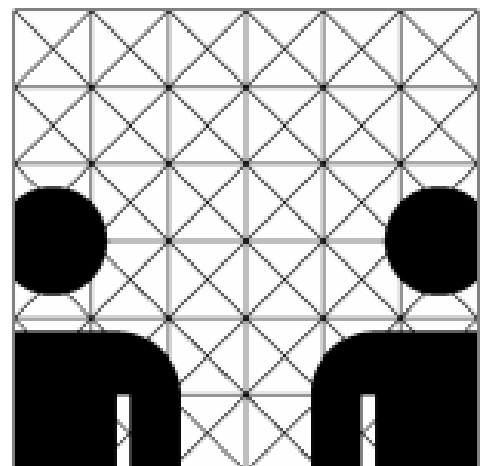


Prüfungsunterlagen
zur Lehrveranstaltung



Teil 2

Universität Hamburg
Fachbereich Informatik
SoSe 2012



Softwareentwicklung II

SE2

Objektorientierte Programmierung und Modellierung

Guido Gryczan
Axel Schmolitzky
Heinz Züllighoven
et al.

Teil 2

Verzeichnis der Folien

1. **Objektorientierte Modellierung – fachlich und technisch**
2. Literaturhinweise
3. Zur Erinnerung: Objektorientierte Aktivitäten
4. Objektorientierte Methoden – das Urgestein
5. Objektorientierte Methoden – die Anfänge
6. Objektorientierte Methoden – der UML-Einschnitt
7. Struktur- und verhaltensorientierte Herangehensweise
8. Zur Erinnerung: Modelle der Softwareentwicklung
9. Der Gegenstandsbereich
10. Anwendungsorientierte Analyse des Gegenstandsbereichs
11. Zusammenhänge im Gegenstandsbereich
12. Begriffe der anwendungsorientierten Analyse am Beispiel
13. Hilfsmittel bei der anwendungsorientierten Analyse des Gegenstandsbereichs
14. Modell des Gegenstandsbereichs
15. Modell des Gegenstandsbereichs: Beispiel Kooperationsbild
16. Modell des Gegenstandsbereichs: Beispiel Anwendungsfall-Diagramm der UML
17. Modell des Gegenstandsbereichs
18. Modell des Gegenstandsbereichs: Beispiel UML Begriffsdiagramm
19. Elemente des Begriffsmodells in Begriffsdiagrammen
20. Von der Analyse des Gegenstandsbereichs zum Modell des Anwendungssystems
21. Der Werkzeug & Material-Ansatz: eine anwendungsorientierte OO-Methode
22. Fachliche Gegenstände als Ausgangspunkt
23. Wir analysieren den fachlichen Umgang mit Gegenständen
24. Welche Arten von Gegenständen modellieren wir?
25. Verwendung von Begriffen bei der Modellierung
26. Die Entwurfsidee: Menschen bearbeiten Material mit Werkzeugen
27. Beispiel: der ursprüngliche Papier-Pausenplan als Material für das Werkzeug "Pausenplaner"
28. Beispiel: Das Werkzeug „Antragsbearbeiter“ stellt unterschiedliche Materialien dar
29. Vom fachlichen zum technischen Entwurf
30. Das Bibliotheksbeispiel
31. Das Bibliographie-System
32. Der fachliche strukturorientierte Entwurf (1)
33. Der fachliche strukturorientierte Entwurf (2)
34. Der fachliche strukturorientierte Entwurf (3)
35. Das Bibliothekssystem
36. Use Case: Beispiel Bibliothek
37. Use Case: Beispiel Ausleih-Subsystem
38. Der fachliche verhaltensorientierte Entwurf (1)
39. Der fachliche verhaltensorientierte Entwurf (2)
40. Der fachliche verhaltensorientierte Entwurf (3)
41. Vom fachlichen zum technischen Modell
42. CRC-Karten
43. Aufbau von CRC-Karten
44. Zur Erinnerung: Zusicherungen im Vertragsmodell
45. Zustandsdiagramm: Einführung
46. Zustandsdiagramm: Mögliches Protokoll eines Stacks
47. Zustandsdiagramm: Mögliches Protokoll eines Sammelguts im Bibliotheksbeispiel
48. Zusammenfassung & Ausblick

49. Strukturierung interaktiver Anwendungssysteme

- 50. Interaktive Softwaresysteme
- 51. „Reiche“ und „schlanke“ Systeme
- 52. Beispiele für Rich- und Thin-Clients
- 53. Einfache interaktive Softwaresysteme
- 54. Einfache Software: Qualität egal
- 55. Motivation für Entwurfsregeln: Qualität von Software
- 56. Hintergrund: verschiedene Qualitätsbegriffe
- 57. Externe und interne Qualität in der ISO 9126
- 58. Zusätzliche Motivation: Software verändert sich
- 59. Umfangreiche Entwurfsregeln: der WAM-Ansatz
- 60. Überschaubare Regeln: SE2-Entwurfsregeln
- 61. Exkurs: Stufen beim Lernen
- 62. Grundlegend: Trenne Fachlogik und Technik
- 63. Modellierung anwendungsfachlicher Klassen
- 64. Wiederholung: Modelle in der Softwareentwicklung
- 65. Zoom: Modell des Anwendungssystems
- 66. Modellierung technischer Klassen
- 67. Die SE2-Entwurfsregeln
- 68. Allgemeines zu den Entwurfsregeln
- 69. Materialien
- 70. Fachwerte
- 71. (Fachliche) Services
- 72. Werkzeuge
- 73. Werkzeugkonstruktion: Erste Schritte
- 74. Zusammenfassung SE2-Entwurfsregeln

75. Entwurfsmuster

- 76. Was ist unser (technisches) Problem?
- 77. Wir entdecken ein Entwurfsmuster
- 78. Ein „Ueberweiser“
- 79. Operationen an der Schnittstelle des Ueberweisers
- 80. Wir wollen den Ueberweiser interaktiv benutzen
- 81. Grundproblem der Konstruktion: Zwei Klassen müssen sich kennen.
- 82. Grundproblem der Konstruktion: Zwei Klassen müssen sich kennen.
- 83. Grundproblem der Konstruktion: Zwei Klassen müssen sich kennen.
- 84. Lösungsansätze für das Rückkopplungsproblem (1)
- 85. Einschub: UML-Sequenzdiagramme
- 86. Einschub: UML-Sequenzdiagramme
- 87. Lösungsansätze für das Rückkopplungsproblem (1)
- 88. Lösungsansätze für das Rückkopplungsproblem (2)
- 89. Lösungsansätze für das Rückkopplungsproblem (2)
- 90. Lösungsansätze für das Rückkopplungsproblem (3)
- 91. Lösungsansätze für das Rückkopplungsproblem (3)
- 92. Weiterentwicklung des Polling-Ansatzes: Der benachrichtigte Beobachter
- 93. Aufrufe, Ereignisse, Signale
- 94. Beobachten und beobachtet werden: Beobachter und Beobachtbar
- 95. Beobachter und Beobachtbar im Kontext des Überweisers
- 96. Dynamik: Registrieren eines Beobachters
- 97. Dynamik des Beobachter-Musters
- 98. Das Beobachter-Muster
- 99. Muster
- 100. Mikroarchitekturen: Muster in der Architektur
- 101. Mikroarchitekturen in objektorientierter Software
- 102. Beschreibung von Entwurfsmustern
- 103. Unterteilung von Entwurfsmustern
- 104. Zusammenhang Entwurfsmuster und Konstruktionsansatz
- 105. Zum Nutzen von Entwurfsmustern
- 106. Klassifizierung der GoF-Muster zusammengefasst
- 107. GoF: 23 Entwurfsmuster klassifiziert
- 108. Ein Klassiker der Softwaretechnik
- 109. Zusammenfassung Entwurfsmuster

110. Programmierung von Graphical User Interfaces (GUIs)

- 111. Motivation – Nichts ist beständiger als der Wandel
- 112. GUI-Toolkits zur Werkzeugkonstruktion
- 113. Hintergrund: AWT – Heavyweight Components
- 114. Hintergrund: Swing – Lightweight Components
- 115. Unsere erste Swing-Applikation
- 116. Swing-Komponenten: eine Auswahl -1-
- 117. Swing-Komponenten: eine Auswahl -2-
- 118. Auf dem Weg zu strukturierten Swing-Oberflächen: Komponenten erzeugen und schachteln
- 119. Hierarchischer Aufbau einer Swing-Oberfläche
- 120. Top-Level Container
- 121. Struktur eines JFrame
- 122. Werkzeugkonstruktion: Nächste Schritte
- 123. Kontrollfluss in Anwendungen – Traditionelles Prinzip: Eingabe, Verarbeitung, Ausgabe
- 124. Eingabe, Verarbeitung, Ausgabe – Merkmale
- 125. Welche Art von Systemen wollen wir bauen (können)?
- 126. Merkmale reaktiver Software
- 127. Reaktive Programmierung – die grundlegende Idee
- 128. Ereignisverarbeitung mit Ereignissen (Events)
- 129. Anbindung der Applikation an die Oberfläche – das Prinzip
- 130. Das Konzept der Listener
- 131. Das Konzept der Listener (II)
- 132. Das Konzept der Listener im schematischen Ablauf
- 133. Java Spezial: Listener mit anonymen inneren Klassen
- 134. Anonyme innere Klassen
- 135. Komponenten und Listener: Beispiele
- 136. Zur Vervollständigung: Events, Listener
- 137. Events und Listener: Weitere Beispiele
- 138. Event / ActionListener zusammengefasst
- 139. Werkzeugkonstruktion: Ereignisverarbeitung
- 140. Layout festlegen
- 141. Container mit Layout
- 142. Layout-Manager
- 143. Verschiedene Layout-Manager
- 144. Layouts im Beispiel: FlowLayout
- 145. Werkzeugkonstruktion: Layout
- 146. Zusammenfassung GUI-Programmierung
- 147. Zusammenfassung SE2-Entwurfsregeln

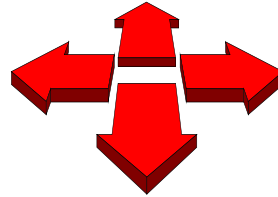
148. Übersicht Klassenentwurf

- 149. Motivation: Qualität von Software
- 150. Qualität von Klassenentwürfen
- 151. Wie kommen wir zu einem guten Entwurf?
- 152. Leitbild: Entwurf nach Zuständigkeiten
- 153. Wiederholung: Trenne Fachlogik und Technik
- 154. Kopplung, objektorientiert
- 155. Lose Kopplung
- 156. Beispiel für implizite Kopplung
- 157. Geeignete Schnittstellen wählen
- 158. Konkretisierung: Verwende Interfaces
- 159. Beispiel: geschachtelte GUI
- 160. Lösung: Abhängigkeit per Interface explizit machen
- 161. Entwurfsentscheidungen kapseln
- 162. Geheimnisprinzip
- 163. Zyklen vermeiden
- 164. Beispiel: keine Paket-Zyklen durch Entwurfsregeln
- 165. Law of Demeter, Original
- 166. Lange Aufrufzeilen = hohe Kopplung
- 167. Law of Demeter, heute
- 168. Law of Demeter, Beispiele

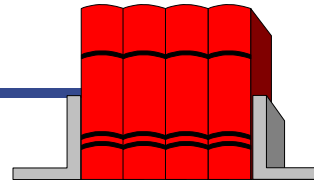
- 169. Kohäsion, objektorientiert
- 170. Kohäsion von Methoden
- 171. Kohäsion von Klassen
- 172. Geeignete Bezeichner wählen
- 173. Englische oder deutsche Begriffe im Quelltext?
- 174. Englisch oder Deutsch: kontextabhängig!
- 175. Deutsch und Englisch gemischt?
- 176. Konkretisierung: möglichst genaue Bezeichner wählen
- 177. Code-Duplizierung vermeiden
- 178. Große Einheiten vermeiden
- 179. Zusammenfassung Klassenentwurf

Objektorientierte Modellierung – fachlich und technisch

- Einordnung objektorientierter Methoden
- Modelle der Softwareentwicklung
- Fachliche Modellierung
- Ein Bibliotheksbeispiel
 - strukturorientiert
 - verhaltensorientiert
- Technische Modellierung



Literaturhinweise



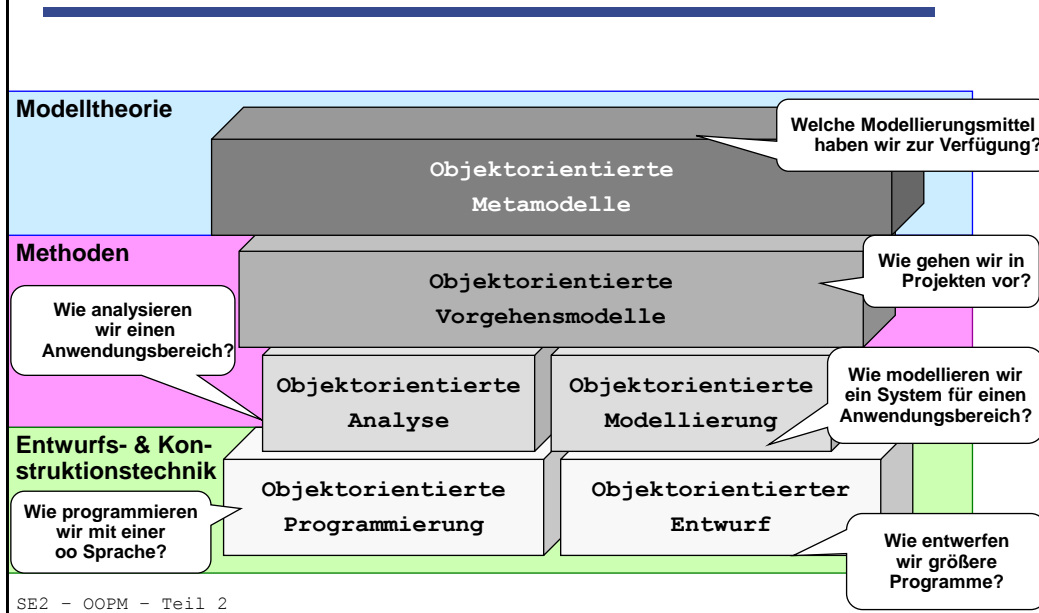
Heinz **Züllighoven** et al.: *Object-Oriented Construction Handbook*.
dpunkt-Verlag, 2004.

[Einige Begriffe und Konzepte für diesen Teil.]

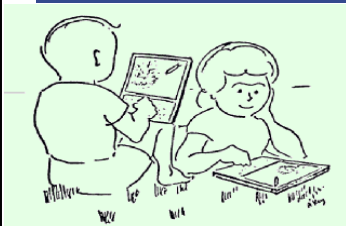
Martin **Hitz**, Gerti **Kappel**, Elisabeth **Kapsammer**, Werner
Retschitzegger: *UML @ Work. Objektorientierte Modellierung mit
UML 2. 3.*, aktualisierte und überarbeitete Auflage, dpunkt.verlag,
2005.

[Mehr zu den UML-Modellen dieses Teils. Siehe auch die
Materialien zu SE2]

Zur Erinnerung: Objektorientierte Aktivitäten



Objektorientierte Methoden – das Urgestein



Die Idee hinter Smalltalk:

Software repräsentiert Arbeitsmaterial

Alan Kay:

"Computing is Simulation"

"Do not automate the work you are engaged in, only the materials."

- Reactive engine (1969)
- Personal dynamic media
- The Dynabook

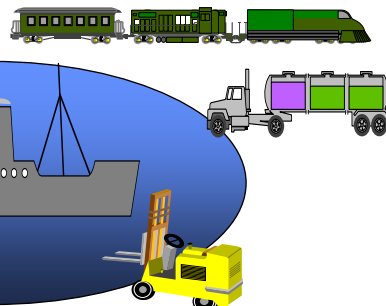
Die Motivation von Simula:

Objekte simulieren Dinge

Ole-Johan Dahl, Kristen Nygaard:

Anwendungen im Bereich
Simulation und Operations
Research

- Simula I (1964)
- Simula 67



Objektorientierte Methoden – die Anfänge

- Seit Simula (1967) wurden immer wieder Papiere zur objektorientierten Programmierung veröffentlicht. Die Themen "Analyse" und "Entwurf" wurden nur am Rande behandelt.
- Objektorientierte Methoden, die sich mit dem gesamten Software-Entwicklungsprozess beschäftigten, wurden erst in den 80er-Jahren vorgestellt.
- Forschung und Praxis wurden geprägt von:
 - Grady Booch: *Object-Oriented Design* (1982)
 - Bertrand Meyer: *Object-Oriented Software Construction* (1988)
 - Sally Schlaer, Stephen J. Mellor: *Object-Oriented Systems Analysis* (1988)
 - Peter Coad: *Object-Oriented Analysis and Object-Oriented Design* (1990/991)
 - Rebecca Wirfs-Brock et al.: *Designing Object-Oriented Software* (1990)
 - James Rumbaugh et al.: *Object-Oriented Modeling and Design* (1991)
 - Ivar Jacobson: *Object-Oriented Software Engineering* (1991)

SE2 – OOPM – Teil 2

5

Objektorientierte Methoden – der UML-Einschnitt

- Die *UML* reduzierte die Vielfalt der Methodenlandschaft
 - Grady Booch, James Rumbaugh: *Unified Method* (1994)
 - Booch, Jacobson, Rumbaugh: *UML - the Unified Modeling Language*
 - Jacobson, Booch, Rumbaugh: *The Unified Software Development Process (UP)*
 - Mary Loomis, Jim Odell et al. (im Auftrag der Object Management Group - OMG): *UML 1.0* (1997)
- Aktuell erscheinen im wesentlichen Arbeiten zu einzelnen Aspekten der Objektorientierung wie
- Agile Vorgehensweisen
 - Entwurfsmuster
 - Testen und Qualitätssicherung

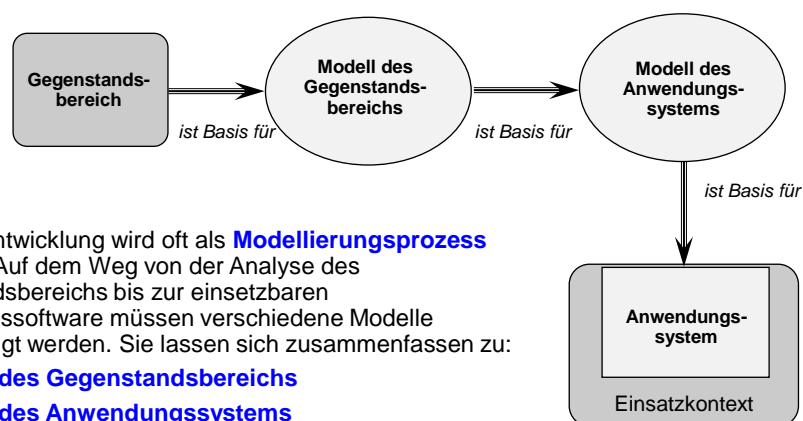
SE2 – OOPM – Teil 2

6

Struktur- und verhaltensorientierte Herangehensweise

- Verallgemeinernd können wir feststellen, dass es zwei grundlegende Herangehensweisen bei den objektorientierten Methoden gibt:
 - **Verhaltensorientiert:**
Der Einsatzkontext wird in seiner Dynamik betrachtet. Der mögliche und sinnvolle Umgang mit den fachlichen Gegenständen eines Gegenstandsbereichs wird analysiert.
 - **Strukturorientiert:**
Der Einsatzkontext wird eher strukturell betrachtet. Die fachlichen Gegenstände werden auf ihre relevanten Daten untersucht, die im IT-System gespeichert werden sollen.
- Verhaltens- oder strukturorientierte Vorgehensweisen bestimmen Analyse, Modellierung und Konstruktion. Sie führen zu unterschiedlichen Anwendungssystemen mit verschiedenen Benutzungsmodellen.
- Wir stellen im Folgenden vorrangig eine verhaltensorientierte Vorgehensweise vor, diskutieren aber auch einen strukturorientierten Entwurf.

Zur Erinnerung: Modelle der Softwareentwicklung

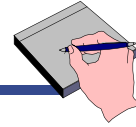


- Software-Entwicklung wird oft als **Modellierungsprozess** betrachtet. Auf dem Weg von der Analyse des Gegenstandsbereichs bis zur einsetzbaren Anwendungssoftware müssen verschiedene Modelle berücksichtigt werden. Sie lassen sich zusammenfassen zu:
 - **Modell des Gegenstandsbereichs**
 - **Modell des Anwendungssystems**
- Objektorientierte Methoden unterscheiden sich auch darin, ob und wie sie diese Modelle erstellen.

Der Quellcode ist Teil des Modells des Anwendungssystems; das ablaufende Programm ist Teil des Anwendungssystems.



Der Gegenstandsbereich

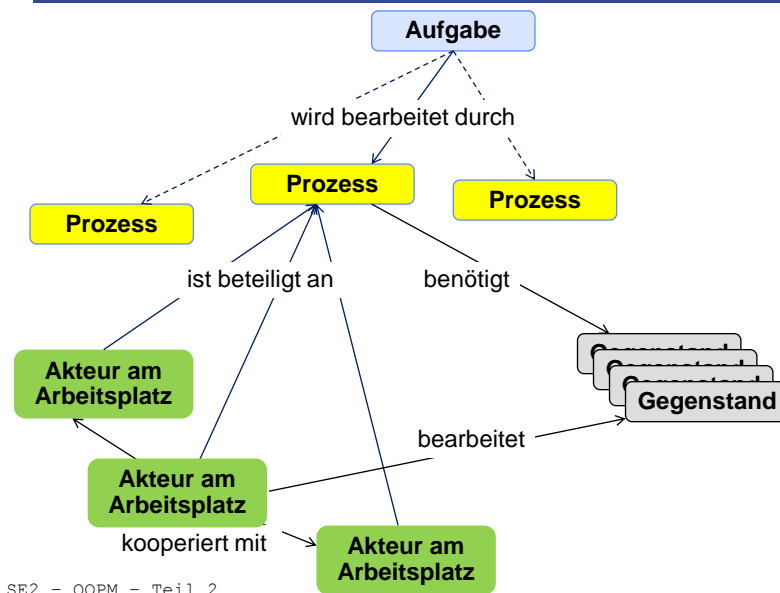


- Die **Analyse** des **Gegenstandsbereichs** ist Ausgangsbasis für das **fachliche Modell** des Gegenstandsbereichs.
- Der Gegenstandsbereich kann eine **Organisation**, ein **Bereich** innerhalb einer Organisation oder eine **Arbeitsplatz** sein.
- Der Gegenstandsbereich muss weit genug gefasst werden, um die **relevanten fachlichen Zusammenhänge** für die Konstruktion von Modellen zu verstehen.
- In einem Gegenstandsbereich wird typischerweise auch eine entsprechende **Anwendungsfachsprache** verwendet.

Anwendungsorientierte Analyse des Gegenstandsbereichs

- **Anwendungsorientierung** bedeutet für die Analyse des Gegenstandsbereichs, dass die **Entwickler** die an den einzelnen Arbeitsplätzen im Gegenstandsbereich anfallenden **fachlichen Aufgaben** verstehen müssen.
- Um die **fachlichen Aufgaben** zu identifizieren und zu verstehen, analysieren die Entwickler die **Prozesse**, durch die die fachlichen Aufgaben erledigt werden.
- Um die Prozesse zu verstehen, betrachten die Entwickler die Art und Weise, wie im Rahmen dieser Prozesse mit **Gegenständen gearbeitet** wird.
- Die Analyse betrachtet den **einzelnen Arbeitsplatz** und die **Kooperation zwischen Arbeitsplätzen**.
- Anwendungsorientierte Analyse führt zu einem **Modell des Gegenstandsbereichs in den Begriffen der Anwendungsfachsprache** und hilft beim fachlichen Modell des Anwendungssystems

Zusammenhänge im Gegenstandsbereich



11

Begriffe der anwendungsorientierten Analyse am Beispiel



Der **Gegenstandsbereich** ist meist eine Organisation oder eine Abteilung.



Aufgaben werden in **Arbeitsprozessen** von verschiedenen **Personen** erledigt.



Personen **bearbeiten** am **Arbeitsplatz** die entsprechenden **Unterlagen** oder **Materialien**. Dabei kooperieren sie.



Arbeitsmaterial wird mit geeigneten **Mitteln** bearbeitet.

