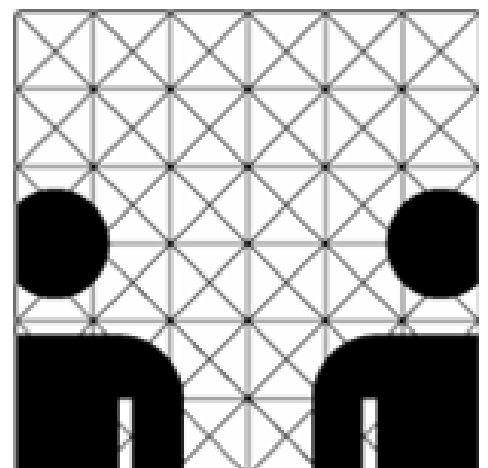


Prüfungsunterlagen  
zur Lehrveranstaltung



Teil 1

Universität Hamburg  
Fachbereich Informatik  
SoSe 2012





# Softwareentwicklung II

## SE2

### Objektorientierte Programmierung und Modellierung

Guido Gryczan  
Axel Schmolitzky  
Heinz Züllighoven  
et al.

#### Teil 1

#### Verzeichnis der Folien

1. Symbole und Zeichen
2. Symbole und Zeichen (II)
3. Grundlegende Lehrbücher
4. Mehr zu Java
5. Weitere Grundlagenwerke
6. Englischsprachig und weiterführend
  
- 7. Einführung: Abstraktion und Modelle**
8. Objektorientierte Aktivitäten in SE1
9. Objektorientierte Aktivitäten in SE2
10. Objektorientierte Aktivitäten allgemein
11. Modelle als zentraler Gegenstand der Softwareentwicklung
12. Beispiel für wissenschaftliche Modelle: Atommodelle
13. Ein Modell als abstrahierende Repräsentation
14. Modell als Abbildung, Verkürzung, Mittel zum Zweck
15. Modelle der Softwareentwicklung
16. Beispiel für die Modelle der Softwareentwicklung
17. Modellierung bedeutet Abstraktion
18. Beispiele für Abstraktion und Repräsentation: Zahlen
19. Von der Abstraktion zur Modell-Manipulation
20. Beispiel für Manipulation von Repräsentationen bei Modellen
21. Abstraktion und Modellbildung am Beispiel Kinosaal
22. Modellieren erfordert Abstraktion vom Original und Interpretation des Modells
23. Beziehungen zwischen Original und Modell
24. Software als Modell des Anwendungsbereichs
25. Anforderungen an die Modellierung von Software
26. Verifikation von Modellen
27. Validation und Bewertung von Modellen
28. Abstraktion und Modellierung bei der Programmierung (1)
29. Abstraktion und Modellierung bei der Programmierung (2)
  
- 30. Testen von objektorientierten Programmen**
31. Literaturhinweise
32. Motivation für das Testen
33. Aus SE1: Wann ist Software überhaupt „korrekt“?
34. Testen als Teil des „Programmverstehens“
35. Aus SE1: Statische und dynamische Tests
36. Begriff: Statischer Test
37. Wiederholung und Präzisierung: Was ist Testen?
38. Grundsätze zum Testen (1)
39. Grundsätze zum Testen (2)
40. Grundsätze zum Testen (3)
41. Austesten?
42. Fehlerarten in Software (nach B. Meyer)
43. Von Ursache zu Entdeckung: eine Motivation für das Vertragsmodell
44. Fehler: Ursache, Wirkung und Entdeckung
45. Nochmals: Weshalb trotzdem Testen?
46. Begriffsvielfalt beim Testen

- 47. Aus SE1: Positives und negatives Testen
- 48. Aus SE1: Modultest und Integrationstest
- 49. Begriff: Komponententest
- 50. Begriff: Integrationstest
- 51. Begriff: Systemtest
- 52. Begriff: Abnahmetest
- 53. Begriff: Test nach Änderungen / Regressionstest
- 54. Testen objektorientierter Software: nicht einfach!
- 55. Probleme des oo Testens: Kapselung
- 56. Probleme des oo Testens: Komplexe Abhängigkeiten
- 57. Testgetriebene Entwicklung
- 58. Ausblick: Weitere Testarten

## **59. Das Vertragsmodell**

- 60. Vertrag
- 61. Das Vertragsmodell der objektorientierten SW-Entwicklung
- 62. Benutzt-Beziehung und Vertragsmodell
- 63. Die Bestandteile des Vertragsmodells
- 64. Der Mechanismus des Vertragsmodells
- 65. Ein Beispiel aus einer Bibliothek
- 66. Auf dem Weg zum Vertragsmodell (1)
- 67. Rechte und Pflichten des Klienten
- 68. Rechte und Pflichten des Lieferanten
- 69. Auf dem Weg zum Vertragsmodell (2)
- 70. Der Klient muss seine Verpflichtungen prüfen können
- 71. Auf dem Weg zum Vertragsmodell (3)
- 72. Operationen an der Schnittstelle
- 73. Zustandsabhängigkeit von Operationen
- 74. Klassen-Invarianten
- 75. Korrektheit einer Klasse
- 76. Kein eingebautes Vertragsmodell in Java
- 77. Manuelle Umsetzung des Vertragsmodells in Java
- 78. Überprüfung von Verträgen in Java
- 79. Vertragsprüfung in Java mit assert
- 80. Diskussion: assert für Vertragsmodell in Java
- 81. Vertragsmodell und Ausnahmen
- 82. Vertragsmodell und Testen: Vorsicht bei Negativtests!
- 83. Zwischenergebnis Vertragsmodell

## **84. Abstraktion: Polymorphie und Vererbung**

- 85. Motivation: Vererbung
- 86. „Inheritance Considered Harmful“
- 87. Bisher: Abstraktionsmittel Interface
- 88. Aus SE1: Die Doppelrolle einer Klasse
- 89. Die objektorientierte Klasse: Typ und Implementation
- 90. Statik von OO Systemen: Geflechte von Typen
- 91. Wiederholung: Statischer und dynamischer Typ
- 92. Wiederholung: Statischer und dynamischer Typ (II)
- 93. Dynamisches Binden
- 94. Vererbung: Auf Typ- und Implementationsebene
- 95. Polymorphie nach Cardelli und Wegner (1985)
- 96. Kurzer Einschub: Overloading und Coercion
- 97. Subtyp-Polymorphie
- 98. Übersicht: Zentrale „Vererbungs“-konzepte
- 99. Klassifizierung
- 100. Typabstraktion (bekannt aus SE1)
- 101. Typhierarchien
- 102. Beispiel: Typ-Hierarchie im Java Collections Framework
- 103. Beispiel: Javas Exception-Hierarchie
- 104. Code-Wiederverwendung
- 105. Einige weitere Begriffe
- 106. Einfach- und Mehrfachvererbung

- 107. Mehrfachvererbung
- 108. Vererbung verschiedener, aber gleichnamiger Merkmale
- 109. Diamant-Vererbung
- 110. Vorläufige Zusammenfassung

### **111. Übersicht Subtyping**

- 112. Allgemeines
- 113. Entwurf von Typhierarchien: Erkennen gleichartiger Umgangsformen ...
- 114. ... als Grundlage einer Typhierarchie
- 115. Eine schlechte Verwendung von Subtyping: "Box" als Supertyp von "Ordner"
- 116. Subtyping soll spezielle Klassifikation ausdrücken ("verstehen als", "is-a")
- 117. Die Softwarearchitektur spiegelt die fachlichen Begriffe
- 118. Noch einmal: Erkennen gleichartiger Umgangsformen ...
- 119. ... als Grundlage einer Typhierarchie
- 120. Subtyping in Java
- 121. Interfaces und Subtyping
- 122. Redefinieren vs. Redeklamieren
- 123. Grenzen beim Redeklamieren
- 124. Formale Definition von Subtyping
- 125. Ko- und Kontravarianz
- 126. Zulässig: Kovarianz für Ergebnistypen
- 127. Zulässig: Kontravarianz für Parametertypen
- 128. Nicht typsicher: Kovarianz für Parametertypen
- 129. Zusicherungen und Subtyping: Das Prinzip
- 130. Zusicherungen und Subtyping: Ein Eiffel-Beispiel
- 131. Zusicherungen und Subtyping in Programmiersprachen
- 132. Invarianten und Subtyping
- 133. Ko- und Kontravarianz und Zusicherungen
- 134. Zusammenfassung Polymorphie und Vererbung

### **135. Übersicht Implementationsvererbung**

- 136. Motivation: Wiederverwendung
- 137. Generalisierung und Spezialisierung
- 138. Erben zum Anpassen
- 139. Erben zum Vermeiden von Redundanz
- 140. Erben zum Vervollständigen
- 141. Übersicht: Umgang mit geerbten Eigenschaften
- 142. Erben und hinzufügen
- 143. Vererbung zwischen Klassen in Java
- 144. Verwendung von Geerbtem
- 145. Private Eigenschaften und Vererbung
- 146. Klasse Object als Oberklasse
- 147. Erben und redefinieren
- 148. Redefinieren: Überschreiben
- 149. Redefinieren: Erweitern (mit super-Aufruf)
- 150. Aufrufe redefinierter Operationen
- 151. Abstrakte Methoden: Deklarieren ohne Definieren
- 152. Abstrakte Klassen
- 153. Abstrakte Methoden und ihre Definition
- 154. Eigenschaften abstrakter Klassen
- 155. Abstrakte Methoden sind aufrufbar!
- 156. Wie kann das überhaupt funktionieren?
- 157. Auswertung von Selbst-Aufrufen in Java
- 158. Schablonenmethode und Einschubmethode
- 159. Verwendung von abstrakten Oberklassen
- 160. Abstrakte Oberklassen zur Spezifikation
- 161. Vererbung und Konstruktoren in Java
- 162. Beispiel: Eigene Exception-Klasse definieren
- 163. Weiteres Beispiel für Konstruktoren und Vererbung
- 164. Konstruktoren und Vererbung, mehrstufiges Beispiel
- 165. Klienten- und Erbenschnittstelle
- 166. Modellierung der Erbenschnittstelle

- 167. Modellierung der Erbenschnittstelle
- 168. Aufgepasst: Der Operator instanceof in Java
- 169. Zusammenfassung Implementationsvererbung
- 170. Diskussion Vererbung et al.
- 171. Diskussion Vererbung et al. (2)

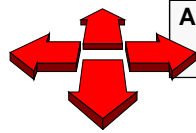
## **172. Fehlerbehandlung mit Exceptions**

- 173. Fehlersituationen in Programmen
- 174. Programmierfehler versus Umgebungsfehler
- 175. Fehlersituationen: intern und extern
- 176. Umgang mit Fehlersituationen
- 177. Traditionelle Ansätze zur Fehlerbehandlung
- 178. Kritik an den traditionellen Ansätzen
- 179. Ausnahmen als eigenständiges Sprachkonzept
- 180. Exceptions in Java sind Objekte
- 181. Geprüfte und ungeprüfte Exceptions
- 182. Javas Schlüsselwörter für die Ausnahmebehandlung
- 183. Ausnahmebehandlung in Java (1)
- 184. Ausnahmebehandlung in Java (2)
- 185. Das Auslösen einer Exception
- 186. Das Umgehen mit geprüften Exceptions
- 187. Das Behandeln von Exceptions
- 188. Ausnahmen differenziert behandeln
- 189. Vererbung: Ausschnitt aus Javas Exception-Hierarchie
- 190. Beispiel: Eigene Exception-Klasse definieren
- 191. Exceptions als Abstraktionshilfe
- 192. Zusammenfassung

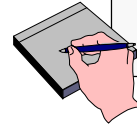
## **193. Namensräume und Modularisierung**

- 194. Das Modulkonzept
- 195. Die Grundidee: ein Software-System besteht aus Modulen
- 196. Weiteres Konzept: Module können Module enthalten
- 197. Klassisch: Modul – Exportschnittstelle
- 198. Klassisch: Modul – Importschnittstelle
- 199. Imperative Sprachen: Beispiel für Export in Modula-2
- 200. Imperative Sprachen: Beispiel für Import in Modula-2
- 201. Wesentliche übertragbare Modulkonzepte
- 202. Klassen sind auch Module
- 203. Geschachtelte Klassen in Java
- 204. Geschachtelte und innere Klassen in Java
- 205. Statische geschachtelte Klassen in Java
- 206. Auch sehr ähnlich zu Modulen: Pakete in Java
- 207. Pakete als Namensräume in Java
- 208. Import von Paketelementen
- 209. Bereits bekannt: Import von Paketelementen
- 210. Namensvergabe für Pakete in Java
- 211. Beschränkter Zugriff: Modifikatoren in Java
- 212. Zugriffsrechte in Java: nicht einschränkbar zwischen Paketen
- 213. Zugriffsrechte in Java: Elemente von Paketen
- 214. Zugriffsrechte in Java: Elemente von Klassen (vollständig)
- 215. Übersicht: Zugriffsmodifikatoren für Elemente von Klassen
- 216. Wer's lieber in der UML mag...
- 217. Jenseits von Packages: SuperPackages?
- 218. Ausblick: Incoming und Outgoing Interfaces
- 219. Darstellung von Incoming und Outgoing Interfaces
- 220. Zusammenfassung: Modul und Klasse

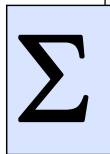
## Symbole und Zeichen


**Ausblick:**

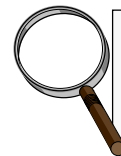
sagt, was kommt.


**Zentraler Begriff:**

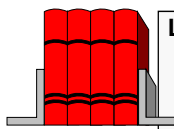
wird hier eingeführt oder erläutert.


**Zwischensumme:**

hebt für einzelne Themen hervor, was uns wichtig ist.


**Vertiefung:**

hier werden Hintergründe etwas genauer beleuchtet.


**Literaturliste:**

Grundlage für den nachfolgenden Teil mit Bewertung.

© <Autor>

**Literaturreferenz:**

auf Bücher aus der Literaturliste.

## Symbole und Zeichen (II)


**Merker:**

kommentiert, stellt Bezüge her und hebt etwas hervor.


**Imperativer Begriff:**

hier wird ein Begriff aus der imperativen Programmierung verwendet, der sich von der OO-Terminologie unterscheidet.


**Java-Beispiel:**

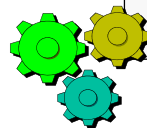
erläutert ein Konzept oder Konstrukt am Beispiel der Sprache Java.


**Negativbeispiel:**

warnet vor häufigen, aber bedenklichen Konstruktionen.


**UML-Notation:**

Diagramm in der Notation der Unified Modeling Language.


**Softwaretechnik:**

diskutiert Erfahrungen und Prinzipien.

## Grundlegende Lehrbücher

David J. Barnes, Michael Kölling: **Java lernen mit BlueJ – Eine Einführung in die objektorientierte Programmierung**, 3. Aufl., Pearson Studium, 2006. (deutsche Übersetzung von: **Objects First with Java - A Practical Introduction using BlueJ**, 3. Aufl., Pearson Education, 2006.)  
[Der aktuelle „Objects First“ Ansatz mit BlueJ. Gut geeignet zum Selbststudium.]

Cornelia Heinisch, Frank Müller, Joachim Goll: **Java als erste Programmiersprache**, 5. überarb. u. erw. Aufl., Teubner, Stuttgart, 2007.  
[Eine gute konventionelle Einführung in Java.]

Reinhard Schiedermeier: **Programmieren mit Java**, 2. Aufl., Pearson Studium, 2010.  
[Ebenfalls eine solide konventionelle Einführung in Java]

## Mehr zu Java

Reinhard Schiedermeier, Klaus Köhler: **Das Java-Praktikum**, dpunkt Verlag, 2008.  
[Eine sehr nützliche Sammlung von Aufgaben zu Java.]

Ken Arnold, James Gosling, David Holmes: **The Java Programming Language**, Fourth Edition, Addison-Wesley, 2005.  
[Der Java-Klassiker. Knapp und ohne didaktischen Anspruch. Eher zum Einlesen für erfahrene Programmierer.]

David Flanagan: **Java in a Nutshell**, 5. Aufl., O'Reilly Media, 2005.  
[Der Java-Nachschlage-Klassiker. Kurz und knapp (auf 1224 Seiten) durch die wesentlichen Java-Bestandteile und -Packages.]

Joshua Bloch: **Effective Java Programming Language Guide**, 2. Aufl., Addison-Wesley Longman, 2008.  
[Die Fallstricke von Java ausführlich und sehr kompetent. Eher für Fortgeschrittene.]

James Gosling, Bill Joy, Guy Steele: **The Java Language Specification**, Third Edition, Addison-Wesley, Juli 2005.  
[Die offizielle Sprachdefinition. Für die, die es genau wissen wollen.]



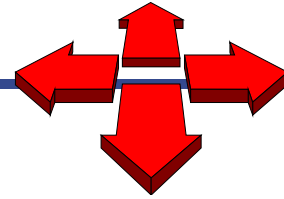
## Weitere Grundlagenwerke

- Peter Rechenberg, Gustav Pomberger, **Informatik-Handbuch**, Hanser-Verlag, 4., aktualis. u. erw. Aufl. 2006. 1251 S.  
[Handbuch der wesentlichen Gebiete der Informatik.]
- Duden Informatik**, Dudenverlag, Ausgabe 2003.  
[Grundbegriffe kurz und grundlegend definiert.]
- Grady Booch, James Rumbaugh, Ivar Jacobson, **The Unified Modeling Language Reference Manual**, Addison-Wesley, 2004.  
[Die Referenz von den „Erfindern“ der UML]
- OMG Unified Modeling Language Specification**, Version 2.1.1, 2007. <http://www.omg.org/technology/documents/formal/uml.htm>  
[Die aktuelle Standardversion der UML.]

## Englischsprachig und weiterführend

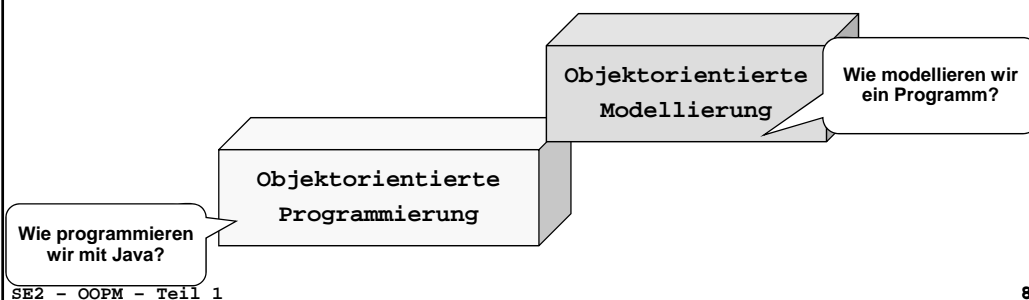
- Robert W. Sebesta, **Concepts of Programming Languages**, Addison-Wesley Educational Publishers, 8. Auflage, 2007.  
[Gute und verständliche Einführung in die Definition von Programmiersprachen]
- Bertrand Meyer: **Object-oriented Software Construction**. Second Edition. Prentice Hall, 1997.  
[Der Klassiker unter den oo-Programmierbüchern (am Beispiel von Eiffel). Viele allgemeingültige und wertvolle Hinweise. Engagiert und bissig.]
- Heinz Züllighoven et al.: **Object-Oriented Construction Handbook**. dpunkt-Verlag, 2004.  
[Unser Diskussionsbeitrag. Für Fortgeschrittene (und die, die es werden wollen :-)]

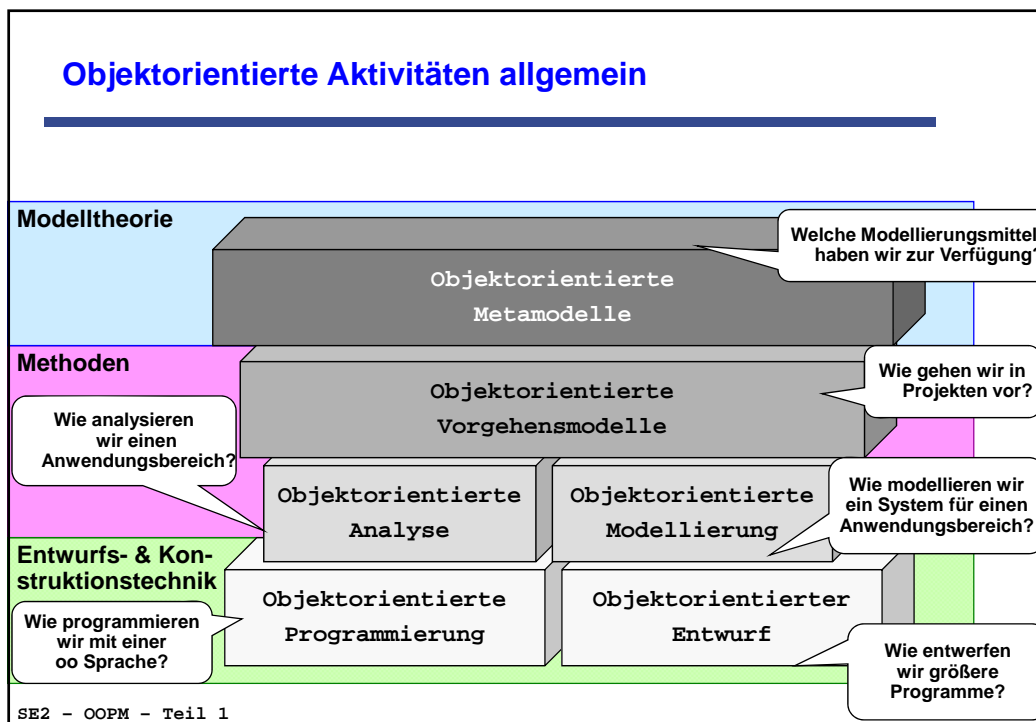
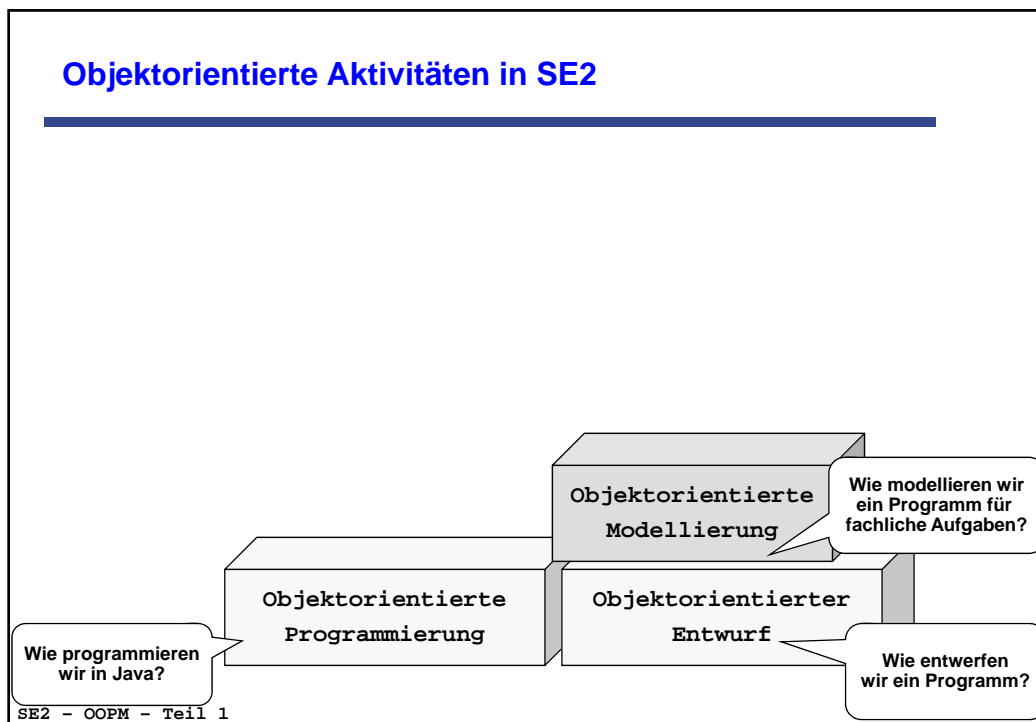
## Einführung: Abstraktion und Modelle



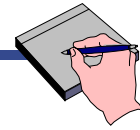
- Aktivitäten bei der Softwareentwicklung
- Der allgemeine Modellbegriff
- Modelle als Abstraktionen
- Abstraktion als zentrales Hilfsmittel der Informatik
- Beziehungen zwischen Modell und Original

## Objektorientierte Aktivitäten in SE1





## Modelle als zentraler Gegenstand der Softwareentwicklung



**Modelle** sind in der Wissenschaft und im Ingenieurwesen von zentraler Bedeutung:

- „Von einem **Modell** spricht man oftmals als Gegenstand wissenschaftlicher Methodik und meint damit, dass eine zu untersuchende Realität durch bestimmte Erklärungsgrößen im Rahmen einer wissenschaftlich handhabbaren Theorie abgebildet wird.“
- Da im Allgemeinen nicht alle Aspekte der untersuchten Realität in Modellen abbildbar sind, wird Modellbildung oftmals als Reduktion, Konstruktion oder Abstraktion bezeichnet.“

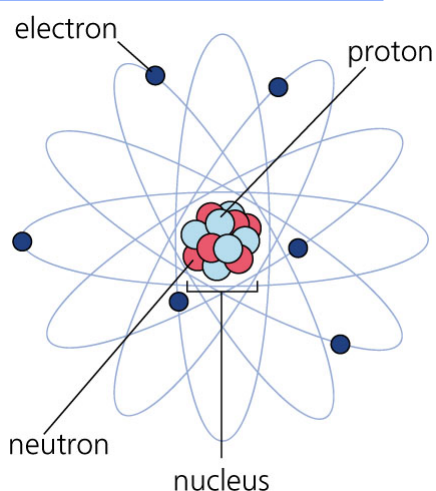
nach Wikipedia

- “Scientists spend a great deal of time building, testing, comparing and revising models, and much journal space is dedicated to introducing, applying and interpreting these valuable tools. In short, models are one of the principal instruments of modern science.”

