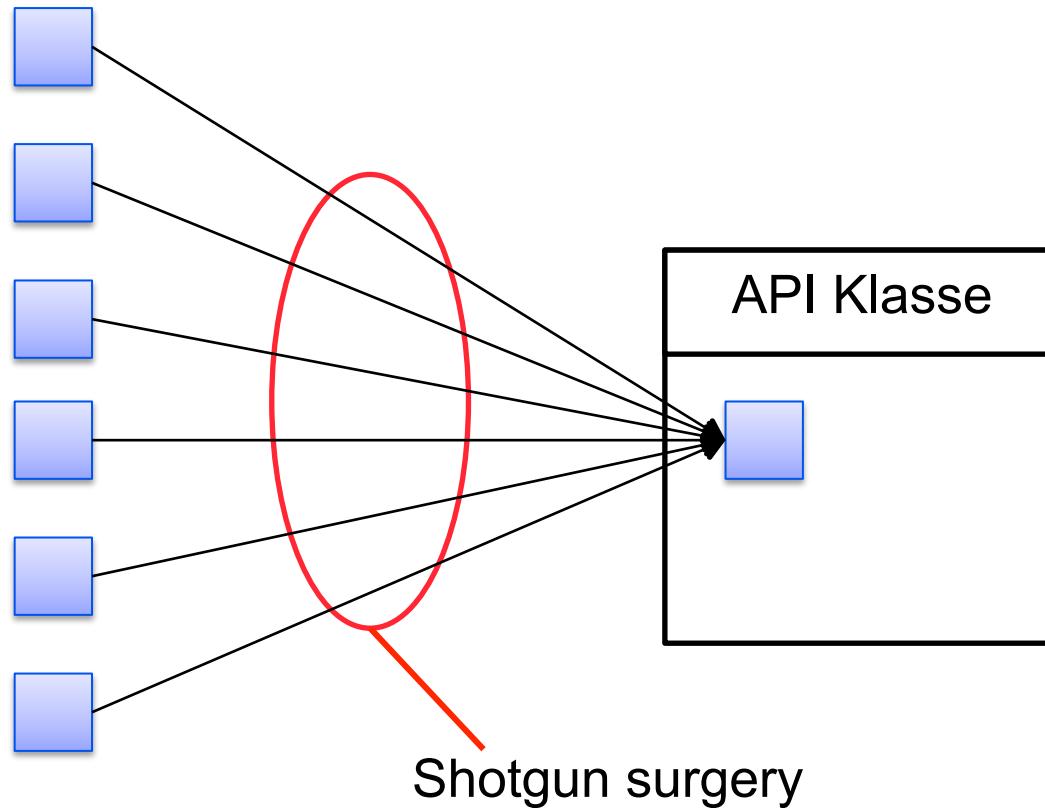
- Methodenketten
- Geschachtelte Funktionen
- Ziel: Lesbarkeit

Aber: charakteristische Eigenschaften von Smells:

- Methoden-Ketten
- Verletzung Law of Demeter
- Evtl. Gestreute Kopplung
- Evtl. Enge Kopplung

[SpeicherJanke2010]

APIs weisen Smells auf



[SpeicherJanke2010]