12

Hilfsmittel bei der anwendungsorientierten Analyse des Gegenstandsbereichs



- Um den Gegenstandsbereich zu verstehen, sind verschiedenen **Analysetechniken** nützlich
 - Interviews mit Anwendungsexperten
 - Rollenspiele
 - „Teilnehmende Beobachtung“ (Hospitieren) vor Ort
 - Ethnographische evtl. videogestützte Analysen

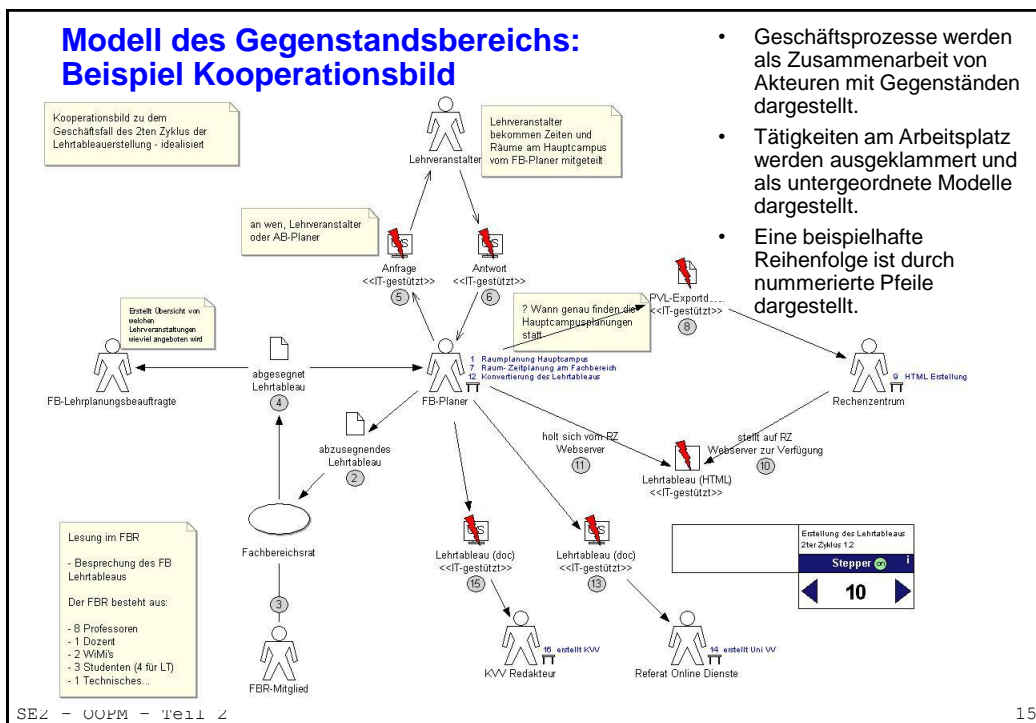


Modell des Gegenstandsbereichs



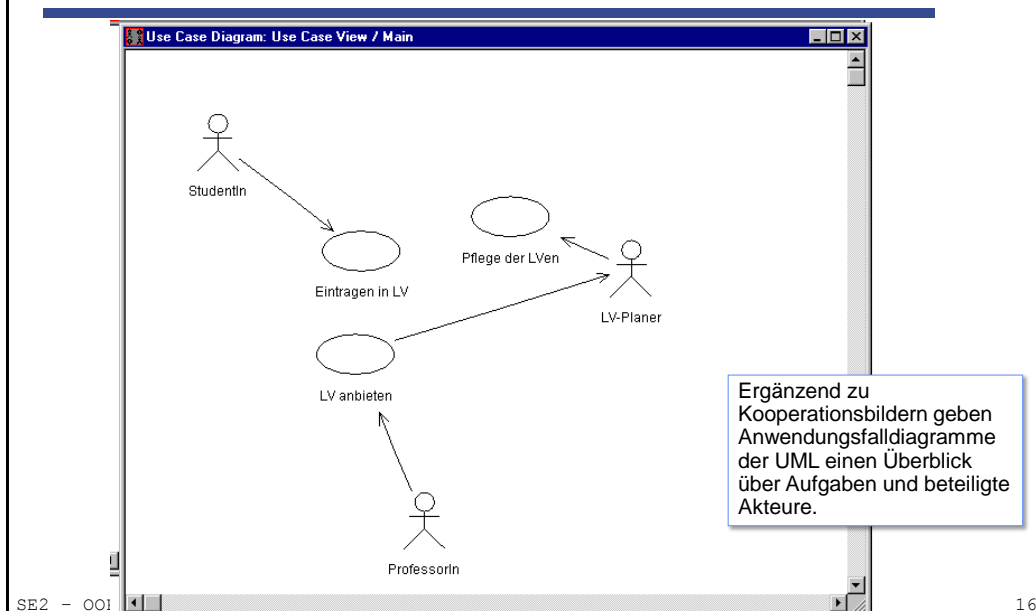
- Das **Modell des Gegenstandsbereichs** stellt anwendungsorientiert diejenigen Aspekte des Gegenstandsbereichs dar, die durch ein Anwendungssystem unterstützt werden sollen. Es wird also schon mit Blick auf den Einsatzkontext entworfen.
- Das Modell orientiert sich an den **Aufgaben** und den dabei relevanten **Gegenständen** mit ihren **Umgangsformen**. Es zeigt die **Prozesse** und die damit verbundenen **Begriffe**.
- Das Modell des Gegenstandsbereichs kann in verschiedenen Formen **repräsentiert** werden, z.B.
 - als **Prozessmodell** durch Kooperationsbilder, Use Cases oder Aktivitätsdiagramme.
 - als **Begriffs-** oder **Objektmodell** durch Objekt- oder Klassendiagramme der UML oder als Glossareinträge.

Teil 2: Modellierung und Entwurf interaktiver Softwaresysteme: Muster, Regeln, Rahmenwerke



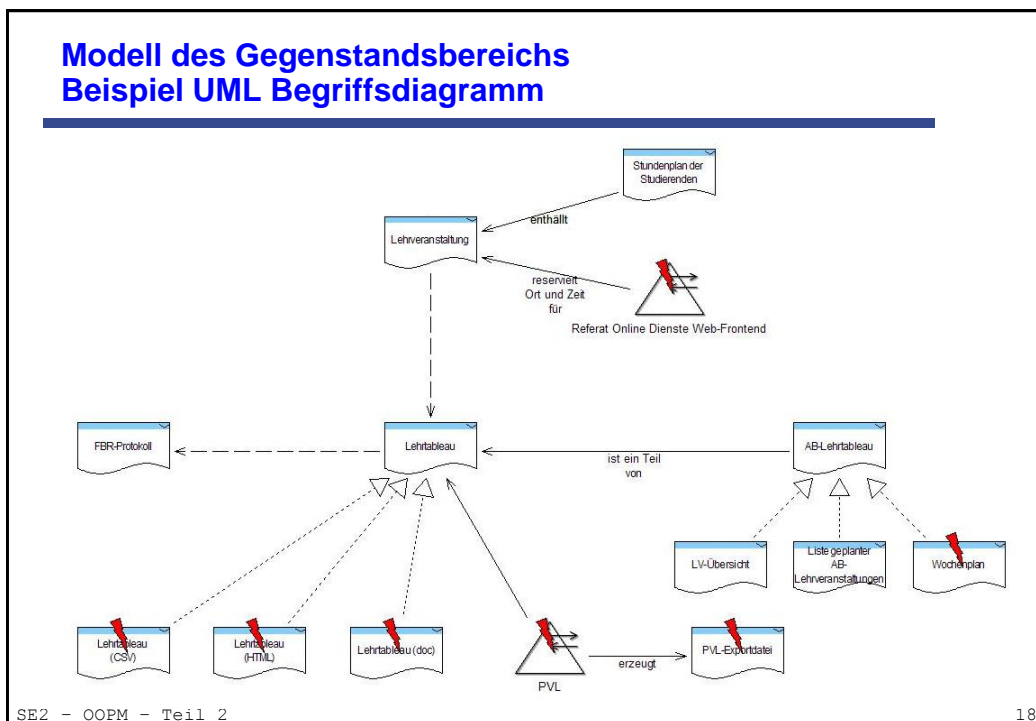
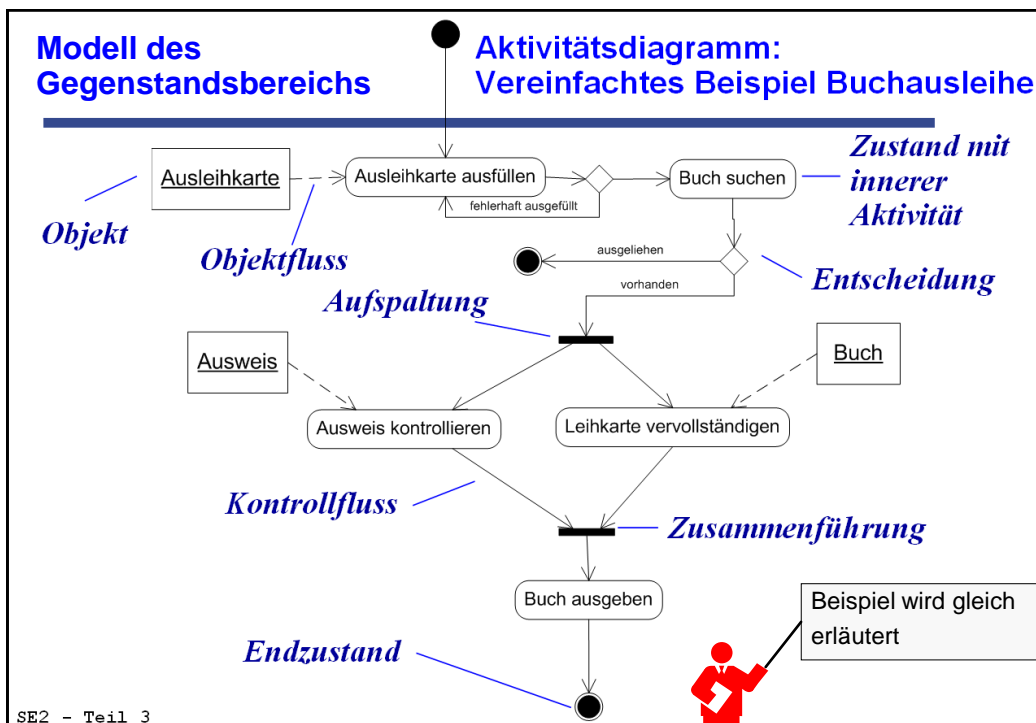
15

Modell des Gegenstandsbereichs: Beispiel Anwendungsfall-Diagramm der UML

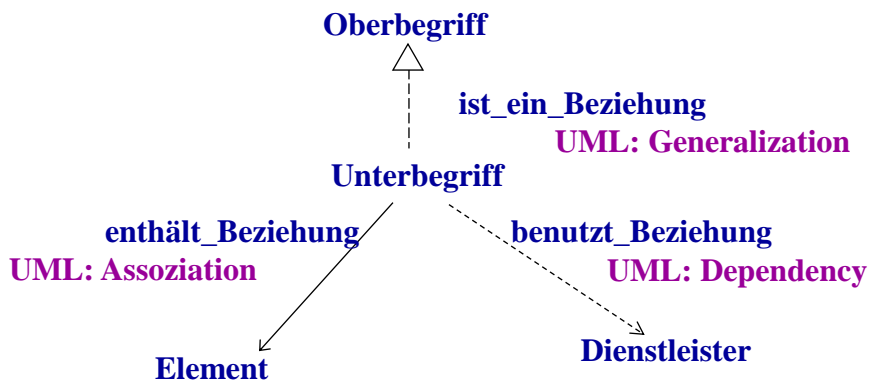


16

Teil 2: Modellierung und Entwurf interaktiver Softwaresysteme: Muster, Regeln, Rahmenwerke



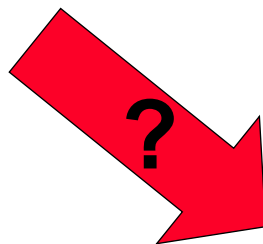
Elemente des Begriffsmodells in Begriffsdiagrammen



SE2 - OOPM - Teil 2

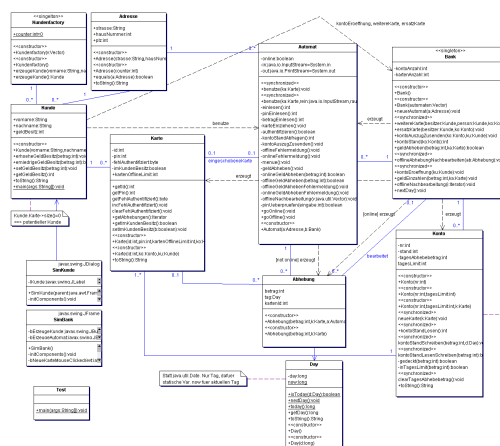
19

Von der Analyse des Gegenstandsbereichs zum Modell des Anwendungssystems



Objektorientierte Methoden geben
Anleitung zur Lösung des Problems:

Wie komme ich von der Analyse des Gegenstandsbereichs zum Entwurf eines Anwendungssystems?

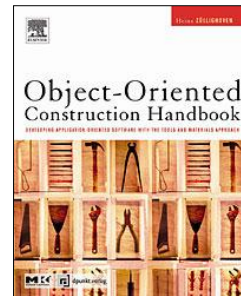


SE2 - OOPM - Teil 2

20

Der Werkzeug & Material-Ansatz: eine anwendungsorientierte OO-Methode

- Der **Werkzeug & Material-Ansatz** wurde über viele Jahre in Forschung und Praxis entwickelt und ist eine wesentliche methodische Grundlage des Arbeitsbereichs Softwaretechnik.
- Er gibt eine Anleitung zur Entwicklung interaktiver Softwaresysteme.
- Grundprinzipien sind
 - Anwendungsorientierung
 - Strukturähnlichkeit zwischen Gegenstandsbereich und Anwendungssystem
- Im Folgenden erläutern wir Herangehensweisen und Techniken, die für die Übungen relevant sind.

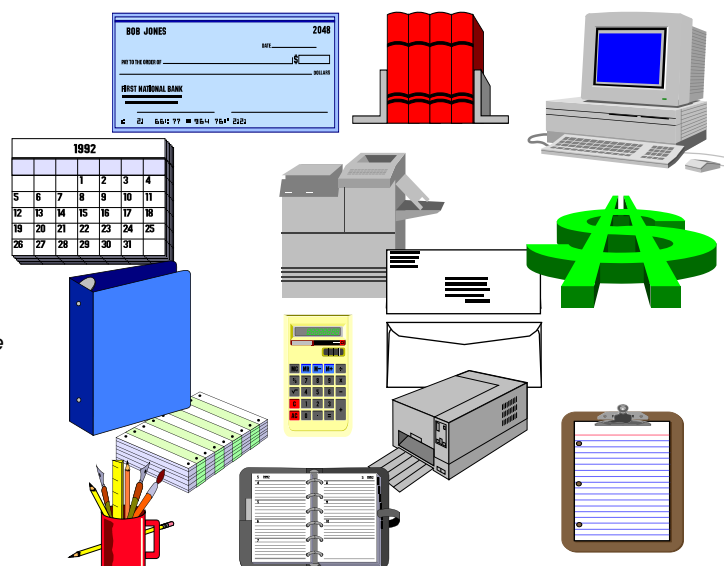


Züllighoven, H., et. al., *Object-Oriented Construction Handbook*, dpunkt.verlag/Copublication with Morgan-Kaufmann, Oktober 2004, 544 Seiten,
<http://www.dpunkt.de/buch/3-89864-254-2.html>

Fachliche Gegenstände als Ausgangspunkt

- Wir orientieren uns an den Gegenständen, die zur Erledigung der fachlichen Aufgaben verwendet werden.
- Diese Gegenstände sind ein guter Ausgangspunkt zur Beantwortung der Frage:

Wie können Arbeitsprozesse am Arbeitsplatz sinnvoll unterstützt werden?



Wir analysieren den fachlichen Umgang mit Gegenständen

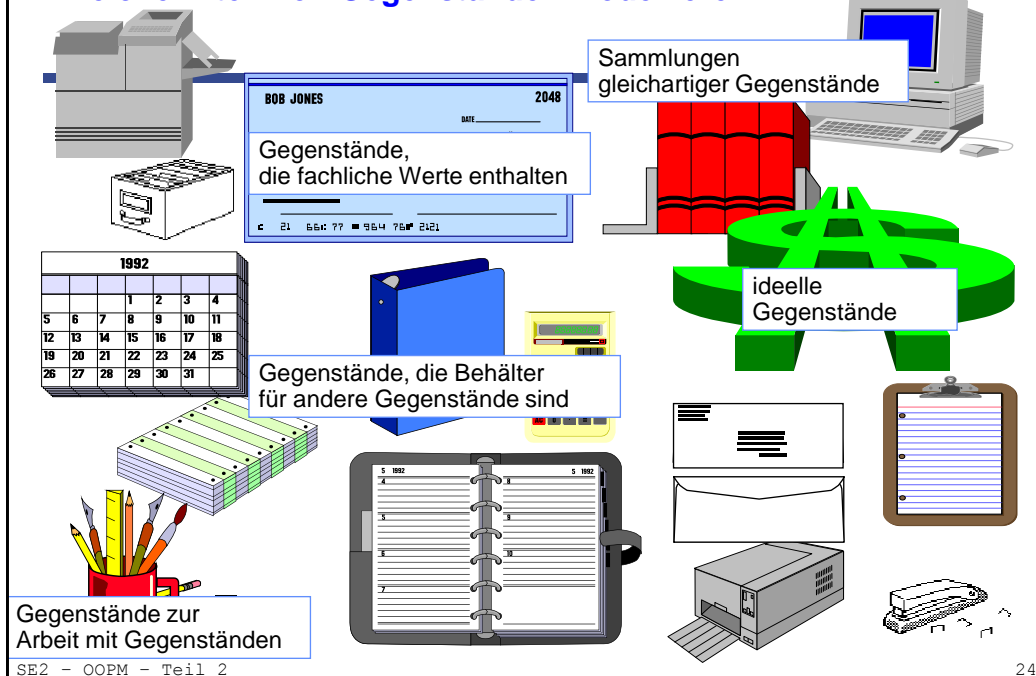
- **Umgangsformen:**

Die **Art und Weise, wie** mit **Gegenständen** im Rahmen der verschiedenen **Aufgaben gearbeitet** wird.

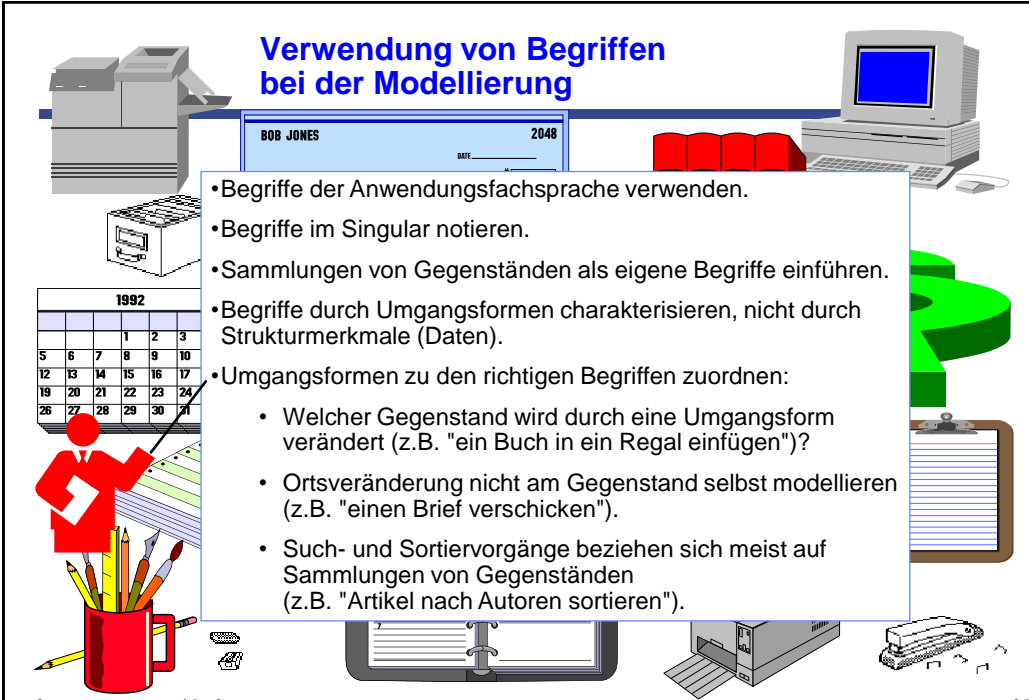
- **Wir untersuchen:**

- Welche **Informationen** werden an den Gegenständen "abgelesen"?
- Welche **Veränderungen** werden an den Gegenständen vorgenommen und welche **Aktionen** werden ausgelöst, ohne dass sie zerstört oder in andersartige Gegenstände transformiert werden?

Welche Arten von Gegenständen modellieren wir?



Verwendung von Begriffen bei der Modellierung



- Begriffe der Anwendungsfachsprache verwenden.
- Begriffe im Singular notieren.
- Sammlungen von Gegenständen als eigene Begriffe einführen.
- Begriffe durch Umgangsformen charakterisieren, nicht durch Strukturmerkmale (Daten).
- Umgangsformen zu den richtigen Begriffen zuordnen:
 - Welcher Gegenstand wird durch eine Umgangsform verändert (z.B. "ein Buch in ein Regal einfügen")?
 - Ortsveränderung nicht am Gegenstand selbst modellieren (z.B. "einen Brief verschicken").
 - Such- und Sortiervorgänge beziehen sich meist auf Sammlungen von Gegenständen (z.B. "Artikel nach Autoren sortieren").

SE2 – OOPM – Teil 2 25

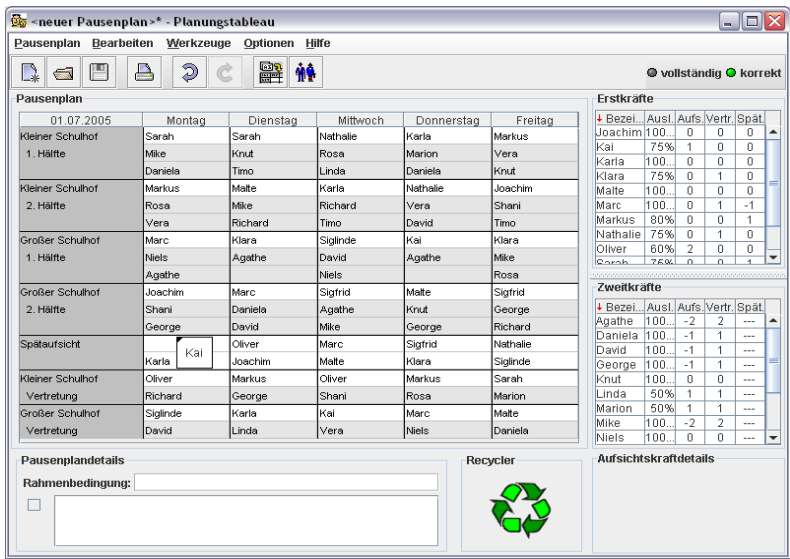
Die Entwurfsidee: Menschen bearbeiten Material mit Werkzeugen

- **Arbeiten** bedeutet oft, dass Menschen einen Arbeitsgegenstand mit geeigneten Werkzeugen bearbeiten. Dies gilt nicht nur im Handwerk.
- **Materialien** sind also Gegenstände, die im Rahmen einer Aufgabe Teil des Arbeitsergebnisses werden. Sie werden durch Werkzeuge und Automaten bearbeitet und verkörpert fachliche Konzepte. Viele Eigenschaften vorhandener Arbeitsgegenstände lassen sich sinnvoll auf Softwarematerialien übertragen.

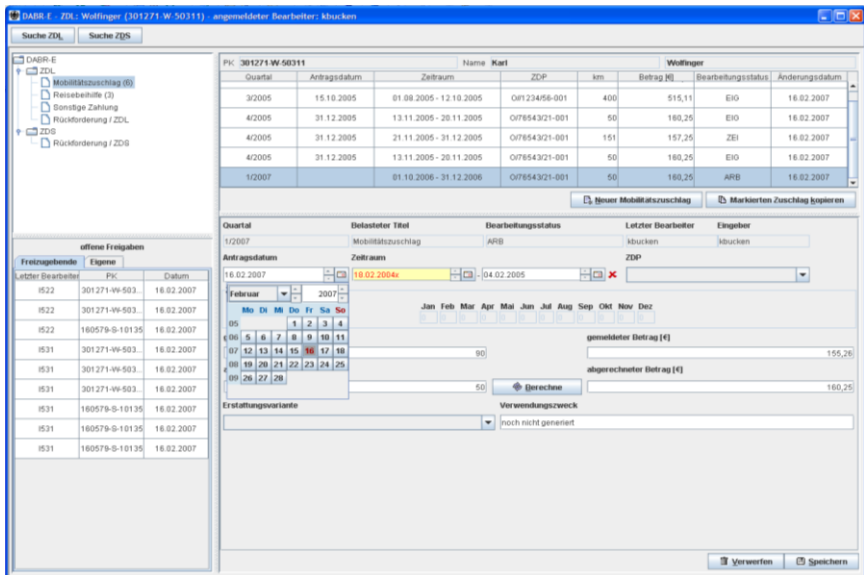


- **Werkzeuge** sind Gegenstände, mit denen Menschen im Rahmen einer Aufgabe Materialien verändern oder sondieren. Sie eignen sich meist für verschiedene Zwecke und unterschiedliche Materialien. Sie müssen geeignet gehandhabt werden. Werkzeuge vergegenständlichen wiederkehrende Arbeitshandlungen. Eine direkte Abbildung von (manuellen) Werkzeugen in Software ist selten sinnvoll.