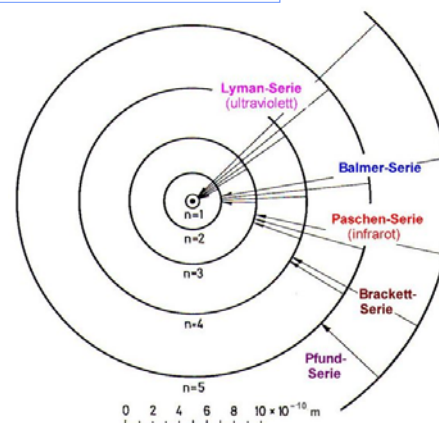
nach Stanford Encyclopedia of Philosophy

## Beispiel für wissenschaftliche Modelle: Atommodelle

### Atommodell nach Rutherford

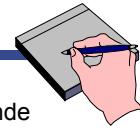


### Atommodell nach Bohr



$$F_E = \frac{1}{4\pi\epsilon_0} \frac{e^2}{r^2} = ma_n = \frac{mv^2}{r}$$

## Ein Modell als abstrahierende Repräsentation



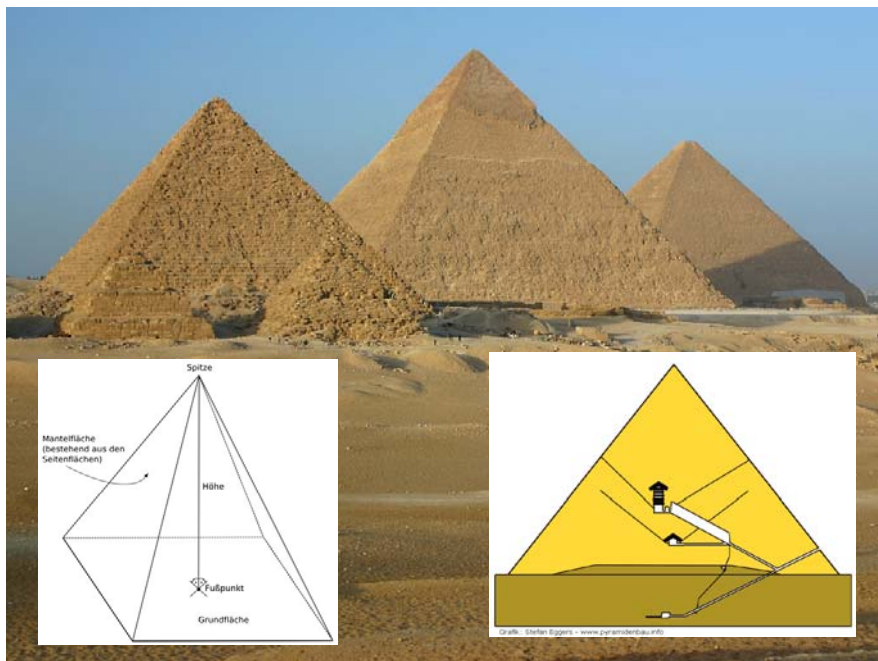
Es gibt verschiedene Modellbegriffe. In der Informatik wird häufig der folgende Ansatz von **Stachowiak** verwendet:

**Ein Modell** ist gekennzeichnet durch:

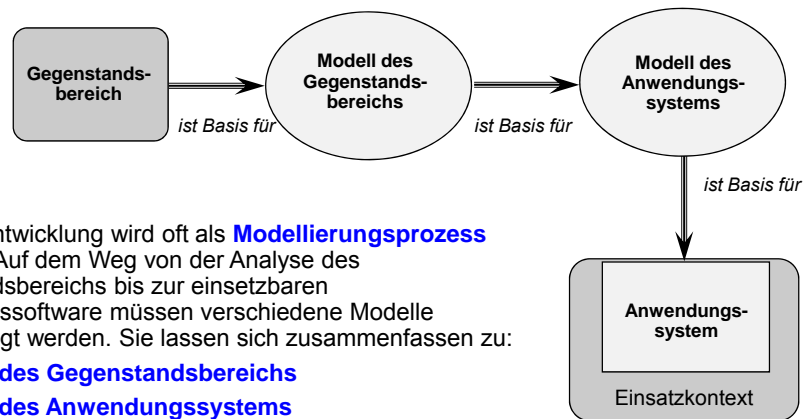
- **Abbildung**  
Ein Modell ist immer ein Abbild von etwas, eine Repräsentation natürlicher oder künstlicher Originale, die selbst wieder Modelle sein können.
- **Verkürzung**  
Ein Modell erfasst nicht alle Attribute des Originals, sondern nur diejenigen, die dem Modellschaffer oder Modellnutzer relevant erscheinen.
- **Pragmatismus**  
Ein Modell ist einem Original aus pragmatischen Gründen zugeordnet. Die Zuordnung wird durch die Fragen *Für wen?*, *Warum?* und *Wozu?* begründet. Ein Modell wird von jemandem innerhalb einer bestimmten Zeitspanne und zu einem bestimmten Zweck für ein Original eingesetzt. Das Modell wird somit interpretiert.

nach Stachowiak, H.: *Allgemeine Modelltheorie*, Springer-Verlag, Wien, New York, 1973.

## Modell als Abbildung, Verkürzung, Mittel zum Zweck



## Modelle der Softwareentwicklung



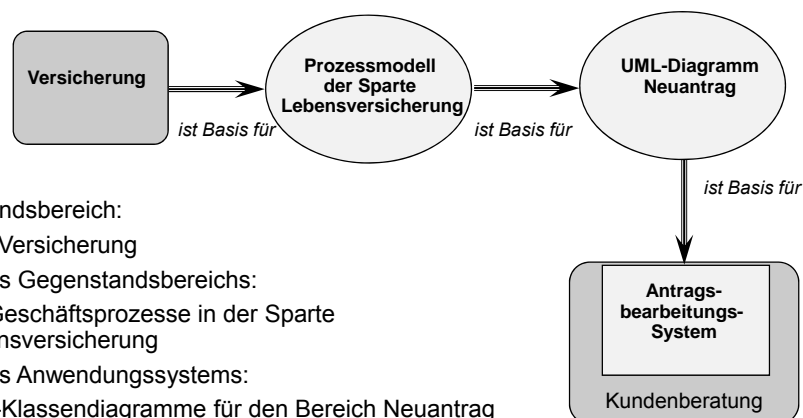
- Software-Entwicklung wird oft als **Modellierungsprozess** betrachtet. Auf dem Weg von der Analyse des Gegenstandsbereichs bis zur einsetzbaren Anwendungssoftware müssen verschiedene Modelle berücksichtigt werden. Sie lassen sich zusammenfassen zu:

- **Modell des Gegenstandsbereichs**
- **Modell des Anwendungssystems**

- Objektorientierte Methoden unterscheiden sich auch darin, ob und wie sie diese Modelle erstellen.

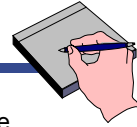
Der Quellcode ist Teil des Modells des Anwendungssystems; das ablaufende Programm ist Teil des Anwendungssystems.

## Beispiel für die Modelle der Softwareentwicklung



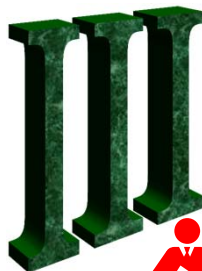
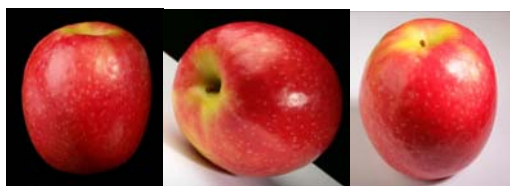
- Gegenstandsbereich:
  - Eine Versicherung
- Modell des Gegenstandsbereichs:
  - Die Geschäftsprozesse in der Sparte Lebensversicherung
- Modell des Anwendungssystems:
  - UML-Klassendiagramme für den Bereich Neuantrag einer Lebensversicherung
- Anwendungssystem:
  - Am Arbeitsplatz einer Kundenberaterin eingesetztes System zur Antragsbearbeitung

## Modellierung bedeutet Abstraktion



- **Abstraktion** ist das stärkste menschliche Hilfsmittel, um komplexe Dinge und Ereignisse zu verstehen.
- Der Schlüssel zur Abstraktion liegt darin, ausgewählte **gemeinsame Eigenschaften** von Gegenständen, Situationen und Prozessen in der Realität zu **erkennen** und die **Unterschiede** zu **ignorieren**.
- Wenn wir die relevanten gemeinsamen Eigenschaften erkannt haben, können wir zukünftige Ereignisse vorhersagen und kontrollieren.
- Oft verwenden wir Zeichen und Bilder, um diese relevanten Eigenschaften zu repräsentieren.
- In der Wissenschaft werden die Repräsentationen manipuliert, um zu Aussagen über zukünftige oder mögliche Ereignisse und Ergebnisse in der Realität zu kommen.

## Beispiele für Abstraktion und Repräsentation: Zahlen

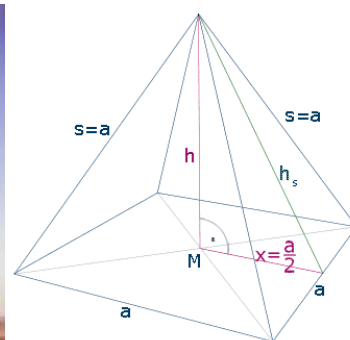
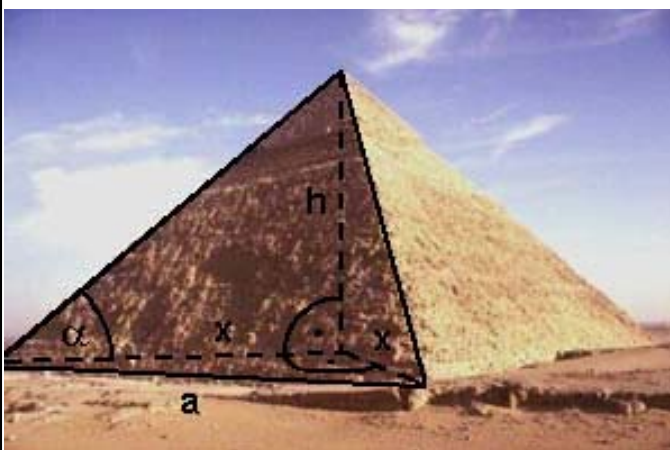


Wir abstrahieren von allen Eigenschaften außer "Anzahl" und repräsentieren den Wert "drei" mit unterschiedlichen Symbolen/Zeichen

## Von der Abstraktion zur Modell-Manipulation

- **Abstraktion:**  
Identifiziere die relevanten gemeinsamen Eigenschaften von Gegenständen, Situationen und Prozessen in der Realität und ignoriere die Unterschiede.
- **Repräsentation:**  
Wähle geeignete Symbole für diese Abstraktion, die Grundlage der notwendigen Kommunikation sein können.
- **Manipulation:**  
Wähle die Regeln zur Transformation der Repräsentation, so dass ähnliche Veränderungen der Repräsentation auf mögliche Veränderungen in der Realität übertragen werden können.
- **Axiomatisierung/Formalisierung:**  
Identifiziere formale Gesetzmäßigkeiten, mit denen sich nachweisen lässt, dass die Manipulation der Repräsentation und die Regeln für die Veränderungen in der Realität korrekt sind.

## Beispiel für Manipulation von Repräsentationen bei Modellen



Um die Höhe der realen Pyramide zu berechnen, repräsentieren wir sie als Dreiecke, für die wir entsprechende Rechenvorschriften kennen.



## Abstraktion und Modellbildung am Beispiel Kinosaal



$$\text{Verkaufspreis} = \text{Anzahl} * (\text{NettoPreis} + \text{MWSt})$$

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 |
| 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 0 |
| 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 0 |
| 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 0 |
| 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 |
| 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 |
| 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 |
| 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 |
| 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 |
| 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

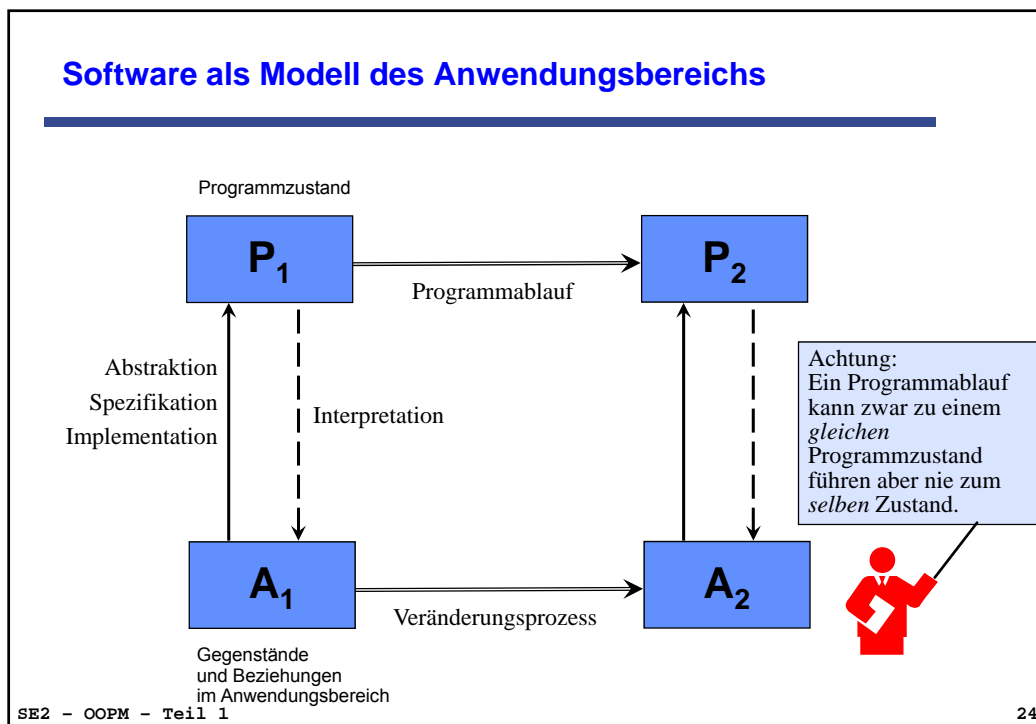
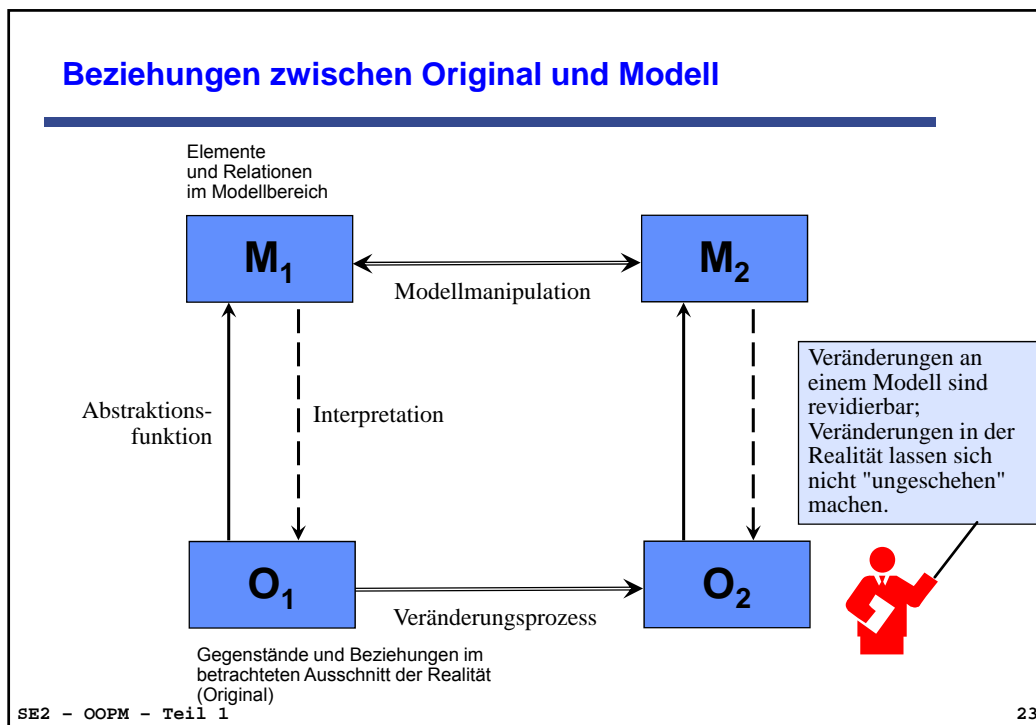
Wir abstrahieren vom Kinosaal die Anzahl der Reihen und Sitze.  
Wir repräsentieren diese Abstraktion als binäre Matrix. Wir setzen belegte Plätze in der Matrix entsprechend auf 1. Wir berechnen Preise anhand eines algorithmischen Terms.

## Modellieren erfordert Abstraktion vom Original und Interpretation des Modells

Ein **Modell** zeichnet sich durch **Abstraktion** vom Original für einen bestimmten Zweck aus:

- Modelle können Zusammenhänge in der realen Situation erklären.
- Sie können zukünftige, mögliche oder sinnvolle Veränderungen des Originals zeigen (vorhersagen).
- Sie ermöglichen durch Manipulation einer Repräsentation gefahrloses oder seiteneffektfreies „Probearbeiten“.

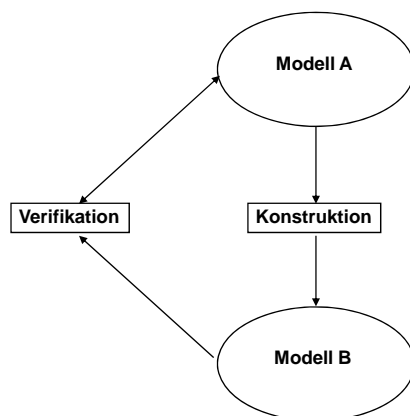
Die meisten Zwecke kann ein Modell nur dann erfüllen, wenn wir seinen (veränderten) Zustand wieder auf das Original beziehen, d.h. wenn wir es **interpretieren**.



## Anforderungen an die Modellierung von Software

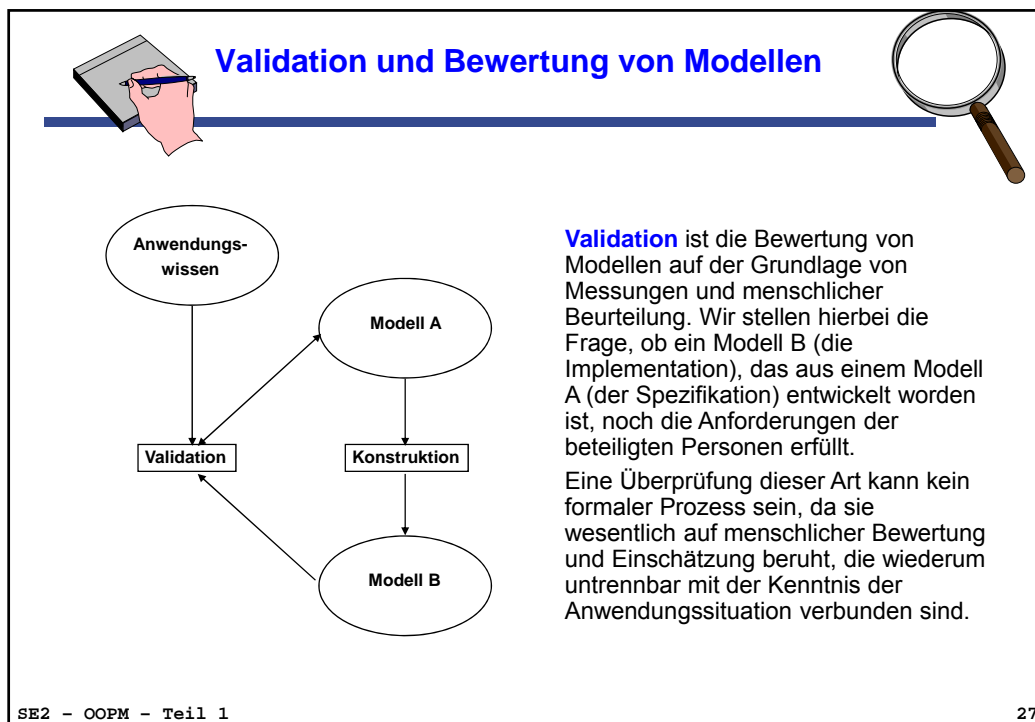
- **Software muss valide sein**, d.h. sie muss für die beteiligten Personen ihren Zweck im Anwendungsbereich erfüllen.  
Dazu müssen wir:
  - Die angemessenen Abstraktionen und Repräsentationen wählen.
    - verständlich für die Beteiligten (Benutzer, Entwickler)
      - weiterentwickelbar
      - benutzbar
      - interpretierbar
  - Eine angemessene Formalisierung wählen:
    - korrekt, d.h. es erfüllt die vorgegebene Spezifikation
    - verständlich
    - konsistent
  - Eine angemessene Implementation wählen
    - effektiv, d.h. das vorgegebene Ziel wird erreicht
    - effizient, d.h. mit möglichst geringem Aufwand (Laufzeit und Speicherbedarf)

## Verifikation von Modellen



**Verifikation** überprüft die Korrektheit und Konsistenz eines Modells der Software-Entwicklung bezogen auf sich selbst und die vorhergehenden Modelle.

Idealerweise bedeutet das, die Korrektheit der Transformation zwischen zwei Modellen nachzuprüfen. Wenn diese Modelle mit den Mitteln eines formalen Kalküls beschrieben sind, dann ist Verifikation ein formaler Prozess, der lokal, d.h. ohne Kenntnis anderer Modelle oder des Kontextes, durchgeführt werden kann.



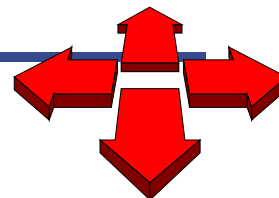
## Abstraktion und Modellierung bei der Programmierung (1)

- Bisher betrachtet:
  - **Prozessabstraktion**
    - *Modell*: Schnittstelle mit Signatur
    - *Abstraktion*: Von Operationen/Methoden ist nur die Signatur bekannt; von der konkreten Implementation der Prozesse wird abstrahiert.
  - **Datenabstraktion**
    - *Modell*: Zustandsfelder und Zugriffsoperationen
    - *Abstraktion*: Werte von Feldern sind nur über Zugriffsoperationen zugänglich; vom konkreten Typ wird abstrahiert.
  - **objektorientierte Abstraktion**
    - *Modell*: Objekte mit Methoden
    - *Abstraktion*: Objekte sind nur über sichtbare Dienstleistungen zugänglich. Von Zustandsfeldern und Implementationen wird abstrahiert.

## Abstraktion und Modellierung bei der Programmierung (2)

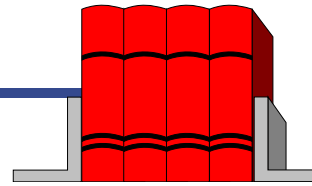
- Die nächsten Schritte:
  - **Klassenhierarchien:**
    - Wir organisieren Klassen als Konzepte/Begriffe auf unterschiedlichen Ebenen von Abstraktionen.
  - **Unterschiedliche Granularität:**
    - Wir organisieren unsere Entwurfs- und Konstruktionselemente in geschachtelten, unterschiedlich großen Einheiten (Klassen, Packages, Teilsysteme).

## Testen von objektorientierten Programmen



- Grundbegriffe
- Probleme und Ansätze des objektorientierten Testens
- Integrationstests
- Testgetriebene Vorgehensweisen
- Weitere Testarten

## Literaturhinweise



- Andreas [Spillner](#), Tilo [Linz](#), *Basiswissen Softwaretest*. 296 Seiten, Dpunkt Verlag 2005.  
[DAS deutschsprachige Basisbuch zum Thema Testen; dient auch als Grundlage für den Certified Tester des ISTQB]
- Robert [Binder](#), *Testing Object Oriented Systems. Models, Patterns and Tools*. 1200 Seiten - Addison-Wesley Professional. November 1999.  
[Standard-Buch über objektorientiertes Testen]
- Johannes [Link](#), Frank [Adler](#), Achim [Bangert](#), *Unit Tests mit Java. Der Test-First-Ansatz*. 348 Seiten - Dpunkt Verlag 2005.  
[Gutes Buch über Test First mit JUnit]

## Motivation für das Testen

- Wir haben bereits festgestellt:
  - „Ein fehlerfreies Softwaresystem gibt es derzeit nicht und wird es in naher Zukunft wahrscheinlich nicht geben, sobald das System einen gewissen Grad an Komplexität und Umfang an Programmzeilen umfasst.“ [Spillner, Linz]
- Dazu kommt:
  - Wir wollen (äußere und innere) Qualität eines Programms bestimmen.
  - Wir wollen die Funktionstüchtigkeit eines Programms erhöhen.
  - Wir wollen ein Programm besser verstehen, um es weiter zu entwickeln oder seine technische Qualität zu erhöhen.

### Aus SE1: Wann ist Software überhaupt „korrekt“?

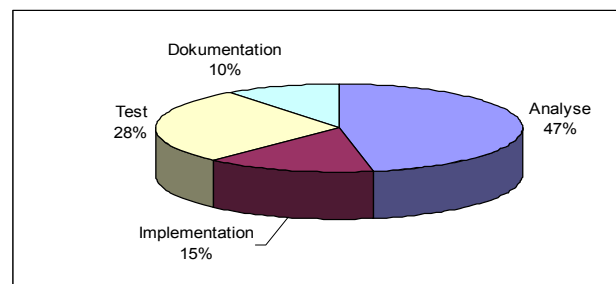
- Die **Korrektheit** von Software kann immer nur in Relation zu ihrer **Spezifikation** gesehen werden - eine Software-Einheit ist korrekt, wenn sie ihre Spezifikation erfüllt.
- Der formale Nachweis, dass eine Software-Einheit ihre Spezifikation erfüllt, ist sehr aufwendig und schwierig.
- Voraussetzung für einen formalen Nachweis der Korrektheit ist, dass die Spezifikation selbst **formal definiert** ist. Dies ist nur sehr selten der Fall, meist sind Spezifikationen problembedingt nur informell formuliert.
- Auch wenn eine formale Spezifikation vorliegt: **Wie kann nachgewiesen werden, dass die Spezifikation selbst „korrekt“ ist?**

Ergo: Für umfangreiche interaktive Programme sind formale Korrektheitsbeweise heute nicht machbar.

**Korrektheit** ist in der Praxis somit häufig ein unscharfer Begriff. Sehr viel nützlicher für die Praxis der Softwareentwicklung ist somit der Begriff der **Validität**.

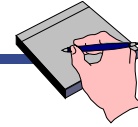
### Testen als Teil des „Programmverstehens“

- Laut Balzert kostet das Verstehen von Software bei der **Weiterentwicklung/Wartung** den größten Aufwand.
- Testen fördert das Verständnis von Software.



[Balzert98]

## Aus SE1: Statische und dynamische Tests



- Linz und Spillner unterscheiden in **Basiswissen Softwaretest** grundlegend zwischen statischen und dynamischen Tests.
- Ein **statischer Test** (häufig auch **statische Analyse** genannt) bezieht sich auf die Übersetzungszeit und analysiert primär den Quelltext. Statische Tests können **von Menschen** durchgeführt werden (Reviews u.ä.) oder mit Hilfe von **Werkzeugen**, wenn die zu testenden Dokumente einer formalen Struktur unterliegen (was bei Quelltext zutrifft).
- **Dynamische Tests** sind alle Tests, bei denen die zu testende Software ausgeführt wird.