Beispiel: der ursprüngliche Papier-Pausenplan als Material für das Werkzeug "Pausenplaner"



Beispiel: Das Werkzeug „Antragsbearbeiter“ stellt unterschiedliche Materialien dar



Vom fachlichen zum technischen Entwurf

- Mit dem Modell des Gegenstandsbereichs (Prozess- und Begriffsmodell) haben wir gute Voraussetzungen für den **Entwurf** des Modells eines objektorientierten Anwendungssystems.
- Objektorientiert können wir die Elemente des fachlichen Entwurfs technisch einfach umsetzen.

fachliches Modell	softwaretechnisches Modell
Gegenstand	Objekt
Umgangsform	Operation/Typ
Begriff	Klasse
Generalisierung, Spezialisierung	Sub-, Supertyp, Vererbung
Komposition	Aggregation, Assoziation
Begriffshierarchie	Typ-, Klassenhierarchie

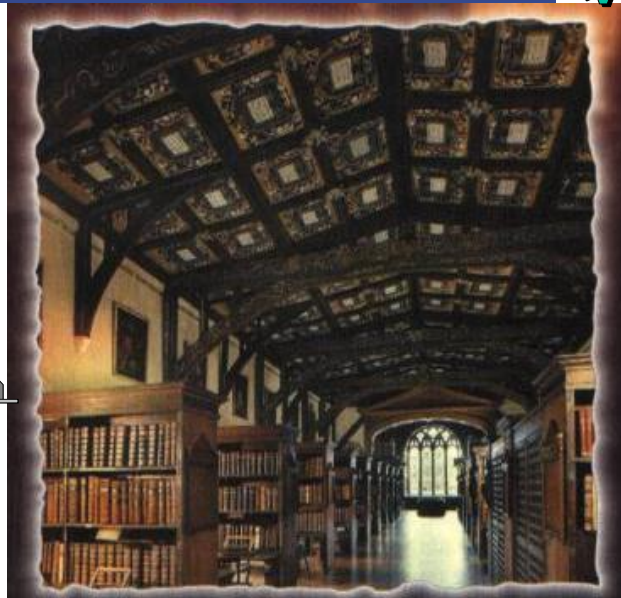
- Wir verdeutlichen anschließend die verhaltens- und die strukturorientierte Vorgehensweise an einem Beispiel.

Das Bibliotheksbeispiel

- Thema: Eine Bibliothek
- Strukturorientiertes Beispiel:
Ein Bibliographie-System
- Verhaltensorientiertes Beispiel:
Eine Bibliothekssystem zur Unterstützung der Aufgaben einer Bibliothek

– Budde, Kuhlenskamp, Sylla, Züllighoven: *Bib – ein Bibliographie-System*. In: H.J. Hoffmann (Hrsg.): *Smalltalk verstehen und anwenden*. München, Hanser, 1987.

– Das verhaltensorientierte Bibliotheksbeispiel wurde über viele Jahre bei SWT in LVs (z.B. STE) verwendet.



Das Bibliographie-System

- **Die Aufgabenstellung:**
Ein Bibliographie-System ist ein Hilfsmittel, um Übersichten über Texte (z.B. Bücher oder Artikel) zu halten. Es enthält bibliographische Einträge ... Jeder bibliographische Eintrag enthält Daten, die einen Text identifizieren.
- **Vorgehensweise bei Analyse und Entwurf:**
 - Welches System könnte nützlich sein?
 - Wie können die relevanten fachlichen Daten im System verwaltet werden?



SE2 – OOPM – Teil 2

31

Der fachliche strukturorientierte Entwurf (1)

Buch	Techn. Report	Zeitschrift
Kürzel	Kürzel	Kürzel
Autorenliste	Autorenliste	Zeitschrift
Titel	Titel	Jahrgang
Verlag	Reportangaben	Nummer
Ort	Institution	Verlag
Jahr	Ort	Ort
	Jahr	Jahr

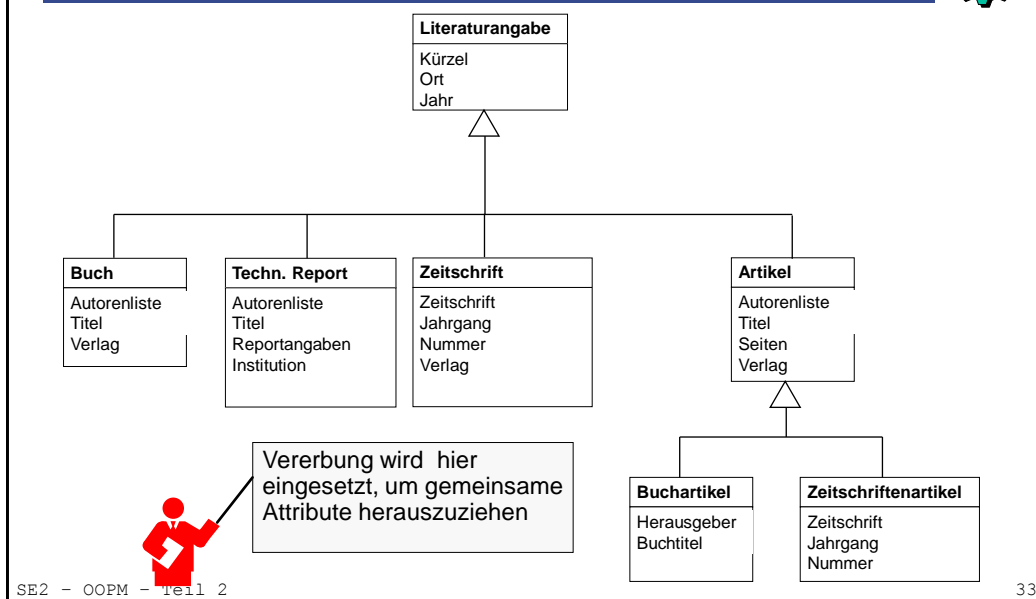
Buchartikel	Zeitschriftenartikel
Kürzel	Kürzel
Autorenliste	Autorenliste
Titel	Titel
Seiten	Seiten
Herausgeber	Zeitschrift
Buchtitel	Jahrgang
Verlag	Nummer
Ort	Verlag
Jahr	Ort
	Jahr

- **Leitfrage bei der strukturorientierten Analyse:**
Welche Daten charakterisieren die Gegenstände eines Bibliographie-Systems?

SE2 – OOPM – Teil 2

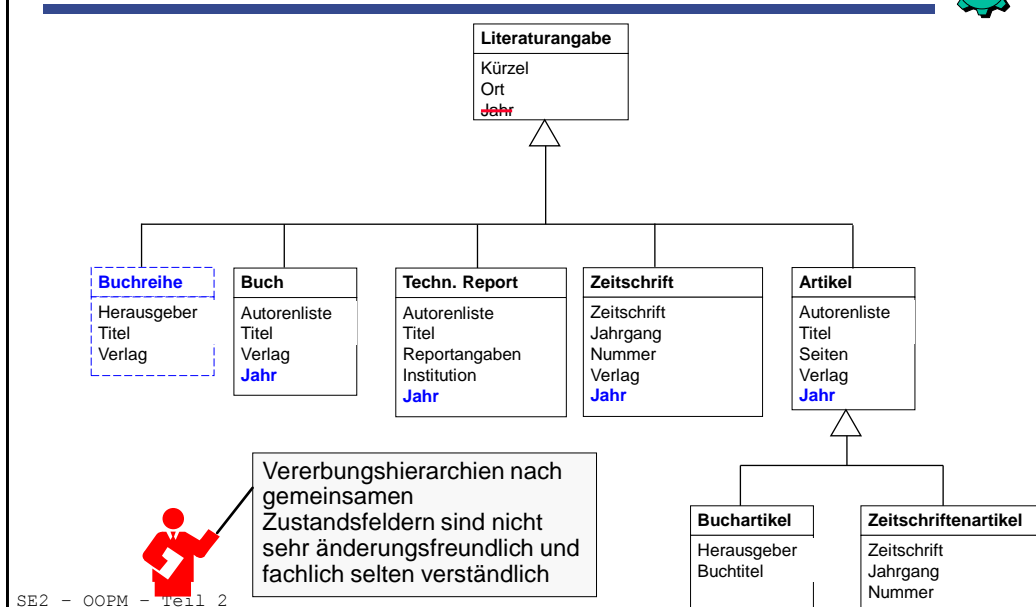
32

Der fachliche strukturentwurf (2)

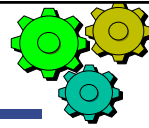


33

Der fachliche strukturentwurf (3)



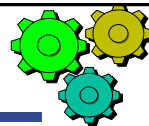
Das Bibliothekssystem



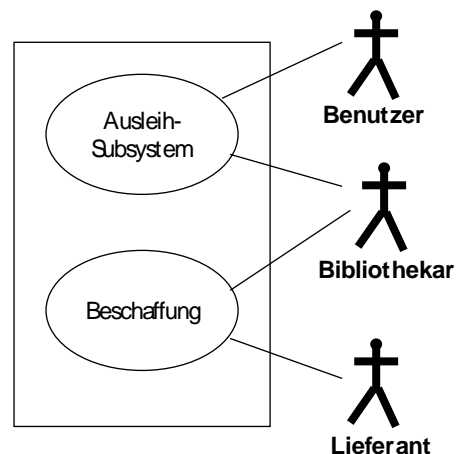
- **Die Aufgabenstellung:**
Die Aufgabe der Bibliothek ist das Sammeln, Erschließen und Benutzbarmachen von Literatur für die Mitarbeiter einer Firma. Die Arbeit mit der Bibliothek soll rationalisiert werden. Der Verwaltungsaufwand ist so weit wie möglich zu reduzieren. Dabei sind Routineaufgaben zu minimieren.
- **Vorgehensweise bei Analyse und Entwurf:**
 - Was sind die Aufgaben in einer Bibliothek?
 - Welche Aufgaben sollen durch das Anwendungssystem unterstützt werden?



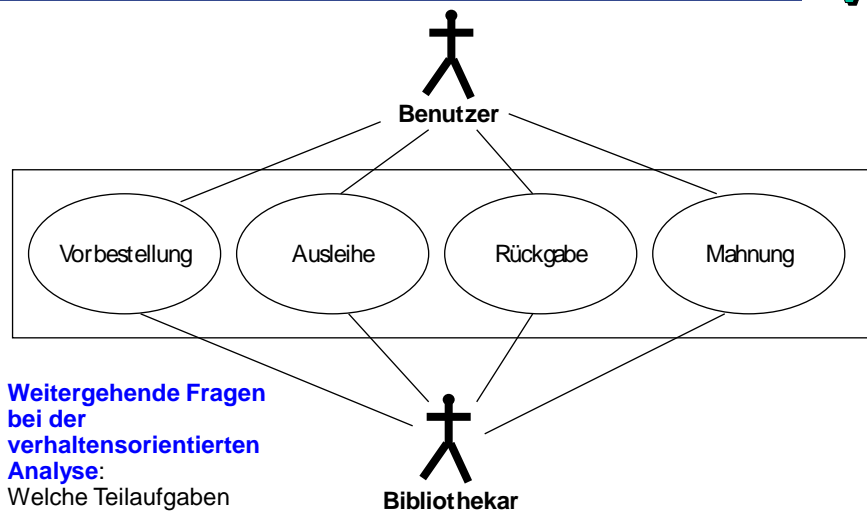
Use Case: Beispiel Bibliothek



- **Leitfrage bei der verhaltensorientierten Analyse:**
Wer ist an welchen Aufgaben in der Bibliothek beteiligt?



Use Case: Beispiel Ausleih-Subsystem

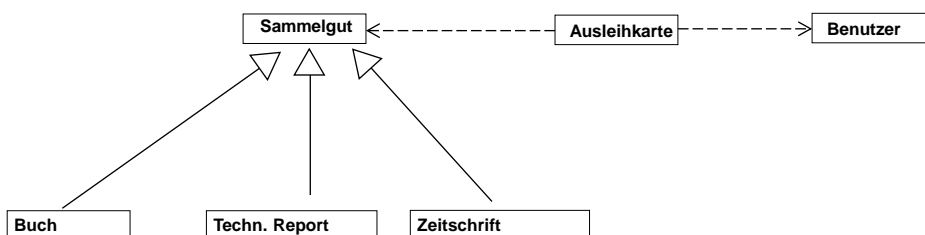


- **Weitergehende Fragen bei der verhaltensorientierten Analyse:**
Welche Teilaufgaben werden von wem in der Bibliothek erledigt?

SE2 – OOPM – Teil 2

37

Der fachliche verhaltensorientierte Entwurf (1)



Das Begriffsmodell wurde unmittelbar aus den Use Cases abgeleitet. Dort werden die Ober- und Unterbegriffe verwendet.

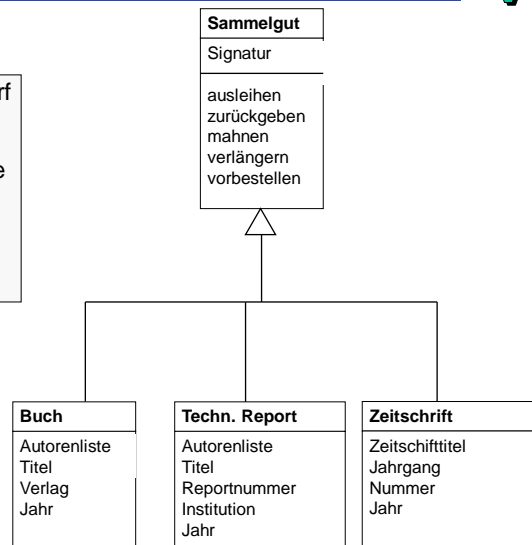
SE2 – OOPM – Teil 2

38

Der fachliche verhaltensorientierte Entwurf (2)



Der fachliche Klassenentwurf ist aus den Umgangsformen abgeleitet, die in den Use Cases beschrieben sind. Die Attribute sind notiert, dienen aber nicht der Klassifikation.



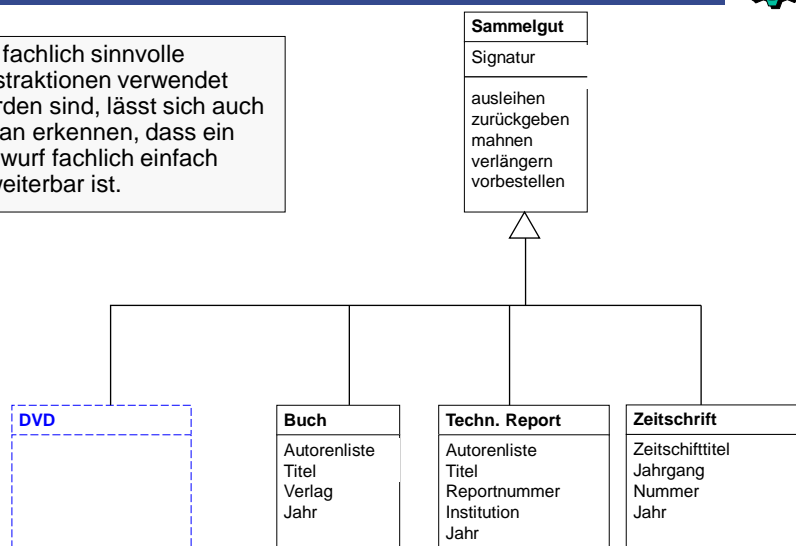
SE2 – OOPM – Teil 2

39

Der fachliche verhaltensorientierte Entwurf (3)



Ob fachlich sinnvolle Abstraktionen verwendet worden sind, lässt sich auch daran erkennen, dass ein Entwurf fachlich einfach erweiterbar ist.



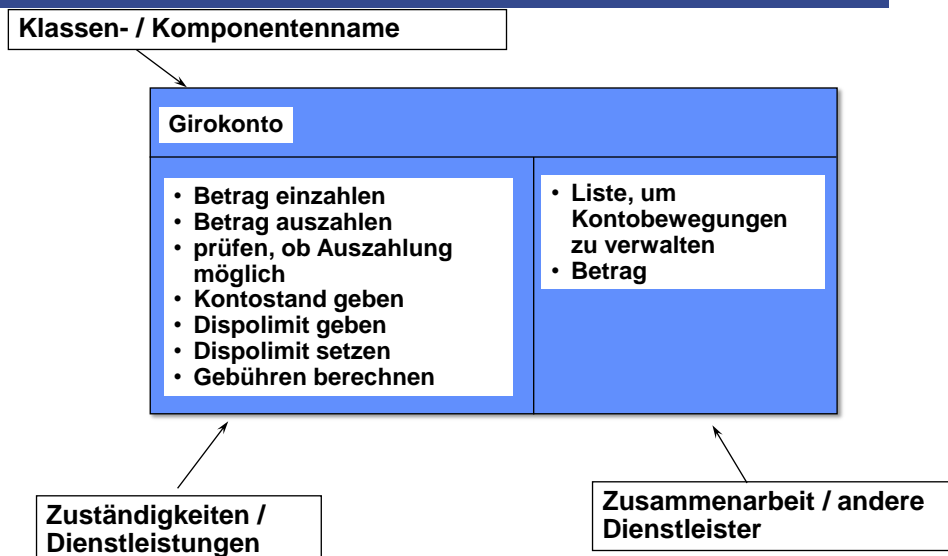
SE2 – OOPM – Teil 2

40

Vom fachlichen zum technischen Modell

- Das Modell des Gegenstandsbereichs wird im Entwurf in ein **Modell des Anwendungssystems** überführt.
- Zu den anwendungsfachlichen Merkmalen werden die für die Konstruktion des Systems notwendigen **technischen Charakteristika** ergänzt.
- Beschrieben wird das Modell des Anwendungssystems vorrangig anhand der **Elemente des objektorientierten Programmiermodells**.
- Darstellungsmittel sind z.B. CRC-Karten, Anwendungsfälle (technische Use Cases), Klassen- und Objektdiagramme.

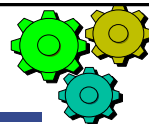
CRC-Karten



Aufbau von CRC-Karten

- **Klassen- / Komponentename** (Class / Component Name)
Der Klassen- / Komponentename ist ein Begriff des Anwendungsgebiets. Er ist Teil der Fachsprache.
- **Zuständigkeiten** (Responsibility)
Die Zuständigkeiten charakterisieren die von der Klasse / Komponente erbrachten Dienstleistungen. Sie sind ein in sich zusammenhängendes Angebot an potentielle Klienten.
- **Zusammenarbeit** (Collaborators)
In diesem Teil werden andere Anbieter von Dienstleistungen (also: andere Klassen / Komponenten) benannt, an die Zuständigkeiten delegiert werden, um die eigene Dienstleistung zu erbringen.

Zur Erinnerung: Zusicherungen im Vertragsmodell



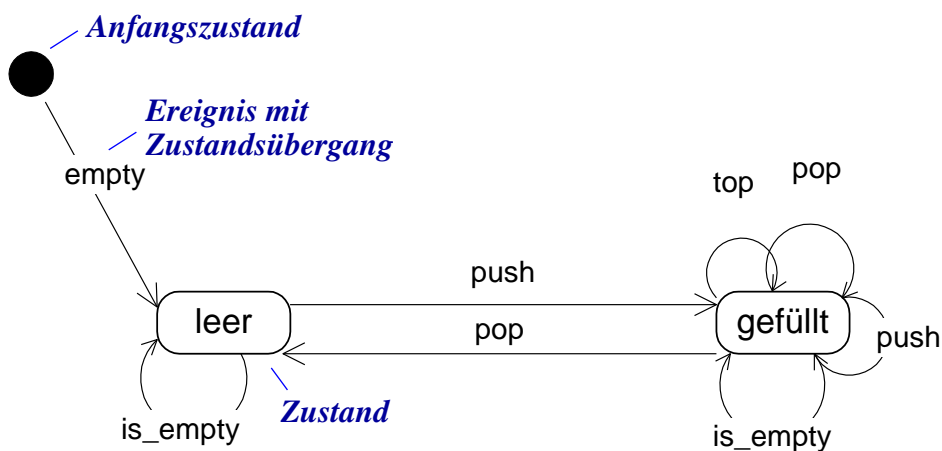
Vor-, Nachbedingungen und Invarianten legen das mögliche Verhalten von Objekten fest. Wir stellen fest, dass ein Buch verschiedene gültige Zustände im Ausleihprozess durchlaufen kann. Das gesamte Modell der zulässigen Operationen abhängig vom jeweiligen Zustand ist aus dem Vertragsmodell nur schwer herauszulesen. Hier hilft ein explizites Zustandsdiagramm.

Buch
ausleihen (b : Benutzer) require: ist_ausleihbar() ensure: ist_ausgeliehen()
zurückgeben (b : Benutzer) require: ist_ausgeliehen() ensure: ist_ausleihbar()
ist_ausleihbar () : boolean ist_ausgeliehen () : boolean

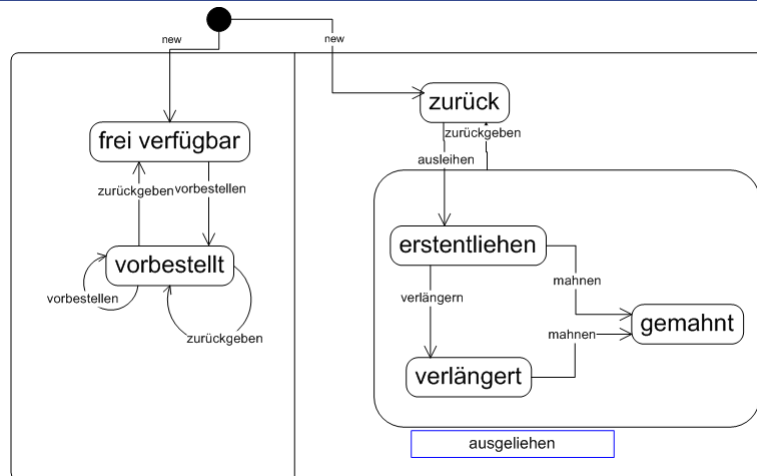
Zustandsdiagramm: Einführung

- Ein Zustandsdiagramm (State Machine Diagram) beschreibt die möglichen **Folgen von Zuständen** eines **Modell-Elements**, i.A. eines Objekts einer bestimmten Klasse
 - während seines **Lebenslaufs** (Erzeugung bis Entsorgung),
 - während der **Ausführung** einer **Operation** oder **Interaktion**.
- Modelliert werden
 - die **Zustände**, in denen sich die Objekte einer Klasse befinden können,
 - die möglichen **Zustandsübergänge** (Transitionen) von einem Zustand zum anderen,
 - die **Ereignisse**, die Transitionen auslösen,
 - **Aktivitäten**, die in Zuständen bzw. im Zuge von Transitionen ausgeführt werden.

Zustandsdiagramm: Mögliches Protokoll eines Stacks



Zustandsdiagramm: Mögliches Protokoll eines Sammelguts im Bibliotheksbeispiel



SE2 – OOPM – Teil 2

47

Zusammenfassung & Ausblick



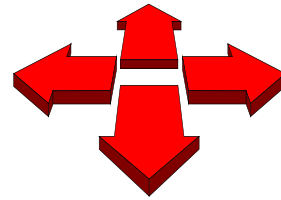
- Es gibt verschiedene Methoden der objektorientierten Modellierung und Konstruktion; **UML** und **UP** sind die kommerziell derzeit dominierenden Ansätze.
- Wir haben mit dem **Werkzeug & Material-Ansatz** eine anwendungs- und verhaltensorientierte Variante vorgestellt.
- Dabei bilden die **Aufgaben im Einsatzkontext** und die dabei verwendeten **Gegenstände** den Mittelpunkt.
- **Verhaltensorientierte Ansätze** führen fachlich und technisch zu Entwürfen, die sich von strukturorientierten Ansätzen unterscheiden.
- Für die **einfache Benutzung** ist die Darstellung von bekannten Gegenständen sehr hilfreich.
- Zu einem verhaltensorientierten Ansatz passen das **Vertragsmodell** und ein **fachlich orientiertes Zustandsmodell**.

SE2 – OOPM – Teil 2

48

Strukturierung interaktiver Anwendungssysteme

- Ausgangspunkt: Interaktive Softwaresysteme
- Motivation für Entwurfsregeln: Softwarequalität
- Konstruktionsregeln für einfache interaktive Systeme



Interaktive Softwaresysteme

- Interaktive Softwaresysteme sind gekennzeichnet durch eine **häufige Interaktion** mit der Benutzerin / dem Benutzer.
- Moderne interaktive Systeme bieten eine **grafische Benutzungs-schnittstelle** (im Gegensatz zu textuellen Kommandoschnittstellen, die ebenfalls stark interaktiv sein können, siehe etwa die Unix-Shells).
- Der Rechenaufwand auf Seite des Computers für die eigentliche Antwort ist gering; aufwändig ist meist eher die Berechnung ihrer **Darstellung** (nicht zufällig besitzen moderne Arbeitsplatzrechner neben einer leistungsfähigen CPU auch eine leistungsfähige Grafikkarte).
- Im Gegensatz dazu: eine aufwändige Verarbeitung durch den Computer anstoßen, der dann völlig selbstständig rechnen kann.
 - Beispiele: Umwandeln von Video-Dateien von einem Format in ein anderes, Klimasimulationen, Genom-Sequenzierung.

„Reiche“ und „schlanke“ Systeme



- Interaktive Systeme sind seit etlichen Jahren selbstverständlich. Als grundlegende Differenzierung unterscheiden wir
 - **Desktop-Systeme** (auch engl.: **Rich-Clients**): Die Verarbeitung der Anweisungen des Benutzers erfolgt primär auf dem Rechner (Desktop, Laptop, mobiles Gerät), mit dem der Benutzer direkt interagiert.
 - **Web-Systeme** (auch engl.: **Thin-Clients**): Die Verarbeitung erfolgt überwiegend auf einem Server (Web-Server), der Rechner vor Ort dient primär dazu, die Interaktion (meist in einem Browser) zu vermitteln und zu visualisieren.
- Die Grenzen zwischen diesen beiden grundsätzlichen Ansätzen verschwimmen inzwischen immer mehr.



Beispiele für Rich- und Thin-Clients

- | | |
|--|--|
| <ul style="list-style-type: none"> • Bekannte Desktop-Systeme: <ul style="list-style-type: none"> • MS Office, OpenOffice • Photoshop • iTunes • Eclipse • Umfangreicher Download • Meist explizites Update auf neue Versionen | <ul style="list-style-type: none"> • Bekannte Web-Anwendungen: <ul style="list-style-type: none"> • Web-Mail • Facebook (auch über Apps) • Google Docs • XING, LinkedIn • Meist nur ein Browser auf Client-Seite notwendig • Update transparent auf dem Server |
|--|--|

Wir fokussieren in SE2 auf **interaktive Desktop-Systeme**, die Konstruktion von Web-Systemen ist ein fortgeschrittenes Thema für weiterführende Veranstaltungen.

Einfache interaktive Softwaresysteme

- Im Folgenden stellen wir einen Satz an **Entwurfsregeln** vor, die die Konstruktion von **einfachen interaktiven Softwaresystemen** erleichtern.
- Diese Systeme sind **einfach**, weil in ihnen wichtige Eigenschaften von vielen Softwaresystemen **nicht berücksichtigt** sind, u.a.:
 - **Persistenz**: die dauerhafte Speicherung von Daten, Informationen oder digitalen Materialien ist ein anspruchsvolles Thema für sich, das unter anderem in „Grundlagen von Datenbanken“ thematisiert wird; die SE2-Systeme arbeiten **ohne Datenbanken**.
 - **Verteilung**: Fast immer gilt es bei realer Software, räumlich getrennte Akteure (Rechner, Menschen, Arbeitsplätze) miteinander zu verknüpfen. Verteilte Software ist ebenfalls sehr anspruchsvoll, die SE2-Systeme sind deshalb **Einzelplatzsysteme**.
- Trotz dieser Vereinfachungen ist die Konstruktion von interaktiven Softwaresystemen immer noch **anspruchsvoll**.
- So anspruchsvoll, dass wir einige **Konstruktionsregeln** für SE2 aufgestellt haben, die ihre Erstellung erleichtern sollen.

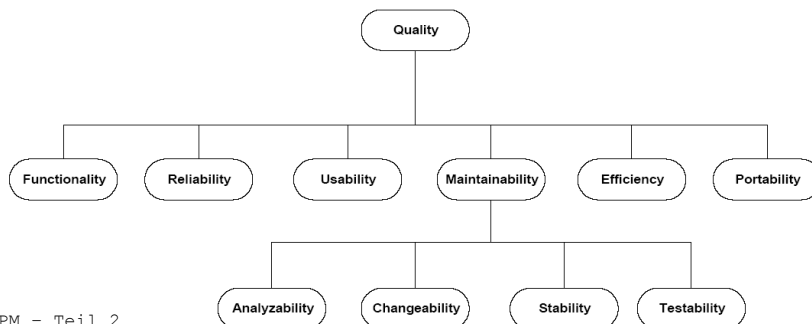
Einfache Software: Qualität egal

„There is software that's easy to write.
That would be software that is small,
written by one person over a short
period of time, used by that one person,
used once and then thrown away.
Everything else is difficult.“



Motivation für Entwurfsregeln: Qualität von Software

- Wir streben nach **möglichst hoher Qualität** bei der Softwareerstellung.
- Die Qualität von Software kann nach B. Meyer differenziert werden in
 - **Äußere bzw. externe Qualität** (Funktionalität, Zuverlässigkeit, Benutzbarkeit, Effizienz) und
 - **Innere bzw. interne Qualität** (Verständlichkeit, Wartbarkeit, Modularität).
- In der Benutzung zählt nur die äußere Qualität, die aber über interne Qualität erreicht und gesichert wird.



SE2 - OOPM - Teil 2

55

Hintergrund: verschiedene Qualitätsbegriffe

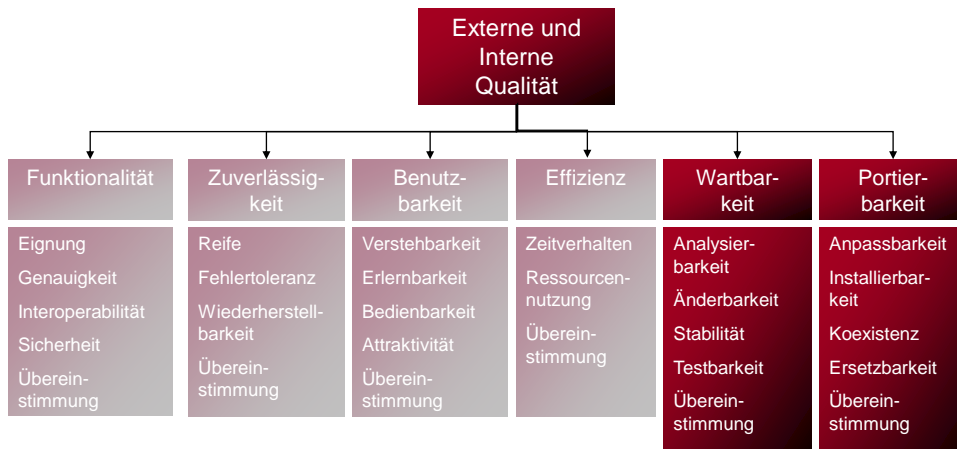
- **Der transzendente Ansatz**
 - Qualität ist universell erkennbar, absolut, einzigartig; sie kann nicht gemessen werden, sie lässt sich nur durch Erfahrung bewerten.
- **Der produktbezogene Ansatz**
 - Objektiv bewertbare Eigenschaften eines Produktes werden spezifiziert und vermessen; Produkte können somit in Rangordnungen gelistet werden.
- **Der benutzerbezogene Ansatz**
 - Der Benutzer legt die Qualität fest; je besser ein Produkt Bedürfnisse befriedigt, desto höher die Qualität; Q. ist somit nicht objektiv bewertbar.
- **Der prozessbezogene Ansatz**
 - Qualität wird durch den Erstellungsprozess bestimmt; ein exakt spezifizierter, detailliert kontrollierter Prozess führt zu einem hochwertigen Produkt.
- **Der Kosten/Nutzen-bezogene Ansatz**
 - Qualität ist eine Funktion von Kosten und Nutzen; Q. heißt, einen bestimmten Nutzen zu einem akzeptablen Preis zu erhalten.

SE2 - OOPM - Teil 2

© Balzert, Lehrbuch der Softwaretechnik

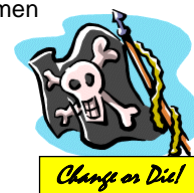
56

Externe und interne Qualität in der ISO 9126



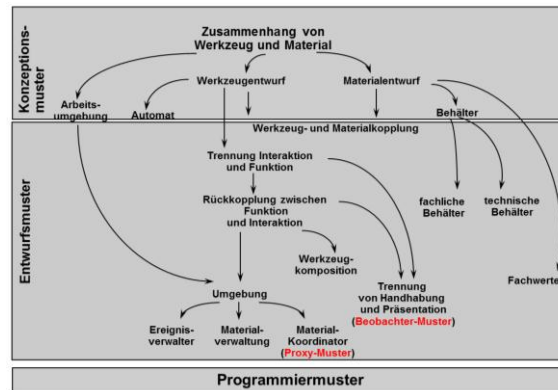
Zusätzliche Motivation: Software verändert sich

- Software ist keine Prosa, die einmal geschrieben wird und dann unverändert bleibt.
- Software wird erweitert, korrigiert, gewartet, portiert, adaptiert, ...
- Diese Arbeit wird von unterschiedlichen Personen vorgenommen (manchmal über Jahrzehnte).
- Für Software gibt es deshalb nur zwei Optionen:
 - Entweder sie wird gewartet.
 - Oder sie stirbt.
- **Software, die nicht gewartet werden kann, wird sterben!**



Umfangreiche Entwurfsregeln: der WAM-Ansatz

- Der **WAM-Ansatz** umfasst neben einem detaillierten Vorgehensmodell und etlichen Dokumenttypen für die objektorientierte Analyse und Modellierung auch einen umfangreichen Satz an Konstruktionshinweisen für interaktive Systeme.
- Diese Hinweise halten jahrelange Konstruktions-erfahrung aus professionellen Entwicklungsprojekten in Form von Mustern fest, die im WAM-Ansatz in **Konzeptions-, Entwurfs- und Programmiermuster** differenziert werden.



SE2 - OOPM - Teil 2

59

Überschaubare Regeln: SE2-Entwurfsregeln

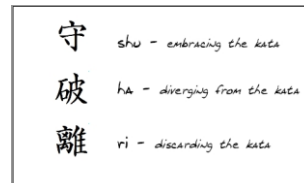
- Für eine Lehrveranstaltung im zweiten Semester ist der WAM-Ansatz in allen seinen Ausprägungen deutlich zu umfangreich; die Grundideen des Ansatzes sind jedoch auch für kleine Systeme bereits anwendbar.
- Im Folgenden beschreiben wir deshalb einige an diesem Ansatz orientierte **Entwurfs- und Konstruktionsregeln**, die das Erstellen interaktiver Anwendungen erleichtern sollen.
- Obwohl diese Regeln vom **WAM-Ansatz** motiviert sind, sind sie eigenständig nutzbar und anwendbar; eine umfassende Kenntnis des WAM-Ansatzes ist **keine Voraussetzung** für ihre Nutzung.
- Wir weisen darauf hin, dass Konstruktionsregeln immer die **Gefahr** bergen, sklavisch befolgt zu werden (wie auch Quelltextkonventionen). Dies ist insbesondere deshalb problematisch, weil Regeln selten alle möglichen Situationen abdecken können; diese Regeln sollen in erster Linie dazu dienen, **Ungeübten** die softwaretechnische Konstruktion zu erleichtern.

SE2 - OOPM - Teil 2

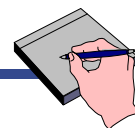
60

Exkurs: Stufen beim Lernen

- **Alistair Cockburn** beschreibt in seinem Buch „Agile Software Development“ **drei Stufen in der persönlichen Entwicklung auf dem Weg zu kompetenter Softwareentwicklung** und orientiert sich dabei an asiatischen Kampfsportarten wie dem **Aikido**; diese Stufen finden aber auch beispielweise Anwendung beim japanischen Brettspiel **Go**.
- Die Stufen sind
 - **Shu** – *beschützen, verteidigen, einhalten, befolgen*
 - Das Lernen der Form / **Kopieren**
 - **Ha** – *zerreißen, durchbrechen*
 - Das Überschreiten der Form / **Abweichen**
 - **Ri** – *sich entfernen, trennen, abschneiden*
 - Eigene Wege finden / **Freie Verwendung**
- In SE2 ist der Fokus auf dem **Erlernen guter Form**; in der konkreten Anwendung wird es immer wieder Bedarf für Abweichung geben; letztlich sollen die Regeln aufgrund reicher eigener Erfahrung überflüssig werden.



Grundlegend: Trenne Fachlogik und Technik



- **Anwendungsfachliche Klassen**, die vor allem die Fachlogik modellieren, sollten deutlich von rein **technisch motivierten Klassen** unterscheidbar sein.



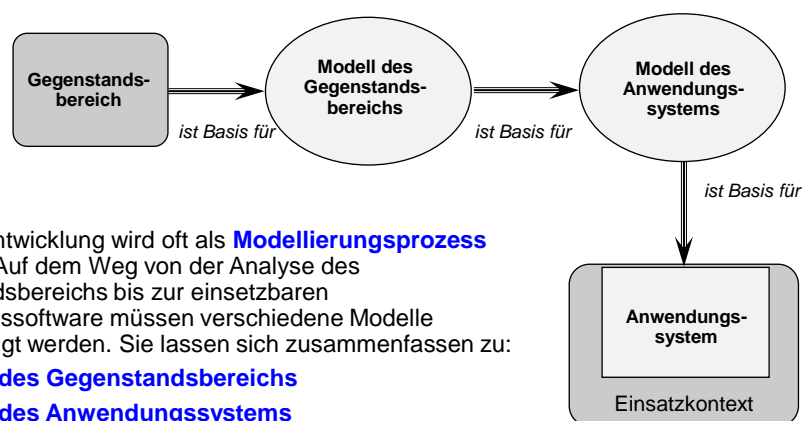
Modellierung anwendungsfachlicher Klassen

- Nah am Anwendungsbereich
- Abstraktionen der jeweiligen Domäne
- Mit den Anwendern diskutierbar
- Oft sehr spezifisch, in jedem Projekt anders
- Wiederverwendung nur möglich, wenn domänenspezifische Klassen bereits vorliegen.
- Andere häufig verwendete Begriffe: **Domain Objects**, **Business Objects**

Aber: innerhalb
eines Projektes oft
das Stabilste!



Wiederholung: Modelle in der Softwareentwicklung

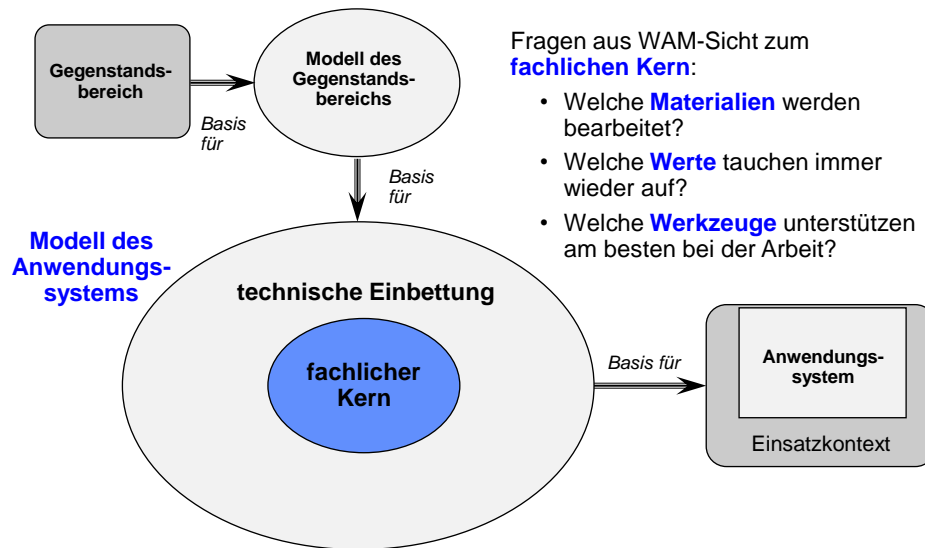


- Software-Entwicklung wird oft als **Modellierungsprozess** betrachtet. Auf dem Weg von der Analyse des Gegenstandsbereichs bis zur einsetzbaren Anwendungssoftware müssen verschiedene Modelle berücksichtigt werden. Sie lassen sich zusammenfassen zu:
 - **Modell des Gegenstandsbereichs**
 - **Modell des Anwendungssystems**
- Objektorientierte Methoden unterscheiden sich auch darin, ob und wie sie diese Modelle erstellen.

Der Quellcode ist Teil des Modells des Anwendungssystems; das ablaufende Programm ist Teil des Anwendungssystems.



Zoom: Modell des Anwendungssystems



SE2 – OOPM – Teil 2

65

Modellierung technischer Klassen

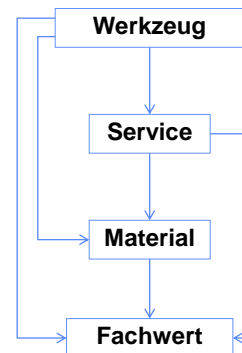
- Neben dem anwendungsfachlichen Kern gibt es immer eine große Anzahl technischer Klassen, die aus Anwendungssicht **Hilfsklassen** sind.
- Sie modellieren **technische Aspekte** einer Anwendung, beispielsweise:
 - Grafische Oberflächen
 - Web-Oberfläche
 - Persistenz in Datenbanken
 - Verteilung
- Häufig deutlich umfangreicher als der fachliche Kern.
- Rückgriff auf umfangreiche Bibliotheken möglich (GUI-Bibliotheken, OR-Mapper, Middleware).
- **Technische und fachliche Klassen sollten gut voneinander getrennt sein.** Dies erleichtert die Wartung erheblich.
- Auch die SE2-Konstruktionsregeln basieren auf dieser grundsätzlichen Unterscheidung.

SE2 – OOPM – Teil 2

66

Die SE2-Entwurfsregeln

- Die **SE2-Entwurfsregeln** benennen vier **Elementtypen**, aus denen sich ein interaktives System zusammensetzt:
 - Materialien** realisieren veränderliche, anwendungsfachliche Gegenstände.
 - Fachwerte** sind anwendungsfachliche Werte; sie sind unveränderlich.
 - Werkzeuge** bieten eine grafische Benutzungsschnittstelle und ermöglichen das interaktive Bearbeiten von Materialien.
 - Services** bieten materialübergreifend fachliche Dienstleistungen an, die systemweit zur Verfügung stehen sollen.



Die Pfeile zeigen die **erlaubten Benutz-Beziehungen** zwischen den Elementtypen. Jeder Elementtyp kann außerdem Elemente vom eigenen Typ benutzen (hier nicht dargestellt).

Allgemeines zu den Entwurfsregeln

- In seiner programmtechnischen Umsetzung kann ein Elementtyp aus mehreren Klassen bestehen. Dies gilt insbesondere für Werkzeuge.
- Für jeden Elementtyp (Werkzeug, Material, Service, Fachwert) sollte es ein **eigenes Java-Paket** geben. Darunter kann es bei Bedarf noch eine weitere Aufteilung geben, z.B. im Werkzeug-Paket für verschiedene Werkzeuge.
- Die Elementtypen Material, Fachwert und Service werden durch **fachliche Klassen** modelliert, die somit auch eine **fachlich motivierte Schnittstelle** haben; Werkzeuge hingegen bieten ihre fachliche „Schnittstelle“ interaktiv gegenüber dem Benutzer an, die Schnittstellen der implementierenden (technisch motivierten) Klassen sind eher technisch geprägt.
- Für alle fachlichen Klassen gilt:
 - Sie sichern ihre Konsistenz über das Vertragsmodell.
 - Es gibt eine zugehörige Testklasse.

Materialien



- Materialien modellieren **anwendungsfachliche „Gegenstände“**, wie CD, DVD, Videospiel, Kunde. Die modellierten Gegenstände müssen im Anwendungsbereich nicht tatsächlich gegenständlich sein; auch abstrakte Dinge, die fachlich relevant sind, werden als Material modelliert (Bsp.: Versicherungspolice, Konto, Vertragsentwurf, Partitur, Film).
- Materialien werden im Arbeitsprozess mit Werkzeugen erzeugt, bearbeitet und beseitigt. Sie vergegenständlichen **Arbeitsergebnisse**.
- Ein Material hat **ausschließlich fachliche Aufgaben**. Das heißt, dass ein Material keinerlei technische Aufgaben übernimmt, die beispielsweise die Persistenz betreffen oder das UI-Framework (in SE2: Swing).
- Materialien **kennen ausschließlich** andere **Materialien** und **Fachwerte**.
- Jeder fachlich relevante Materialtyp wird durch **eine eigene Klasse** modelliert!
- Für jedes einzelne Material im Anwendungsbereich gibt es nur genau ein passendes Material-Exemplar einer solchen Klasse im Softwaresystem.

Fachwerte



- Bei der Modellierung eines Anwendungsbereichs gibt es immer auch Begriffe, die eher wertartige Dinge beschreiben, wie **Kontonummer** oder **Geldbetrag**.
- Wir beschreiben solche Begriffe über **Fachwerte**.
- Fachwerte sind fachlich motivierte **Werte**. Werte sind ein allgemeineres Konzept, das beispielsweise auch Zahlen und Zeichenketten umfasst.
 - Ein Wert ist **unveränderlich**.
 - Wir beschreiben Werte programmiertechnisch über **Werttypen**.
 - Werttypen sind besondere Typen mit einer **unveränderlichen Wertemenge**; Werte werden somit konzeptuell nicht erzeugt, sondern bei Bedarf aus der Wertemenge ausgewählt.
- Fachwerte bilden die Grundkonstanten in einem Anwendungssystem.
- Wir werden uns Werttypen in einer der nächsten Vorlesungen ausführlich ansehen.

(Fachliche) Services

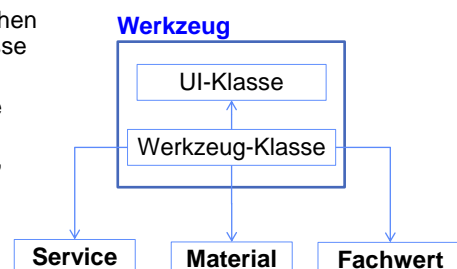


- Services bieten **fachliche Dienstleistungen** an, die systemweit zur Verfügung stehen sollen. Sie dienen in der Regel dafür, **materialübergreifende** Operationen anzubieten.
- Im Gegensatz zu Materialien gibt es von jedem Service **nur jeweils ein Exemplar** (Beispiel: Kundenstamm).
- Services können **Materialien verwalten**. Sie kapseln dabei aus Sicht der Anwendung häufig auch die **Persistenz** von Materialien (also ihre dauerhafte Speicherung).
- Services liefern **Referenzen auf Original-Materialien**, nicht auf Kopien.
- Services muss es **nicht in jedem interaktiven System** geben.
- Services werden von Werkzeugen benutzt, kennen diese aber nicht. Services **kennen nur andere Services, Materialien und Fachwerte**.
- Services werden an zentraler Stelle erzeugt und „verdrahtet“, beispielsweise in einer Startup-Klasse, und den Werkzeugen bei Bedarf als Konstruktorparameter übergeben.

Werkzeuge



- Werkzeuge dienen zur **Benutzerinteraktion**. Mit einem Werkzeug können Benutzerinnen und Benutzer über dessen **grafische Schnittstelle** interaktiv Materialien ansehen und bearbeiten.
- Ein Werkzeug übernimmt **genau eine fachliche Aufgabe**, die in einem kurzen Satz gut beschreibbar sein sollte.
- In SE2 zerlegen wir ein Werkzeug immer in eine **Werkzeug-Klasse** und eine **UI-Klasse**:
 - Die Werkzeug-Klasse **vermittelt** zwischen der grafischen Schnittstelle der UI-Klasse und den fachlichen Klassen.
 - Die UI-Klasse eines Werkzeugs hat die Aufgaben, die **GUI-Komponenten** der **grafischen Schnittstelle** zu erzeugen, zu layouten und zu verwalten.