## Begriff: Statischer Test

- Statische Tests werden **nicht am laufenden System**, sondern an seinen Dokumenten (Quellcode und Dokumentation) ausgeführt.
- Sie sollen Fehler identifizieren und die Qualität verbessern.
- Statische Tests sind ein umfangreiches Gebiet. Hier werden nur die wesentlichen Verfahren benannt:
  - **Reviews:**  
Prüfung der Dokumente durch Personengruppen nach festgelegten Regeln.
  - **Statische Analysen** (meist werkzeuggestützt):
    - Datenflussanalyse
    - Kontrollflussanalyse
    - Berechnung von Metriken



## Wiederholung und Präzisierung: Was ist Testen?

- (Dynamisches) Testen von Software
  - Ein Testobjekt wird zur Überprüfung stichprobenartig ausgeführt.
  - Randbedingungen müssen festgelegt werden.
  - Die Soll-Eigenschaften werden anschließend mit den Ist-Eigenschaften verglichen.

erwartetes vs. geliefertes Verhalten

- Ziele des Testens:
  - Fehlerwirkungen nachweisen
  - Qualität bestimmen
  - Vertrauen und Verständnis schaffen
  - durch Analyse Fehlerwirkungen vorbeugen

nach [Spillner, Linz]

## Grundsätze zum Testen (1)



In den letzten 40 Jahren haben sich folgende Grundsätze zum Testen herauskristallisiert und können somit als Leitlinien dienen:

- **Testen garantiert nicht Fehlerfreiheit.**  
Mit Testen wird die Anwesenheit von Fehlerwirkungen nachgewiesen. Testen kann nicht zeigen, dass keine Fehlerzustände im Testobjekt vorhanden sind!  
»Program testing can be used to show the presence of bugs, but never to show their absence!« Edsger W. Dijkstra, 1970
- **Vollständiges Testen ist nicht möglich.**  
Vollständiges Testen – Austesten – ist (abgesehen von wenigen Ausnahmen) nicht möglich.  
(Beispiel kommt)

## Grundsätze zum Testen (2)



- **Mit dem Testen frühzeitig beginnen.**  
Testen ist keine späte Phase in der Softwareentwicklung, es soll damit so früh wie möglich begonnen werden. Durch frühzeitiges Prüfen (z.B. Reviews) parallel zu den konstruktiven Tätigkeiten werden Fehler(zustände) früher erkannt und somit Kosten gesenkt.
- **Wo viele Fehler sind, verbergen sich meist noch mehr.**  
Fehlerzustände sind in einem Testobjekt nicht gleichmäßig verteilt, vielmehr treten sie gehäuft auf. Dort wo viele Fehlerwirkungen nachgewiesen wurden, finden sich vermutlich auch noch weitere.

## Grundsätze zum Testen (3)

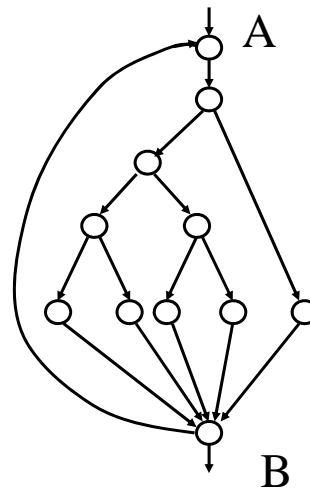


- **Tests müssen gewartet werden.**  
Tests nur zu wiederholen, bringt keine neuen Erkenntnisse. Testfälle sind zu prüfen, zu aktualisieren und zu modifizieren. Tests müssen also, genau wie Software, dynamisch Veränderungen angepasst werden, sonst sterben sie.
- **Testen ist abhängig vom Umfeld.**  
Sicherheitskritische Systeme sind anders (intensiver, mit anderen Verfahren, ...) zu testen als beispielsweise der Internetauftritt einer Einrichtung.
- **Erfolgreiche Tests garantieren nicht Benutzbarkeit.**  
Ein System ohne Fehlerwirkungen bedeutet nicht, dass das System auch den Vorstellungen der späteren Nutzer entspricht.

## Austesten?

- Ein einfaches Programm soll getestet werden, das aus vier Verzweigungen (IF-Anweisungen) und einer umfassenden Schleife besteht und somit fünf mögliche Wege im Schleifenrumpf enthält.
- Unter der Annahme, dass die Verzweigungen voneinander unabhängig sind und bei einer Beschränkung der Schleifendurchläufe auf maximal 20, ergibt sich folgende Rechnung:  

$$5^1 + 5^2 + \dots + 5^{18} + 5^{19} + 5^{20}$$
- Wie lange dauert das Austesten bei 100.000 Tests pro Sekunde?

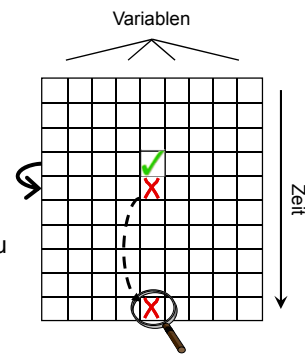


## Fehlerarten in Software (nach B. Meyer)

- Meyer differenziert Begriffe zu Fehlern in Softwaresystemen in folgender Weise:
  - **Programmierfehler (engl.: error):**  
Eine falsche Entscheidung, die während der Softwareentwicklung getroffen wurde.
  - **Programmfehler (engl.: defect):**  
Sie sind Folge von Programmierfehlern, „stecken“ in einer Software und **können** bei ihrer Ausführung bewirken, dass sich die Software nicht so verhält, wie dies gedacht war.
  - **Laufzeitfehler (engl.: fault):**  
Sie treten als Folge von Programm- oder Hardware-Fehlern zur Laufzeit auf. Ihr Effekt ist ein Programmabbruch, eine Fehlermeldung oder eine aus Sicht des Benutzers inakzeptable Systemreaktion.

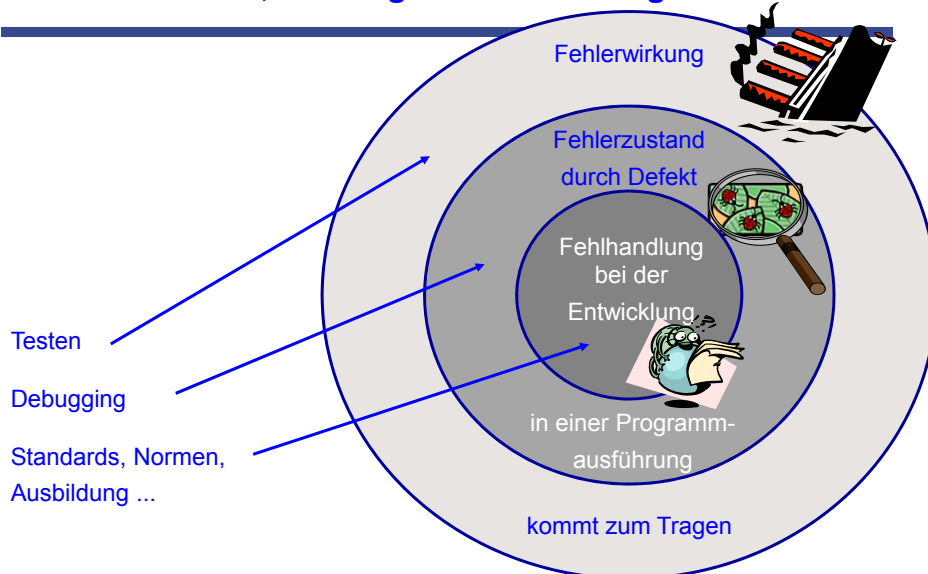
## Von Ursache zu Entdeckung: eine Motivation für das Vertragsmodell

- Zeller differenziert Laufzeitfehler **bei der Suche nach Fehlern** in Software etwas genauer:
  - **defect** – der eigentliche **Defekt**, etwa eine fehlerhafte Anweisung in einer Quelltextzeile;
  - **infection** – die **Verfälschung** des Speicherzustandes aufgrund eines Defektes, der zur Ausführung kommt;
  - **failure** – den **extern beobachtbaren Fehler**, etwa durch einen Programmabbruch.
- Je **größer der Abstand** vom Wirken eines Defektes bis zu seiner Entdeckung, desto **schwieriger und damit teurer** das Finden der Fehlerursache.
- **Das Vertragsmodell, konsequent umgesetzt, unterstützt beim schnelleren Aufdecken von Programmierfehlern!**



Zeller, A.: "Why Programs Fail – A Guide to Systematic Debugging", dpunkt-Verlag, 2006.

## Fehler: Ursache, Wirkung und Entdeckung



nach © 2007 - 2010 by GTB V1.0 / 2007; A. Spillner

## Nochmals: Weshalb trotzdem Testen?

- Der **Quelltext kann leichter weiterentwickelt werden**, weil die Veränderungen einfach auf Korrektheit geprüft werden können.
- Die **Debugging-Zeiten reduzieren sich**, weil durch die Tests Fehler schneller lokalisiert werden können.
- **Schnittstellen werden einfach**, da jeder Programmierer lieber einfachere Schnittstellen testet. Dadurch wird auch vermieden, dass Technologie auf Vorrat gebaut wird.
- **Testklassen zeigen die vom Entwickler einer Klasse vorgesehene Verwendung** und können als ein Teil der Dokumentation des Quelltextes verstanden werden.
- Bei sog. „**Test First-Ansätzen**“ (kommt noch) kann das Testen als eine **Form der Spezifikation** der zu implementierenden Methode verstanden werden.

## Begriffsvielfalt beim Testen



- Es gibt viele Arten des Testens und unterschiedliche Begriffe dazu. Wir unterscheiden grundlegend:
  - **Funktionale Tests:**  
Das von außen sichtbare Ein-/Ausgabeverhalten des Testobjekts wird geprüft. Üblich sind dabei sog. Black-Box-Verfahren, mit denen die funktionalen Anforderungen an ein Programm geprüft werden.
  - **Nicht-funktionale Tests:**  
Prüfen qualitativer Eigenschaften einer Software. Üblich sind Lasttest, Performanztest, Massentest, Stresstest, Test im Dauerbetrieb, Test auf Robustheit, Test auf Benutzerfreundlichkeit (Usability).
  - **Strukturbezogene Tests:**  
Beziehen sich auf die interne Struktur und Architektur der Software (nach White-Box-Verfahren).
  - **Tests bezogen auf den Entwicklungsprozess:**  
Den klassischen Aktivitäten im Entwicklungsprozess lassen sich Tests zuordnen: Komponententest, Integrationstest, Systemtest, Abnahmetest, Test nach Änderungen.

### Aus SE1: Positives und negatives Testen



- Die gesamte Funktionalität einer Software-Einheit sollte durch eine Reihe von Testfällen überprüft werden.
- Ein **Testfall** besteht aus der Beschreibung der erwarteten Ausgabedaten für bestimmte Eingabedaten.
- Wenn nur **erwartete/gültige** Eingabewerte getestet werden, spricht man von **positivem Testen**.
- Wenn **unerwartete/ungültige** Eingabewerte getestet werden, spricht man von **negativem Testen**.
- Positive Tests erhöhen das Vertrauen in die **Korrektheit**, negative Tests das Vertrauen in die **Robustheit**.



Vorsicht:  
Positiv/Negativ hat  
hier nichts mit  
true/false zu tun!



### Aus SE1: Modultest und Integrationstest



- Wenn die Einheiten eines Systems (Methoden, Klassen) isoliert getestet werden, spricht man von einem **Modultest** (engl.: unit test). Modultests sind eher technisch motiviert und orientieren sich an den programmiersprachlichen Einheiten eines Systems.
- Wenn alle getesteten Einzelteile eines Systems in ihrem Zusammenspiel getestet werden, spricht man von einem **Integrationstest** (engl.: integration test).
- Da erfolgreiche Modultests die Voraussetzung für Integrationstests sind, betrachten wir vorläufig nur Modultests näher.
- Die Methoden zum Modultest lassen sich grob in **Black-Box**-, **White-Box**- und **Schreibtischtests** unterteilen.
- Black-Box- und White-Box-Tests sind **dynamische Tests** (das Testobjekt wird ausgeführt), Schreibtischtests sind **statische Tests**.

## Begriff: Komponententest



- Die kleinsten sinnvollen Bausteine im Entwicklungsprozess werden getestet.
  - **Unit-Test** oder **Modul-Test** sind eigentlich Begriffe aus der traditionellen imperativen Programmierung. Dort werden einzelne Prozeduren getestet.
  - Heute spricht man meist von **Komponententest**, der sich auf Methoden in Klassen, Klassen und ganze Subsysteme beziehen kann.
  - Durch **JUnit** (siehe SE1) hat der Begriff Unit-Test als automatisierter Regressionstest eine neue Interpretation bekommen.
- Geprüft werden die einzelnen Software-Bausteine isoliert von den anderen Systemteilen.
- Dabei soll sichergestellt werden, dass das Testobjekt die einzelnen funktionalen Anforderungen korrekt und vollständig realisiert.
- Teststrategien sind der **Whitebox-Test** oder **Test-first-Ansätze** (kommt noch).

## Begriff: Integrationstest



- Die einzelnen (getesteten) Komponenten der Software werden zu Teilsystemen oder Baugruppen zusammengesetzt, die dann getestet werden.
- Geprüft wird das **Zusammenspiel der Software-Bausteine im Verbund**.
- Dabei soll sichergestellt werden, dass die Schnittstellen korrekt realisiert sind und die Protokolle (zulässigen Aufrufsequenzen) erfüllt werden.
- Teststrategien sind z.B.:
  - **Top-down-Integration**, d.h. der Test beginnt mit der Komponente, die andere ruft aber selbst nicht gerufen wird. Schrittweise werden zunächst Platzhalter hinzugefügt und dann durch untergeordnete Komponenten ersetzt.
  - **Bottom-up-Integration**, d.h. zunächst werden die Komponenten getestet, die keine anderen aufrufen. Dann werden die übergeordneten Komponenten hinzugefügt.
  - **Backbone-Integration**, d.h. ein Systemkern (Backbone) wird erstellt, dann werden schrittweise nachgeordnete Komponenten integriert.

### Begriff: Systemtest



- Das **integrierte System** wird **insgesamt getestet**.
- Das **System wird komplett in einer Testumgebung getestet**, die der späteren "Produktionsumgebung" möglichst vollständig entspricht.
- Dabei soll sichergestellt werden, dass das System alle funktionalen und nicht-funktionalen Anforderungen vollständig realisiert.
- Teststrategien sind alle Testarten für funktionale und nicht-funktionale Anforderungen.

### Begriff: Abnahmetest



- Das **einsatzbereite System** wird vor der Übergabe an den Auftraggeber/Nutzer getestet.
- Das System wird in einer **Testumgebung** oder in der **Produktionsumgebung** getestet.
- Dabei können unterschiedliche Ziele verfolgt werden:
  - Test auf vertragliche Akzeptanz
  - Test auf Benutzerakzeptanz
  - Test auf Akzeptanz durch den Systembetreiber
  - Feldtest beim Hersteller oder beim Kunden (Alpha- und Beta-Tests)



### Begriff: Test nach Änderungen / Regressionstest



- Das veränderte System wird getestet.
- Dabei soll sichergestellt werden, dass alle alten und ggf. die neuen Anforderungen vollständig realisiert sind.
- Teststrategien beim sog. Regressionstest sind:
  - Wiederholung aller Tests, die Fehlerwirkungen erzeugt haben.
  - Vollständiger Test aller Programmkomponenten, die verändert worden sind.
  - Test aller neuen Programmkomponenten.
  - Vollständiger Test des gesamten Systems

### Testen objektorientierter Software: nicht einfach!

- Die **objektorientierte Programmierung** besitzt gegenüber der klassischen imperativen Programmierung **strukturelle und dynamische Besonderheiten**, welche **beim Testen** zu berücksichtigen sind.
- Das Thema Testen von objektorientierten Programmen ist sehr umfangreich und wird in SE2 nur im Überblick behandelt.
- In den nächsten Vorlesungen werden wir wiederholt darauf zurückkommen, wenn grundlegende Konzepte eingeführt sind (beispielsweise Vererbung).
- Ausführliche Informationen sind in den Referenzen zu finden.

## Probleme des oo Testens: Kapselung

- Der Mechanismus der **Kapselung** bedeutet, dass die **Implementation einer Operation verborgen** ist. Dieses softwaretechnisch wertvolle Prinzip erschwert den Test einer Klasse:
  - Reine Black-Box-Tests (siehe SE1) decken erfahrungsgemäß nur ein Drittel bis zur Hälfte der Zustände oder Ausführungspfade einer Klasse ab, da nur die nach außen sichtbare Struktur getestet werden kann.
- Beim Testen ist es oft wichtig zu wissen,
  - welchen **konkreten Zustandsraum** ein Objekt haben kann,
  - wie die **Klasse strukturell eingebettet** ist,
  - welche **Abhängigkeiten** sich daraus ergeben.

Dazu ist es notwendig, direkt auf den gekapselten Zustand eines Objektes zuzugreifen.

## Probleme des oo Testens: Komplexe Abhängigkeiten

- Objektorientierte Software besteht aus Objekten. Objekte kommunizieren miteinander und ändern ihren Zustand durch Austausch von Nachrichten.
- Ob und wie ein Objekt auf eine Nachricht reagiert, definiert sich durch seinen eigenen Zustand und ggf. den Zustand, den es an einem anderen Objekt beobachten kann.
- Dadurch **bilden Objekte zur Laufzeit ein zeitabhängiges Netzwerk von kommunizierenden Einheiten** mit mehreren „Einstiegspunkten“ und ohne zentrale Instanz, die den Kontrollfluss steuert.
- Das macht das Testen von Kontrollflüssen sehr komplex:
  - **Welches Netz von Objekten muss für einen Test aufgebaut werden?**
  - **In welchem Zustand müssen die Objekte sein?**
  - **Bei welchem Objekt beginnt der Test?**

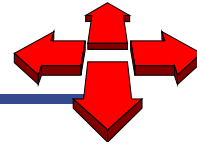
## Testgetriebene Entwicklung

- **Agile Methoden** empfehlen, für jede Klasse eine eigene Testklasse zu schreiben.
- Beim sog. **Test First-Ansatz** sollen Testklassen **vor** den zu testenden Klassen geschrieben werden. Vor der Implementation der Methoden einer Klasse wird durch entsprechende Tests spezifiziert, welches Problem mit welchen Randbedingungen gelöst werden soll. So lässt sich auch eine gute Anweisungsüberdeckung erreichen.
- Testen und Programmieren sollen **in schnellem Wechsel** aufeinander folgen. Jede neue Klasse und Operation wird sofort getestet. Am Ende des Tages müssen alle Testfälle bei der Integration korrekt durchlaufen.
- Wir werden noch sehen: Insbesondere systematisches **Refactoring** erfordert Testklassen und automatisches Testen für eine sichere Veränderung in Einzelschritten.

## Ausblick: Weitere Testarten

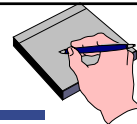
- **Akzeptanztests** sind Tests einer lauf- und einsatzfähigen Software-Version aus Sicht der Benutzung. Sie werden von Anwendern und Auftraggebern aus deren jeweiligen Blickwinkel (z.B.: Ist das System benutzbar? Erfüllt das System die vertraglichen Anforderungen?) durchgeführt.
- **Benchmark-Tests** messen die Performanz eines Systems gegen ein Vergleichssystem oder einen vorgegebenen Wert (z.B. Antwortzeit max. 0,5 Sekunden).
- **Lasttests** (Load Tests) prüfen das Verhalten des Systems bei praxisrelevanter Beanspruchung. Werden Extremsituationen (z.B. sehr große Benutzerzahlen oder hoher Datendurchsatz) getestet, spricht man von **Stresstests**.
- **Robustheitstests** prüfen, wie ein System auf Fehler, Ausnahmen oder nicht-spezifizierte Benutzereingaben reagiert.
- **Installationstests** prüfen, wie sich ein System in unterschiedlichen Installationskontexten verhält.

## Das Vertragsmodell



- Aus SE1 kennen wir (generische) Java-Datentypen wie **Stack**, **Queue**, **List** und **Set**.
- Das **Verhalten** solcher Datentypen kann programmiersprachen-unabhängig spezifiziert werden, in Form so genannter **Abstrakter Datentypen** (engl.: abstract data types, häufig abgekürzt mit **ADT**).
- Bevor wir uns ADTs näher ansehen, wollen wir zuerst eine pragmatische Anwendung der theoretischen Konzepte abstrakter Datentypen betrachten: das **Vertragsmodell** der objektorientierten Softwareentwicklung.
- Das Vertragsmodell ermöglicht unter anderem, das Grundkonzept der Objektorientierung, das Verhältnis zwischen **Klient** und **Dienstleister**, klarer zu fassen.

## Vertrag

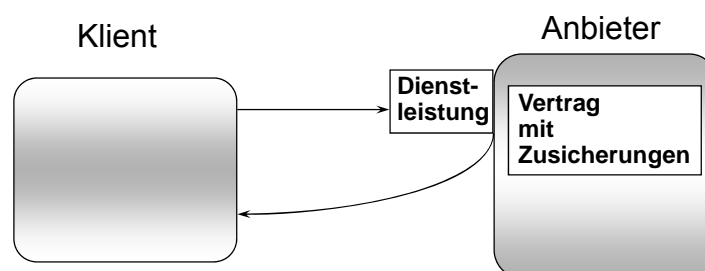


*Nach Wikipedia:*

Ein **Vertrag** ist eine von zwei oder mehreren Vertragspartnern geschlossene Übereinkunft. Er dient der Herbeiführung eines von den Parteien gewollten Erfolges. Der Vertrag kommt durch übereinstimmende Willenserklärungen zustande.

*Im Vertragsmodell:*

Ein **Klient** und ein **Anbieter** schließen einen **Vertrag** über eine Dienstleistung. Die Bedingungen des Vertrags sind als wechselseitige Zusicherungen formuliert und werden vom Anbieter verwahrt.



## Das Vertragsmodell der objektorientierten SW-Entwicklung

Die Metapher:

- Die Benutzt-Beziehung zwischen Klassen wird als Vertragsverhältnis zwischen **Klient** (engl.: Client) und **Lieferant** (engl.: Supplier) interpretiert:
- Eine Klasse als Lieferant bietet eine Dienstleistung an, die eine andere Klasse als Klient nutzen will.
- Im Vertrag wird formuliert,
  - welche Vorleistung der Klient erbringen muss,
  - damit der Lieferant seine Dienstleistung liefert
  - und dies auch garantiert.
- Der Begriff **Vertragsmodell** ist unsere deutsche Übersetzung des englischen **Design by Contract**, wie es von Meyer geprägt wurde.



## Benutzt-Beziehung und Vertragsmodell

- Eine **Benutzt-Beziehung** zwischen zwei Klassen **A** und **B** wird (in Java) auf zwei Arten hergestellt:
  - In Klasse **A** werden Variablen vom Typ **B** deklariert.
  - In einer Methode der Klasse **A** werden Operationen an Exemplaren der Klasse **B** aufgerufen.
- Das **Vertragsmodell** bezieht sich auf:
  - den Aufruf von Operationen,
  - die Überprüfung von Aufrufparametern sowie des Zustands des gerufenen Exemplars.

## Die Bestandteile des Vertragsmodells

- **Der Vertrag:**
  - Ein Vertrag bezieht sich immer auf eine Operation einer Klasse.
  - Er wird in der Lieferanten-Klasse festgelegt.
  - Die Vertragsbedingungen werden als Zusicherungen spezifiziert.
- **Zusicherungen:**
  - Zusicherungen sind **boolesche Ausdrücke** (Prädikate). Es gibt:
    - **Vorbedingungen** (vor der Ausführung der Operation)
    - **Nachbedingungen** (nach der Ausführung der Operation)
- Zusätzlich gibt es **Invarianten**:
  - Bedingungen, die im Vertragsmodell immer gelten sollen, werden als **Klassen-Invarianten** festgehalten, die ebenfalls **boolesche Ausdrücke** sind.

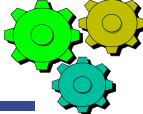
## Der Mechanismus des Vertragsmodells


- Ein Vertrag wird bei jedem Operationsaufruf geprüft:
  - Der Klient muss sicherstellen, dass die Vorbedingungen der Operation erfüllt sind. Beim Lieferant muss geprüft werden, ob die Vorbedingung gilt.
  - Wenn die Vorbedingung erfüllt ist, führt der Lieferant die Operation aus.
  - Der Lieferant garantiert dann durch die Nachbedingungen, dass die Leistung erbracht ist.
- Die Klasseninvariante ist eine allgemeine Randbedingung des Vertrags. Sie muss bei jedem Operationsaufruf gelten.

**Merke:**

Wenn  
der **Klient** die **Vorbedingungen** erfüllt,  
dann  
garantiert der **Lieferant** die **Nachbedingungen**.

## Ein Beispiel aus einer Bibliothek





Die fachliche Schnittstelle der Klasse **Buch** ist aus den Aufgaben und Tätigkeiten von BibliothekarInnen und BibliotheksbenutzerInnen abgeleitet.


**Buch**


ausleihen (b : Benutzer)  
 zurückgeben (b : Benutzer)  
 mahnen (b : Benutzer)  
 verlängern (b : Benutzer, f : Frist)  
 vorbestellen (b : Benutzer)

- Probleme bei der Verwendung
  - Wir haben ein Vorverständnis von der Bedeutung der Operationen an der Schnittstelle, kennen aber die genaue Semantik nicht.
  - Vermutlich lässt sich nicht jede Operation an einem Buch-Objekt jederzeit aufrufen.
  - Es gelten vermutlich Regeln für die Verwendung der Operationen ("ein Buch muss zurückgegeben sein, ehe es ausgeliehen werden kann").

SE2 – OOPM – Teil 1 65

## Auf dem Weg zum Vertragsmodell (1)





Wir formulieren Vor- und Nachbedingungen aus fachlicher Sicht.

Vorbedingungen werden durch **require** gekennzeichnet, Nachbedingungen durch **ensure** (Der Syntax der Sprache Eiffel folgend).

**Buch**

ausleihen (b : Benutzer)  
     **require:** NOT (ist\_ausgeliehen())  
     **ensure:** ist\_ausgeliehen()

zurückgeben (b : Benutzer)  
     **require:** ist\_ausgeliehen()  
     **ensure:** NOT (ist\_ausgeliehen())

...

SE2 – OOPM – Teil 1 66

## Rechte und Pflichten des Klienten

- Der Klient hat die **Pflicht**, die Operation nur aufzurufen, wenn der **Zustand des Lieferanten** es erlaubt.
- Dann hat der Klient aber auch das **Recht**, die **ordnungsgemäße Erfüllung** des Vertrags zu erwarten.



Einhalten der Vorbedingungen



Erfüllung der Nachbedingungen

- Das Vertragsmodell wurde bisher nur in der Sprache Eiffel direkt umgesetzt.



```

public class Buch
{
    //Vorbedingung: NOT (ist_ausgeliehen())
    public void ausleihen(Benutzer b)
    { ... }
    //Nachbedingung: ist_ausgeliehen()

    public boolean ist_ausgeliehen()
    { ... }

    private boolean _ausgeliehen;
    private Benutzer _b;
}

```

Lieferant

## Rechte und Pflichten des Lieferanten

- Der Lieferant hat die **Pflicht**, die **versprochene Leistung** zu erbringen.
- Er hat aber gleichzeitig das **Recht** bei Nichteinhaltung des Vertrags von Seiten des Klienten, die **Operation nicht auszuführen**.



Erfüllung der Nachbedingungen



Einhalten der Vorbedingungen

**Klient**

```

Benutzer einBenutzer = new Benutzer("hz");
Buch einBuch = new Buch();

einBuch.ausleihen (einBenutzer); ✓

einBuch.ausleihen (einBenutzer); ✗

```

```

public class Buch
{
    //Vorbedingung: NOT (ist_ausgeliehen())
    public void ausleihen(Benutzer b)
    { ... }
    //Nachbedingung: ist_ausgeliehen()

    public boolean ist_ausgeliehen()
    { ... }

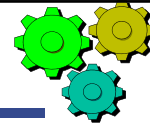
    private boolean _ausgeliehen;
    private Benutzer _b;
};

```

**Lieferant**

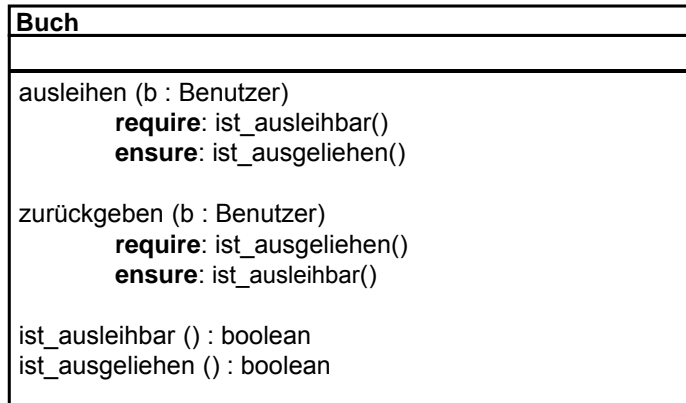


## Auf dem Weg zum Vertragsmodell (2)



Damit der Klient seine Verpflichtungen erfüllen kann, muss er die entsprechenden Vorbedingungen des Lieferanten prüfen können.