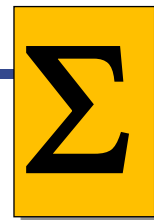


Werkzeugkonstruktion: Erste Schritte

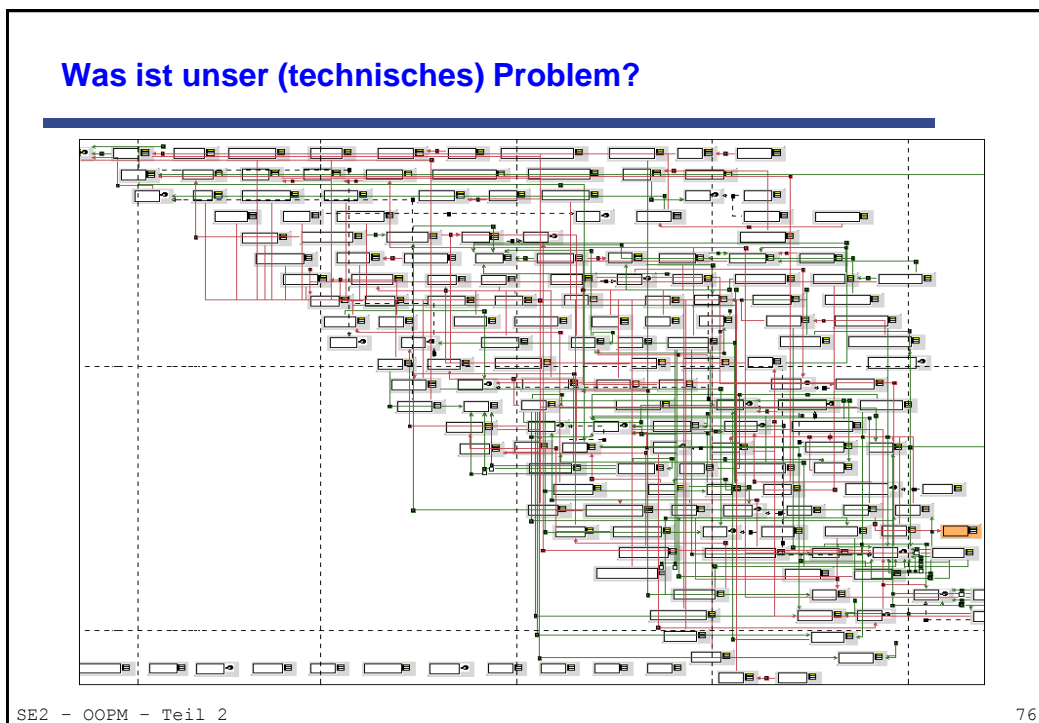
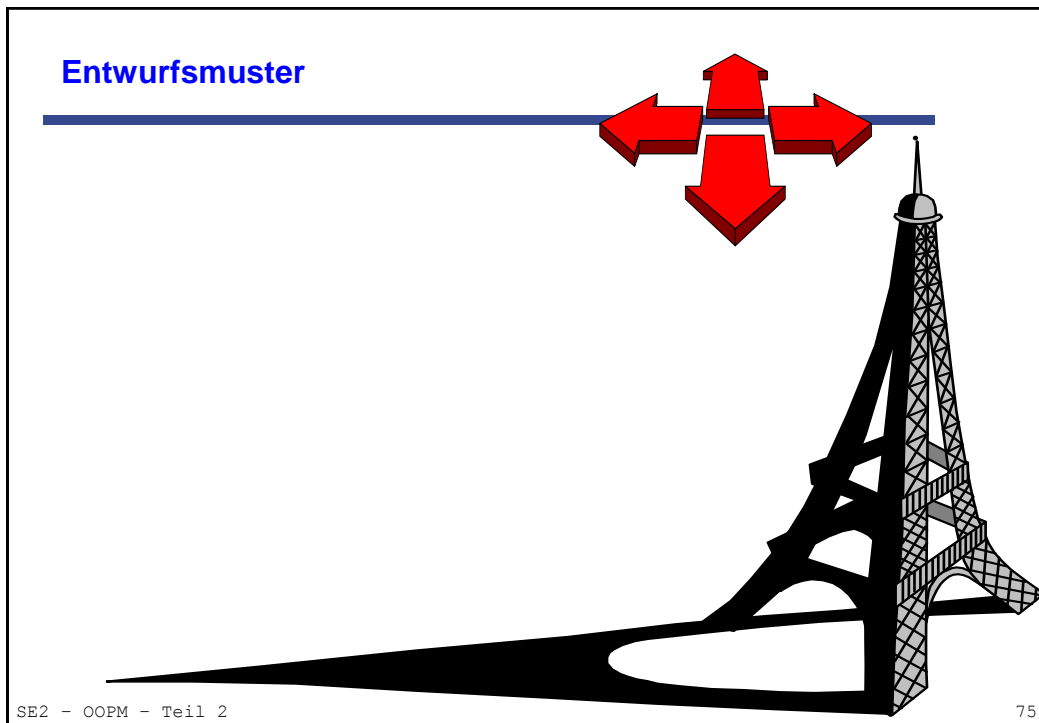


- Relevante Entwurfsregeln bis zu diesem Punkt:
 - Die Werkzeug-Klasse erhält ihr **Material** als Konstruktorparameter, über Setter (wenn das Material austauschbar sein soll) oder holt es sich über Services.
 - Der Werkzeug-Klasse werden benötigte **Services als Konstruktorparameter** übergeben.
 - Die Werkzeug-Klasse **erzeugt** ein Exemplar ihrer **UI-Klasse** im eigenen Konstruktor.
 - Die UI-Klasse eines Werkzeugs hat die Aufgaben, die **GUI-Komponenten** zu **erzeugen**, zu layouten und zu **verwalten**.
- Bevor wir diese Unterteilung weiter betrachten können, benötigen wir einiges an Grundlagenwissen über **Entwurfsmuster** und **grafische Benutzungsschnittstellen**.

Zusammenfassung SE2-Entwurfsregeln

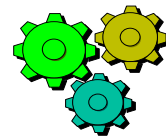


- Die **SE2-Entwurfsregeln** für interaktive Anwendungen (insbes. Rich-Clients) geben konstruktive Hinweise für die Strukturierung von Softwaresystemen mit einer grafischen Oberfläche.
- Sie definieren vier Elementtypen:
 - **Materialien** – veränderbare fachliche Objekte, die üblicherweise Arbeitsergebnisse modellieren;
 - **Fachwerte** – unveränderliche fachliche Abstraktionen;
 - **Services**, die systemweit fachliche Dienstleistungen anbieten und häufig Materialien verwalten;
 - **Werkzeuge** zur interaktiven Bearbeitung von Materialien.
- Der Entwurf von Materialien, Fachwerten und Services ist primär fachlich anspruchsvoll, während die Konstruktion von Werkzeugen vor allem technisch anspruchsvoll ist.



Wir entdecken ein Entwurfsmuster

„Definiere eine 1-zu-n Beziehung zwischen Objekten, so dass die Änderung des Zustands eines Objekts dazu führt, dass alle abhängigen Objekte benachrichtigt und automatisch aktualisiert werden.“



SE2 – OOPM – Teil 2

77

Ein „Ueberweiser“

```
public class Ueberweiser {
    public void setzeAuftraggeber( int ktoNr, int blz ) {...}
        -- setzt das Konto des Auftraggebers
    public void setzeEmpfaenger( int ktoNr, int blz ) {...}
        -- setzt das Konto des Empfaengers
    public void ueberweisen( float betrag )
        -- ueberweist 'betrag' vom 'AuftraggeberKonto' auf das 'EmpfaengerKonto'
    public float empfaengerSaldo() {...}
        -- liefert den aktuellen Kontostand des Empfaengers
    public float auftraggeberSaldo {...}
        -- liefert den aktuellen Kontostand des Auftraggebers
    public boolean gueltigesKonto( int ktonr, int blz ) {...}
        -- prüft, ob ein Konto mit 'ktonr' bei der Bank 'blz' existiert
    public boolean istUeberweisungMoeglich() {...}
        -- prüft, ob die Konten von Auftraggeber und Empfaenger bereits bestimmt worden sind
    public boolean istAuftraggeberSaldoVeraendert () {...}
        -- liefert 'true', wenn das Saldo des Auftraggebers verändert wurde
    public boolean istEmpfaengerSaldoVeraendert () {...}
        -- liefert 'true', wenn das Saldo des Empfängers verändert wurde
}
```

SE2 – OOPM – Teil 2

78

Operationen an der Schnittstelle des Ueberweisers

Wir unterscheiden:

- **verändernde Operationen**, die eine Veränderung des Zustands bewirken
setzeAuftraggeber, setzeEmpfaenger, ueberweisen,
- **sondierende fachliche Operationen**, mit denen Informationen erfragt werden
können empfaengerSaldo, auftraggeberSaldo,
- **sondierende boolesche Operationen** mit denen u.a. Parameterwerte und
Reihenfolgebedingungen getestet werden können istEmpfaengerSaldoVeraendert,
istAuftraggeberSaldoVeraendert, gueltigesKonto, istUeberweisungMoeglich.



Die Schnittstelle des Ueberweisers macht keine Annahmen über die Art ihrer Benutzung. (Graphikkomponente? Eine andere Klasse?)

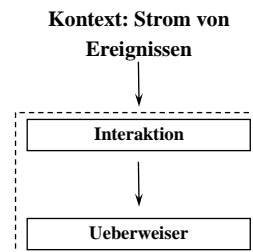
Wir wollen den Ueberweiser interaktiv benutzen

Zu lösende Aufgaben:

- Entgegennahme eines Stroms von Ereignissen, die durch Aktionen des Benutzers ausgelöst werden,
- Abstraktion vom zugrundeliegenden Fenstersystem unter dessen Verwendung Ein- und Ausgabe erfolgt (z.B. mit Knöpfen, Menüs).

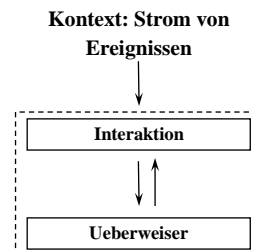
Grundproblem der Konstruktion: Zwei Klassen müssen sich kennen.

- Eine Aktion des Benutzers löst eine Reaktion des Programms aus.
- Das bedeutet technisch: die Interaktion benutzt den Ueberweiser.



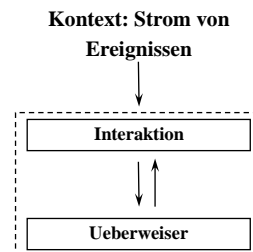
Grundproblem der Konstruktion: Zwei Klassen müssen sich kennen.

- Eine Aktion des Benutzers löst eine Reaktion des Programms aus.
- Das bedeutet technisch: die Interaktion benutzt den Ueberweiser.
- Der Ueberweiser soll seinen Zustand an der Oberfläche zeigen.
- Das bedeutet technisch: Der Ueberweiser benutzt die Interaktion.



Grundproblem der Konstruktion: Zwei Klassen müssen sich kennen.

- **Eine Aktion des Benutzers löst eine Reaktion des Programms aus.**
- Das bedeutet technisch: die Interaktion benutzt den Ueberweiser.
- **Der Ueberweiser soll seinen Zustand an der Oberfläche zeigen.**
- Das bedeutet technisch: Der Ueberweiser benutzt die Interaktion.
- **Sich zyklisch benutzende Klassen müssen wir vermeiden.**
- **Wir erkennen das Grundproblem unserer Konstruktion: Das Rückkopplungsproblem**

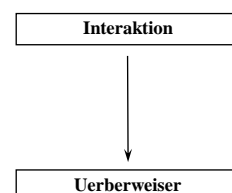


Lösungsansätze für das Rückkopplungsproblem (1)

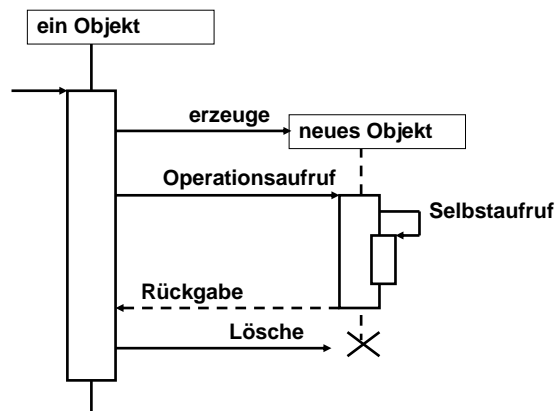
Die Interaktion kennt die Effekte von Operationsaufrufen und ruft geeignete **sondierende Operationen**, um einen veränderten Zustand des Ueberweisers darzustellen.



Das Geheimnisprinzip ist verletzt, da die Interaktion Kenntnis über die Umsetzung von Aufrufen besitzt !



Einschub: UML-Sequenzdiagramme



- Objektinteraktionen in zeitlicher Reihenfolge.
- Die an einem Szenario beteiligten Objekte und Klassen und die Operationsaufrufe die zwischen ihnen ausgeführt werden, um die Funktionalität eines UseCases zu erfüllen.
- Pragmatik (in UML): Soll während der Analysephase eingesetzt werden. Außerdem: "Keep it simple, stupid."

SE2 – OOPM – Teil 2

85

Einschub: UML-Sequenzdiagramme

- Ein Sequenzdiagramm besitzt zwei Dimensionen
 - vertikal: Zeit
 - horizontal: beteiligte Objekte
- Der Zeitablauf erfolgt von oben nach unten
 - normalerweise ist nur die Reihenfolge der Nachrichten signifikant
 - in Echtzeitanwendungen kann die Zeitachse auch eine Metrik besitzen
- Die horizontale Ordnung der Objekte ist nicht signifikant.

- **Ein Sequenzdiagramm ist die konkretisierte Darstellung eines Szenarios:**

- Beschreibt die im Szenario auftretenden Ereignisse in ihrer zeitlichen Abfolge.
- Benennt die am Szenario beteiligten Objekte.

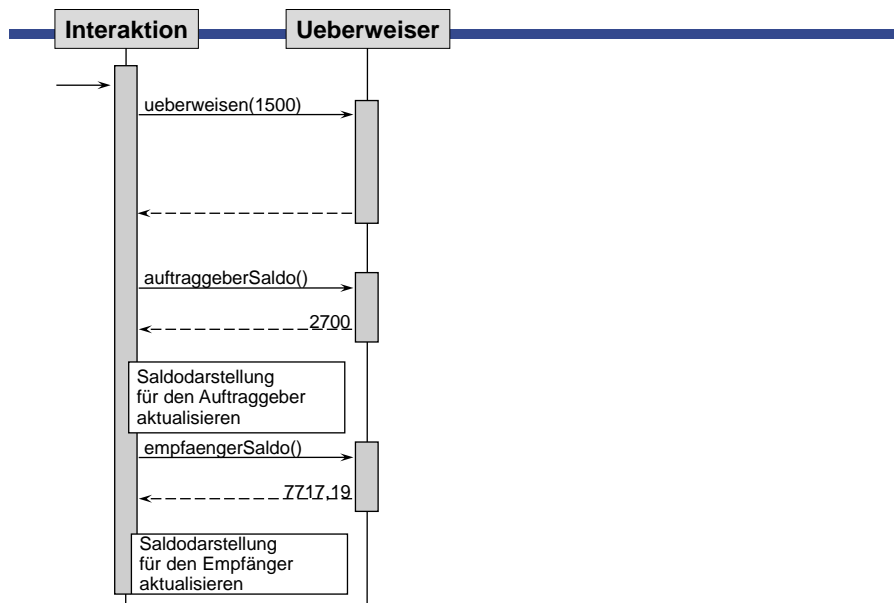
- **Ein Sequenzdiagramm zeigt:**

- den Nachrichtenaustausch zwischen den Objekten
- die Lebenszeit der Objekte
- die Aktivitätszeiten der Objekte

SE2 – OOPM – Teil 2

6

Lösungsansätze für das Rückkopplungsproblem (1)



SE2 – OOPM – Teil 2

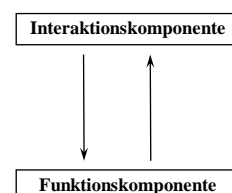
87

Lösungsansätze für das Rückkopplungsproblem (2)

Am Ende eines **Operationsaufrufs** ruft der Ueberweiser seinerseits die Interaktion, um einen veränderten Zustand aktualisiert darstellen zu lassen.



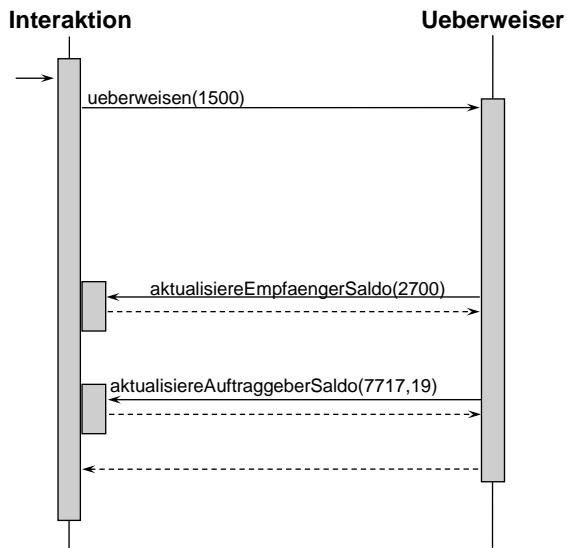
Die Aufgabentrennung von Ueberweiser und Interaktion ist verletzt, da der Ueberweiser über die Realisierung der Darstellung Kenntnis besitzt !



SE2 – OOPM – Teil 2

88

Lösungsansätze für das Rückkopplungsproblem (2)



SE2 - OOPM - Teil 2

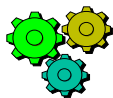
89

Lösungsansatz für das Rückkopplungsproblem (3)

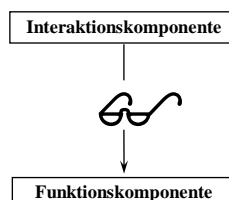
Nach jedem Systemereignis ruft die Interaktion die überprüfenden Funktionen des Ueberweisers, um einen veränderten Zustand unmittelbar nachzuvollziehen. Wir bezeichnen diesen Lösungsansatz als **Polling-Ansatz**.



Die Interaktion ist fast ausschließlich mit Polling beschäftigt !



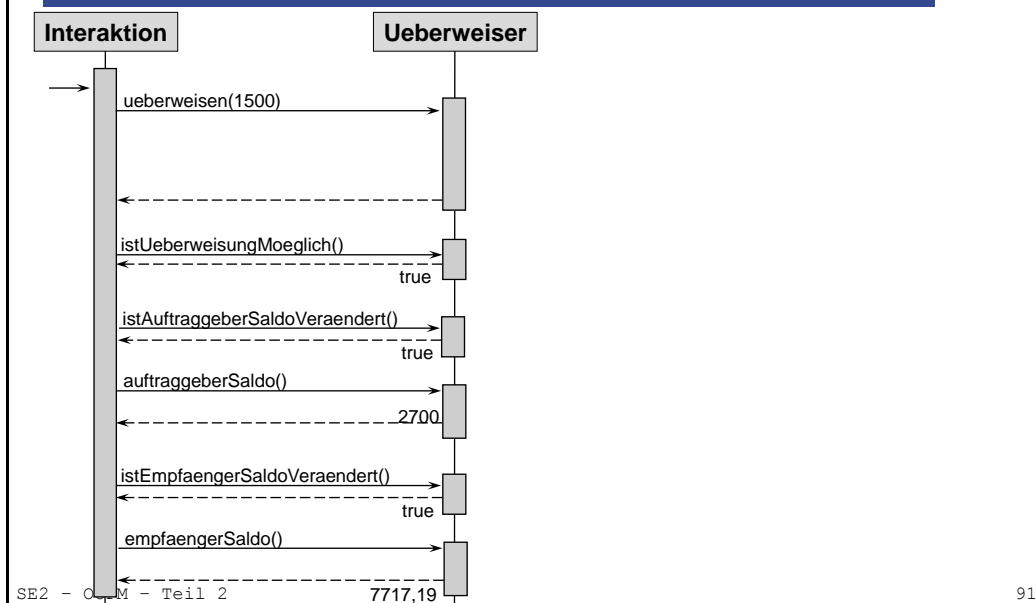
Softwaretechnisch ist **Polling** die sauberste der drei Lösungen und vermeidet die Nachteile der ersten beiden Lösungen.



SE2 - OOPM - Teil 2

90

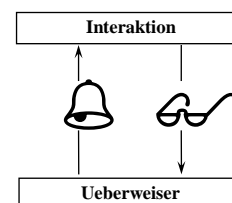
Lösungsansätze für das Rückkopplungsproblem (3)



Weiterentwicklung des Polling-Ansatzes: Der benachrichtigte Beobachter

•Konzept:

- Der Ueberweiser signalisiert Zustandsänderungen.
- Die Interaktion beachtet diese Signale und ruft entsprechende **sondierende Operationen**.
- Folge: Die Interaktion muß im Ueberweiser bekannt sein.



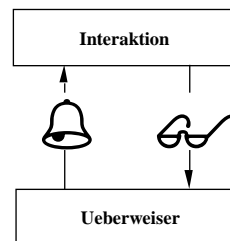
Aufrufe, Ereignisse, Signale

Wir unterscheiden:

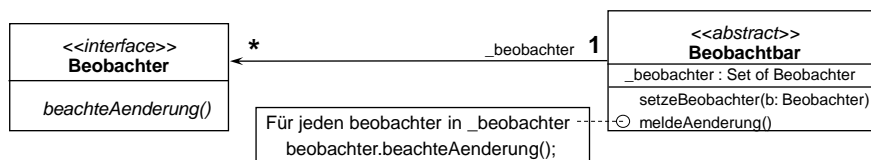
- Beim **Aufruf** (Call) kennt der Rufer (Klient) den Gerufenen (Anbieter) und erwartet eine bestimmte Dienstleistung.
- Ein **Ereignis** (Event) wird vom Erzeuger oder Verteiler an einen bestimmten Empfänger weitergeleitet. Eine Reaktion des Empfängers wird nicht erwartet.
- Ein **Signal** (Broadcast) wird vom Erzeuger an die Umgebung ausgesandt. Prinzipiell sind die Empfänger des Signals anonym. Die Reaktion auf ein Signal ist meist ein Aufruf des Erzeugers.



Da uns die gängigen oo Sprachen keinen eigenen Ereignis- oder Signalmechanismus anbieten, konstruieren wir die Benachrichtigung mit Hilfe des Aufrufmechanismus.



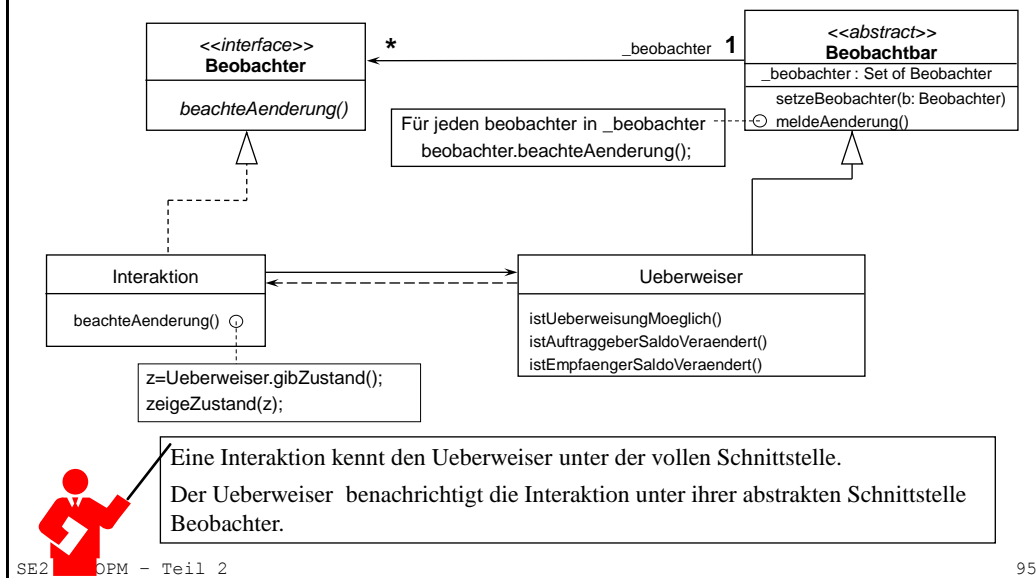
Beobachten und beobachtet werden: Beobachter und Beobachtbar



Beteiligte:

- Das **Interface Beobachter**: Jede Klasse, die eine andere Klasse beobachten können möchte, muss dieses Interface implementieren. Exemplare implementierender Klassen registrieren sich bei einem beobachtbaren Exemplar durch einen Aufruf von `setzeBeobachter()`. In `beachteAenderung()` wird eine Reaktion auf das Änderungssignal (der beobachtbaren Klasse) implementiert.
- Die **Klasse Beobachtbar**: Eine Klasse, die beobachtbar sein will, erbt von dieser Klasse. Registrierte Beobachter werden durch einen Aufruf von `meldeAenderung()` über eine Zustandsänderung benachrichtigt.

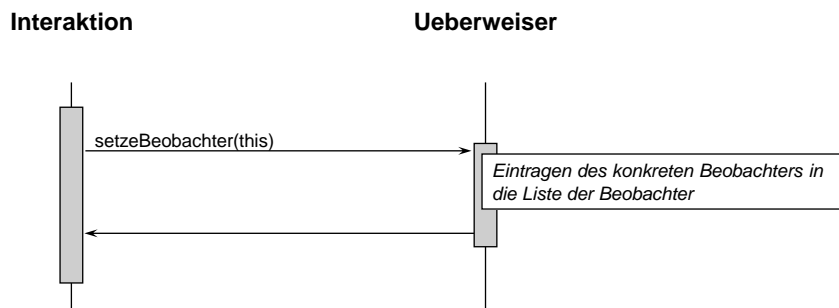
Beobachter und Beobachtbar im Kontext des Überweisers



SE2 – OOPM – Teil 2

95

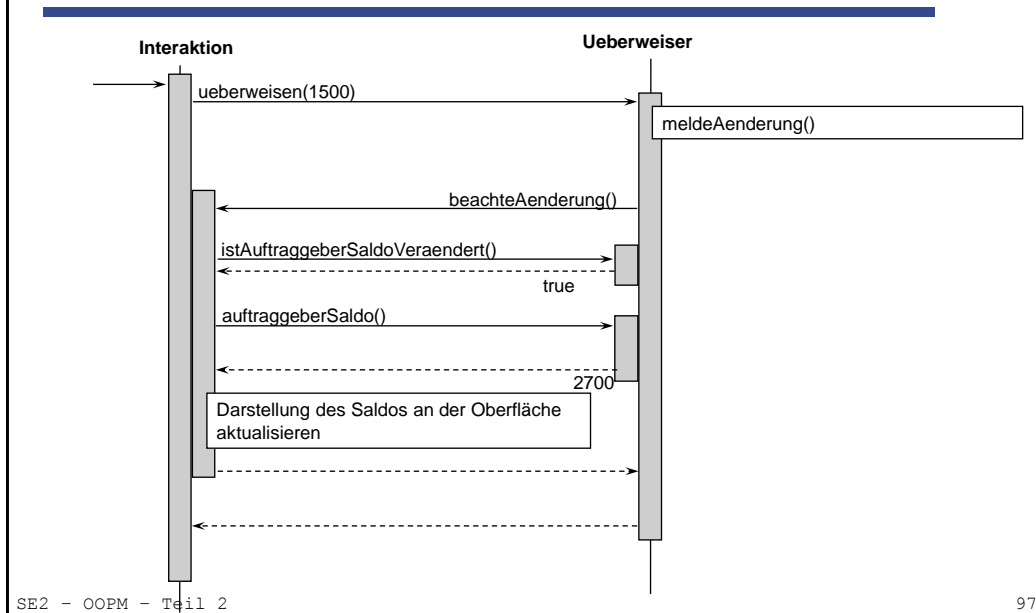
Dynamik: Registrieren eines Beobachters



SE2 – OOPM – Teil 2

96

Dynamik des Beobachter-Musters



SE2 - OOPM - Teil 2

97

Das Beobachter-Muster

Zweck

Das Beobachter-Muster wird zum Synchronisieren von Objektänderungen verwendet. „Definiere eine 1-zu-n-Abhängigkeit zwischen Objekten, so dass die Änderung des Zustandes eines Objekts dazu führt, dass alle abhängigen Objekte benachrichtigt und automatisch aktualisiert werden.“ [GHJV, S. 11]

Anwendbarkeit/Motivation (http://www.dle.informatik.uni-siegen.de/lehre/EI/informatik1/vorlesung_material/21_MVC.pdf)

- Konsistenz (in sich stimmig, keine Widersprüche) der Objekte sicherstellen
- Klassen nicht miteinander koppeln → Wiederverwendbarkeit
- Vorgehensweise: publiziere und abonniere (publish – subscribe)

SE2 - OOPM - Teil 2

98

Muster



Muster (engl. **pattern**)

Ein Muster ist eine Abstraktion von einer konkreten Form, die wiederholt in bestimmten, nicht willkürlichen Kontexten auftritt.



- Der Musterbegriff ist hier sehr allgemein. Er ist weder auf Softwareentwicklung noch auf eine bestimmte Verwendung von Mustern zugeschnitten.
- Die (bekannte) Definition
"A pattern is a solution to a recurring problem in a context"
ist auf die Lösung von Entwurfsproblemen ausgerichtet.

Mikroarchitekturen: Muster in der Architektur

Christopher Alexander:

"Each pattern describes a **problem** which occurs over and over again in our environment, and then describes the **core of the solution** to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice."

[Alexander, Ishikawa, Silverstein. A pattern language. Oxford University Press, 1977]

"Each pattern is a three part rule, which expresses a relation between a certain **context**, a **problem**, and a **solution**."

[Alexander. The timeless way of building. Oxford University Press, 1979]

Die meisten Autoren im Bereich Design Patterns schließen sich Alexander an (z.B. Gamma et al., Beck, Johnson, Schmidt, Buschmann, Coplien).



Mikroarchitekturen in objektorientierter Software

Ein **Entwurfsmuster** (engl. **design pattern**)

- beschreibt abstrakt eine bewährte Lösung für ein bestimmtes und häufig wiederkehrendes Problem des **objektorientierten Softwareentwurfs**
- **entsteht durch die Analyse und Überarbeitung vorhandener Designlösungen, setzt also Entwurfs- und Programmiererfahrung voraus.**
- kann in seiner **Struktur** verstanden werden als eine Menge von Klassen, die festgelegte Verantwortlichkeiten haben und in einer definierten Vererbungs- bzw. Benutzungsbeziehung zueinander stehen
- kann in seiner **Dynamik** verstanden werden als eine Menge von Objekten, die nach einem beschreibbaren Prinzip interagieren bzw. erzeugt werden
- kann immer nur zusammen mit dem Entwurfsproblem beschrieben werden, das es lösen soll.

Beschreibung von Entwurfsmustern

Eine **Musterbeschreibung** besteht meist aus den folgenden Teilen:

- dem **Namen** des Entwurfsmusters,
- dem **Problem**, das mit Hilfe des Musters gelöst werden soll,
- der **Kontext**, in dem sich das Problem stellt,
- der **Lösung**, mit der die Organisation von Klassen in einer Klassenhierarchie und die Gestaltung ihrer Interaktion vorgegeben werden,
- die (positiven und negativen) **Konsequenzen** der Musteranwendung.

Von Gamma et al. wird eine feiner ausgearbeitete Notation verwendet:

- Name
- Ziel
- Motivation
- Struktur
- Teilnehmer
- Zusammenarbeit
- Implementation
- Anwendbarkeit
- Konsequenzen

Unterteilung von Entwurfsmustern

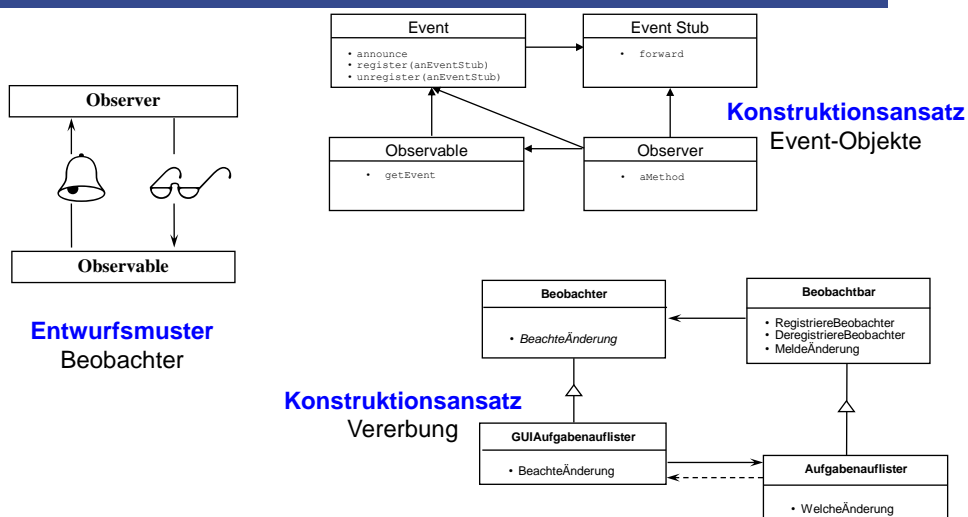
Der **allgemeine Teil eines Entwurfsmusters**:

- beschreibt die abstrakten Modellierungselemente und Bezüge für den softwaretechnischen Entwurf,
- beschreibt die zusammenarbeitenden Objekte und Klassen, die maßgeschneidert sind, um ein allgemeines Entwurfsproblem in einem bestimmten Kontext zu lösen,
- muß durch unterschiedliche Konstruktionsansätze konkretisiert werden.

Ein **Konstruktionsansatz** als der konstruktive Teil eines Entwurfsmusters:

- beschreibt die konkreten Elemente und Bezüge des softwaretechnischen Entwurfs,
- ist i.d.R. eine Implementationsvariante, die einen Aspekt des Musters betont und auf eine bestimmte Programmiersprache zugeschnitten ist.
- kann unmittelbar in eine programmiersprachliche Implementation umgesetzt werden.

Zusammenhang Entwurfsmuster und Konstruktionsansatz



Zum Nutzen von Entwurfsmustern

Entwurfsmuster

- helfen, existierende Softwareentwürfe zu analysieren und zu reorganisieren
- erleichtern die Einarbeitung in Software-Architekturen (z.B. Klassenbibliotheken, Rahmenwerke), solange sie auf der Basis von bekannten Entwurfsmustern dokumentiert sind
- sind "Mikroarchitekturen", die sich von erfahrenen Entwicklern als Bausteine innerhalb größerer Software-Architekturen wiederverwenden lassen (Wiederverwendung von Design-Lösungen statt Wiederverwendung von Code).
- stellen uns die Elemente einer Sprache, in der wir über Software-Architekturen nachdenken und kommunizieren können.
- sollen die softwaretechnische Qualität von Entwürfen erhöhen (z.B. ihre Wiederverwendbarkeit und Erweiterbarkeit).

Klassifizierung der GoF-Muster zusammengefasst

		Aufgabe		
		Erzeugungsmuster	Strukturmuster	Verhaltensmuster
Gültigkeitsbereich	Klassenbasiert	Objekterzeugung wird in Unterklassen verlagert	Vererbung wird genutzt, um Klassen zusammenzuführen	Vererbung wird verwendet, um Algorithmen und Kontrollfluß zu beschreiben
	Objektbasiert	Objekterzeugung wird an ein anderes Objekt delegiert	Möglichkeiten Objekte zusammen zu führen	Objekte arbeiten zusammen, um eine Aufgabe auszuführen, die ein einzelnes Objekt nicht in der Lage wäre zu erfüllen

GoF: 23 Entwurfsmuster klassifiziert

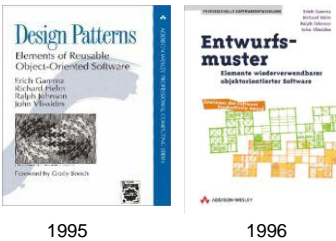
		Aufgabe		
		Erzeugungsmuster	Strukturmuster	Verhaltensmuster
Gültigkeitsbereich	Klassenbasiert	Fabrikmethode	Adapter (klassenbasiert)	Interpreter Schablonenmethode
	Objektbasiert	Abstrakte Fabrik Erbauer Prototyp Singleton	Adapter (objektbasiert) Brücke Dekorierer Fassade Fliegengewicht Kompositum Proxy	Befehl Beobachter Besucher Iterator Memento Strategie Vermittler Zustand Zuständigkeitskette

Zusammensetzung von Klassen und Objekten

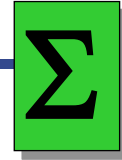
Zusammenarbeit von Klassen und Objekten

Ein Klassiker der Softwaretechnik

„Design Patterns: Elements of Reusable Object-Oriented Software“
der Gang of Four, mit 23 Entwurfsmustern.

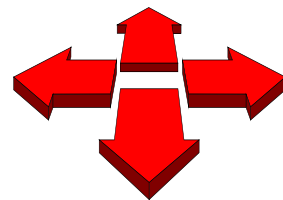


Zusammenfassung Entwurfsmuster



- Entwurfsmuster sind ein wesentliches Hilfsmittel des objektorientierten Software-Entwurfs.
- Zum softwaretechnischen Handwerkszeug gehört die Kenntnis über:
 - Erzeugungs-
 - Struktur- und
 - Verhaltensmuster
- Entwurfsmuster sind das zentrale konzeptionelle Bindeglied zwischen
 - den Mitteln einer objektorientierten Programmiersprache und
 - Überlegungen zur Architektur großer Softwaresysteme.

Programmierung von Graphical User Interfaces (GUIs)

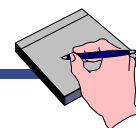


- Motivation
- GUI-Komponenten
- Reagieren auf Ereignisse
- Das Layout von GUI-Komponenten

Motivation – Nichts ist beständiger als der Wandel

- Technologiezyklen unterscheiden sich von fachlichen Zyklen.
- Wir müssen die fachlichen Kerne unserer Anwendungen so konstruieren, dass
 - die unausweichlichen Entwicklungen in der (GUI-)Technologie beherrschbar bleiben, und
 - fachlich motivierte Änderungen/Erweiterungen schnell, kostengünstig, und mit hoher Qualität realisiert werden können.
- Forderung deshalb: Trenne **Präsentation/Handhabung** und **Funktionalität**.
 - Es sollte aus formaler Sicht irrelevant sein, ob eine fachliche Operation über eine grafische oder eine textuelle Benutzungsschnittstelle angestoßen wird.
- Die Modell-Elemente unserer Entwurfsregeln folgen dieser Forderung:
 - **Werkzeuge** sind zuständig für die **Präsentation/Handhabung**.
 - **Services, Materialien** und **Fachwerte** bilden die **Funktionalität** ab.

GUI-Toolkits zur Werkzeugkonstruktion



- Betriebssysteme bieten seit einigen Jahren Unterstützung für grafische Oberflächen (z.B. Windows API, Macintosh Toolbox, Motif, ...).
- Diese APIs sind in der Regel sehr plattformabhängig und nicht objektorientiert.
- Um diese Systeme leichter zu handhaben, gibt es sogenannte **GUI-Toolkits** für Java, z.B. das **AWT** (Abstract Windowing Toolkit) und **Swing**.
- Diese
 - vereinfachen den Umgang,
 - erleichtern die Portierung,
 - stellen objektorientierte Schnittstellen zur Verfügung.
- Swing und AWT sind **objektorientierte** Toolkits zur Anbindung grafischer Benutzungsschnittstellen mit Java.
- Mit ihnen ist es möglich, den Quelltext zur Werkzeugkonstruktion (Handhabung/Präsentation) sauber von der Funktionalität zu trennen.

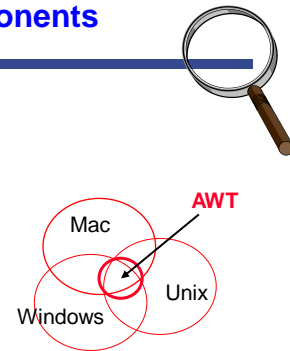
Swing baut auf dem AWT auf:

- etliche Komponenten wurden hinzugefügt;
- einige AWT-Komponenten wurden ersetzt;
- einige AWT-Komponenten werden in Swing weiterhin benutzt.



Hintergrund: AWT – Heavyweight Components

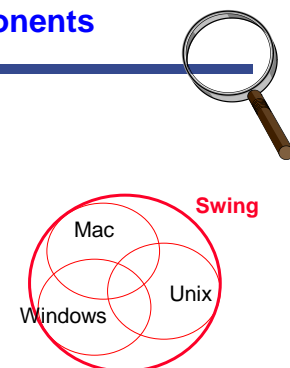
- AWT stützt sich auf das zugrunde liegende Betriebssystem ab. **Widgets** des Systems werden in Java zur Verfügung gestellt.
- Vorteile:
 - Gute Verarbeitungsgeschwindigkeit
 - Look & Feel des Systems ohne großen Aufwand unterstützt
- Nachteile:
 - Es muss ein gemeinsamer Nenner gefunden werden, damit das gleiche API auf verschiedenen Systemen realisiert werden kann.
 - Die Implementierung muss an jede Plattform angepasst werden.



Der Begriff „Widget“ leitet sich von „Window“ und „Gadget“ ab.

Hintergrund: Swing – Lightweight Components

- Swing-Komponenten sind leichtgewichtige Komponenten, d. h. sie stützen sich nicht auf Komponenten des zugrundeliegenden Betriebssystems ab, sondern implementieren die Optik selbst.
- Vorteile:
 - Hohe Anzahl von Komponenten einfach zu entwickeln, da plattformunabhängig
 - Kein Portierungsaufwand, nur einmalige Entwicklung
 - Verschiedene Look & Feels austauschbar
- Nachteile:
 - benötigen mehr Rechenleistung
 - verschiedene Look & Feels müssen aufwendig nachprogrammiert werden



Unsere erste Swing-Applikation

- **Look:**

- Ein Knopf mit dem Text:
- Ein Ausgabefeld



- **Feel:**

- Wenn der Knopf gedrückt wird, dann wird ein Zähler hochgezählt.

- **Die Fragen des Tages:**

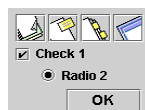
- Welche GUI-Komponenten (der GUI-Bibliothek **Swing**) stehen uns zur Konstruktion einer graphischen Benutzungsoberfläche zur Verfügung?
- Wie können wir diese Komponenten mit unseren fachlichen Klassen verbinden?
- Welche Möglichkeiten gibt es für das Layout einer graphischen Benutzungsschnittstelle?



Swing-Komponenten: eine Auswahl -1-



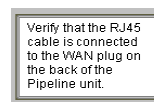
JLabel



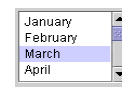
JButton



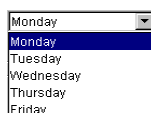
JTextField



JTextArea



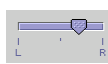
JList



JChoice



JScrollPane



JSlider



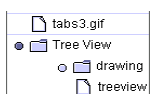
JToolTip



JToolBar



JProgressBar



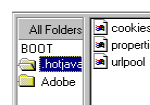
JTree



JTabbedPane



JTable

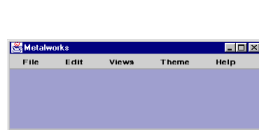


JSplitPane

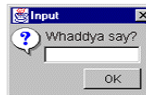


JMenu...

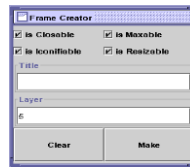
Swing-Komponenten: eine Auswahl -2-



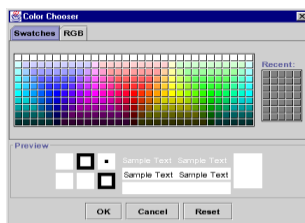
JFrame



JDialog



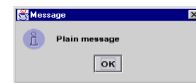
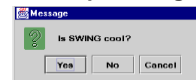
JInternalFrame



JColorChooser



JFileChooser

JOptionPane
.showMessageDialogJOptionPane
.showWarningDialogJOptionPane
.showInputDialogJOptionPane
.showConfirmDialog

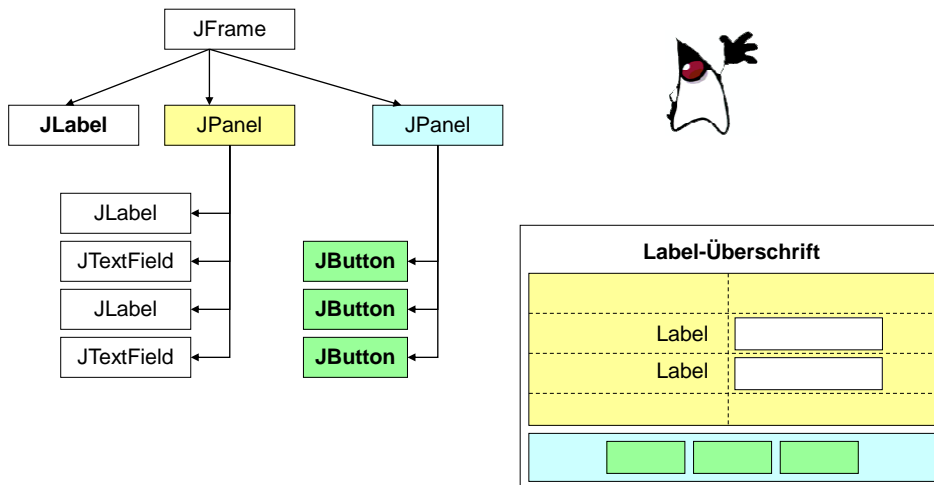
Auf dem Weg zu strukturierten Swing-Oberflächen: Komponenten erzeugen und schachteln



- Die einzelnen Komponenten einer Swing-Oberfläche werden hierarchisch angeordnet.
- Komponenten können unterschieden werden in **Containerkomponenten** und **atomare Komponenten**.
- Containerkomponenten können beliebige Komponenten (auch wieder Container) enthalten.
- Container bieten eine Schnittstelle, mit der Komponenten eingetragen werden können (Operation **add**).



Hierarchischer Aufbau einer Swing-Oberfläche



SE2 – OOPM – Teil 2

119

Top-Level Container

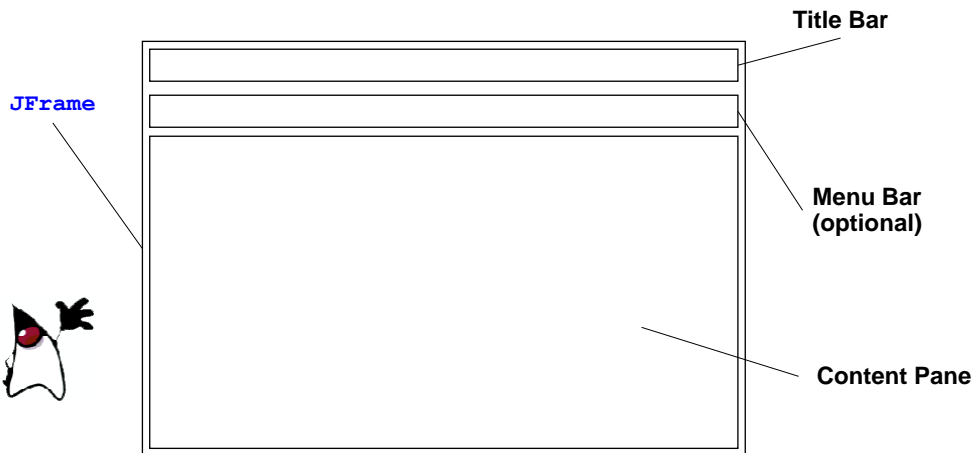
- An der Spitze der hierarchischen Struktur einer Oberfläche stehen die **Top-Level Container**. Sie korrespondieren mit den Fenstern, die vom jeweiligen Betriebssystem zur Verfügung gestellt werden.
- Top-Level Container in Swing sind beispielsweise **JFrame**, **JDialog**, **JOptionPane**, **JApplet**.
- Ein **JFrame** enthält unter anderem einen so genannten **Content Pane**. Dies ist der Container, in den die Hauptkomponenten der Oberfläche eingetragen werden.



SE2 – OOPM – Teil 2

120

Struktur eines JFrame



SE2 – OOPM – Teil 2

121

Werkzeugkonstruktion: Nächste Schritte



- Wir zerlegen Werkzeuge immer in eine **Werkzeug-Klasse** und eine **UI-Klasse**.
- Die Werkzeug-Klasse **vermittelt** zwischen der grafischen Schnittstelle der UI-Klasse und den fachlichen Klassen.
 - Die Werkzeug-Klasse **erzeugt** ein Exemplar ihrer **UI-Klasse** im eigenen Konstruktor.
- Die UI-Klasse eines Werkzeugs hat die Aufgaben, die **GUI-Komponenten** zu **erzeugen**, zu layouten und zu **verwalten**.
 - Eine UI-Klasse **erbt nicht von UI-Framework-Klassen** wie **JFrame** oder **JPanel**, um die eigene Schnittstelle schmal zu halten. Sie definiert als oberste UI-Komponente üblicherweise einen **JFrame**.
 - Eine UI-Klasse stellt die für ihre Werkzeug-Klasse relevanten **UI-Elemente über Getter** an ihrer Schnittstelle zur Verfügung.
 - Eine UI-Klasse hat keine Abhängigkeiten zu anderen Elementtypen und verwendet nur Importe aus dem UI-Framework.
 - Eine UI-Klasse sollte als **paketinterne** Klasse deklariert werden.