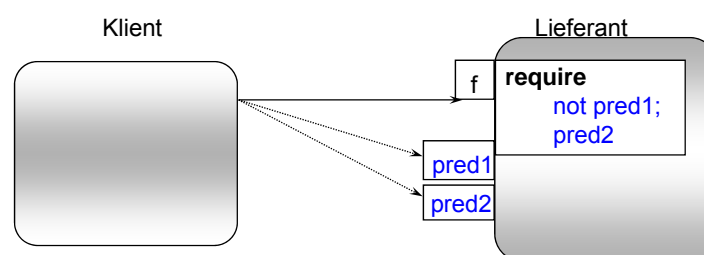
Dazu muss die Schnittstelle des Lieferanten entsprechende sondierende Operationen enthalten. Dabei wird der Zusammenhang zwischen den Operationen ausleihen() und zurückgeben() deutlich.



SE2 – OOPM – Teil 1

69

## Der Klient muss seine Verpflichtungen prüfen können



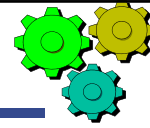
### Vertragsmodell und Zusicherungen:

- Damit ein Vertrag sinnvoll ist, muss der Klient die Einhaltung der Vorbedingung einer zu rufenden Operation prüfen können.
- Daher müssen alle in einer Vorbedingung verwendeten Terme geeignet als Prädikate (sondierende Operationen) an der Schnittstelle sichtbar sein.
- Da die Einhaltung der Nachbedingung Aufgabe des Lieferanten ist, müssen die entsprechenden Terme nicht Teil der Schnittstelle werden.

SE2 – OOPM – Teil 1

70

### Auf dem Weg zum Vertragsmodell (3)



Alle in Vorbedingungen verwendeten Terme werden geeignet in Prädikate (sondierende boolsche Operationen) umgesetzt.

- Prädikate sind total, d.h. sie können in jedem Zustand des Objekts gerufen werden; sie dürfen also selbst keine Vorbedingungen haben.
- Prädikate haben keinen von außen sichtbaren Effekt auf den Zustand des Objekts.

| Buch               |           |
|--------------------|-----------|
|                    |           |
| ist_ausleihbar()   | : boolean |
| ist ausgeliehen () | : boolean |



- Die beiden Prädikate ist\_ausleihbar() und ist ausgeliehen() sind immer an Exemplaren der Klasse Buch ausführbar. Bei ihrer Implementation ist darauf zu achten, dass keine Zustandsvariablen durch Zuweisung verändert werden.

### Operationen an der Schnittstelle

Die Schnittstelle einer Klasse sollte den Prinzipien eines ADT entsprechend so aufgebaut sein:

- Die **verändernden Operationen**:  
Verändernde Operationen haben Seiteneffekte, d.h. verändern den Zustand eines Objekts: sie haben selten einen Rückgabewert (z.B. `pop` verändert den Stack; liefert aber kein Element).
- Die **sondierenden fachlichen Operationen**:  
Sie liefern einen fachlichen Rückgabewert (Fachwert) oder ein Rückgabeobjekt.  
(z.B. `top` liefert ein Stack-Element).  
Eine sondierende Operation hat keine Seiteneffekte, d.h. der (von außen sichtbare) Zustand eines Objektes bleibt unverändert.
- Die **sondierenden booleschen Operationen**:  
Sie prüfen den jeweiligen Objektzustand und werden besonders zur Sicherung von Vorbedingungen verwendet.

## Zustandsabhängigkeit von Operationen

Nicht jede Operation eines Objekts ist zu jedem Zeitpunkt aufrufbar:

- Die **verändernden Operationen**:  
Sie sind selten zu jedem Zeitpunkt im Lebenszyklus eines Objekts aufrufbar. Dies sollten die Vorbedingungen deutlich machen. Da ihr Aufruf den Objektzustand verändert, muss **vor jedem Aufruf** erneut die Zulässigkeit geprüft werden.
- Die **sondierenden fachlichen Operationen**:  
Sie sind ebenfalls meist nur an bestimmten Zeitpunkten im Lebenszyklus eines Objekts aufrufbar. Dies sollten die Vorbedingungen deutlich machen. Durch ihren wiederholten Aufruf sollte sich nichts an der Zulässigkeit verändern.
- Die **sondierenden boolschen Operationen**:  
Sie sind zu jedem Zeitpunkt aufrufbar. Sie dürfen keine Vorbedingungen haben.

## Klassen-Invarianten



**Klassen-Invariante** (meist kurz Invariante):

- Eine Klassen-Invariante gilt für alle Exemplare einer Klasse und muss von allen Operationen berücksichtigt werden. Sie beschreibt (semantische) Randbedingungen einer Klasse insgesamt.
- Formal ist sie eine boolesche Aussage über alle Exemplare einer Klasse, die *vor* und *nach* Ausführung jeder Operation der Klasse gelten muss.

- Beispiel 1:  
Wir modellieren zunächst, dass ein Buch ausgeliehen oder ausleihbar ist:  
`Invariant: ist_ausgeliehen() ^ ist_ausleihbar()`
- Beispiel 2:  
Wenn wir berücksichtigen, dass ein Buch vorbestellt werden kann, dann erweitert sich die Invariante zu:  
`Invariant: ist_ausgeliehen() ^ ist_ausleihbar();  
NOT ist_ausleihbar() ^  
    ( ist_vorhanden() & NOT ist_vorbestellt() )`

## Korrektheit einer Klasse

### Korrektheit bezogen auf die Zusicherungen:

- Die Korrektheit eines Objektes kann nur in sog. **stabilen Zuständen** geprüft werden, d.h. vor und nach der Ausführung von Operationen.
- Dann muss auch jeweils die Klasseninvariante gelten.
- Zwischenzeitlich kann die Invariante (z.B. während des Aufrufs einer privaten Methode) verletzt sein.

- Eine Klasse K ist im Sinne des Vertragsmodells korrekt, wenn:
  - **beim Erzeugen eines Objekts** die Vorbedingung des gerufenen Konstruktors erfüllt ist, und die Nachbedingung und die Invariante vom gerufenen Konstruktor erfüllt wird.
  - **für jede gerufene Operation** die Vorbedingung und die Invariante gelten, und die Nachbedingung und die Invariante von der gerufenen Operation erfüllt wird.



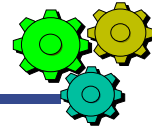
## Kein eingebautes Vertragsmodell in Java

- Java bietet **keine Sprachunterstützung** für das Vertragsmodell; d.h. Vor- und Nachbedingungen und Invarianten sind **nicht Teil des Sprachmodells**.
- **James Gosling**, der Designer von Java, hat in einem Interview gesagt, dass er anfangs das Vertragsmodell in die Sprache integrieren wollte. Das mangelnde Verständnis der meisten Programmierer der damaligen Zeit für das Konzept hat ihn dann aber davon abgehalten, was er inzwischen bedauert.
- Wenn wir das Vertragsmodell in Java umsetzen wollen, müssen wir es deshalb **„von Hand“ (manuell) programmieren**.



„The C Family of Languages: Interview with Dennis Ritchie, Bjarne Stroustrup, and James Gosling“, Java Report, 5(7), July 2000.

## Manuelle Umsetzung des Vertragsmodells in Java



- Wenn wir das Vertragsmodell in Java umsetzen wollen, dann müssen wir sowohl die **Dokumentation** der Verträge als auch deren **Überprüfung** bedenken.
  - Die **Dokumentation** sollte im **Schnittstellenkommentar** einer Operation stehen, damit die Vertragsinformationen für einen Klienten ersichtlich sind.
  - Die **Überprüfung** kann nur innerhalb des Methodenrumpfes erfolgen, der die Operation implementiert:
    - Vorbedingungen sollten unmittelbar zu Beginn geprüft werden.
    - Nachbedingungen sollten beim Methodenausgang geprüft werden; bei einer **return**-Anweisung kann die Prüfung aber nur unmittelbar vor dieser Anweisung erfolgen.
  - Die Invarianten werden meist nur im Klassenkommentar notiert.

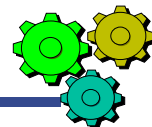


- Die Trennung von Dokumentation und Überprüfung führt zu einem potenziellen Wartungsproblem: Bei Änderungen kann es leicht zu Inkonsistenzen zwischen Programmtext und Kommentar kommen!

SE2 – OOPM – Teil 1

77

## Überprüfung von Verträgen in Java



- Für die manuelle Überprüfung gibt es verschiedene Möglichkeiten:
  - Eine **spezifische Fehlerbehandlung für jeden Einzelfall** mit jeweils passendem Exception-Typ.
  - Eine zentrale Implementation, z.B. über Klassenmethoden einer Contract-Klasse wie im **Framework JWAM**:
 

```
public static void require (boolean precondition,
                             Object contractor, String description);
```

**Vorteil:** Knappere Darstellung, einheitliche Behandlung  
**Nachteil:** Die Aufrufe lassen sich bei Bedarf nicht leicht entfernen.
  - Verwendung der **assert-Anweisung** von Java.
 

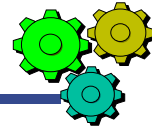
**Vorteil:** ebenfalls knapper, zusätzlich (zur Laufzeit) abschaltbar  
**Nachteil:** In Java bedeutet abschaltbar leider, dass das Anschalten vergessen werden kann (die Überprüfung von Assertions ist standardmäßig **nicht** angeschaltet).



SE2 – OOPM – Teil 1

78

## Vertragsprüfung in Java mit assert

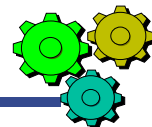


- Die **assert-Anweisung** von Java gibt es in zwei Varianten:  
`assert BooleanExpression ;`  
`assert BooleanExpression : Expression ;`
- Die zweite Variante kann für das Vertragsmodell verwendet werden, indem der boolesche Ausdruck die eigentliche Zusicherung formuliert, während der Ausdruck nach dem Doppelpunkt einen String mit der Beschreibung der Zusicherung liefert.
- Beispiel für unsere altbekannte Konto-Klasse:

```
/**
 * Zahle einen Betrag auf dieses Konto ein.
 * @require betrag >= 0
 */
public void einzahlen (int betrag)
{
    assert betrag >= 0 : "Vorbedingung verletzt: betrag >= 0";
    ...
}
```



## Diskussion: assert für Vertragsmodell in Java



- Sun Microsystems rät explizit von der Verwendung der assert-Anweisung für Vorbedingungen ab, da die Überprüfung von Asserts eine Laufzeit-Option ist, die ausgeschaltet werden kann.
- Diese Position ist aus der **Sicht eines API-Anbieters** formuliert; Bibliothekscode sollte bei der Überprüfung seiner Vorbedingungen nicht davon abhängig sein, ob zufällig auf der ausführenden Virtual Machine die Assertions geprüft werden. Stattdessen sollten die Vorbedingungen explizit überprüft werden und bei einer Verletzung eine passende Runtime-Exception ausgelöst werden (etwa eine **InvalidArgumentException**).
- Wir teilen dieses Argument nicht ganz, denn zumindest **innerhalb eines Software-Entwicklungsprojektes** besteht die Kontrolle über die ausführende Virtual Machine.
- Weiterhin ist das Vertragsmodells primär eine Hilfe bei der Entwicklung; in einem ausgelieferten System müssen die Verträge nicht mehr zwingend überprüft werden.
- Aber auch in einem laufenden System können durch die Prüfung Inkonsistenzen früher erkannt werden...



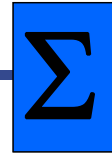
## Vertragsmodell und Ausnahmen

- **Verträge müssen erfüllt werden** oder es gibt eine **Vertragsverletzung**; eine Vertragsverletzung aber ist ein **Programmierfehler**!
- Folglich sollten fehlgeschlagene Zusicherungen zu einem **Programmabbruch** führen, bevor Schlimmeres passieren kann.
- Moderne Sprachen bieten dazu das Konzept von **Ausnahmen** (engl.: exceptions), das wir (zumindest für Programmabbrüche) bereits in SE1 kennen gelernt haben (mehr zu Ausnahmen/Exceptions später).
- Nach dem Vertragsmodell tritt eine **Ausnahmesituation** auf, wenn
  - der Klient die Vorbedingungen nicht liefert (Fehlerursache liegt beim Klienten), oder
  - der Lieferant die Nachbedingungen und Invarianten nicht erfüllen kann (Fehlerursache liegt beim Lieferanten).
- **Grundsätzlich:**  
**Ein Vertrag wird erfüllt oder die Vertragsverletzung wird festgestellt**, d.h. Operationen sind erfolgreich oder schlagen mit einer Ausnahme fehl.

## Vertragsmodell und Testen: Vorsicht bei Negativtests!

- Durch das Vertragsmodell können die **Zuständigkeiten** von **Klient** und **Dienstleister** klarer getrennt und definiert werden.
- Das Vertragsmodell liefert **Hinweise auf Testfälle**: Wenn ein Klient sich an die Vorbedingungen hält, kann er die Nachbedingungen erwarten – und sie in geeigneten Testfällen überprüfen.
- Eine Merkgel aus SE1 lautete: Positivtests müssen, Negativtests können sein! Anders gesagt: **Korrektheit muss, Robustheit kann** geprüft werden.
- Im Zusammenhang mit dem Vertragsmodell gilt: **Negativtests zum Vertragsmodell** (also bewusstes Verletzen der Vorbedingungen, um zu überprüfen, ob der Dienstleister sie überprüft) sind zu **vermeiden**.
- Warum das?
- Weil die **assert-Prüfungen gefahrlos ausschaltbar** sein sollten. Dies wäre bei Negativtests des Vertragsmodells jedoch nicht der Fall, da die Testfälle fehlschlagen würden, die erwarten, dass bei einer verletzten Vorbedingung beispielsweise eine bestimmte Exception geworfen wird.

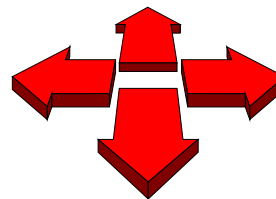
## Zwischenergebnis Vertragsmodell



- Das **Vertragsmodell** ist eine Interpretation der **Benutzt-Beziehung zwischen Klassen**.
- Es unterstützt
  - bei der Formulierung von Konsistenzbedingungen über Klassen und Objekte;
  - die Entwicklung verständlicher und fachlich „runder“ Klassen;
  - eine disziplinierte Fehlerbehandlung.
- **Zusicherungen** (Vor- und Nachbedingungen) und **Randbedingungen** (Invarianten) sind beim Dienstleister (der Angebote macht) formuliert.
- Vor- und Nachbedingungen und Invarianten sollen **keine Repräsentationsdetails**, sondern „vertragsrelevante“ Daten enthalten.
- Wir werden das Vertragsmodell noch einmal im Zusammenhang mit **Vererbung** diskutieren.

## Abstraktion: Polymorphie und Vererbung

- Polymorphie-Begriff
- Übersicht über Vererbungskonzepte
- Typhierarchien: Subtyping





## Motivation: Vererbung

- Vererbung ist nach Wegner die **definierende Eigenschaft** objektorientierter Programmiersprachen.
- Ohne ein Verständnis von Vererbung kein vollständiges Verständnis für OOP!
- Aber: der Begriff ist stark **überladen**; Vererbung wurde auch schon als das „**Goto der Neunziger Jahre**“ bezeichnet.

Was heißt eigentlich Vererbung?

Wegner, P.: "Dimensions of Object-Based Language Design", Proc. OOPSLA '87, Orlando, Florida; in ACM SIGPLAN Notices, Vol. 22:12, 1987.

## „Inheritance Considered Harmful“

- In Anlehnung an Dijkstras „**GoTo Statement Considered Harmful**“.
- Sinnvoll: Unterscheidung der **Konzepte**, die durch Vererbung unterstützt werden sollen, und der **Mechanismen**, die verschiedene Sprachen anbieten.
- „**Harmful**“ werden die **Mechanismen**, die von Programmiersprachen angeboten werden, wenn sie mit **zu vielen Konzepten** überladen werden (wie beim GoTo).
- Vererbung ist insbesondere dann „**harmful**“, wenn die Klassen großer Systeme unsystematisch mit Vererbung von einander abhängig werden (ähnlich wie beim GoTo).

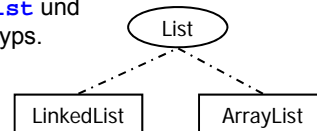
These:

**Vererbung ist das GoTo der Objektorientierung!**



## Bisher: Abstraktionsmittel Interface

- Wir haben bisher **Interfaces** primär als Abstraktion von verschiedenen Implementationen eines Datentyps kennen gelernt.
- Beispiel: Datentyp **List** als Interface, **LinkedList** und **ArrayList** als Implementationen dieses Datentyps.



- Wir wenden uns nun den Konzepten zu, die die Grundlage für Polymorphie und Vererbung in Programmiersprachen bilden.

## Aus SE1: Die Doppelrolle einer Klasse

- Aus Sicht der Klienten einer Klasse ist interessant:
  - Welche **Operationen** können an Exemplaren der Klasse aufgerufen werden?
  - Welchen Typ haben die Parameter einer **Operation** und welches Ergebnis liefert sie?
  - Was sagt die Dokumentation (Kommentare, javadoc) über die Benutzung?
- Für die Implementation der **Methoden** einer Klasse ist relevant:
  - Wie sind die Operationen in den **Methodenrümpfen** umgesetzt?
  - Welche **Exemplarvariablen/Felder** definiert die Klasse?
  - Welche (privaten) **Hilfsmethoden** stehen in der Klasse zur Verfügung?

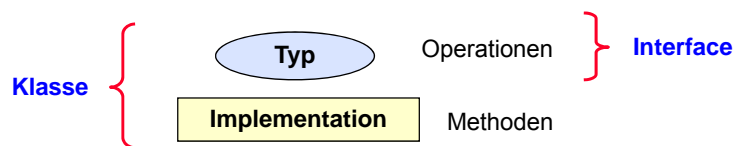


**Außensicht,  
öffentliche  
Eigenschaften,  
Dienstleistungen,  
Typ**

**Innensicht,  
private  
Eigenschaften,  
Implementation**

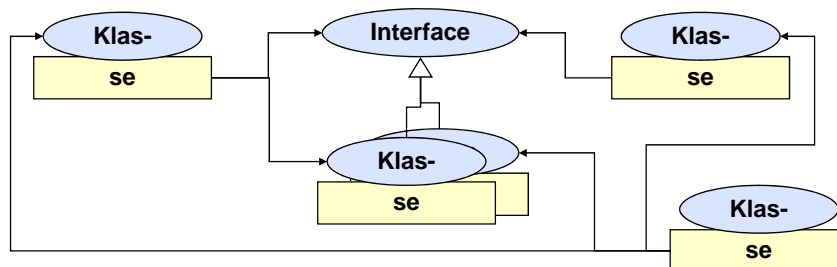
## Die objektorientierte Klasse: Typ und Implementation

- Die Unterscheidung von Schnittstelle bzw. Typ und Implementation kennen wir bereits aus SE1. Eine **Klasse definiert beides**.
- Interfaces** hingegen sind reine Typinformationen, ohne Implementation.

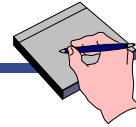


## Statik von OO Systemen: Geflechte von Typen

- Ein objektorientiertes (Java-)System besteht in seiner statischen Sicht aus einer **Menge von Typen** (Klassen und Interfaces) und **Implementationen**.
- Diese **benutzen** sich gegenseitig ausschließlich über ihre **Schnittstellen**, indem sie Operationen aufrufen.
- Zu einem Interface kann es verschiedene Implementationen geben, die auch nebeneinander in einem System zum Einsatz kommen können.  
(Bsp.: Interface **List** mit Implementationen **LinkedList** und **ArrayList**)



## Wiederholung: Statischer und dynamischer Typ



- In objektorientierten Sprachen muss der statische vom dynamischen Typ einer **Referenzvariablen** unterschieden werden.
- Der **statische Typ einer Variablen** wird durch ihren **deklarierten Typ** definiert. Er heißt statisch, weil er zur Übersetzungszeit feststeht.  
`List<String> liste1; // List<String> ist hier der statische Typ von liste1`
- Der statische Typ legt die **Operationen** fest, die über die Variable aufrufbar sind.  
`liste1.add("Simpson"); // add ist hier eine Operation`
- Ein Compiler kann bei der Übersetzung prüfen, ob die genannte Operation tatsächlich im statischen Typ definiert ist.



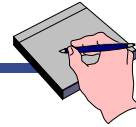
## Wiederholung: Statischer und dynamischer Typ (II)



- Der **dynamische Typ einer Referenzvariablen** hängt von der Klasse des Objektes ab, auf das die Variable zur Laufzeit verweist.  
`liste1 = new LinkedList<String>(); // 1. dynamischer Typ von liste1`
- Er bestimmt die Implementation und ist dynamisch in zweierlei Hinsicht:
  - Er kann erst zur Laufzeit ermittelt werden.
  - Er kann sich während der Laufzeit ändern.  
`liste1 = new ArrayList<String>(); // neuer dynamischer Typ von liste1`
- Ein **Objekt** hingegen **ändert seinen Typ nicht**; es bleibt sein Leben lang ein Exemplar seiner Klasse.
- Der dynamische Typ einer Variablen (bzw. der Typ des referenzierten Objektes) entscheidet darüber, welche **konkrete Methode** bei einem **Operationsaufruf** ausgeführt wird. Da diese Entscheidung erst zur Laufzeit getroffen werden kann, wird dieser Prozess **dynamisches Binden** (einer Methode) genannt.



## Dynamisches Binden



### Dynamisches Binden:

Da erst zur Laufzeit ein konkretes Objekt den **dynamischen Typ** einer Variablen bestimmt, kann der Compiler beim Aufruf einer Operation durch einen Klienten zur Übersetzungszeit nicht festlegen, **welche Methode** tatsächlich **aufzurufen** ist; diese Entscheidung muss deshalb zur Laufzeit (**dynamisch**) getroffen werden.

In Java werden lediglich die Aufrufe privater Exemplarmethoden statisch gebunden. Alle anderen Aufrufe an Exemplare werden dynamisch gebunden!

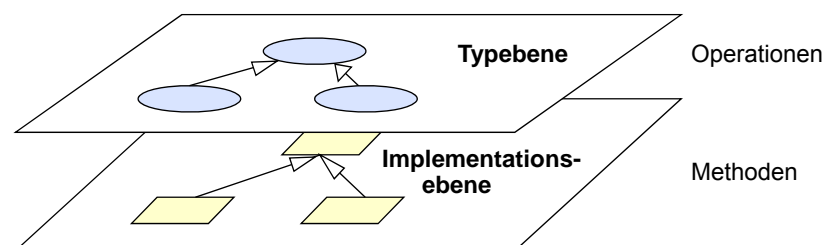


## Vererbung: Auf Typ- und Implementationsebene

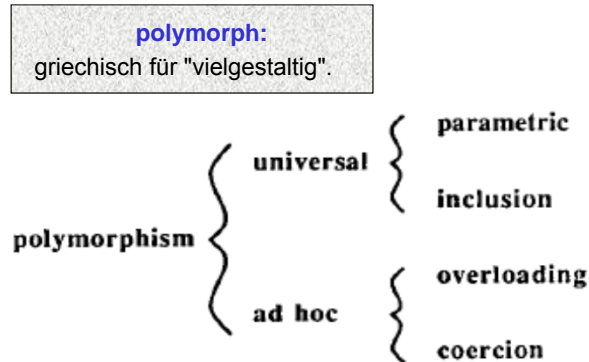
- Mit Vererbung werden **hierarchische Beziehungen** modelliert.
- Die Unterscheidung von Schnittstelle bzw. Typ und Implementation ermöglicht eine differenzierte Betrachtung von Hierarchien auf Typ- und Implementationsebene.

Typ

Implementation



## Polymorphie nach Cardelli und Wegner (1985)



Entsprechung  
in Java:

Generizität (seit Java 5)

Vererbung et al.

Operatoren, insbes. +;  
Überladen von Methoden-  
und Konstruktornamen

Typumwandlungen  
für primitive Typen

© Cardelli, L., Wegner, P.: "On Understanding Types, Data Abstraction and Polymorphism", *Computing Surveys*, Vol. 17:4, S. 471-522, 1985.

## Kurzer Einschub: Overloading und Coercion



Was bedeuten diese Ausdrücke?

- $3 + 5$
- $3.0 + 5$
- $3 + 5.0$
- $3.0 + 5.0$

Eine Möglichkeit: '+' ist **überladen** mit vier verschiedenen Operationen:

- int **plus1** (int, int)
- float **plus2** (float, int)
- float **plus3** (int, float)
- float **plus4** (float, float)

Oder: '+' ist **überladen** mit zwei verschiedenen Operationen:

- int **plus1** (int, int)
- float **plus2** (float, float)



In Java ist der Operator '+' weiter überladen; er ist auch bei Strings anwendbar.

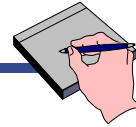
Und in gemischten Ausdrücken werden ints in floats **umgewandelt** (Coercion):

- $3.0 + 5 \rightarrow 3.0 + 5.0$
- $3 + 5.0 \rightarrow 3.0 + 5.0$

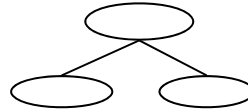
Oder: '+' ist **nur für floats definiert** und ints in Additionen werden **immer** in floats **umgewandelt**.

Anmerkung: Lediglich zur Verdeutlichung und Allgemeingültigkeit der Darstellung wird hier float als Typ verwendet. Für Java müsste korrekterweise float durch double ersetzt werden.

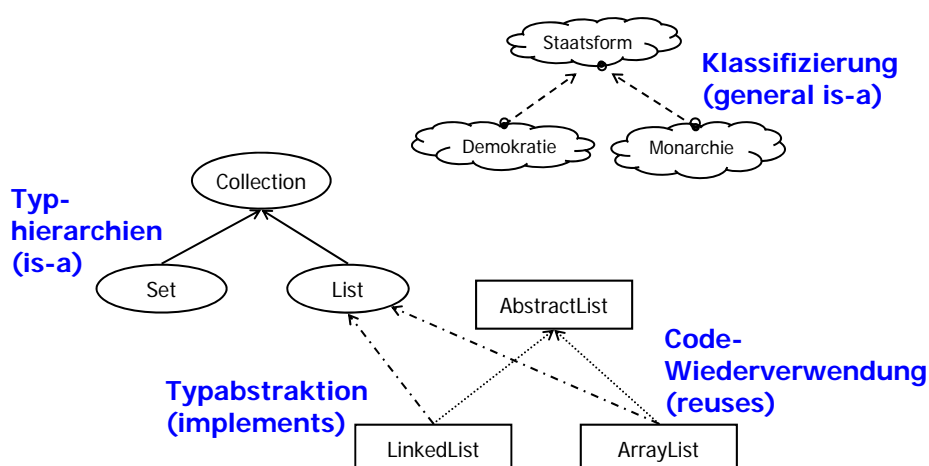
## Subtyp-Polymorphie



- **Inclusion Polymorphism** nach Cardelli u. Wegner
- Im Kontext objektorientierter Sprachen meist kurz **Polymorphie** genannt.
- Eng mit **Ersetzbarkeit** (engl. substitutability) verknüpft: Variable eines Supertyps kann auf Exemplare von Subtypen verweisen.
- Somit auch eng verknüpft mit der Unterscheidung von **statischem** und **dynamischem Typ** einer Referenz-Variablen.
- Erfordert **dynamisches Binden**!
- Zentrales **technisches Konzept** objektorientierter Sprachen
- Voraussetzung für Typhierarchien und Typabstraktion

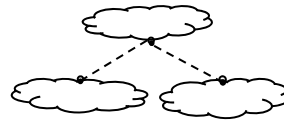


## Übersicht: Zentrale „Vererbungs“-konzepte



## Klassifizierung

- Allgemeines Verständnis von **Ist-ein-Beziehungen**
- Vgl. **Taxonomien** in der Biologie
- **Ontologien**, semantische Netze
- Beispiele:
  - ein Quadrat **ist ein** Rechteck.
  - ein Emu **ist ein** Vogel.
- Häufig im Zusammenhang mit Vererbung genannt; wird in ihrer allgemeinen Form jedoch **nicht** durch die Mechanismen **von Programmiersprachen unterstützt!**

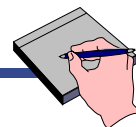


**Subclassing  $\neq$  Subtyping  $\neq$  Is-a**

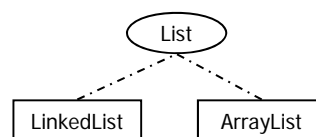
LaLonde u. Pugh, Journal of Object-oriented Programming, January 1991

- Wir gehen in dieser Veranstaltung nicht weiter auf Klassifizierung ein.

## Typabstraktion (bekannt aus SE1)



- Die Implementierung eines abstrakt, aber vollständig beschriebenen Typs (vgl. **abstrakter Datentyp**) kann auf unterschiedliche Weise erfolgen.
- Beispiel: Der Typ **List** kann mit einer **LinkedList** und mit einer **ArrayList** implementiert werden.
- Idealerweise wird für einen Klienten nur der Typ sichtbar, die **Implementation** ist **austauschbar**.
- Typischerweise definieren die Implementationen keine zusätzlichen Operationen.
- Unterschiede zeigen sich möglicherweise in der Effizienz (falls nicht Teil der Spezifikation). Je nach Benutzungsprofil wirken sich die Implementationen unterschiedlich aus.
- Setzt **dynamisches Binden** nur dann voraus, wenn mehrere Implementationen im gleichen Programm aktiv sein sollen!
- In Java über **Subtyp-Polymorphie** realisierbar.

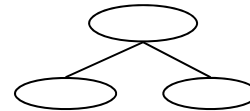




## Typhierarchien

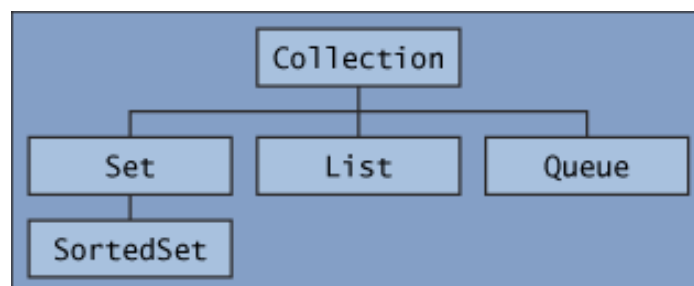


- Durch das Anordnen von **Typen** in einer hierarchischen Beziehung entstehen **Super-** und **Subtypen**.
- Ein **Subtyp** definiert mindestens alle Operationen seines Supertyps; typischerweise bietet ein Subtyp **weitere Operationen** an.
- Auch hier gilt **Ersetzbarkeit**: Ein Supertyp ist durch jeden seiner Subtypen ersetzbar.
- Das Bilden von Typhierarchien wird auch **Subtyping** genannt.
- Basiert auf **Subtyp-Polymorphie**.
- Wir gehen noch ausführlich auf Subtyping ein.

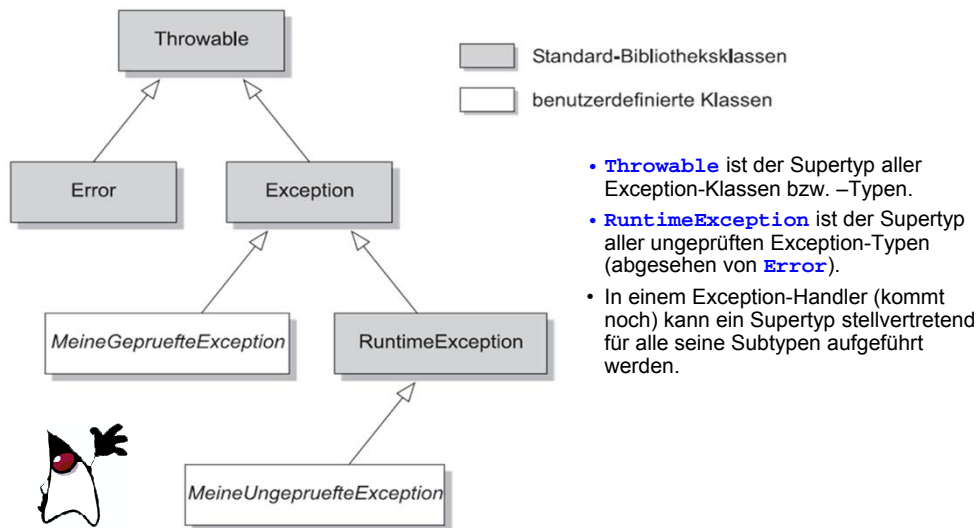


## Beispiel: Typ-Hierarchie im Java Collections Framework

- Der Typ **Collection** ist der Supertyp der Typen **Set**, **List** und **Queue** (und transitiv auch der Supertyp von **SortedSet**).
- An allen Stellen, an denen eine **Collection** erwartet wird, kann ein Exemplar eines der Subtypen eingesetzt werden.
- An allen Stellen, an denen ein **Set** erwartet wird, kann auch ein Exemplar von **SortedSet** eingesetzt werden.



### Beispiel: Javas Exception-Hierarchie

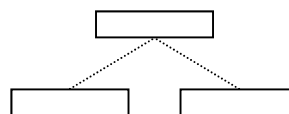


SE2 – OOPM – Teil 1

103

### Code-Wiederverwendung

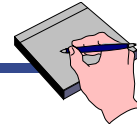
- Auch **Implementationsvererbung** (engl.: inheritance) oder **Subclassing** genannt.
- Eine **Subklasse** erbt die **Methoden** und **Felder** ihrer **Superklasse**.
- Der geerbte Code wird für spezielle Anforderungen angepasst, indem Methoden **definiert**, **überschrieben** oder **erweitert** werden.
- Theoretisch (und auch praktisch, wie etwa in der Sprache **Sather**) durch einfaches Kopieren von Quelltexten realisierbar (**statisch**).
- Meist jedoch ebenfalls über dynamisches Binden realisiert.
- Wir gehen in einer späteren Vorlesung noch ausführlich auf Code-Wiederverwendung ein.



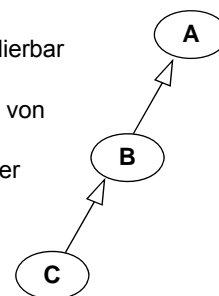
SE2 – OOPM – Teil 1

104

## Einige weitere Begriffe



- Wenn wir uns auf **Hierarchien von Typen** beziehen, werden wir im Folgenden die Begriffe **Super- und Subtyp** verwenden.
- Wenn wir uns auf **Hierarchien von Implementationen** (Klassen) beziehen, werden wir im Folgenden die Begriffe **Ober- und Unterklasse** verwenden.
- Subtyp- und Unterklassenbeziehungen sind unter anderem **transitiv**; eine Typ kann deshalb **mehrere Supertypen** haben und eine Klasse **mehrere Oberklassen**.
- Ist eine solche Beziehung zwischen zwei Typen/Klassen formulierbar ohne die Beteiligung weiterer, nennen wir sie **unmittelbar**.
  - Im Beispiel rechts ist C ein Subtyp von B und transitiv auch von A.
  - A ist jedoch nur der **unmittelbare Supertyp** von B und B der **unmittelbare Supertyp** von C.



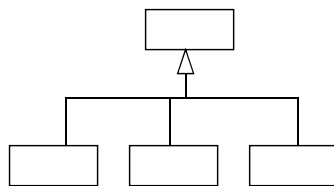
SE2 – OOPM – Teil 1

105

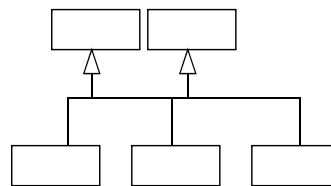
## Einfach- und Mehrfachvererbung



### Einfachvererbung



### Mehrfachvererbung



- Sind Vererbungshierarchien baumförmig, d.h. hat eine Klasse nur **eine unmittelbare Oberklasse** und beliebig viele Unterklassen, dann sprechen wir von **Einfachvererbung**.
- Hat eine Klasse mehr als eine unmittelbare Oberklasse, sprechen wir von **Mehrfachvererbung**.



Java erlaubt nur Einfachvererbung zwischen Klassen, bietet aber Mehrfachvererbung zwischen Interfaces.

SE2 – OOPM – Teil 1

106

## Mehrfachvererbung

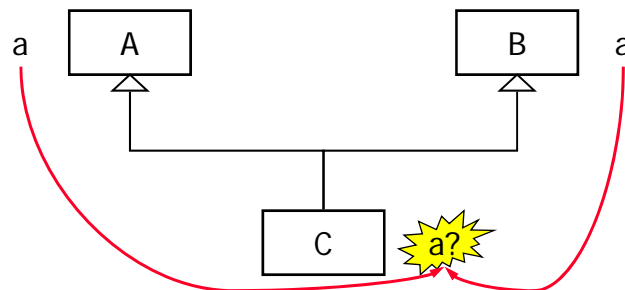
- Die Notwendigkeit von Mehrfachvererbung in Programmiersprachen wurde Anfang der 90er Jahre heiß in der Forschergemeinde diskutiert.
- Vorläufiges Ergebnis für neuere Sprachen (wie Java, C#):
  - **Mehrfach-Subtyping ist nützlich und gewünscht.**
  - **Mehrfach-Implementationsvererbung ist (eher) kompliziert.**
- Das größte Problem bei Mehrfachvererbung ist der Umgang mit **Namenskollisionen**.
- Allgemein unterscheidet man dabei:
  - **Vererbung verschiedener, aber gleichnamiger Merkmale**
  - **Vererbung eines Merkmals über verschiedene Wege (Diamant-Vererbung)**

SE2 – OOPM – Teil 1

107

## Vererbung verschiedener, aber gleichnamiger Merkmale

- Horizontale Namenskollision: verschiedene geerbte Eigenschaften wurden voneinander unabhängig mit gleichen Namen in Superklassen definiert.



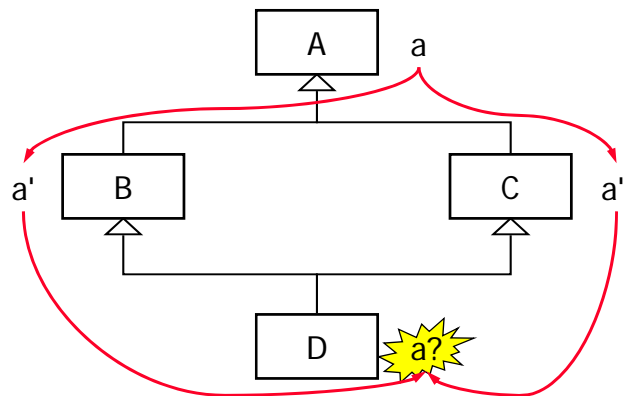
SE2 – OOPM – Teil 1

108

### Diamant-Vererbung



- Ein Merkmal wird über verschiedene Vererbungspfade geerbt. Auf jedem Vererbungspfad können Eigenschaften verändert worden sein.



SE2 – OOPM – Teil 1

109

### Vorläufige Zusammenfassung



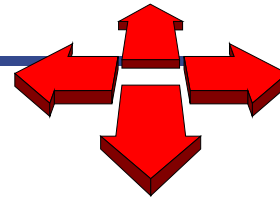
- Vererbung** ist eine zentrale Eigenschaft objektorientierter Programmiersprachen; **aber**: Vererbung ist auch einer der am stärksten missbrauchten und missverstandenen **Sprachmechanismen**.
- Vererbung als Begriff ist stark **überladen**; viele verschiedene **Konzepte** werden darunter zusammengefasst. Die wichtigsten sind:
  - Subtyp-Polymorphie** auf Typebene für das Formulieren von **Typhierarchien** (Subtyping) und für **Typabstraktion**.
  - Implementationsvererbung** für das hochflexible Kombinieren von ausführbaren Quelltext-Elementen (vor allem Methoden).
- SoftwaretechnikerInnen sollten diese sehr verschiedenen Konzepte klar voneinander trennen können; wir werden sie uns deshalb getrennt voneinander im Folgenden näher ansehen.
- In der Praxis treten diese beiden Konzepte meist gemeinsam auf; umso wichtiger ist ein klares Verständnis der Unterschiede.

SE2 – OOPM – Teil 1

110

## Übersicht Subtyping

- Fachliche Typhierarchien
- Technische Eigenschaften des Subtyping
- Ko- und Kontravarianz
- Subtyping und Zusicherungen



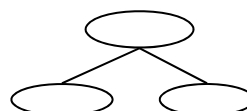
SE2 – OOPM – Teil 1

111

## Allgemeines

- Beim Subtyping werden **Typen** hierarchisch miteinander in Beziehung gesetzt.
- Eine Subtyp-Beziehung sollte eine **ist-ein-Beziehung** ausdrücken (aber nicht jede ist-ein-Beziehung ist eine Subtyp-Beziehung).
- Nur eine genaue Kenntnis der technischen Grundlagen von Subtyping versetzt uns in die Lage, **fachliche Hierarchien** in einem Anwendungsbereich geeignet in **Typ-Hierarchien** abzubilden.

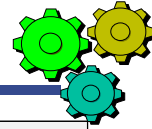
**Fokus beim Subtyping:**  
**Ersetzbarkeit!**



SE2 – OOPM – Teil 1

112

## Entwurf von Typhierarchien: Erkennen gleichartiger Umgangsformen ...



**Modellierung** der Umgangsformen  
mit Materialien in einem Bürokontext:

### Stapel

Stapel drauflegen  
Dokument einfügen  
Dokument auswählen  
Dokument entnehmen  
auflösen  
ausbreiten

### Ordner

-mit Text beschriften  
-Inhaltsverzeichnis führen  
-Dokument einfügen  
-Dokument auswählen  
-Dokument entnehmen  
-Zwischenblätter einlegen

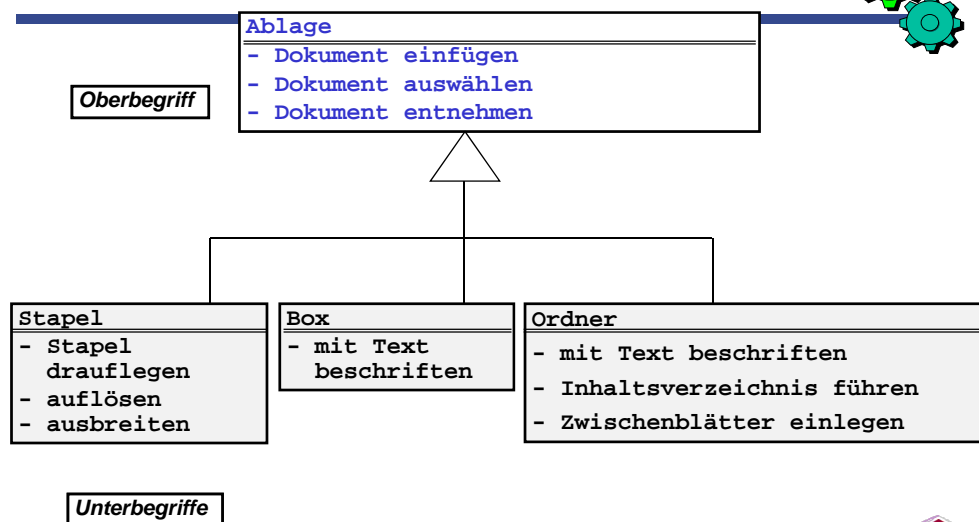
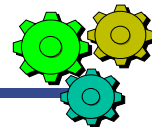
### Ablage

Dokument einfügen  
Dokument auswählen  
Dokument entnehmen

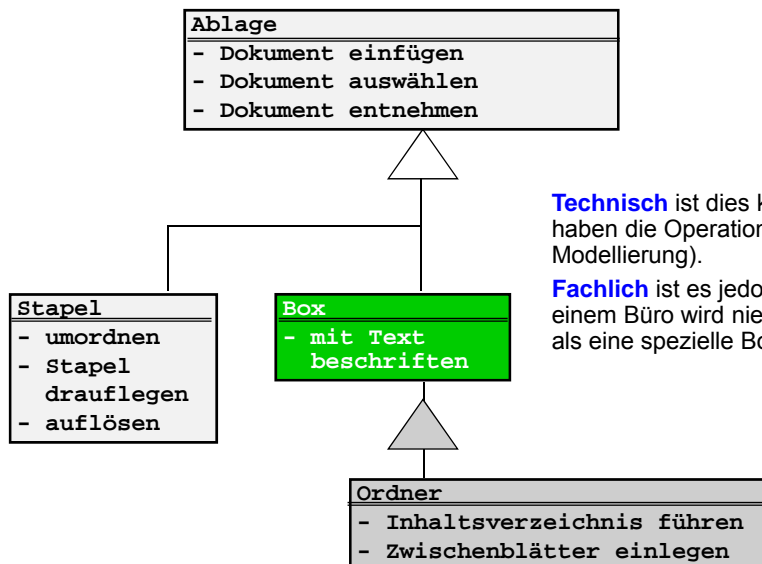
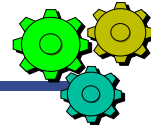
### Box

Dokument einfügen  
Dokument auswählen  
Dokument entnehmen  
mit Text beschriften

## ... als Grundlage einer Typhierarchie



### Eine schlechte Verwendung von Subtyping: "Box" als Supertyp von "Ordner"



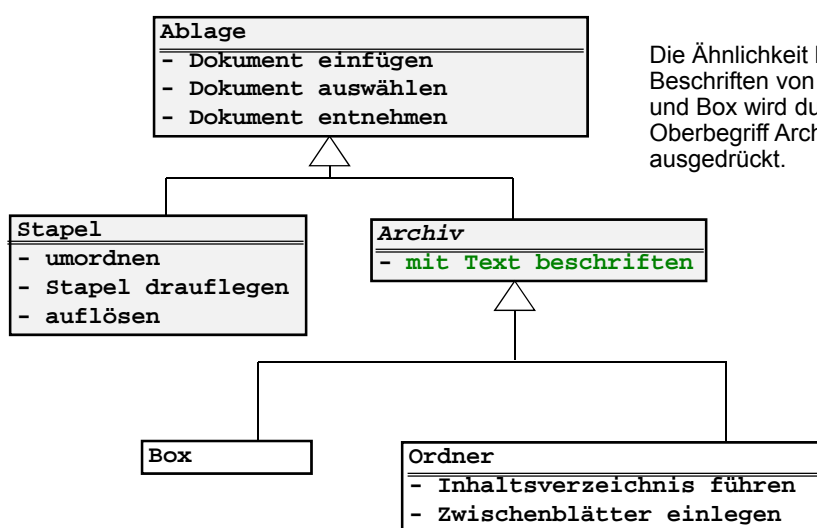
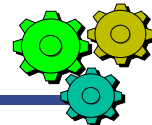
**Technisch** ist dies korrekt (alle Typen haben die Operationen aus der Modellierung).

**Fachlich** ist es jedoch **unsauber**: In einem Büro wird niemand einen Ordner als eine spezielle Box ansehen.

SE2 – OOPM – Teil 1

115

### Subtyping soll spezielle Klassifikation ausdrücken ("verstehen als", "is-a")

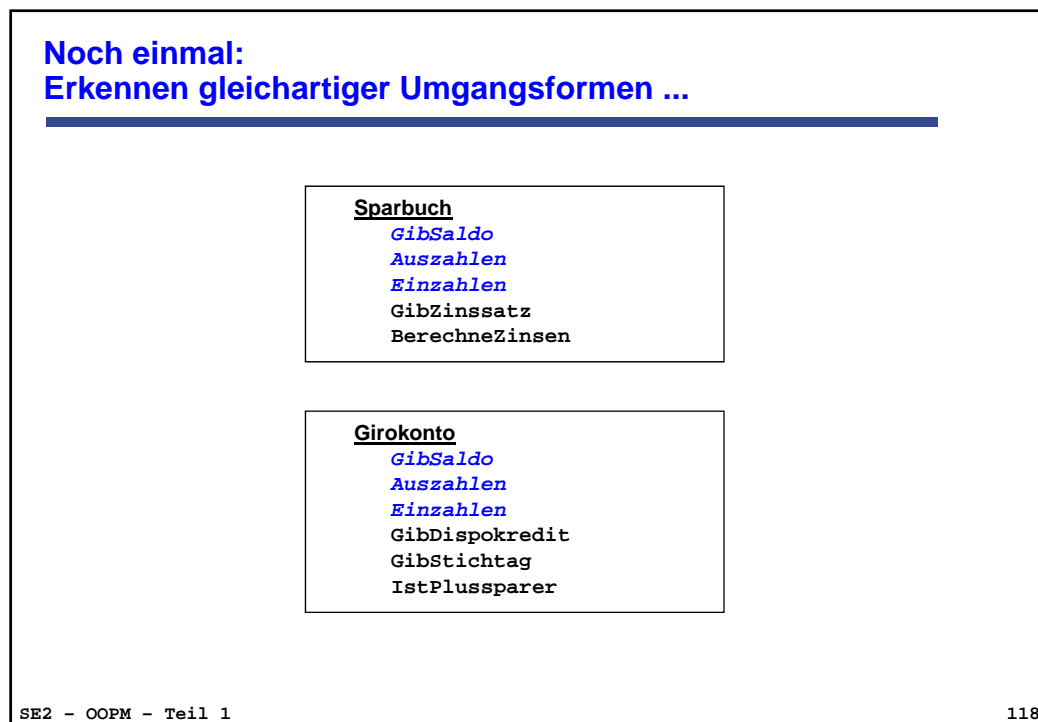
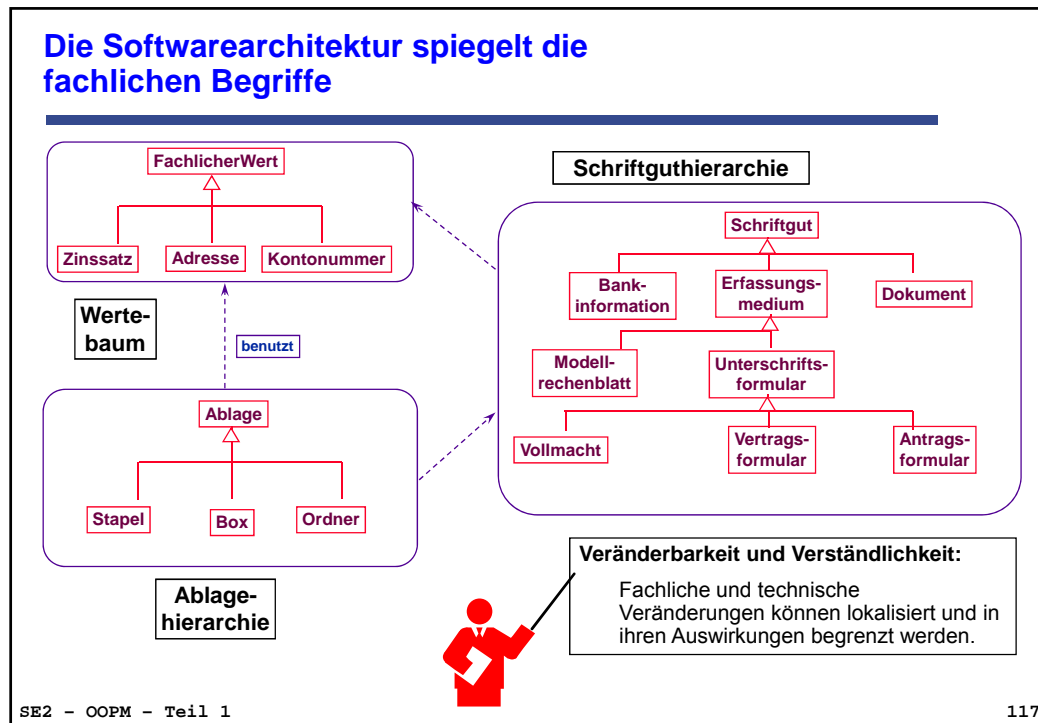


Die Ähnlichkeit beim Beschriften von Ordner und Box wird durch den Oberbegriff Archiv ausgedrückt.

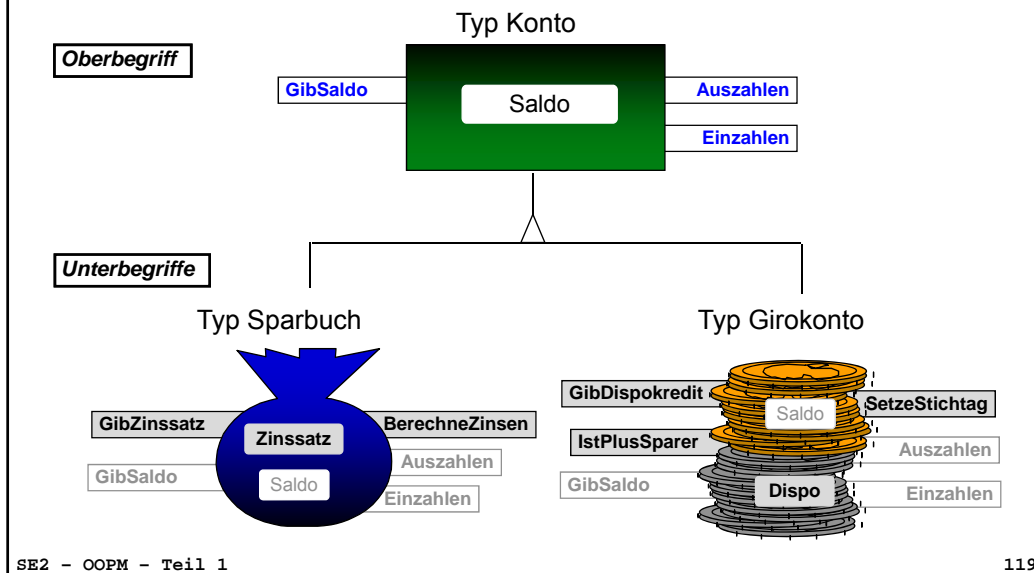
SE2 – OOPM – Teil 1

116





### ... als Grundlage einer Typhierarchie



SE2 – OOPM – Teil 1

119

### Subtyping in Java

- In Java besteht eine **Subtyp-Beziehung**...
  - zwischen **Interfaces**, die über die **extends**-Beziehung verknüpft sind;
  - zwischen einer **Klasse** und einem **Interface**, die über eine **implements**-Beziehung verknüpft sind (denn eine Klasse definiert in Java immer auch einen Typ);
  - zwischen **Klassen**, die miteinander über eine **extends**-Beziehung verknüpft sind (denn Klassen definieren in Java immer Typen).