SE2 – OOPM – Teil 2

122

Kontrollfluss in Anwendungen – Traditionelles Prinzip: Eingabe, Verarbeitung, Ausgabe

OPERATO 005

HAUPTMENUE

- 1 Umsatzverarbeitung
- 2 Beratungsunterstützung/Abfragen
- 3 Kontoauszug
- 4 Bestandspflege
- 5 3270
- 6 Daten- und Sachgebietserfassung
- 7 SB-Verwaltung
- 8 Abstimmung Arbeitsplatz
- 9 Systemdaten

- Auswahl

ENTER F10=EXPERTE F13=ABMELDUNG

SE2 - OOPM - Teil 2

123

Eingabe, Verarbeitung, Ausgabe – Merkmale

- **Prinzipielle Struktur des Programmtextes:**

```

While not ende do
    input, process, output.
process:
    if lastInput = X
    then doX
    else if lastInput = Y
    then doY
    else error
  
```

- **Vorteile**

- Vollständige Kontrolle des Programms ist beim Anwendungsentwickler.
- *Effiziente* Programmentwicklung möglich

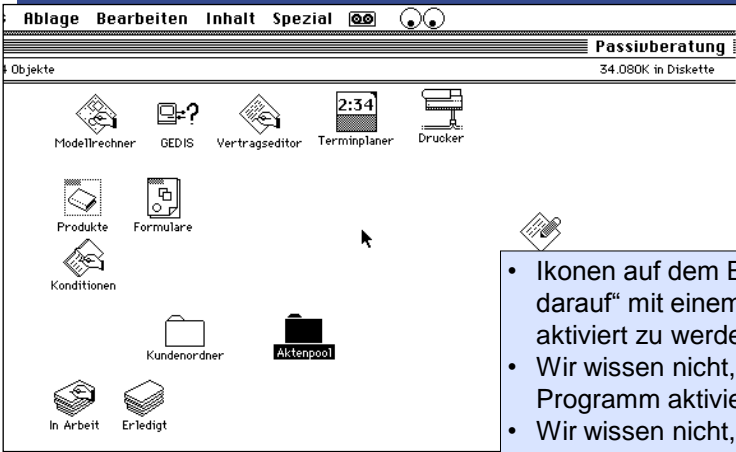
- **Nachteile**

- Keine Anleitung für die Trennung von fachlichem und GUI-spezifischem Code.
- Austausch der Oberfläche (des GUIs) tendenziell schwierig.

SE2 - OOPM - Teil 2

124

Welche Art von Systemen wollen wir bauen (können)?



Passiüberlegung
34.080K in Diskette

Objekte

Modellrechner GEDIS Vertragseditor Terminplaner Drucker

Produkte Formulare

Konditionen

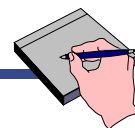
Kundenordner Aktenpool

In Arbeit Erledigt

- Ikonen auf dem Bildschirm „warten darauf“ mit einem Doppelklick aktiviert zu werden. Sie sind **reaktiv**.
- Wir wissen nicht, wann welches Programm aktiviert wird.
- Wir wissen nicht, in welcher Reihenfolge aktivierte Programme vom Benutzer verwendet werden.

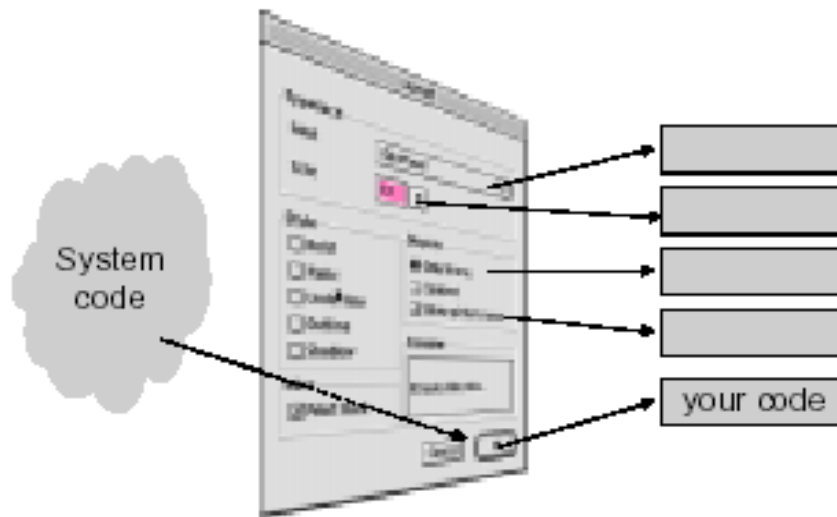
SE2 – OOPM – Teil 2 125

Merkmale reaktiver Software



- **Grundsätzlich:**
 - Steuerung des Kontrollflusses liegt außerhalb des Quelltextes des Anwendungsentwicklers.
- **Vorteile:**
 - Kenntnisse über die Spezifika des **Eventing** sind nicht notwendig.
 - Trennung von GUI- und Applikationscode wird erleichtert. Daraus folgen bessere Möglichkeiten der Änderbarkeit.
- **Nachteile:**
 - (Aufwendige) Einarbeitung in/Verständnis für die zugrundeliegenden GUI-Bibliotheken notwendig.

Reaktive Programmierung – die grundlegende Idee



© Michael Kölling, Monash University, Australia

SE2 – OOPM – Teil 2

127

Ereignisverarbeitung mit Ereignissen (Events)

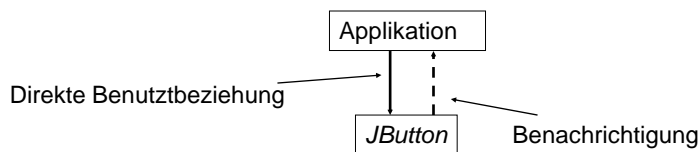
- Jede Mausbewegung, jeder Mausklick und jeder Tastendruck wird vom Systemcode einer GUI registriert.
- Für jede dieser Aktionen wird ein **Ereignis/Event** (vom engl. event) erzeugt.
- Events können sehr elementare Aktionen sein (Mausbewegung) oder sich aus mehreren Aktionen zusammensetzen (ein „Mausklick“ besteht beispielsweise aus den Aktionen „Mausknopf gedrückt“ und „Mausknopf wieder losgelassen“).
- Für viele Komponenten gibt es **High-Level-Events**, die die typischen Aktionen auf einer Komponenten modellieren (Bsp.: „button pressed“ auf einem Button).
- Ein solches Event wird an alle Teile des Anwendungssystems verschickt, die sich für diese Aktion bei einer GUI-Komponente **angemeldet** haben.

SE2 – OOPM – Teil 2

128

Anbindung der Applikation an die Oberfläche – das Prinzip

- Der Anwender löst an der Oberfläche **Ereignisse** aus.
- Die Applikation kann auf solche Ereignisse reagieren, indem sie **Listener** implementiert.
- Durch diesen Benachrichtigungsmechanismus kann die Oberflächenkomponente von der Applikation verwendet und die Applikation von der Komponente benachrichtigt werden,
 - ➔ ohne dass eine direkte zyklische Benutzt-Beziehung entsteht und
 - ➔ ohne dass die Komponente alle ihre Listener explizit kennen muss.



SE2 – OOPM – Teil 2

129

Das Konzept der Listener

- Damit Anwendungscode vom Systemcode aufgerufen werden kann, muss der Anwendungscode eine Schnittstelle haben, die dem System bekannt ist.
- Java stellt zu diesem Zweck die **Listener-Interfaces** zur Verfügung.
- Beispiel **ActionListener**:

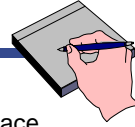
```
public interface ActionListener
{
    void actionPerformed(ActionEvent event);
}
```



SE2 – OOPM – Teil 2

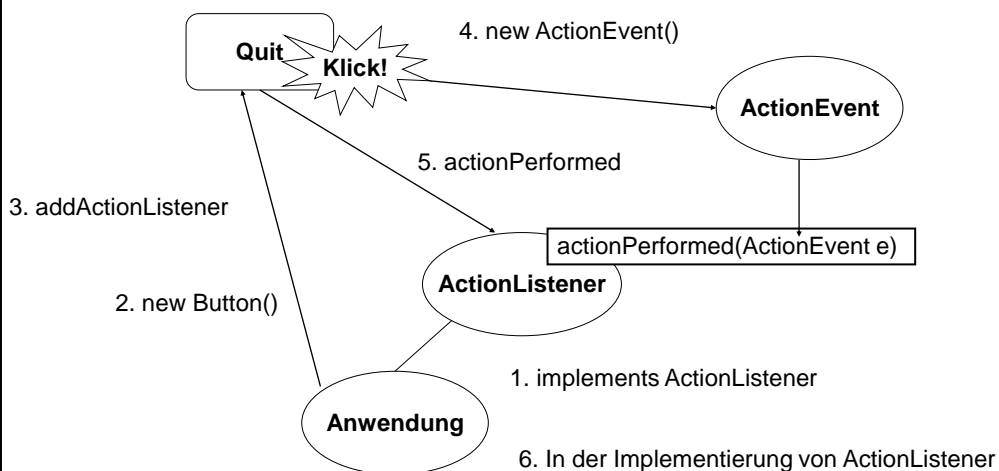
130

Das Konzept der Listener (II)

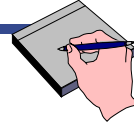


- Im Anwendungscode wird eine Klasse geschrieben, die dieses Interface implementiert. In der Implementierung von `actionPerformed` steht dann der Quelltext, der auf das Event reagiert.
- Damit dieser Code wirklich aufgerufen werden kann, meldet der Anwendungscode die implementierende Klasse bei der GUI-Komponente als Listener an.
- Wenn ein Ereignis eintritt (etwa ein Buttonklick), erzeugt die Komponente (der Button) ein Event-Objekt und übergibt dieses als Parameter nacheinander **allen Listnern**, die sich bei der Komponente angemeldet haben, durch Aufruf ihrer Operation `actionPerformed`.
- Das Event-Objekt enthält dann Informationen über das Ereignis (auslösende Komponente etc.).

Das Konzept der Listener im schematischen Ablauf



Java Spezial: Listener mit anonymen inneren Klassen



- Ein Sprachmechanismus von Java wurde speziell für eine vereinfachte Implementierung von Listener-Interfaces entworfen: die **anonymen inneren Klassen**.
- Mit diesem Mechanismus kann an einer Stelle, an der ein Exemplar einer ein Interface implementierenden Klasse übergeben werden soll, direkt ein **spezieller Ausdruck** stehen (hier rot hervorgehoben):

```
myButton.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        // geeignete Reaktion; Zugriff auf private Felder
        // des „umgebenden“ Exemplars möglich!
    }
});
```

- Dieser Ausdruck **erzeugt** nicht nur an der Aufrufstelle **ein Exemplar** einer Klasse, sondern **definiert auch** gleich **die erzeugende Klasse selbst** – namenlos, wie hier zu sehen, deshalb der Begriff anonyme innere Klasse.

Anonyme innere Klassen



- Wir erinnern uns: Java erlaubt die Schachtelung von Klassen durch **geschachtelte Klassen**. Neben den statischen geschachtelten Klassen gibt es auch drei Arten von geschachtelten Klassen, die **innere Klassen** genannt werden.
- Ein Exemplar einer inneren Klasse benötigt immer ein Exemplar der Klasse, in die die innere Klasse hineingeschachtelt ist; das umgebende Objekt kann man als das **Wirtobjekt** bezeichnen, das innere Objekt als **Parasit-Objekt**.
- In Java ist das Verhältnis zwischen geschachtelten und umgebenden Klassen **sehr eng**: Zwei Exemplare beider Klassen können wechselseitig auf **alle Exemplarvariablen** (auch auf private) des jeweils anderen **zugreifen**.
- Anonyme innere Klassen können sogar auf die lokalen Variablen der Methode zugreifen, in der sie definiert wurden; allerdings nur, wenn diese als **final** deklariert sind.



Komponenten und Listener: Beispiele

- Wichtige Listener:

- `JRadioButton`: `ActionListener`, `ItemListener`
- `JList`: `ActionListener`, `ListSelectionListener`
- `JComboBox`: `ActionListener`, `ItemListener`
- `TextField`: `ActionListener`



Zur Vervollständigung: Events, Listener

- Es gibt in AWT und Swing verschiedene Typen von Ereignissen. Zu jedem Typ existiert eine **Event-Klasse**.
- Die Event-Objekte **tragen** alle nötigen **Informationen** über das aktuelle Ereignis mit sich. Zum Beispiel verfügen alle Java-Events über die Operationen **getSource** (Event-Quellkomponente) und **getID** (Event-Typ als Konstante). Viele Events verfügen auch über **consume**, um das Event zu „verbrauchen“.
- Event-Objekte werden von angemeldeten **Listnern** verarbeitet. Ein **Listener-Interface** definiert alle notwendigen Antwortmethoden, mit denen auf bestimmte Ereignisse reagiert werden kann.



Events und Listener: Weitere Beispiele

Event-Typ und wichtige Methoden	Listener	Methoden
FocusEvent isTemporary()	FocusListener	focusGained() focusLost()
ItemEvent getItem() getItemSelectable() getStateChange()	ItemListener	itemStateChanged()
TextEvent	TextListener	textValueChanged()
ComponentEvent getComponent()	ComponentListener	componentHidden() componentMoved() componentResized() componentShown()
Es gibt noch mehr Events...		



SE2 – OOPM – Teil 2

137

Event / EventListener zusammengefasst

Der Mechanismus zur Behandlung von Oberflächen-Ereignissen in Java ist der sog. **Event/EventListener-Mechanismus**.

Dieser ist allgemein so aufgebaut:

1. Jede GUI-Komponente implementiert für eine bestimmte Art von Ereignissen eine **add<EventListener>()** – Methode.
2. Über diese Methode kann an einer GUI-Komponente ein Objekt „angemeldet“ werden, welches das Interface **<EventListener>** implementiert.
3. Wird ein entsprechendes Ereignis durch den Benutzer ausgeführt, so werden alle angemeldeten Listener-Objekte benachrichtigt.
4. Informationen über den Ereignistyp sowie weitere evtl. notwendige Informationen werden über ein Event-Objekt übermittelt.

SE2 – OOPM – Teil 2

138

Werkzeugkonstruktion: Ereignisverarbeitung



- Wir haben nun das technische Rüstzeug, um die Entwurfsregeln zur Ereignisverarbeitung nachvollziehen zu können:
 - Eine UI-Klasse **stellt** die für ihre Werkzeug-Klasse relevanten **UI-Elemente** über Getter **in ihrer Schnittstelle zur Verfügung**.
 - Die Werkzeug-Klasse **erzeugt** für diese UI-Widgets **Listener**, die passende Aktionen ausführen, und **registriert** diese an den Widgets.
 - Die Werkzeug-Klasse gibt die anzuzeigenden Materialien oder Fachwerte in die UI, in folgender Weise:
 - Die Werkzeug-Klasse **holt** sich von der UI-Klasse ein **Widget** und **setzt** bei diesem die anzuzeigenden **Informationen** des Materials oder des Fachwerts.
 - Sofern nötig, wird das darzustellende Element über einen **Formatierer** für die jeweilige Darstellung angepasst.

Die Werkzeug-Klasse reagiert auf Ereignisse!

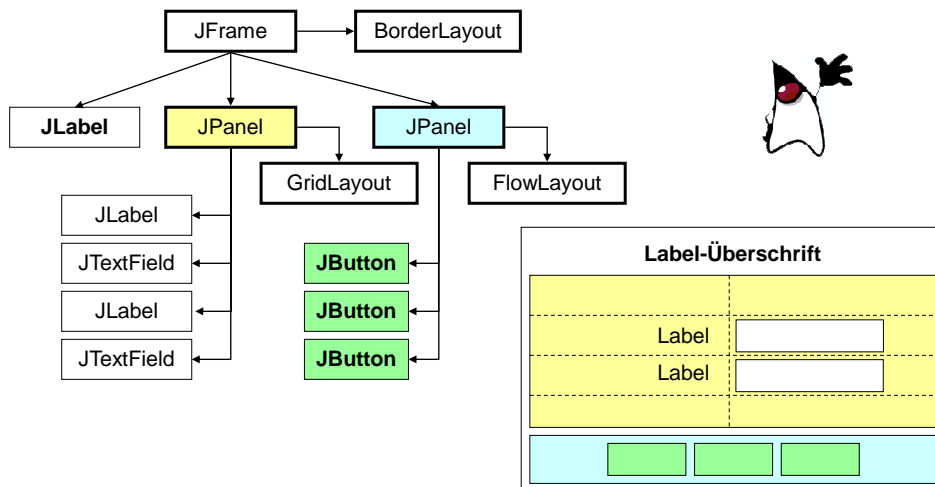
Die Werkzeug-Klasse bestimmt, wie ein Material oder ein Fachwert dargestellt wird!

Layout festlegen

- In Java ordnen **Layout-Manager** die Komponenten in einem Container an. Oberflächen werden also nicht pixelgenau erstellt, sondern immer relativ zueinander.
- Das erleichtert beispielsweise:
 - das **Vergrößern** und **Verkleinern** von Fenstern inklusive Inhalt,
 - die **Darstellung** gleicher, aber unterschiedlich großer Widgets **auf unterschiedlichen Plattformen** (ein typischer Windows-Button kann völlig anders aussehen als ein Mac-Button).
- Jeder Container besitzt als Default einen Layout-Manager.



Container mit Layout



SE2 - OOPM - Teil 2

141

Layout-Manager

- Jeder Container hat einen Layout-Manager, der gesetzt und abgefragt werden kann:

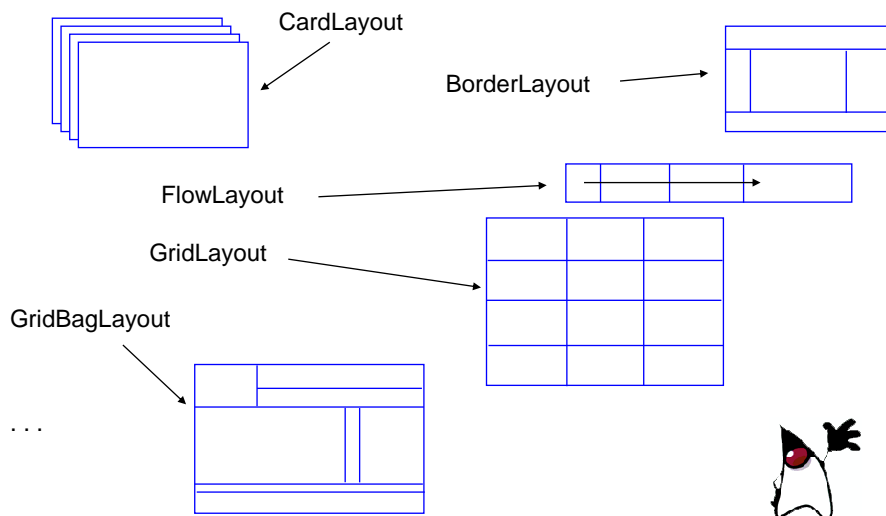

```
void setLayout( LayoutManager );
LayoutManager getLayout();
```
- Das Standard-Layout für den Content-Pane eines **JFrame** ist das **BorderLayout**.
- Für Fortgeschrittene: Man kann eigene Layout-Klassen schreiben und benutzen.
- Es gibt in Java bereits zahlreiche Layout-Manager, von denen wir nur die wichtigsten betrachten....



SE2 - OOPM - Teil 2

142

Verschiedene Layout-Manager



SE2 - OOPM - Teil 2

143

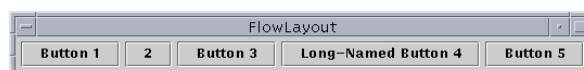
Layouts im Beispiel: FlowLayout

- Das **FlowLayout** ordnet Komponenten in einer Reihe an und bricht die Komponenten notfalls in Zeilen um.
- Die Abstände zwischen den Komponenten und die Ausrichtung zum umgebenden Frame können angegeben werden.
- **FlowLayout** befindet sich im Paket `java.awt`.

```
Container contentPane = getContentPane();
FlowLayout fl = new FlowLayout();
contentPane.setLayout( fl );

contentPane.add(new JButton("Button 1"));
contentPane.add(new JButton("2"));
contentPane.add(new JButton("Button 3"));
contentPane.add(new JButton("Long-Named Button 4"));
contentPane.add(new JButton("Button 5"));

fl.setAlignment( FlowLayout.CENTER ); // LEFT, RIGHT, ...
fl.setHgap( 20 ); // Horizontales spacing
fl.setVgap( 20 ); // Vertikales spacing
```



SE2 - OOPM - Teil 2

144

Werkzeugkonstruktion: Layout



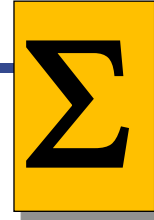
- Relevante Entwurfsregeln zum Thema Layout:
 - Die UI-Klasse eines Werkzeugs hat die Aufgaben, die **GUI-Komponenten** zu erzeugen, zu **layouten** und zu verwalten.
- Mit anderen Worten: Die **UI-Klasse** ist ein „Sammelbehälter“ für die UI-Widgets; sie ist **vollständig für das Layout** der UI-Komponenten **zuständig**.

Zusammenfassung GUI-Programmierung



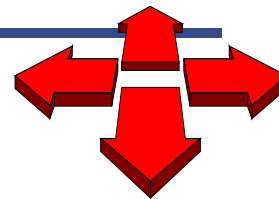
- Wir haben grundlegende Techniken zur Konstruktion **reaktiver Programme** mit einer GUI-Bibliothek kennengelernt.
- Am Beispiel der GUI-Toolkits **AWT** und **Swing** haben wir gesehen, dass
 - es sehr unterschiedliche **GUI-Komponenten** geben kann;
 - **Listener** eine Möglichkeit darstellen, um Anwendungscode durch GUI-Code über **Ereignisse** informieren zu lassen;
 - das **Layout** einer grafischen Benutzeroberfläche sehr flexibel mit Layout-Managern gestaltet werden kann.
- Mit diesen Kenntnissen können wir nun unsere ersten **Desktop-Anwendungen** konstruieren.

Zusammenfassung SE2-Entwurfsregeln



- Die **SE2-Entwurfsregeln** für interaktive Anwendungen (insbes. Rich-Clients) geben konstruktive Hinweise für die Strukturierung von Softwaresystemen mit einer grafischen Oberfläche.
- Sie definieren vier Elementtypen:
 - **Materialien** – veränderbare fachliche Objekte, die üblicherweise Arbeitsergebnisse modellieren;
 - **Fachwerte** – unveränderliche fachliche Abstraktionen;
 - **Services**, die systemweit fachliche Dienstleistungen anbieten und häufig Materialien verwalten;
 - **Werkzeuge** zur interaktiven Bearbeitung von Materialien.
- Der Entwurf von Materialien, Fachwerten und Services ist primär fachlich anspruchsvoll, während die Konstruktion von Werkzeugen vor allem technisch anspruchsvoll ist.

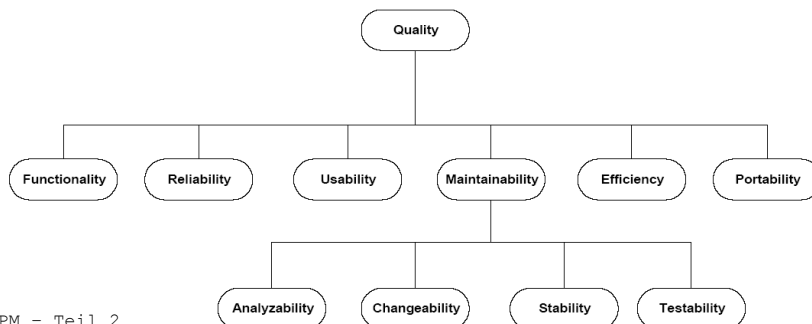
Übersicht Klassenentwurf



- Erneute Motivation: Softwarequalität
- Richtlinien für den Klassenentwurf
 - Anwendungsfachliche und technische Klassen
 - Kopplung
 - Kohäsion

Motivation: Qualität von Software

- Wir streben nach **möglichst hoher Qualität** bei der Softwareerstellung.
- Die Qualität von Software kann nach B. Meyer differenziert werden in
 - **Äußere bzw. externe Qualität** (Funktionalität, Zuverlässigkeit, Benutzbarkeit, Effizienz) und
 - **Innere bzw. interne Qualität** (Verständlichkeit, Wartbarkeit, Modularität).
- In der Benutzung zählt nur die äußere Qualität, die aber über interne Qualität erreicht und gesichert wird.