Das Schlüsselwort **extends** ist in Java überladen: Es wird sowohl für Subtyping zwischen Interfaces verwendet als auch für Subtyping und Subclassing zwischen Klassen.

SE2 – OOPM – Teil 1

120

## Interfaces und Subtyping

- Ein **Interface** in Java ist eine **reine Typbeschreibung** und ist deshalb immer Teil einer Subtyp-Beziehung.
- Die häufigsten Verwendungen von Interfaces sind...
  - für **Typabstraktion**: Ein Interface beschreibt einen abstrakten Datentyp, der von mehreren Klassen implementiert werden kann. Der Name des Interfaces ist meist ein **Substantiv** (**Set**, **List**, **Map**, etc.).
  - für **adjektivische Abstraktionen**: Ein Interface beschreibt nur eine Teilfunktionalität/Rolle, deren Operationen von implementierenden Klassen neben anderen Operationen angeboten werden. Der Name des Interfaces ist meist ein **Adjektiv** (**Comparable**, **Iterable**, etc.).
  - für **reine Typhierarchien**: Mehrere Interfaces beschreiben Abstraktionen, die in einer hierarchischen Beziehung zueinander stehen; siehe etwa die Interfaces im Java Collection Framework (**Collection** als Supertyp von **List** und **Set** etc.).
- Eine weniger übliche Verwendung stellen die so genannten **Marker-Interfaces** dar: Ein Marker-Interface definiert keine Operationen, sondern dient nur zur Typprüfung. Beispiele sind **java.lang.Cloneable** und **java.io.Serializable**.

## Redefinieren vs. Redeklamieren



- **Redefinieren** ist das Ändern der Implementation einer Operation in einer Subklasse (später mehr dazu beim Thema Implementationsvererbung).
  - **eine andere Methode für dieselbe Operation**
- **Redeklamieren** ist das Ändern der Signatur einer Operation in einem Subtyp bzw. einer Subklasse.
  - **Änderung der Schnittstelle**

## Grenzen beim Redeclarieren

- Aus Gründen der Typsicherheit sollte die **Ersetzbarkeit** erhalten bleiben.
- In statisch typsicheren Sprachen ist ein Redeclarieren deshalb nur in sehr begrenztem Maße möglich.
- Der Name\* einer Operation und die Anzahl (und Reihenfolge) der Parameter müssen gleich bleiben.
- Einzig möglich: **Typänderungen** für Parameter und Ergebnisse

\*Eiffel erlaubt auch das Umbenennen von Methoden beim Redeclarieren. Dies trägt nicht zum besseren Verständnis des Quelltextes bei.

## Formale Definition von Subtyping

Der **Typ einer Operation** kann geschrieben werden als  $Op(S):T$  für eine Operation, die einen Parameter vom Typ  $S$  bekommt und ein Ergebnis vom Typ  $T$  liefert. Der Typ einer weiteren Operation,  $Op(R):U$ , ist ein **Subtyp** der ersten Operation, wenn  $S$  ein Subtyp von  $R$  ist und  $U$  ein Subtyp von  $T$ :

$$(Op(R): U) \prec (Op(S): T), \text{ wenn } S \prec R \text{ und } U \prec T$$

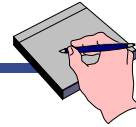
Diese Definition gilt analog für Operationen mit  $n$  Parametern ( $n \geq 0$ ) und mit  $m$  Ergebnistypen ( $m \geq 0$ ).

Ein **Objekttyp** definiert eine Menge von  $n$  Operationen  $m_i$  mit Typ  $T_i$  für  $1 \leq i \leq n$ , verkürzt geschrieben als  $ObjektTyp\{m_i: T_i\}_{1 \leq i \leq n}$ .

Ein Objekttyp ist ein **Subtyp** eines anderen Objekttyps, wenn er mindestens die Operationen des anderen Typs definiert und jede dieser Operationen ein Subtyp des Operationstyps des anderen Objekttyps ist:

$$ObjektTyp\{m_j: S_j\}_{1 \leq j \leq m} \prec ObjektTyp\{m_i: T_i\}_{1 \leq i \leq n}, \\ \text{wenn } n \leq m \text{ und für alle } i \leq n \text{ ist } S_i \prec T_i$$

## Ko- und Kontravarianz



- Die formale Definition von Subtyping ist ein Ergebnis der **objektorientierten Typtheorie**.
- Alle Theorie ist grau; aus der Definition lassen sich jedoch wichtige und praktisch relevante Aussagen für die statische Typsicherheit ableiten:
  - **Ergebnistypen** können **kovariant** (d.h. der Spezialisierungsrichtung folgend) angepasst werden.
  - **Typen von Parametern** können **kontravariant** (d.h. entgegen der Spezialisierungsrichtung) angepasst werden.
- Insbesondere heißt dies auch, dass **kovariante** Anpassungen von **Parametertypen** nicht statisch typsicher sind; entsprechende Gegenbeispiele lassen sich leicht konstruieren.
  - An diesem Umstand scheitert beispielsweise die statische Typsicherheit in **Eiffel**!

## Zulässig: Kovarianz für Ergebnistypen

```
interface Person {
    public Person clone();
}
```

Redeclaration, da  
Änderung der Signatur.

```
interface Student extends Person {
    public Student clone();
}
```

Es bleibt **dieselbe** Operation.

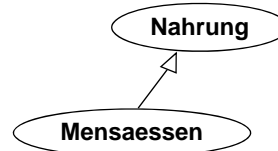
In Java bis 1.4 nicht zugelassen, obwohl die VM es schon immer unterstützt hat.  
**Seit Java 1.5 zugelassen!**

### Zulässig: Kontravarianz für Parametertypen

```
class Nahrung { ... }
class Mensaessen extends Nahrung { ... }
```

```
interface Person {
    void verzehre(Mensaessen param);
}
```

```
interface Student extends Person {
    void verzehre(Nahrung param);
}
```



**Redeclaration**, da  
Änderung der Signatur  
(kein Overloading wie  
in Java).

theoretisch interessant,  
praktisch irrelevant.

### Nicht typischer: Kovarianz für Parametertypen

```
class Nahrung { ... }
class DiabetischeNahrung extends Nahrung { ... }
```

```
interface Person {
    void verzehre(Nahrung param);
}
```

```
interface Diabetiker extends Person {
    void verzehre(DiabetischeNahrung param);
}
```



**Kovariant**, da  
Redeclaration **mit** der  
Vererbungsrichtung.

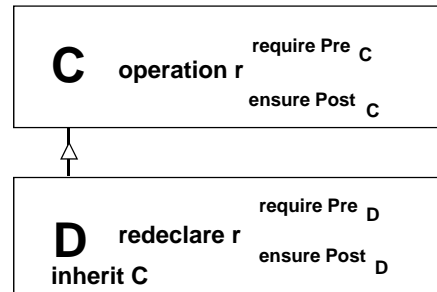
#### Benutzung:

```
Person p = new DiabetikerImpl();
Nahrung nahrung = new Nahrung();
p.verzehre(nahrung);
```

## Zusicherungen und Subtyping: Das Prinzip

Metapher:

- **Verträge** können an **Zulieferer** weitergegeben werden.
- Ein Zulieferer darf **höchstens** die Voraussetzungen fordern, die der Anbieter gefordert hat.
- Ein Zulieferer muss **mindestens** die Leistung erbringen, die der Anbieter versprochen hat.



Bei der Redeklaration einer Operation muss die Unterklasse den Vertrag von der Oberklasse übernehmen. Es muss gelten:

**Pre<sub>D</sub> schwächer als Pre<sub>C</sub> :**  $\text{Pre}_C \Rightarrow \text{Pre}_D$

**Post<sub>D</sub> stärker als Post<sub>C</sub> :**  $\text{Post}_D \Rightarrow \text{Post}_C$

Die Prüfung der Vertragsübernahme ist ein i.a. unentscheidbares Problem.

SE2 – OOPM – Teil 1

129

## Zusicherungen und Subtyping: Ein Eiffel-Beispiel

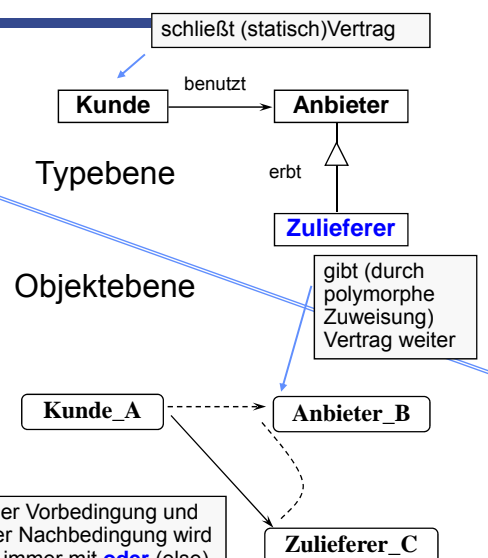
```

Class Anbieter
feature
  Dienstleistung (x : INTEGER) is
    require x > 5
    do
    ...
    ensure x > 10
    end;
end
  
```

```

Class Zulieferer
inherit Anbieter redefine Dienstleistung
feature
  Dienstleistung (x : INTEGER) is
    require else x > 3
    do
    ...
    ensure then x > 12
    end;
end
end
  
```

Ein Verschärfen der Vorbedingung und ein Aufweichen der Nachbedingung wird verhindert, indem immer mit **oder** (else) bzw. mit **und** (then) verknüpft wird.



SE2 – OOPM – Teil 1

130

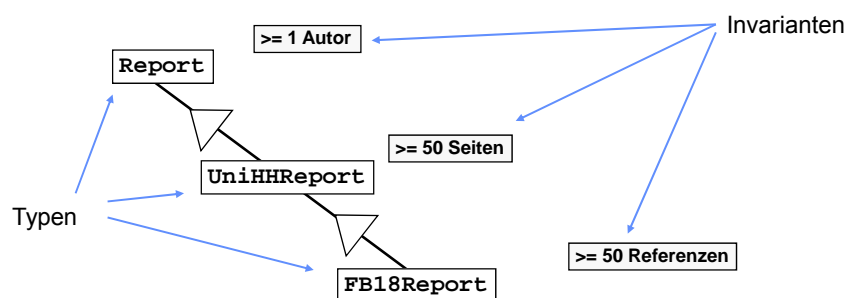
## Zusicherungen und Subtyping in Programmiersprachen

- Da ein Compiler nicht überprüfen kann, ob die Regeln für Subtyping und Zusicherungen eingehalten werden, wurden in der Sprache **Eiffel** defensive Regeln für das Vertragsmodell festgelegt:
- **Vor- und Nachbedingungen:**
  - Eine Vorbedingung kann in Subtypen nur durch eine **oder-Verknüpfung** erweitert werden - die Bedingung wird höchstens abgeschwächt.
  - Eine Nachbedingung kann in Subtypen nur durch eine **und-Verknüpfung** erweitert werden - die Bedingung wird höchstens verschärft.
- **Invarianten:**
  - Eine Invariante kann in Subtypen nur durch eine **und-Verknüpfung** ergänzt werden - die Bedingung wird höchstens verschärft.

SE2 – OOPM – Teil 1

131

## Invarianten und Subtyping



- Die (Gesamt-) Invariante des Typs **FB18Report** ist **>= 1 Autor und >= 50 Seiten und >= 50 Referenzen**



- **Invarianten** werden mitvererbt („ge-undet“), d.h. entlang der Vererbungsrelation verschärft.
- Die Randbedingungen für **Verträge** werden somit anspruchsvoller.

SE2 – OOPM – Teil 1

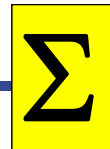
132



## Ko- und Kontravarianz und Zusicherungen

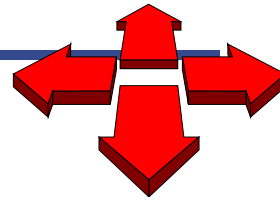
- Den Regeln für Ko- und Kontravarianz und denen für Subtyping und Zusicherungen liegt **dasselbe Prinzip** zugrunde:
  - Ein Klient sollte sich auf die statisch vereinbarten Verträge (Signatur und Zusicherungen einer Operation) verlassen können.
  - Wenn aufgrund von Subtyp-Polymorphie ein Exemplar eines Subtyps „hinter“ einer Variablen steckt, dann sollte sich dieses Exemplar stets so verhalten, wie der statische Typ es verspricht.
- Eine **kovariante Parametertypanpassung** widerspricht diesem Prinzip ebenso wie die **Verschärfung einer Vorbedingung**:
  - Das gerufene Exemplare akzeptiert in beiden Fällen nur einen kleineren Wertebereich;
  - der Klient liefert potenziell etwas aus dem größeren Wertebereich;
  - es kann deshalb, trotz pflichtbewusstem Klienten, zur Laufzeit zu Fehlern kommen!

## Zusammenfassung Polymorphie und Vererbung



- **Vererbung** ist eine zentrale Eigenschaft objektorientierter Programmiersprachen; **aber**: Vererbung ist auch einer der am stärksten missbrauchten und missverstandenen **Sprachmechanismen**.
- Ein zentrales Vererbungskonzept auf Typebene ist (**Subtyp-**) **Polymorphie** für das Formulieren von **Typhierarchien** (Subtyping) und für **Typabstraktion**.
  - Grundidee dabei immer: Aus Sicht eines Klienten können sich hinter einer Schnittstelle verschiedenartige (polymorphe) Typen und Implementationen verbergen, von deren Unterschieden auf Ebene der Benutzung bewusst abstrahiert werden soll.
- **Implementationsvererbung** für das Kombinieren von ausführbaren Quelltext-Elementen (vor allem Methoden) werden wir in einer eigenen Vorlesung ausführlich betrachten.

## Übersicht Implementationsvererbung

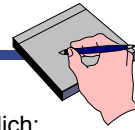


- Motivation
- Erben und Erweitern
- Erben und Anpassen
- Abstrakte Methoden und abstrakte Klassen
- Selbstaufrufe
- Vererbung und Konstruktoren
- Die Erbenschnittstelle

## Motivation: Wiederverwendung

- **Wiederverwendung** von bereits entwickelter Software ist ein wichtiger Aspekt effizienter Softwareentwicklung.
  - „**Das Rad nicht jedesmal neu erfinden.**“
- In objektorientierten Systemen gibt es zwei grundsätzliche Formen von Wiederverwendung:
  - Einfache Benutzung – „**Use**“
    - systematisch in Software-Bibliotheken
  - Implementationsvererbung – „**Reuse**“
    - Erben zum Anpassen
    - Erben zum Vermeiden von Redundanz
    - Erben zum Vervollständigen
- **Implementations-** oder **Code-Vererbung** (engl.: inheritance) ist das Organisieren von **ausführbaren** Software-Elementen in Hierarchien.

## Generalisierung und Spezialisierung



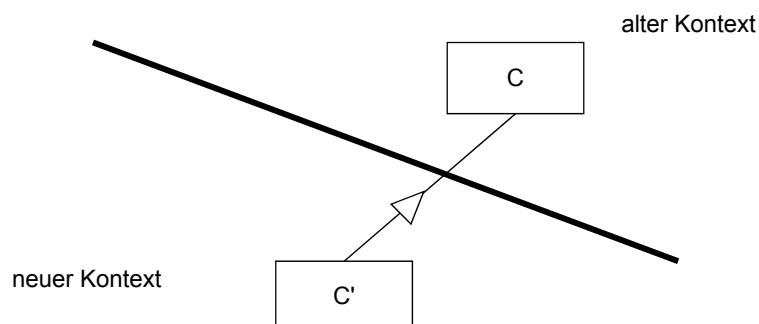
- Zwei prinzipielle Sichtweisen auf Implementationshierarchien sind möglich:
  - **Generalisierung** (bottom-up):
    - Gleiche Eigenschaften mehrerer Klassen lassen sich generalisieren und als eine eigene Software-Einheit formulieren.
    - Gemeinsamkeiten werden zusammengefasst.
    - Die Oberklasse wird aus bereits existierenden Klassen „herausgezogen“.
    - Dient dem Vermeiden von Redundanz.
  - **Spezialisierung** (top-down):
    - Neue Unterklassen werden als Spezialisierungen von bestehenden Klassen formuliert.
    - Als Unterklassen oder abgeleitete Klassen verfeinern/erweitern sie bereits existierende Konzepte.
    - Wiederverwendung im Sinne der Anpassung für andere/neue Kontexte.

SE2 – OOPM – Teil 1

137

## Erben zum Anpassen

- In einen neuen Kontext passt eine bestehende Klasse nicht vollständig hinein, kann aber mit kleineren **Modifikationen** auf die neuen Anforderungen zugeschnitten werden.
- Meist eine 1-zu-1-Beziehung

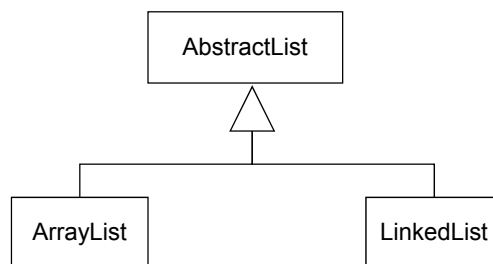


SE2 – OOPM – Teil 1

138

## Erben zum Vermeiden von Redundanz

- Statt gleiche oder ähnliche Teile eines Systems redundant (an mehreren Stellen) zu realisieren, sollen **Gemeinsamkeiten in Oberklassen** zusammengefasst werden.
- Typischerweise eine 1-zu-n-Beziehung
- Beispiel: Java Collections Framework



SE2 – OOPM – Teil 1

139

## Erben zum Vervollständigen

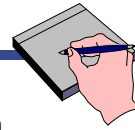
- Der überwiegende Teil eines gewünschten Verhaltens wurde in einer Reihe von Klassen festgelegt, mit Unterklassen sollen offene Stellen ergänzt werden (engl. auch **code injection**); der Weg zu **Rahmenwerken** (engl.: **frameworks**).
- Typisches Beispiel: Grafische Benutzeroberflächen



SE2 – OOPM – Teil 1

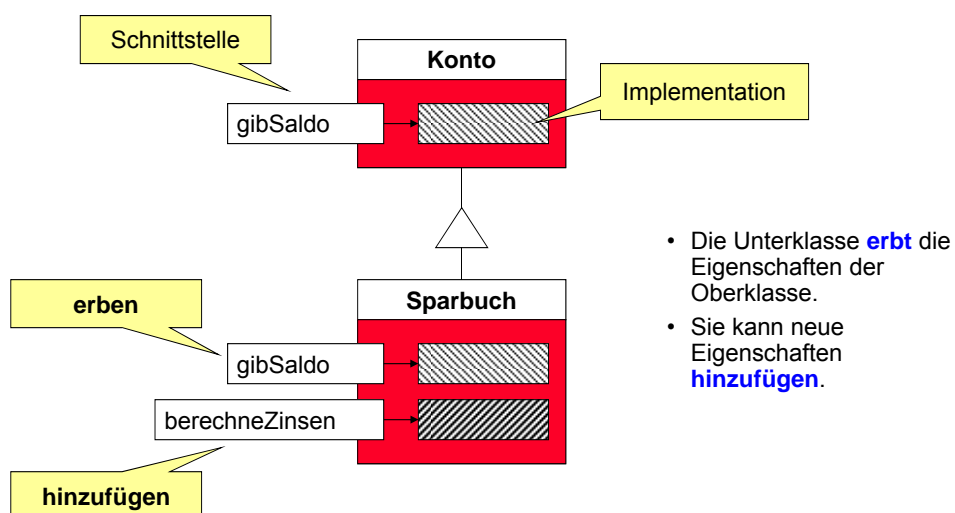
140

## Übersicht: Umgang mit geerbten Eigenschaften



- **Ober- und Unterklasse:** Alle Eigenschaften der Oberklasse sind durch Vererbung zunächst auch Eigenschaften einer Unterklasse; sie bilden einen festen Bestandteil der Unterklasse.
  - Die geerbten **Felder** der Oberklasse werden **vollständig** und (üblicherweise) **unverändert** übernommen; weitere können in Unterklassen hinzugefügt werden.
  - Die geerbten **Methoden** können in einer Unterklasse spezialisiert werden; dabei werden **Operationen**
    - **definiert** (durch eine Methode implementiert), für die in der Oberklasse keine Methode angegeben war;
    - **redefiniert**, wenn eine neue Methode in der Unterklasse eine gleichnamige Methode einer Oberklasse ersetzt (**überschreibt** oder **erweitert**);
    - **hinzugefügt**, wenn noch keine signaturgleiche Operation in einem Supertyp existiert.

## Erben und hinzufügen



- Die Unterklasse **erbt** die Eigenschaften der Oberklasse.
- Sie kann neue Eigenschaften **hinzufügen**.

## Vererbung zwischen Klassen in Java

- In Java geschieht die Implementationsvererbung ausschließlich über Beziehungen zwischen Klassen, die mit **extends** formuliert werden.
- Vererbungsbeziehung herstellen:
  - Sparbuch** erbt alle Operationen und Felder der Klasse **Konto** durch das Schlüsselwort **extends**.
  - Sparbuch** ergänzt die geerbten Operationen um zwei neue: **gibZinssatz** und **berechneZinsen**.

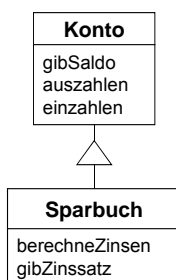
```
class Konto {
    public Konto() {...}
    public void einzahlen(float b) {...}
    public void auszahlen(float b) {...}
    public float gibSaldo() {...}
}
```



```
class Sparbuch extends Konto {
    public Sparbuch(Zinssatz z) {...}
    public Zinssatz gibZinssatz() {...}
    public void berechneZinsen() {...}
}
```

## Verwendung von Geerbtem

- Benutzung von geerbten Operationen:
  - Wird ein Objekt der Klasse **Sparbuch** erzeugt, können neben allen Operationen, die in der Klasse **Sparbuch** definiert werden, **auch alle Operationen, die die Klasse **Konto** anbietet**, verwendet werden.
  - Ein Exemplar der Klasse **Sparbuch** **enthält alle Zustandsfelder** von **Sparbuch** und **Konto**.



```
Sparbuch einSparbuch = new Sparbuch(3);

einSparbuch.einzahlen(1000);
einSparbuch.berechneZinsen();

if ( einSparbuch.gibSaldo() > 500 ) {
    einSparbuch.auszahlen(500);
    System.out.print("Der neue Saldo beträgt ");
    System.out.println(einSparbuch.gibSaldo());
}
```

## Private Eigenschaften und Vererbung

- Private Eigenschaften (Methoden und Felder) werden zwar mitvererbt, sie sind in einer Unterklasse jedoch nicht zugreifbar.
- Im Beispiel:
  - Damit das Feld `_saldo` in der Klasse `Sparbuch` zugreifbar ist, deklarieren wir es nicht `private`, sondern nur `protected`.
  - Es ist dann für erbende Klassen zugreifbar, aber nicht für „normale“ Klienten.
  - `Sparbuch` erweitert die geerbten Felder um das Feld `_zinssatz`.

```
class Konto {
    protected float _saldo;

    public Konto() {...}
    public void einzahlen(float b) {...}
    public void auszahlen(float b) {...}
    public float gibSaldo() {...}
}
```



```
class Sparbuch extends Konto {
    protected float _zinssatz;

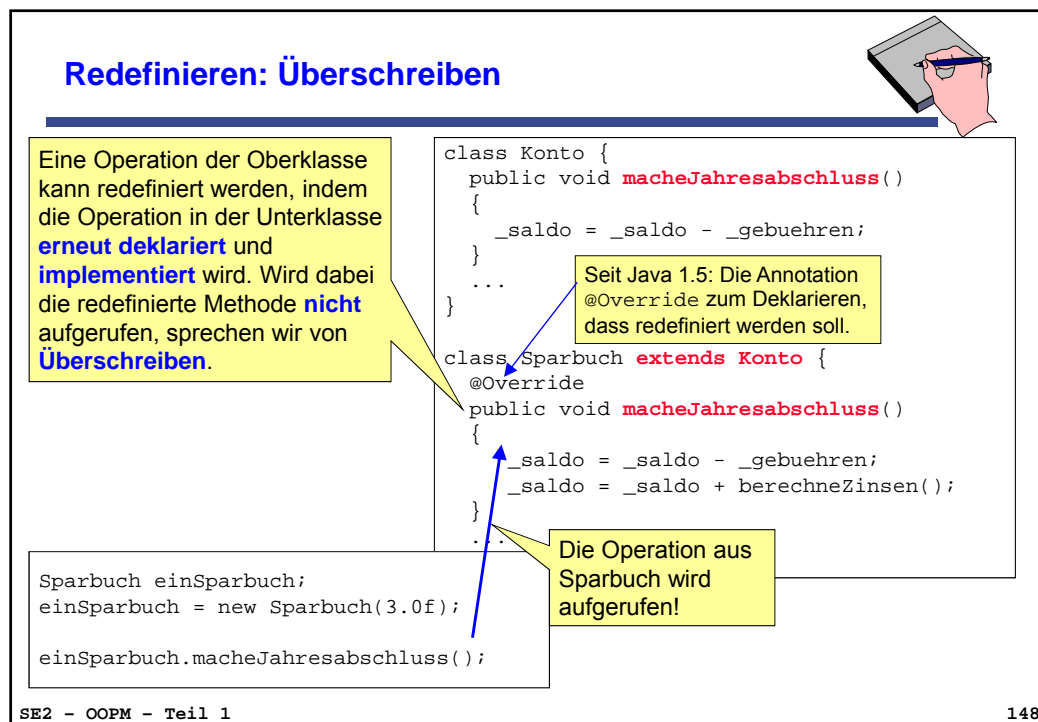
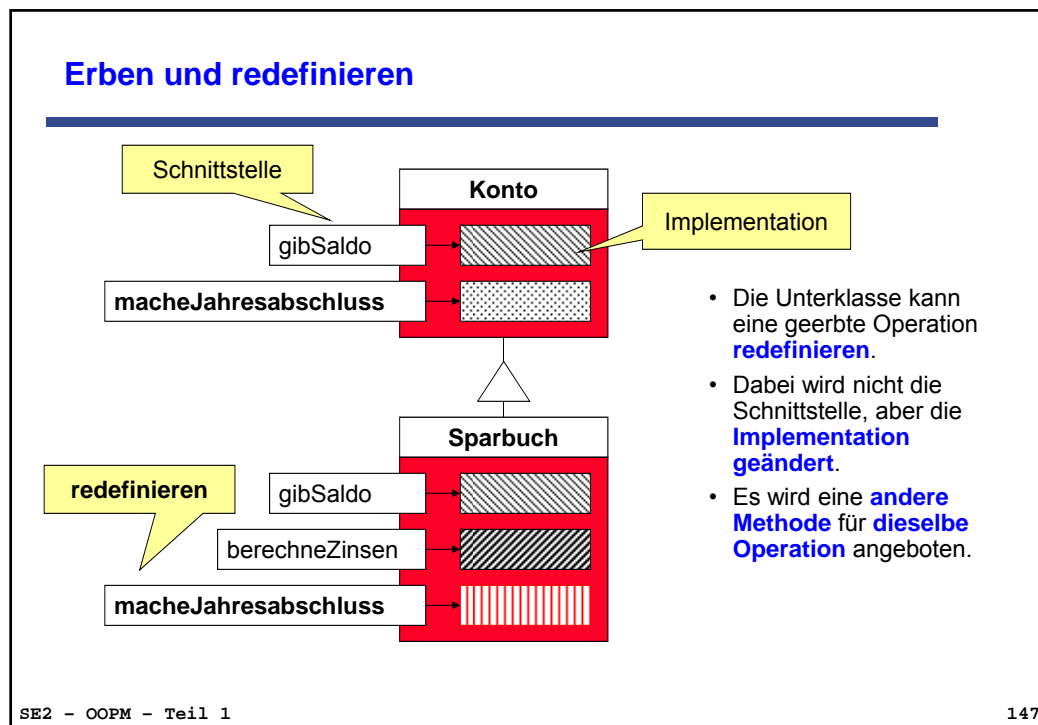
    public Sparbuch(Zinssatz z) {...}
    public Zinssatz gibZinssatz() {...}
    public void berechneZinsen() {...}
}
```

## Klasse Object als Oberklasse

- Und wenn ich nicht erbe?
  - Jede Klasse, die nicht explizit (durch eine `extends`-Klausel) von einer anderen Klasse erbt, **erbt implizit die Klasse Object**, die Teil des Java-Sprachkerns ist.
  - Die Klasse **Object** ist damit direkte oder indirekte **Oberklasse aller anderen Klassen**.
    - Der **Typ Object** definiert somit **Operationen**, die aufgrund dieser impliziten Vererbungsbeziehung **jeder Typ** hat.

```
public String toString() // Darstellung als String
public boolean equals(Object other) // Vergleich
public int hashCode()    // Hashcode berechnen
...
```

- Die **Klasse Object** implementiert diese Operationen durch ihre Methoden, die in erbenden Klassen verändert werden können.





## Redefinieren: Erweitern (mit super-Aufruf)



- Aufruf von Methoden der Oberklasse:
  - Mit dem **Schlüsselwort `super`** wird bei der Redefinition einer Operation die Methode der Oberklasse gerufen; auf diese Weise wird die Methode nicht komplett überschrieben, sondern **erweitert**.
  - Einen „**`super.super`**-Aufruf“ gibt es in Java nicht, deshalb kann man **nur die Implementation der direkten Oberklasse** erreichen!

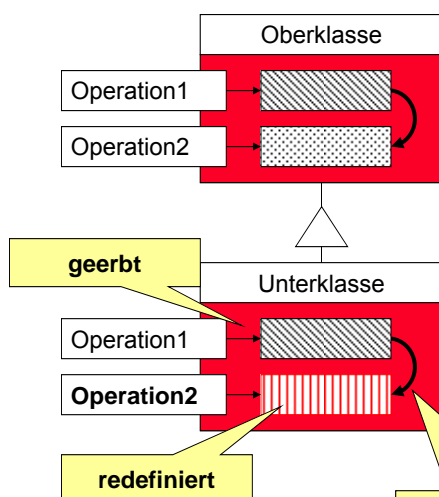
```
class Konto {
    public void macheJahresabschluss()
    {
        _saldo = _saldo - _gebuehren;
    }
    ...
}

class Sparbuch extends Konto {
    @Override
    public void macheJahresabschluss()
    {
        super.macheJahresabschluss();
        _saldo = _saldo + berechneZinsen();
    }
    ...
}
```

SE2 – OOPM – Teil 1

149

## Aufrufe redefinierter Operationen

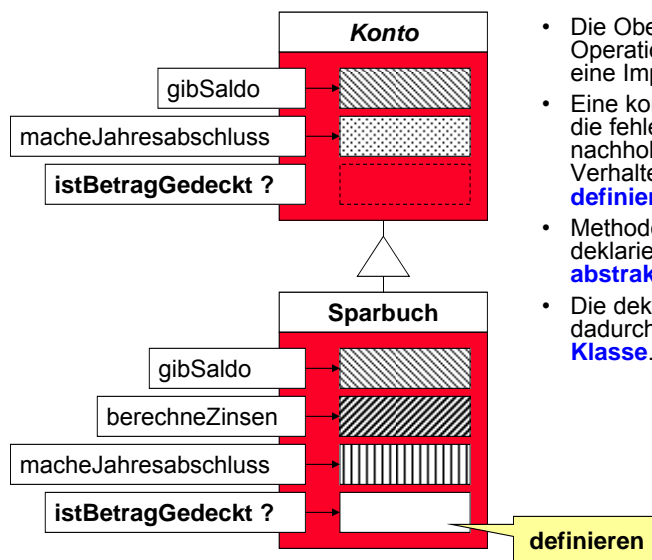


- Innerhalb einer Klassendefinition können sich Methoden **gegenseitig aufrufen** („Selbstaufrufe“ bzw. **self-Aufrufe**).
- Wenn in einer Unterklasse eine Operation redefiniert wird, die in der Oberklasse durch eine andere Methode aufgerufen wird, dann wird in der Unterklasse **die neue Implementation dieser Operation** aufgerufen.
- Hier passiert etwas sehr Wichtiges: Alter Code (der Oberklasse) ruft auf diese Weise neuen Code auf!** Bei klassischer Wiederverwendung mit Bibliotheken ist dies anders herum.

SE2 – OOPM – Teil 1

150

## Abstrakte Methoden: Deklarieren ohne Definieren



- Die Oberklasse kann eine Operation **deklarieren**, ohne eine Implementation anzugeben.
- Eine konkrete Unterklasse muss die fehlende Implementation nachholen, indem sie das Verhalten in einer Methode **definiert**.
- Methoden, die Operationen nur deklarieren, nennt man **abstrakte Methoden**.
- Die deklarierende Klasse wird dadurch zu einer **abstrakten Klasse**.

SE2 – OOPM – Teil 1

151

## Abstrakte Klassen

- Szenario:
  - Die Klasse **Konto** deklariert eine Operation **IstBetragGedeckt**, die sie **nicht implementiert**. Ob ein auszahlender Betrag gedeckt ist oder nicht, hängt von der jeweiligen Kontoart ab und kann daher nur **in den konkreten** abgeleiteten Klassen **Sparbuch** und **Girokonto** **sinnvoll ausimplementiert** werden.
- **Abstrakte Methoden** sind in Java mit dem Schlüsselwort **abstract** gekennzeichnet.
- Eine **Klasse**, die eine abstrakte Methode enthält, muss ebenfalls mit dem Schlüsselwort **abstract** deklariert werden.

```

abstract class Konto
{
    public abstract boolean istBetragGedeckt( float b );
    ...
}
  
```

SE2 – OOPM – Teil 1

152

## Abstrakte Methoden und ihre Definition

```
abstract class Konto
{
    protected float _saldo;
    ...
    public abstract boolean istBetragGedeckt(float b);
}
```

Methoden, die in einer Klasse nur eine Operation **deklarieren**, sie aber nicht implementieren, werden mit dem Schlüsselwort **abstract** markiert.



```
class Sparbuch extends Konto
{
    ...
    @Override
    public boolean istBetragGedeckt(float b)
    {
        return (_saldo >= b);
    }
}
```

Die Klasse Sparbuch **definiert** die in Konto zunächst nur deklarierte Operation **istBetragGedeckt**, indem sie sie mit einer Methode **implementiert**.

Da die Exemplarvariable **\_saldo** in der Oberklasse **protected** deklariert ist, kann in allen abgeleiteten Klassen, also auch in Sparbuch, **direkt darauf zugegriffen** werden.

## Eigenschaften abstrakter Klassen

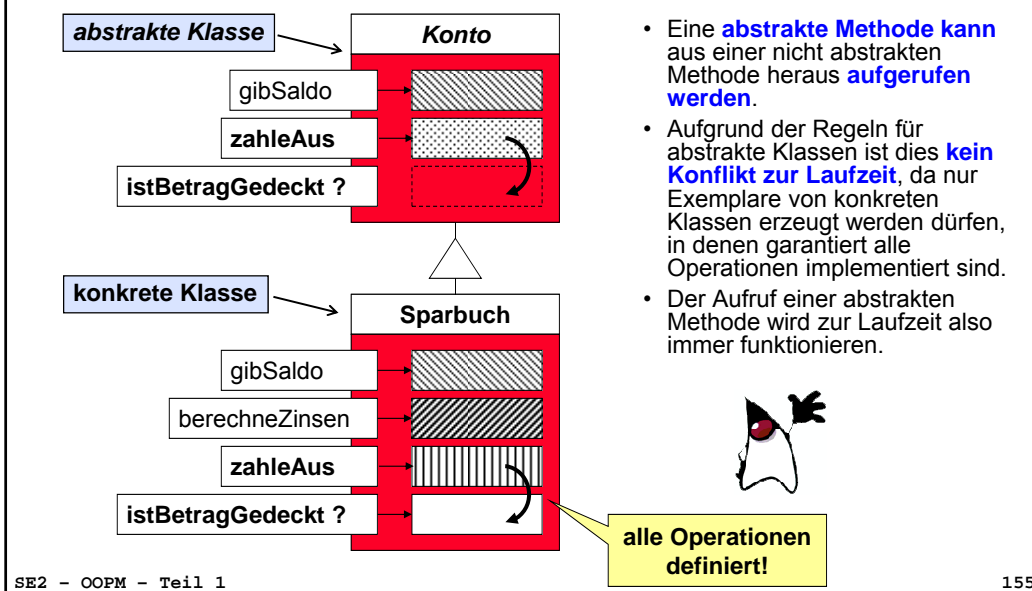


- Eine abstrakte Klasse kann Operationen deklarieren, die lediglich in ihrer Schnittstelle festgelegt sind.
  - Wenn ein Exemplar einer solchen Klasse erzeugt würde, würden bei einem Aufruf dieser Operationen **keine Implementierungen existieren**.
  - Um diesen Fehlerfall zu vermeiden, darf man in Java **keine Exemplare** von abstrakten Klassen **erzeugen**.
  - Eine Unterklasse einer abstrakten Klasse muss **auch abstrakt deklariert sein**, wenn sie nicht alle Operationen implementiert.
  - Eine Klasse ohne abstrakte Methoden wird auch eine **konkrete Klasse** genannt.



In Java kann jede Klasse als abstrakt deklariert werden, auch wenn sie keine abstrakten Methoden enthält!

## Abstrakte Methoden sind aufrufbar!

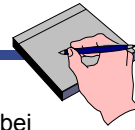


## Wie kann das überhaupt funktionieren?

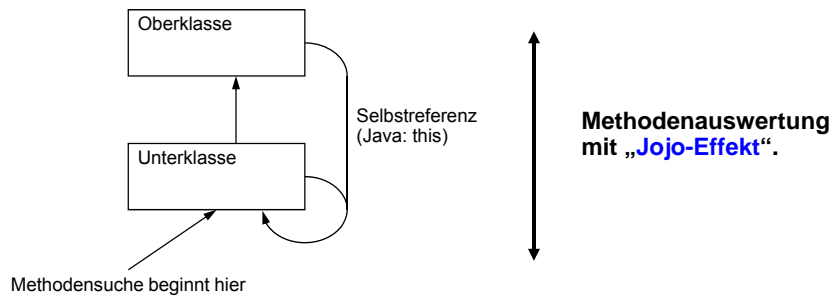
- Zwei **mentale Modelle** erleichtern das Verständnis:
  - „Erben“ heißt „Quelltext kopieren und neu übersetzen“
  - Das Jojo-Modell
- **Quelltext kopieren und neu übersetzen:**
  - Unmittelbar einsichtig: Wenn eine aufgerufene Methode in einer Unterklasse redefiniert wurde, wird dort die redefinierte Methode aufgerufen; wenn die Methode nicht redefiniert wurde, wird die ursprüngliche (kopierte) Methode aufgerufen.
  - Als mentales Modell nützlich, aber selten in Programmiersprachen realisiert (z.B. in Sather).
- Das **Jojo-Modell:**
  - Auch Selbst-Aufrufe werden dynamisch gebunden. Die Suche nach einer geeigneten Implementation beginnt dabei immer in der Klasse des aktuellen Objekts und sucht dann in der direkten Oberklasse, wenn in der durchsuchten Klasse keine Implementation der Operation vorliegt.
  - Unter anderem in Java realisiert.



## Auswertung von Selbst-Aufrufen in Java



- Ein Selbst-Aufruf wird **dynamisch** (also zur Laufzeit) **gebunden** (auch: **spät gebundene Selbstreferenz**, engl.: late-bound self-reference). Dabei wird immer in der Klasse des aktuellen Objektes mit der Methodensuche begonnen.



- Es kann bei der Methodensuche immer wieder dazu kommen, dass eine Methode einer Oberklasse gewählt wird. Aber auch von dort wird jeder Selbstaufruf wieder in der Klasse des aktuellen Objektes begonnen.

## Schablonenmethode und Einschubmethode



- Das Verhältnis zwischen einer aufrufenden (also nicht abstrakten) Methode und der aufgerufenen abstrakten Methode wird im **Schablonenmuster** abstrahiert:
  - Eine **Schablonenmethode** (engl.: template method) ist eine konkrete Methode, die eine oder mehrere abstrakte Methoden aufruft. Sie legt üblicherweise einen **Ablauf**, einen **Algorithmus** oder ein **Teilverhalten** fest.
  - Eine **Einschubmethode** (engl.: hook method) ist eine abstrakte Methode, die meist zusammen mit einer Schablonenmethode vorkommt. Die erbbende Klasse implementiert die Einschubmethode und konkretisiert so einen **Teil des vorgegebenen abstrakten Verhaltens**. Die Unterklasse „schiebt“ quasi konkretes Verhalten ein, daher der Name.
  - Einschubmethoden werden für den Zweck der **Code Injektion** definiert und sind eine wichtige Grundlage für **objektorientierte Rahmenwerke**.

## Verwendung von abstrakten Oberklassen

```
abstract class FloatList
{
    public abstract void append(float f);
    public abstract void start();
    public abstract void next();
    public abstract boolean empty();
    public abstract boolean off();
    public abstract float item();

    public boolean contains(float f){
        boolean ergebnis = false;

        if ( !empty() ) {
            start();
            while (!off() && (item() != f)){
                next();
            }
            ergebnis = !off();
        }
        return ergebnis;
    }
}
```

- Abstrakte Implementationen
  - Oft kann man auf Basis von abstrakten Methoden schon vollständige Algorithmen formulieren.
  - **contains()** kann von der abstrakten Oberklasse FloatList unter Rückgriff auf ihre abstrakten Methoden für die gesamte Klassenfamilie der Float-Listen implementiert werden.

## Abstrakte Oberklassen zur Spezifikation

- Abstrakte Klassen können also für den Zweck definiert werden, **Teile des Verhaltens** von Unterklassen festzulegen.
- In Java kann dieses Verhalten den Unterklassen auch **zwingend vorgeschrieben** werden:
  - Wenn eine **Operation** als **final** deklariert wird, dann kann sie in einer Unterklasse nicht redefiniert werden:

```
final void gibSaldo() { ... }
```

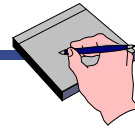
- Wenn eine ganze **Klasse** als **final** deklariert wird, dann können von ihr **keine Unterklassen** definiert werden:

```
final class String {
    ...
}
```



Die Klasse **String** ist in Java **final** deklariert.

## Vererbung und Konstruktoren in Java



- Konstruktoren werden in Java **nicht vererbt**.
  - Eine erbende Klasse bietet deshalb nicht automatisch die gleichen Konstruktoren wie ihre Oberklasse an!
- Bei der Erzeugung eines Objekts einer abgeleiteten Klasse werden jedoch die Konstruktoren **sämtlicher Oberklassen** - von der entferntesten bis zur direkten - **gerufen**.
- Ein Programmierer kann dies **explizit** anstoßen, indem er einen super-Aufruf als **erste Anweisung im Konstruktor** der Unterklasse formuliert; eventuell müssen dabei Parameter übergeben werden, wenn der gerufene Konstruktor der Oberklasse diese fordert.
- Fehlt in einem Konstruktor ein expliziter super-Aufruf, dann fügt der Compiler **automatisch einen parameterlosen Aufruf** (`super()`) ein – auch wenn dieser Aufruf möglicherweise fehlschlägt!
- Wenn in einer Klasse **A** gar kein Konstruktor angegeben ist, dann fügt der Compiler automatisch den folgenden Standard-Konstruktor ein:  

```
public A() { super(); }
```

SE2 – OOPM – Teil 1

161

## Beispiel: Eigene Exception-Klasse definieren

- Die Klasse **Exception** verfügt über ein String-Attribut, das mit einem Konstruktor gesetzt werden kann. Wir können es über einen super-Aufruf zum Ablegen eines **Fehlertextes** nutzen.
- Mit der von **Exception** geerbten Operation `getMessage()` kann ein Klient diesen Text auslesen.

```
class RangeException extends Exception
{
    public RangeException() { super(); }
    public RangeException(String s){ super(s); }
    public RangeException(int i)
    {
        super("Access at position " + i + " was invalid.");
    }
}
```

SE2 – OOPM – Teil 1

162

### Weiteres Beispiel für Konstruktoren und Vererbung

```
class Punkt {
    private double _x, _y;

    public Punkt(){
        this(0.0, 0.0);
    }

    public Punkt(double x, double y){
        _x = x;
        _y = y;
    }
    ...
}

class Pixel extends Punkt {
    private Farbe _farbe;
    ...
    public Pixel(double x, double y, Farbe f){
        super(x,y);
        _farbe = f;
    }
}
```

eigener Standardkonstruktor, der den 2-stelligen Konstruktor `Punkt` ruft.

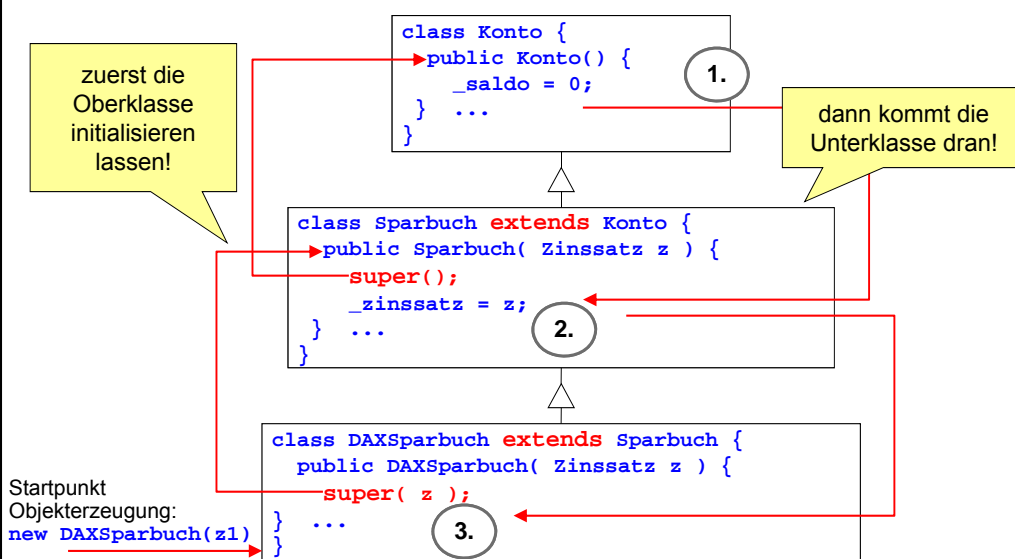
2-stelliger Konstruktor `Punkt` initialisiert die Exemplarvariablen.

Konstruktor `Pixel` ruft den Konstruktor der Oberklasse `Punkt`.

SE2 – OOPM – Teil 1

163

### Konstruktoren und Vererbung, mehrstufiges Beispiel



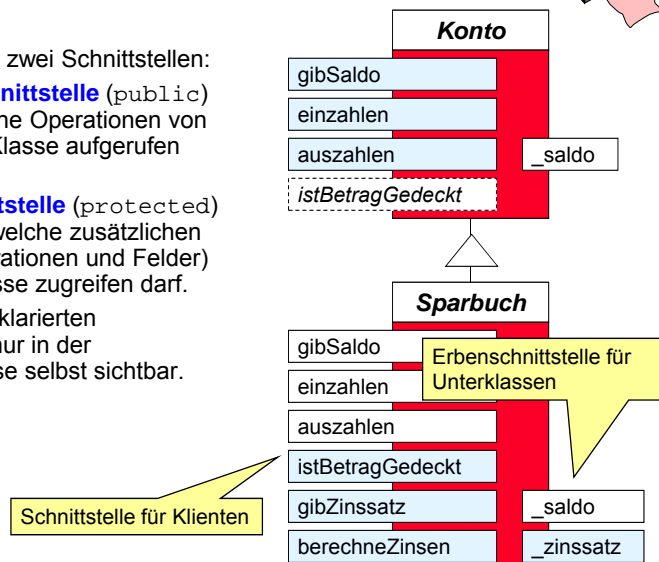
SE2 – OOPM – Teil 1

164



## Klienten- und Erbenschnittstelle

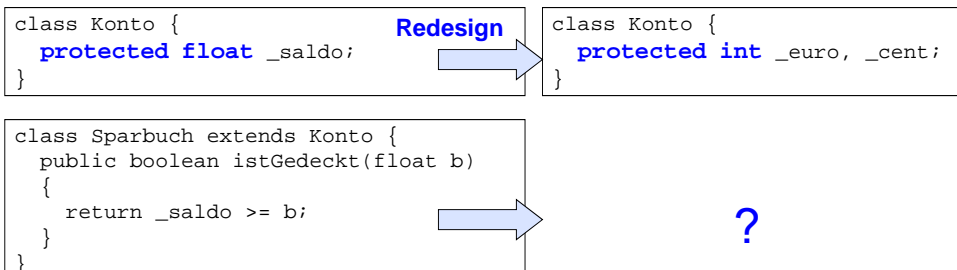
- Eine Klasse verfügt über zwei Schnittstellen:
  - An der **Klientenschnittstelle** (public) wird festgelegt, welche Operationen von einem Klienten der Klasse aufgerufen werden können.
  - An der **Erbenschnittstelle** (protected) wird festgelegt, auf welche zusätzlichen Eigenschaften (Operationen und Felder) eine abgeleitete Klasse zugreifen darf.
  - Alle als `private` deklarierten Eigenschaften sind nur in der deklarierenden Klasse selbst sichtbar.



SE2 – OOPM – Teil 1

165

## Modellierung der Erbenschnittstelle



- Problem:
  - Bei unserem aktuellen Entwurf haben `Sparbuch` und `Girokonto` **direkten Zugriff** auf die Exemplarvariable `_saldo` der Oberklasse.
  - Auf diese Weise **durchbrechen wir die Datenkapselung** in `Konto` und machen unseren Entwurf **änderungsanfällig**.
  - Die **Erbenschnittstelle** sollte die interne Speicherstruktur von `Konto` deshalb **genauso gut kapseln**, wie es auch für die Klientenschnittstelle gefordert wird.

SE2 – OOPM – Teil 1

166

## Modellierung der Erbenschnittstelle

```
class Konto {
    private float _saldo;
    protected float gibSaldo() {
        return _saldo;
    }
}
```

Redesign

```
class Konto {
    private int _euro, _cent;
    protected float gibSaldo() {
        return _euro + (float)_cent/100;
    }
}
```

```
class Sparbuch extends Konto {
    public boolean istGedeckt(float b)
    {
        return gibSaldo() >= b;
    }
}
```

Die Unterklasse bleibt unverändert - trotz der Umstellung der internen Struktur!



Werden die Exemplarvariablen **durch Zugriffsooperationen** auch gegenüber den erbenden Klassen **gekapselt**, so kann die Repräsentation in der Oberklasse geändert werden, ohne dass die Unterklasse von dieser Änderung betroffen ist!

## Aufgepasst: Der Operator instanceof in Java

- In SE1 haben wir gesagt, dass der Operator `instanceof` genau dann `true` liefert, wenn eine gegebene Referenz auf ein **Exemplar der genannten Klasse** verweist.
- Dies ist angesichts der zusätzlichen Möglichkeiten durch Subtyping und Implementationsvererbung nicht mehr ganz korrekt. Eigentlich wird getestet, ob eine gegebene Referenz auf ein **Exemplar des genannten Typs oder eines seiner Subtypen** verweist.
- Beispielsweise liefert  

```
ref instanceof Object
```

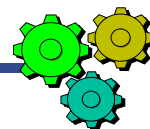
 für jedes Exemplar in Java `true`, auch für solche, die nicht direkte Exemplare der Klasse `Object` sind!
- Immerhin haben wir den Operator in SE1 unter dem Begriff **Typtest** bereits korrekt eingeordnet...

## Zusammenfassung Implementationsvererbung



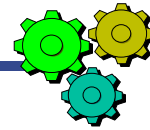
- Mit Implementationsvererbung wird ermöglicht,
  - ... **gemeinsames Verhalten** verschiedener Klassen in einer Oberklasse **zusammenzufassen**.
  - ... dass eine Unterklasse den Code, den sie erbt, um eigene Operationen **erweitert**.
  - ... dass eine Unterklasse den Code, den sie erbt, auch für ihre speziellen Zwecke **anpasst**, indem sie Operationen redefiniert.
- **Abstrakte Klassen** deklarieren für einige Operationen nur ihre Schnittstelle, geben aber nicht ihre Implementation vor.
- Durch Implementationsvererbung wird eine Unterscheidung zwischen **Klientenschnittstelle** und **Erbenschnittstelle** notwendig.
- Java-spezifisch:
  - Alle **Typen** (Klassen und Interfaces) erben die **Operationen** des Typs **Object**, alle **Klassen** erben die Methoden der Klasse **Object**.
  - **Konstruktoren** werden nicht vererbt.

## Diskussion Vererbung et al.



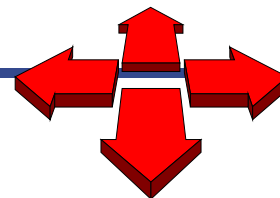
- Die Vererbungskonzepte in objektorientierten Programmiersprachen werden methodisch eingesetzt zur
  - Modellierung von Begriffshierarchien (in Subtyp-Hierarchien),
  - Trennung von Spezifikation und Implementation (Typabstraktion),
  - inkrementellen Übernahme und Veränderung vorhandener ausführbarer Software-Einheiten (Implementationsvererbung),
  - Abstraktion gemeinsamer Merkmale (Redundanzvermeidung auf Code-Ebene).

## Diskussion Vererbung et al. (2)



- Auf Implementationsebene hat sich Einfachvererbung durchgesetzt, während auf die flexiblen Entwurfsmöglichkeiten durch multiples Subtyping (siehe Interfaces in Java) kaum noch verzichtet werden kann.
- Objektorientierte Vererbungskonzepte spielen schon beim fachlichen Entwurf eine große Rolle.
- Bei der Konstruktion großer Systeme nutzen wir
  - Interfaces zur Spezifikation von Schnittstellen,
  - Klassen zur abstrakten und konkreten Implementierung.