SE2 - OOPM - Teil 2

149

Qualität von Klassenentwürfen

- Differenzieren des Begriffs **Entwurf**:
 - Als Bezeichnung für die **bestehende Struktur** eines Systems.
 - Im Sinne von „Hier haben wir einen guten Entwurf.“
 - Welche Elemente existieren? Wie arbeiten diese zusammen?
 - Als Bezeichnung der **Tätigkeit des Entwerfens**:
 - Im Sinne von „Beim Klassenentwurf (sprich: beim Entwerfen der Klassen) haben wir festgestellt, dass die Aufteilung nicht einfach ist.“
 - Beinhaltet neben dem Treffen von Entscheidungen auch eine Planungskomponente.
 - Prozessunterstützung beispielsweise durch CRC-Karten.
- **Aber wann ist ein Klassenentwurf im ersten Sinn „gut“?**

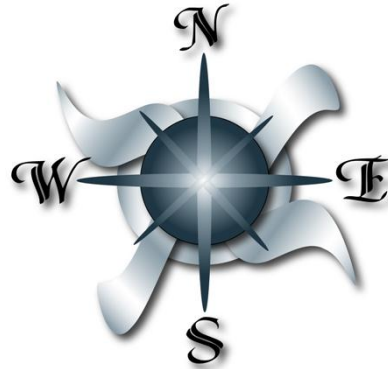
SE2 - OOPM - Teil 2

150

Wie kommen wir zu einem guten Entwurf?

• Richtlinien für den Klassentwurf

- Allgemein: Entwurf nach Zuständigkeiten
- Lose Kopplung
 - Geeignete Schnittstellen wählen
 - Entwurfsentscheidungen kapseln
 - Geheimnisprinzip
 - Zyklen vermeiden
 - Law of Demeter
- Hohe Kohäsion
 - für Klassen und für Methoden
 - Code-Duplizierung vermeiden
 - Geeignete Bezeichner wählen
 - Große Einheiten vermeiden



Leitbild: Entwurf nach Zuständigkeiten

- **Entwurf nach Zuständigkeiten** (engl.: Responsibility-Driven Design) ist eine Entwurfsphilosophie, die von Rebecca Wirfs-Brock et al. Ende der 80er Jahre formuliert wurde.

„Objects are not just simple bundles of logic and data. They are responsible members of an object community.“

- Jedes Objekt in einem objektorientierten System sollte für eine klar definierte Aufgabe zuständig sein.
- Dieser Ansatz geht u.a. auf die Forderung nach „Separation of Concerns“ von Dijkstra zurück.

Aktuelles Buch: Wirfs-Brock, McKean: *Object Design – Roles, Responsibilities and Collaborations*, Addison-Wesley 2002

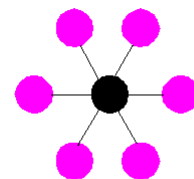
Wiederholung: Trenne Fachlogik und Technik

- Anwendungsfachliche Klassen, die vor allem die Fachlogik modellieren, sollten deutlich von rein technisch motivierten Klassen unterscheidbar sein.

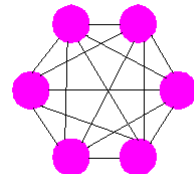


Kopplung, objektorientiert

- **Kopplung** (engl.: coupling) bezeichnet den **Grad der Abhängigkeiten** zwischen den **Einheiten** eines Softwaresystems.
 - **Abhängigkeiten** können aus objektorientierter Sicht sein:
 - Benutzt-Beziehungen
 - Vererbungsbeziehungen
 - **Einheiten** können sein:
 - Methoden
 - Klassen
 - Pakete
 - Subsysteme
- Je mehr Abhängigkeiten in einem System existieren, desto stärker ist die Kopplung.
- Wir streben beim Klassenentwurf nach **möglichst loser Kopplung** (engl.: loose coupling).



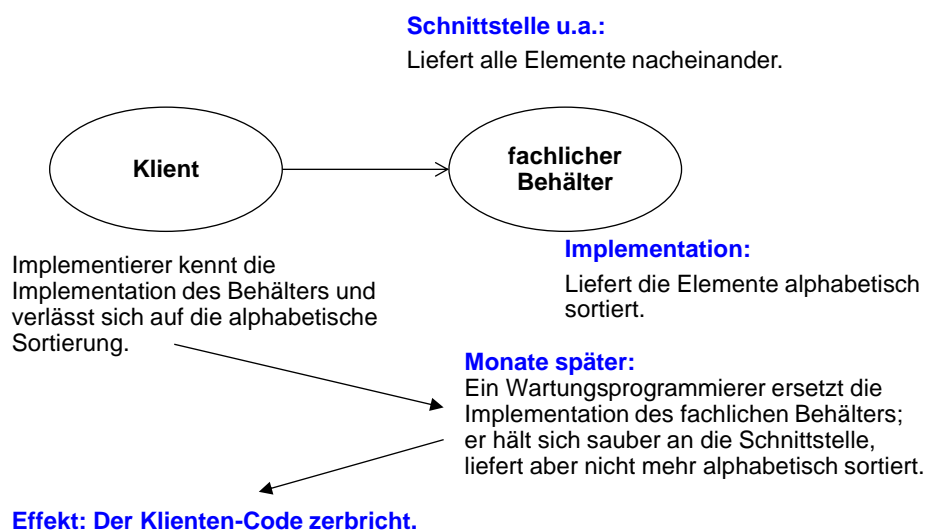
Wenn eine Klasse mit allen anderen Klassen verbunden ist, ist das ihre maximal mögliche Kopplung.



Lose Kopplung

- Lose Kopplung zwischen Klassen ist erstrebenswert, weil...
 - wir den Text einer einzelnen **Klasse besser verstehen** können, wenn wir nicht viele andere Klassendefinitionen lesen müssen;
 - wir eine Klasse **leichter ändern** können, wenn nicht viele andere Klassen von dieser Änderung betroffen sind.
- Mit anderen Worten: lose Kopplung erleichtert die Wartung.
- Wir unterscheiden **explizite** und **implizite** Kopplung:
 - Eine Kopplung ist **explizit**, wenn sie **formal nachweisbar** ist.
Bsp.: Zugriff eines Klienten auf öffentliche Exemplarvariablen ist durch Analysewerkzeuge als explizite enge Kopplung nachweisbar.
 - **Implizite** Kopplung ist **nicht formal nachweisbar**; sie ist deshalb deutlich unangenehmer.
Bsp.: Ein Entwickler implementiert sein Wissen über die Interna eines Dienstleisters in eine Klientenklasse hinein.

Beispiel für implizite Kopplung



Geeignete Schnittstellen wählen

- Über die Schnittstellen von Klassen werden die meisten Abhängigkeiten definiert.
- Für eine Schnittstelle sollte gelten, dass sie möglichst...
 - **vollständig**,
 - **klar**,
 - **konsistent**,
 - **minimal**,
 - und **bequem benutzbar** ist.
- Wie bei vielen Entwurfsrichtlinien gilt auch hier: diese Ziele können sich teilweise widersprechen.

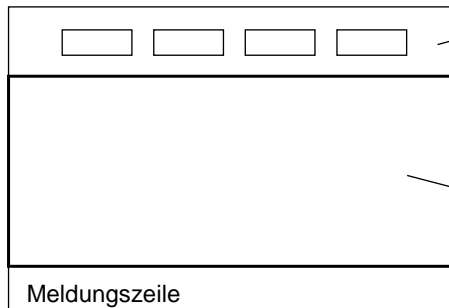


Konkretisierung: Verwende Interfaces

- Wir haben Interfaces als ein explizites Sprachkonstrukt von Java kennen gelernt, mit dem die Schnittstelle einer Klasse explizit modelliert werden kann.
- Häufig empfiehlt es sich, die Kopplung zwischen zwei Klassen zu verringern, indem ein Interface modelliert wird, das nur die Operationen zusammenfasst, die die eine Klasse von der anderen benutzt.
 - Erfüllt unter anderem die Forderung nach schmalen Schnittstellen.
 - Macht eine konkrete Kopplung deutlicher.
 - Kann helfen, statische Zyklen aufzulösen.
- Zentrales Zitat aus der Einleitung zu Gamma et al.:

„Program to an interface, not an implementation.“

Beispiel: geschachtelte GUI



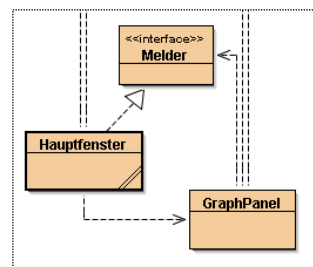
Hauptfenster, ein JFrame mit BorderLayout: JButtons in der Nord-Komponente, ein JLabel in der Südkomponente.

Als Center-Komponente: **GraphPanel**, ein spezieller JPanel, der Knoten eines Graphen darstellen kann.

Problem: **Hauptfenster** muss **GraphPanel** kennen, um beim Klick auf einen Button einen neuen Knoten auf dem **GraphPanel** zu erzeugen. Aber der **GraphPanel** muss auch das **Hauptfenster** kennen, weil Aktionen im Panel zu Meldungen in der Meldungszeile führen können. Was tun?

Lösung: Abhängigkeit per Interface explizit machen

- Das **GraphPanel** muss nicht die gesamte Schnittstelle des **Hauptfensters** kennen. Es benötigt lediglich eine Schnittstelle, über die Meldungen ausgegeben werden können.
- Das **Hauptfenster** implementiert deshalb ein Interface **Melder** mit einer Operation **melde**; wenn das **Hauptfenster** den **GraphPanel** erzeugt, übergibt es als Parameter eine Referenz auf sich selbst vom statischen Typ **Melder**.
- Der direkte Zyklus zwischen den beiden Klassen ist damit, zumindest statisch, aufgelöst.
- Zur Laufzeit besteht der Zyklus aber weiter, beide Objekte verfügen über eine Referenz auf das jeweils andere Objekt.



Entwurfsentscheidungen kapseln

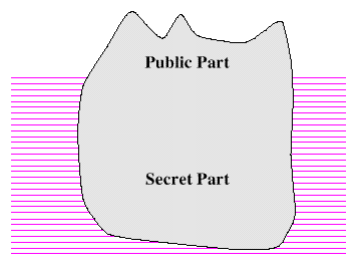
- Modellieren ist das Fällen von Entwurfsentscheidungen.
- Jede Entscheidung wirkt sich auf nachfolgende Entscheidungen aus.
- Es sollten deshalb bei der Modellierung die Entscheidungen **zuerst** getroffen werden, die sich mit **größter Wahrscheinlichkeit nicht ändern** werden (unveränderliche Kernabstraktionen eines Anwendungsbereichs).
- Entscheidungen hingegen, die sich mit **hoher Wahrscheinlichkeit ändern** können (wie die Art der Benutzungsschnittstelle, Technologien), sollten wir **kapseln** und damit **leichter austauschbar** halten.
- Wie identifizieren wir diese? Wir brauchen
 - ausreichend Informationen über den Kontext
 - Erfahrung
 - Augenmaß ...

Geheimnisprinzip

Geheimnisprinzip: Nur relevante Merkmale einer Entwurfs- und Konstruktionseinheit sollten für Klienten sichtbar und zugänglich sein.

Details der Implementierung sollten verborgen bleiben.

Schon allein das Wissen über Interna kann missbraucht werden!



Das **Geheimnisprinzip** (Information Hiding) geht zurück auf die Arbeit von Parnas:
D. L. Parnas, On the criteria to be used in decomposing systems into modules, Communications of the ACM, Vol 15:12, p.1053-1058, Dec. 1972.

Zyklen vermeiden

- Klassen können in einer zyklischen Abhängigkeit zueinander stehen.
- Klassen-Zyklen haben etliche Nachteile:
 - Die beteiligten Klassen lassen sich **schlecht unabhängig** voneinander **testen**.
 - Die **Initialisierungsreihenfolge** der entstehenden Objekte ist möglicherweise **unklar**.
 - Es gibt **keinen** offensichtlichen **Einstiegspunkt**, um sich in den Entwurf einzulesen.
- In ähnlicher Weise gelten diese Aussagen für andere Einheiten des Softwareentwurfs (Pakete, Subsysteme).
- Zyklen sind oft nicht leicht zu erkennen.
- Generell gilt die Regel:
Zyklische Abhängigkeiten sollten vermieden werden.

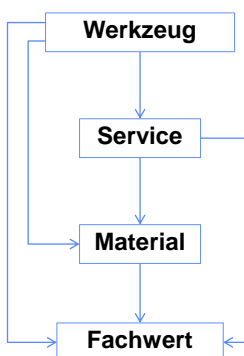


SE2 – OOPM – Teil 2

163

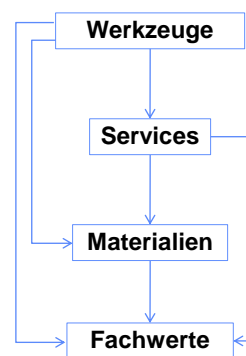
Beispiel: keine Paket-Zyklen durch Entwurfsregeln

Elementtypen:



- Die **SE2-Entwurfsregeln** benennen vier **Elementtypen**, aus denen sich ein interaktives System zusammensetzt, und legen erlaubte Benutzt-Beziehungen fest.
- Für SE2 empfehlen wir die Verwendung von entsprechend benannten **Paketen**: Werkzeuge, Services, Materialien und Fachwerte.
- Wenn die Entwurfsregeln in einem Softwaresystem auf Klassenebene eingehalten sind, dann sind sie transitiv **auch auf Paketebene eingehalten**.

Pakete:



SE2 – OOPM – Teil 2

164

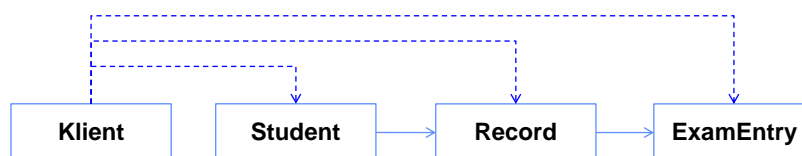
Law of Demeter, Original

- Ursprüngliche Definition (als **Law of Demeter for Functions/Methods**) war bereits lediglich eine **Entwurfsrichtlinie**:
 - Eine **Methode m** eines **Objektes o** sollte ausschließlich Methoden von folgenden Objekten aufrufen:
 - von o selbst;
 - von Parametern von m;
 - von Objekten, die m erzeugt;
 - von Exemplarvariablen von o.
- Umgangssprachlich: „Sprich nur mit Deinen engsten Freunden!“
- Hinweise auf Verletzungen sind lange Methodenaufkettungen in einer Zeile:
 - `student.getRecord().getExamEntry("SE2/2006").addResult(93);`



Lange Aufrufzeilen = hohe Kopplung

- `student.getRecord().getExamEntry("SE2/2011").addResult(93);`



- Wenn wir das Traversieren von Objektgeflechten in einer Methode ausimplementieren, **koppeln** wir die Methode zu stark an eine Struktur, deren Änderung dadurch erschwert wird.

Law of Demeter, heute

- Zeitgemäße Anpassungen:
 - „If you delegate, delegate fully.“
 - „Don't message your delegate object's objects.“
 - „Don't be aware of how some object works. Don't work at a lower level than is necessary.“
- **Tell, don't ask.**
- Auch beim Einhalten des LoD hilft immer wieder die Frage: **Wer sollte für diese Aufgabe zuständig sein?**

Law of Demeter, Beispiele

- Statt:
`student.getRecord().getExamEntry("SE2/2006").addResult(93);`
besser:
`student.scored(markedExam);`
- **Nicht** einen Satelliten nach seinen Antriebsdüsen **fragen** und eine davon veranlassen, etwas mehr Gas zu geben; stattdessen dem Satelliten **sagen**, dass er sich neu ausrichten soll.
- **Getter** und **Setter** als Verstoß gegen das LoD:
 - Avoid thinking in terms of getting and setting variables: „I know you're in there, variable; come out with your hands up. If it wasn't for this pesky object-orientation that's has been foisted upon us and that I don't have the time or inclination to get to understand, I'd be able to read and write your state.“ (Deacon, Object-Oriented Analysis and Design, Addison-Wesley 2005)

Kohäsion, objektorientiert

- Mit **Kohäsion** (engl.: cohesion) bezeichnen wir den Grad (Anzahl und Verschiedenheit) der Aufgaben, die eine Softwareeinheit zu erfüllen hat.
- Wenn eine Einheit für genau eine logische Aufgabe zuständig ist, dann sprechen wir von **hoher Kohäsion** (engl.: high cohesion).
- Auf Ebene des Klassenentwurfs unterscheiden wir
 - Kohäsion von Methoden
 - Kohäsion von Klassen
- Wir streben nach **möglichst hoher Kohäsion**.



Bedeutung des Wortes Kohäsion in der Chemie und der Physik:

„Molekulare Kraft zwischen Molekülen eines Stoffes. Sie bewirkt den **inneren Zusammenhalt** eines Stoffes.“

In der Linguistik:

Verknüpfung von Textelementen (Sätze, Teilsätze, Redeeinheiten) zu einer **sinnvollen Einheit** auf der Oberfläche .

SE2 – OOPM – Teil 2

169

Kohäsion von Methoden

- Eine Methode sollte nur für genau eine wohldefinierte Aufgabe zuständig sein.

```
public enum TrigoFunc {
    sin, cos, tan, arctan, ...
}
...
public double trigo(TrigoFunc func,
    double x) ...
```

Methode mit
niedriger Kohäsion

Methoden mit
hoher Kohäsion

```
public double sin(double x) ...
public double cos(double x) ...
public double tan(double x) ...
public double arctan(double x) ...
...
```

SE2 – OOPM – Teil 2

170

Kohäsion von Klassen

- Eine Klasse (bzw. jedes ihrer Exemplare) sollte genau eine klar umrissene Einheit repräsentieren.

```
public class Program {
    public void initializeCommandStack()...
    void pushCommand(Command command)...
    Command popCommand()...
    void shutdownCommandStack()...
    void InitializeReportFormatting()...
    void formatReport(Report report)...
    void printReport(Report report)...
    void InitializeGlobalData()...
    void ShutdownGlobalData()...
}
```

Klasse mit
niedriger Kohäsion

Klasse mit
hoher Kohäsion

```
public class Film {
    public Film(String titel,
        int laenge,
        FSK fsk,
        boolean ueberlaenge)...

    public String gibTitel()...
    public int gibLaenge()...
    public FSK gibFSK()...
    public boolean hatUeberlaenge()...
}
```

SE2 - OOPM - Teil 2

171

Geeignete Bezeichner wählen

- Hohe Kohäsion zeigt sich unter anderem daran, dass geeignete Bezeichner für Entwurfseinheiten gewählt werden können.
 - Eine **konkrete Klasse** sollte durch ein **treffendes Substantiv** benennbar sein.
 - Eine **Methode** sollte durch ein **treffendes Verb** benennbar sein.
- Situationen, in denen eine geeignete Wahl schwer fällt, lassen häufig auf einen schlechten Entwurf schließen.
- Insbesondere **und-Bezeichner** (wie „einfuegenUndBerechnen“ oder „StarterUndVermittler“) sind ein deutlicher Hinweis auf **niedrige Kohäsion**.

**Extreme Ausprägung: Schreibe Quelltext
so, dass der Kunde ihn auch lesen kann!**

SE2 - OOPM - Teil 2

172

Englische oder deutsche Begriffe im Quelltext?

- Eine immer wieder gestellte Frage: Sollen wir unsere **Klassen** und **Methoden mit deutschen oder mit englischen Namen** versehen?

Englische Namen
sind viel cooler!

Java ist doch
auch englisch,
also sollte
gleich alles
englisch sein.



Wir sind hier an der Uni,
selbstverständlich
sollte da alles englisch
sein!

Außerdem ist das
internationaler!

SE2 - OOPM - Teil 2

173

Englisch oder Deutsch: kontextabhängig!

- Es gibt keine abschließende Antwort auf diese Frage; sie muss **für jedes Projekt neu** beantwortet werden.
- Eine wichtige Entscheidungshilfe: die **Trennung von fachlichen und technischen Klassen**.
 - Technische Klassen sollten mit englischen Namen benannt werden (Frame, Button, Listener, Writer, Transaction, ...).
 - Für die fachlichen Klassen sollte die Sprache abhängig vom fachlichen Kontext gewählt werden:
 - Bei einer Versicherung, die alle fachlichen Begriffe auf Deutsch verwendet, wären englische Begriffe sehr aufgesetzt und würden sehr schnell „unscharf“.
 - In einem Forschungsprojekt über Robotik mit internationaler Beteiligung sollten sicher auch die fachlichen Klassen englisch benannt werden.



SE2 - OOPM - Teil 2

174

Deutsch und Englisch gemischt?

- Durch einen deutschsprachigen fachlichen Kontext kann es zu Bezeichnern kommen, die sich aus einem deutschen und einem englischen Teilbegriff zusammensetzen.
- Das kann man als hässlich empfinden; es ist aber tatsächlich eher eine Chance als ein Nachteil, weil dann sogar innerhalb von Bezeichnern fachlicher und technischer Anteil sauber unterschieden sind.
- Beispiele aus der Mediathek:
 - **BearbeitenButton**: technisch eine Schaltfläche, die fachlich das Bearbeiten eines Mediums ermöglicht.
 - **getMedienart**: technisch ein Getter, der eine fachliche Eigenschaft auslesbar macht.
 - **RückgabeUI**: technisch zuständig für das User Interface, fachlich zuständig für die Rückgabe von Medien.



Konkretisierung: möglichst genaue Bezeichner wählen

- Beobachtung bei Systementwürfen:
 - **Verschmelzen ist leichter als Aufteilen.**
 - Es ist leichter, zwei verschiedene Begriffe zusammenzuführen (etwa, weil sich ihre Differenzierung als unnötig erweist), als einen Begriff in zwei neue aufzuspalten.
- Beispiel: Software für einen **CD-Verleih**.
 - Wir modellieren zu Beginn als zentrale Abstraktion die **Klasse CD**.
 - Später stellen wir fest: Wir müssen die Modellierung eines Albums unterscheiden von der Modellierung eines konkreten Exemplars dieses Albums, das ausgeliehen werden kann (denn es kann mehrere verleihbare Exemplare eines Albums geben).
 - Frage: Was wären geeignete Namen für die beiden Abstraktionen?

Code-Duplizierung vermeiden

- Wenn ein Stück Quelltext in identischer Form an mehreren Stellen eines Systems definiert ist, sprechen wir von **Code-Duplizierung**.
- Code-Duplizierung ist problematisch, weil
 - üblicherweise an einer Stelle nicht erkennbar ist, an welchen anderen Stellen derselbe Quelltext erscheint, und
 - Änderungen an einem der Duplikate eventuell auch an allen anderen Duplikaten ausgeführt werden müssen; dies kann bei der Wartung übersehen werden.
- Code-Duplizierung ist ein **Zeichen niedriger Kohäsion**:
 - Wenn zwei Einheiten dieselbe (Teil-)Aufgabe erledigen, ist bei mindestens einer von beiden die Zuständigkeit falsch zugeordnet.



SE2 - OOPM - Teil 2

177

Große Einheiten vermeiden

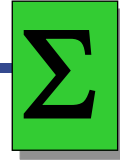
- Ein bekanntes Anti-Pattern - die „**Gott-Klasse**“:
 - Sie ist für alles zuständig und hält 90% des Quelltextes.
- Methoden, die **mehrere Bildschirmseiten** lang sind, sind schlecht wartbar.
- Klassen mit mehreren hundert Methoden oder Exemplarvariablen sind meist zu groß!
- Wenn einzelne Einheiten zu groß werden, ist dies ein Hinweis auf **niedrige Kohäsion**:
 - Wenn eine Einheit **sehr viele** Aufgaben erfüllt, erfüllt sie vermutlich **zu viele** Aufgaben.



SE2 - OOPM - Teil 2

178

Zusammenfassung Klassenentwurf



- Die **innere Qualität** eines Softwaresystems bestimmt seine **Wartbarkeit** und die **Portierbarkeit**.
- Die innere Qualität eines **objektorientierten Systems** wird maßgeblich durch den **Klassenentwurf** bestimmt.
- Zahlreiche **Richtlinien** helfen uns, Entwurfsentscheidungen zu treffen.
- Ein Klassenentwurf sollte insbesondere
 - eine **hohe Kohäsion** seiner Bestandteile
 - und eine **niedrige Kopplung** zwischen diesen aufweisen.