## Fehlerbehandlung mit Exceptions



- Fehlerkategorien
- Traditionelle Techniken der Fehlerbehandlung
- Grundsätzliches zur Ausnahmebehandlung
- Exceptions in Java: Auslösen und Behandeln
- Javas Exception-Hierarchie

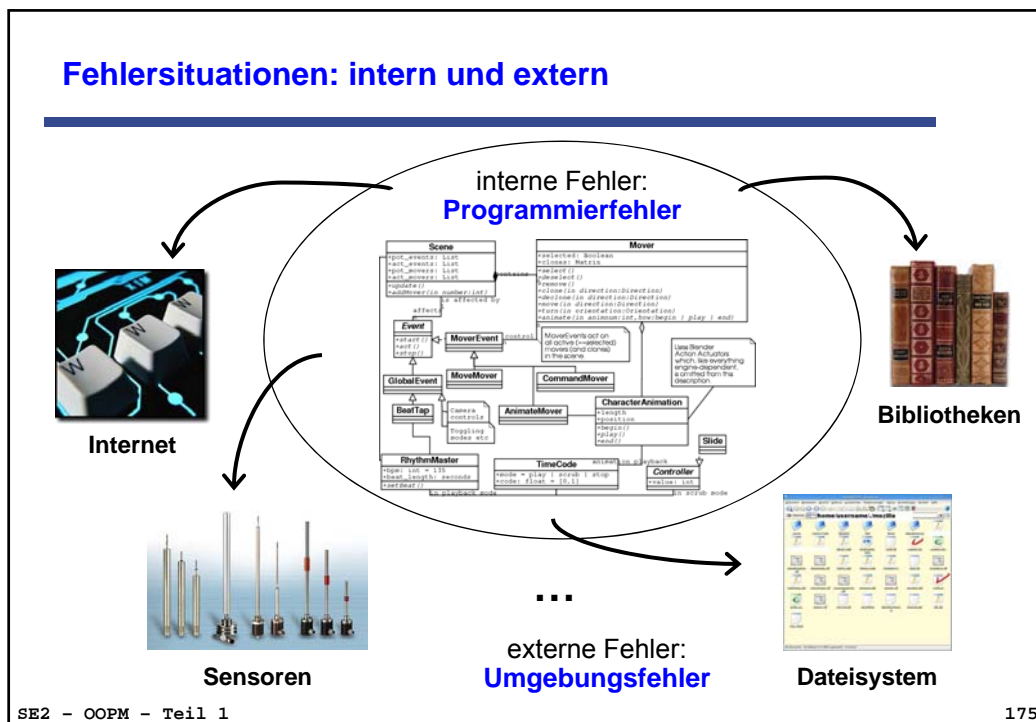
## Fehlersituationen in Programmen

- In Softwaresystemen kann es zu vielfältigen Fehlersituationen kommen, beispielsweise:
  - Eine Implementation erfüllt nicht die Anforderungen ihrer Spezifikation.
  - Bei einem schreibenden Zugriff auf eine Datei hat ein Programm keine Schreibrechte für die Datei.
  - Ein Klient hält sich bei der Anforderung einer Dienstleistung nicht an die Vorbedingungen.
  - Für den Zugriff auf eine Webseite hat der Browser keinen Zugang zum Internet.
  - Bei einem Array-Zugriff wird ein ungültiger Index benutzt.
  - Durch konkurrierenden (gleichzeitigen) Zugriff zweier Prozesse auf ein Objekt gerät dieses in einen inkonsistenten Zustand.
- **Wie können wir systematisch mit solch unterschiedlichen Fehlersituationen umgehen?**

## Programmierfehler versus Umgebungsfehler



- Als Ursachen für Fehlersituationen können fundamental unterschieden werden:
  - **Programmierfehler**
  - **Umgebungsfehler**
- **Programmierfehler** werden bei der Softwareentwicklung gemacht. Obwohl sie prinzipiell vermeidbar sind, treten sie auf. Für Programmierfehler kann es zur Laufzeit keinen **korrekten** Umgang in der Software selbst geben; das Behandeln eines Programmierfehlers in der Software dient ausschließlich der **Robustheit** der Software! Programmierfehler, die zur Übersetzungszeit entdeckt werden, werden behoben und sind dann weg.
- **Umgebungsfehler** liegen außerhalb des Einflussbereichs des Programmerteams und können somit nicht von diesem behoben werden. Sie können immer wieder auftreten und sind prinzipiell vorhersehbar; es können deshalb Vorkehrungen in der Software getroffen werden, um mit diesen Fehlern umzugehen.



### Umgang mit Fehlersituationen

- Wir können in objektorientierten Systemen zwei Situationen unterscheiden:
  - Das Erkennen und Signalisieren eines Fehlers durch einen **Dienstleister**.
  - Das Behandeln eines Fehlers durch einen **Klienten**.
- Für das **Erkennen und Signalisieren** stellen sich Fragen wie:
  - Wann, wie, wie häufig soll ein Dienstleister prüfen, ob er korrekt verwendet wird?
  - Was soll er tun, wenn ein Klient sich fehlerhaft verhält?
- Beim **Behandeln** stellen sich Fragen wie:
  - Wie kann sich ein Klient auf mögliche Fehler einstellen?
  - Wie soll der Klient auf Fehler reagieren? Kann er nach Fehlerart differenzieren und unterschiedlich reagieren?

## Traditionelle Ansätze zur Fehlerbehandlung

- Traditionell werden Fehlersituationen mit Hilfe der folgenden Techniken behandelt:
  1. Das Programm wird aus der fehlerhaften Methode heraus **abgebrochen**.
  2. Die fehlerhafte Methode liefert einen **Return-Wert**, der vom Aufrufer als Fehler-Code interpretiert werden soll.
  3. Die fehlerhafte Methode setzt eine **global lesbare Variable** auf den eingetretenen Fehlerstatus.
  4. Die fehlerhafte Methode ruft eine sog. **Callback-Operation**, die zuvor bei ihr für den Fall eines auftretenden Fehlers bekannt gemacht worden ist.

## Kritik an den traditionellen Ansätzen

- Die traditionellen Techniken besitzen folgende Nachteile:
  1. Bei vielen Anwendungssystemen ist ein Programmabbruch nicht akzeptabel, etwa bei **sicherheitskritischen** Systemen (z.B. Flugzeugsteuerungen).
  2. Nicht immer lässt sich ein **geeigneter Wert** finden, der als Fehlercode interpretiert werden kann, z.B. wenn alle Werte des deklarierten Rückgabetyps auch sinnvolle Rückgabewerte sind.  
Prinzipiell muss jeder Operationsaufruf auf einen Fehlerfall abgeprüft werden, dies erscheint vielen Programmierern als **inakzeptabler Aufwand**.
  3. Fehlerhafte Operationen mit ungeprüften Fehlermeldungen werden aus Sicht eines Aufrufers **scheinbar "normal"** ausgeführt, d.h. der Fehler wird nicht erkannt.
  4. Fehlerbehandlung über Callback-Operationen weist bereits in die Richtung einer Ausnahmebehandlung, **bläht** aber die **Schnittstellen** von Operationen unnötig **auf**.

## Ausnahmen als eigenständiges Sprachkonzept

- In Programmiersprachen wurden für das systematische Umgehen mit Fehlersituationen eigene Sprachkonzepte für so genannte **Ausnahmen** (engl.: exceptions) entwickelt.
  - Eine der ersten Programmiersprachen mit einem Exception-Konzept war **CLU**; die erste weiter verbreitete Sprache war **Ada**.
  - **Eiffel** war die erste objektorientierte Programmiersprache mit Exceptions.
- Mindestens folgende **Ziele** sollen mit einer Sprachunterstützung erreicht werden:
  - Fehler sollen nicht einfach ignoriert werden können.
  - Der Quelltext zur Fehlerbehandlung soll den Normalfall nicht unnötig „verschütten“.
  - Durch eine Typisierung der Fehler sollte je nach Fehlerart unterschiedlich reagiert werden können.

## Exceptions in Java sind Objekte

- Eine Exception ist in Java ein **Exemplar einer Exception-Klasse**.
- Es existieren etliche **vordefinierte Exception-Klassen** in den Java-Bibliotheken.
  - Jede Exception hat damit einen **Typ**, der sich zur Laufzeit abfragen lässt; dies ermöglicht differenziertes Reagieren auf verschiedene Fehlerarten.
- Ein Programmierer kann zusätzlich **eigene** Exception-Klassen definieren.
  - Dies hat den Vorteil, dass die Exemplare dieser Klassen mit beliebigen **zusätzlichen Informationen** über den Grund der Fehlersituation versehen werden können.





## Geprüfte und ungeprüfte Exceptions

- In Java wird grundsätzlich unterschieden zwischen **geprüften** und **ungeprüften** (engl.: checked and unchecked) Exceptions.
- Wenn eine Operation in ihrer Implementation eine **geprüfte Exception** auslösen könnte, muss sie dies **deklarieren**. Geprüfte Exceptions werden für vorhersehbare Fehler eingesetzt, also primär für **Umgebungsfehler**.
  - Beispiele für geprüfte Exceptions: `ServerNotActiveException`, `IOException`, `TimeoutException`
- **Ungeprüfte Exceptions** müssen hingegen **nicht deklariert** werden. Sie werden primär für **Programmierfehler** definiert, da diese Fehler praktisch jederzeit auftreten können und ihre Deklarationen den Quelltext überschwemmen würden.
  - Beispiele für ungeprüfte Exceptions: `ArrayIndexOutOfBoundsException`, `NullPointerException`, `OutOfMemoryException`



Das „geprüft“ und „ungeprüft“ ist **statisch** zu verstehen, denn es bezieht sich auf die **Übersetzungszeit**. Zur Laufzeit führen alle Exceptions zu einer Spezialbehandlung in der Virtual Machine.

## Javas Schlüsselwörter für die Ausnahmebehandlung

- |                |   |
|----------------|---|
| <b>throw</b>   | <b>Auslösen</b> einer Exception.  |
| <b>throws</b>  | <b>Deklarieren</b> eine (geprüften) Exception an der Operationsschnittstelle.   |
| <b>try</b>     | Einleiten eines Blocks, in dem mit dem <b>Auftreten einer Exception</b> gerechnet wird bzw. werden muss.  |
| <b>catch</b>   | Einleiten eines Blocks, in dem eine aufgetretene Exception <b>behandelt</b> (engl.: to handle) werden kann. Der catch-Block wird deshalb auch als <b>Exception-Handler</b> bezeichnet.                                |
| <b>finally</b> | Einleiten eines Blocks, der <b>immer ausgeführt</b> wird, wenn der try-Block betreten wurde, selbst wenn dort eine Exception zum vorzeitigen Beenden des try-Blocks geführt hat und ein catch-Block ausgeführt wurde. |



## Ausnahmebehandlung in Java (1)

- Grundsätzliches zum Auslösen von Exceptions:
  - Eine **Ausnahme** bzw. **Exception** ist ein Signal, dass irgendeine Ausnahmebedingung aufgetreten ist, z.B. ein Fehler beim Öffnen einer Datei oder das Überschreiten von Array-Grenzen.
  - Eine Ausnahme **auszulösen** bedeutet, eine Ausnahmebedingung zu signalisieren.
  - Die auslösende Methode wird abgebrochen und es wird zur aufrufenden Methode **zurückgesprungen**.
  - Eine Ausnahme wird auf diese Weise von Methode zu Methode **weiterpropagiert** und durchläuft die Methoden-Aufrufkette, bis sie abgefangen wird.



## Ausnahmebehandlung in Java (2)

- Grundsätzliches zum Behandeln:
  - Eine Ausnahme **abzufangen** (catch) heißt, sie zu behandeln; entweder werden Aktionen durchgeführt, die den normalen Zustand wieder herstellen, oder es wird organisiert abgebrochen, inklusive möglicher Aufräumarbeiten.
  - Wird eine Ausnahme überhaupt nicht abgefangen, so bahnt sie sich ihren Weg bis zum Aufruf der **main-Methode**.
  - Fällt sie durch die **main-Methode**, wird der Java-Interpreter eine Fehlermeldung ausgeben und die Programmausführung **abbrechen**.



## Das Auslösen einer Exception

- Wann löst man eine Exception aus?
  - Eine Methode, die ihre Aufgabe **nicht erfüllen** kann (z.B., weil bestimmte Bedingungen in ihrer Umgebung nicht erfüllt sind), kann eine Exception auslösen.
  - Bevor sie eine Exception auslösen kann, muss sie ein **Exception-Objekt erzeugen**. Erst durch das **Werfen** dieses Objektes (mit dem Schlüsselwort **throw**) wird eine Exception ausgelöst.
  - **Spezialfall geprüfte Exception**: Für eine geprüfte Exception muss die Methode in ihrem Kopf **deklarieren** (mit dem Schlüsselwort **throws**), dass sie diese Exception im Bedarfsfall auslösen wird.



## Das Umgehen mit geprüften Exceptions



### Regel:

Wenn eine Methode A eine andere Methode B benutzt, die eine geprüfte Exception deklariert, dann müssen in der Methode A Vorkehrungen für den Fall getroffen werden, dass die deklarierte Exception wirklich auftritt.

- Es gibt zwei Möglichkeiten für die Methode A:
  - Sie kann die Exception **behandeln**.
  - Sie kann selbst deklarieren, dass sie diese Exception auslöst, sie also **weiterpropagieren**.



## Das Behandeln von Exceptions

- Allgemeine Struktur eines Exception-Handlers:

```
try
{
    // Anweisungen,
    // die Exceptions auslösen können
}
catch ( <Exception-Typ> e)
{
    // Behandeln der Fehlersituation
}
finally
{
    // Anweisungen, die in jedem Fall
    // ausgeführt werden sollen
}
```

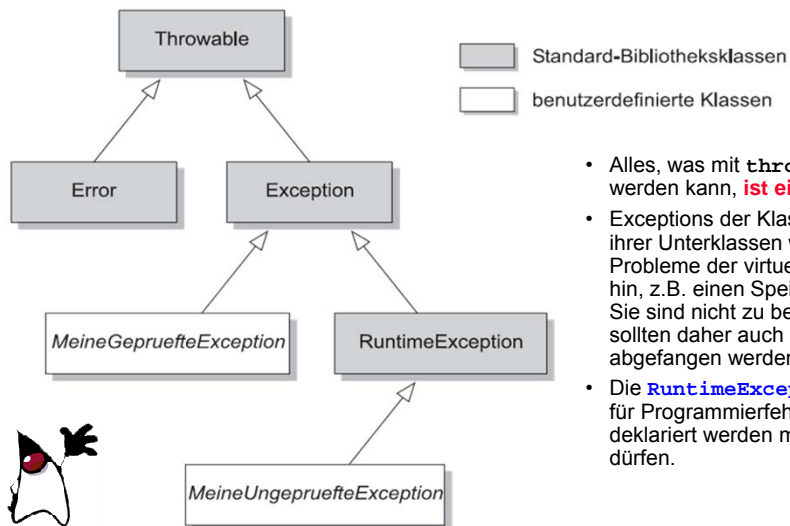


## Ausnahmen differenziert behandeln

- Mit welchem Exception-Handler ?
  - Auf einen try-Block können **mehrere** catch-Blöcke folgen.
  - Für jeden catch-Block wird der Typ der Exceptions spezifiziert, die er behandeln soll.
  - Es wird **maximal ein** Exception-Handler ausgeführt.
  - Bei mehreren möglichen Exception-Handlern wird der **erste passende** gewählt.
  - Ein catch-Block passt zu einer Exception, wenn der Typ der erzeugten Exception gleich oder ein **Subtyp** ist.



## Vererbung: Ausschnitt aus Javas Exception-Hierarchie



- Alles, was mit `throw` geworfen werden kann, **ist ein Throwable**.
- Exceptions der Klasse `Error` oder ihrer Unterklassen weisen auf Probleme der virtuellen Maschine hin, z.B. einen Speicherüberlauf. Sie sind nicht zu beheben und sollten daher auch nicht abgefangen werden.
- Die `RuntimeExceptions` stehen für Programmierfehler, die nicht deklariert werden müssen, aber dürfen.

SE2 – OOPM – Teil 1

189

## Beispiel: Eigene Exception-Klasse definieren

- Die Klasse `Exception` verfügt über ein String-Attribut, das mit einem Konstruktor gesetzt werden kann. Wir können es über einen super-Aufruf zum Ablegen eines **Fehlertextes** nutzen.
- Mit der von `Exception` geerbten Operation `getMessage()` kann ein Klient diesen Text auslesen.

```

class RangeException extends Exception
{
    public RangeException() { super(); }
    public RangeException(String s){super(s); }
    public RangeException(int i)
    {
        super("Access at position " + i + " was invalid.");
    }
}
  
```

SE2 – OOPM – Teil 1

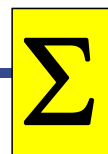
190

## Exceptions als Abstraktionshilfe

“From the Software Engineering point of view, one can regard **exceptions** and **exception handling** as yet another technique for **software abstraction**: being able to abstract away rare, special cases during a first pass in writing and understanding a program. The **anticipated exceptions and their handlers** can then be considered as refinements to the program which appear as **footnotes**, ...”

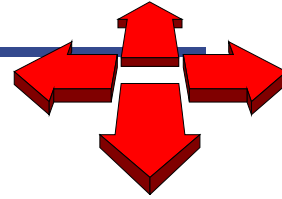
Borgida, A: "Exceptions in Object-oriented Languages", SIGPLAN Notices 21:10, S. 107-119, 1986.

## Zusammenfassung



- In Softwaresystemen kann es sehr **verschiedene Fehlerursachen** für auftretende Fehler geben.
- Eine grundsätzliche Unterscheidung aus Sicht der Softwareentwicklung ist die Unterscheidung in **Umgebungsfehler** und **Programmierfehler**.
- Für den systematischen Umgang mit Fehlern wurde für Programmiersprachen das Konzept der **Ausnahmen** bzw. **Exceptions** entwickelt.
- In Java sind Exceptions **Exemplare von Exception-Klassen**, die zur Laufzeit im Fehlerfall erzeugt und geworfen werden.
- In Java kann in **Exception-Handlern** jede geworfene Exception zur Laufzeit gefangen und behandelt werden.

## Namensräume und Modularisierung



- Das Modulkonzept
- Javas Klassen als Module
- Javas Pakete als Module

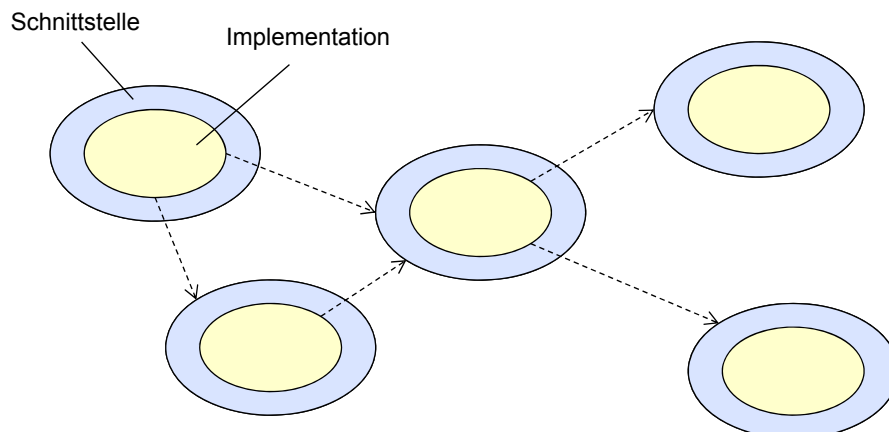
## Das Modulkonzept



- Die klassische imperative Programmierung ist eng mit dem **Modulkonzept** verknüpft.
- Entstanden aus der Notwendigkeit, große Programmtexte in für den Übersetzer fassliche Einheiten zu zerlegen, wurde das Modulkonzept **zum zentralen Organisationskonzept für Entwürfe und Programmtexte**.
- Für nicht-objektorientierte Sprachen sind Module (soweit in der Sprache vorhanden) die Programmeinheiten, in denen fachliche oder technische Entwurfsentscheidungen gekapselt werden (Geheimnisprinzip).
- Die neueren Entwicklungen bei objektorientierten Programmiersprachen zeichnen sich u.a. durch eine neue Interpretation des Modulkonzepts aus (wir stellen das Package-Konzept in Java vor).
- Module werden hier als wichtiges Konstruktionsmerkmal der imperativen Programmierung in Verbindung zur objektorientierten Programmierung gesetzt.

## Die Grundidee: ein Software-System besteht aus Modulen

- Module sind statische Einheiten des Quelltextes mit einer **Schnittstelle** und einer **Implementation**. Sie benutzen sich gegenseitig ausschließlich über ihre Schnittstellen.

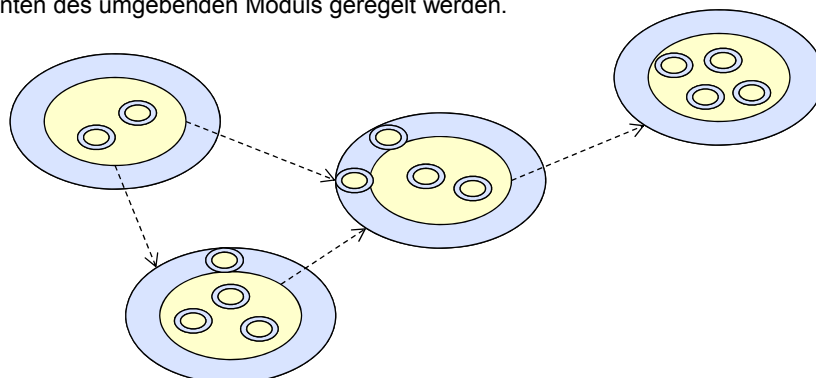


SE2 – OOPM – Teil 1

195

## Weiteres Konzept: Module können Module enthalten

- In großen Systemen sind Module häufig auf **mehreren Ebenen** (beispielsweise Schichten, Subsysteme, Pakete) definiert, um die hohe Anzahl an Einheiten strukturierbar zu halten.
- Wenn Module selbst wieder Module enthalten können, dann muss der Zugriff (**Benennung** und **Schutz**) auf die geschachtelten Module für Klienten des umgebenden Moduls geregelt werden.



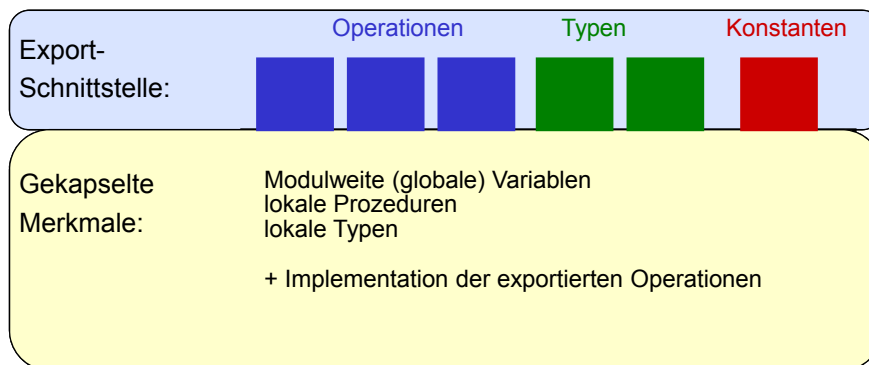
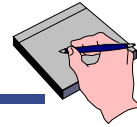
SE2 – OOPM – Teil 1

196



### Klassisch: Modul – Exportschnittstelle

- Um Dienstleistungen über Modulgrenzen hinaus verwendbar zu machen, müssen sie nach außen bekannt gemacht werden.
- Die **Exportschnittstelle** definiert, welche deklarierten Bestandteile ein Modul seiner Umgebung zur Verfügung stellt.

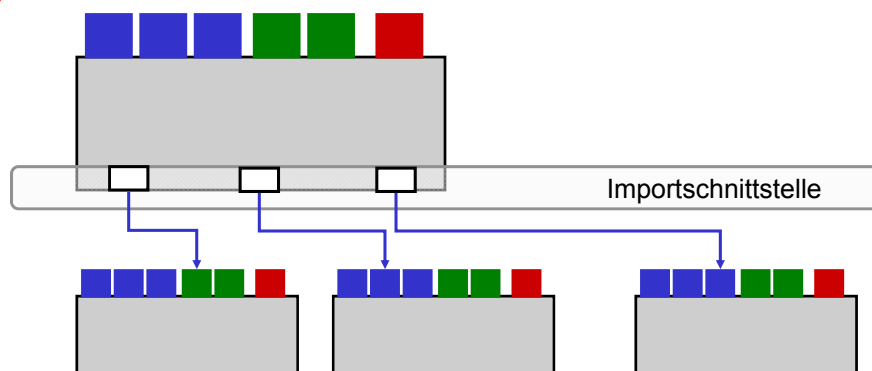


SE2 – OOPM – Teil 1

197

### Klassisch: Modul – Importschnittstelle

- Um Programmobjekte und ihre Bezeichner über Modulgrenzen hinaus zu verwenden, müssen sie gezielt angefordert werden.
- Die **Importschnittstelle** beschreibt explizit, welche deklarierten und exportierten Objekte ein Modul von seiner Umgebung benötigt.



SE2 – OOPM – Teil 1

198

## Imperative Sprachen: Beispiel für Export in Modula-2

- Modula-2 kennt getrennte Einheiten für die Definition der Export-Schnittstelle (Definitions-Modul) und für die Implementation.
- Beispiel für ein sog. **Definitionsmodul**:

```

DEFINITION MODULE Buch;
  TYPE
    BUCH = RECORD
      Nummer : CARDINAL;
      Autor  : ARRAY [1..30] OF CHAR;
      Titel  : ARRAY [1..60] OF CHAR
    END;
  PROCEDURE Einrichten;
  PROCEDURE Auflegen(Element : BUCH);
  PROCEDURE Lesen(VAR Element : BUCH);
  PROCEDURE Entfernen
END Buch.

```

- Das Definitionsmodul **Buch** exportiert:
  - Den Record-Typ **Buch**
  - Die Prozeduren **Einrichten**, **Auflegen**, **Lesen**, **Entfernen**

© K. Murmann, H. Neumann, Fakultät für Informatik, Universität Ulm, 2001

## Imperative Sprachen: Beispiel für Import in Modula-2

- Im sog. Implementationsmodul **Benutzer** werden die Operationen **fac**, **sqrt** des Moduls **MathFunc** importiert:

```

IMPLEMENTATION MODULE Benutzer;
  FROM MathFunc IMPORT fac, sqrt;
  VAR
    ergebnis, wert : CARDINAL;
  BEGIN
    ...
    ergebnis := fac(wert);
    ...
  END Benutzer.

```

- Es können auch alle exportierten Merkmale eines Moduls importiert werden.

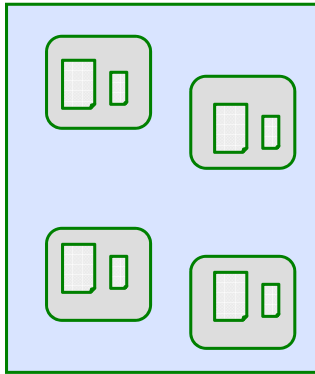
```

IMPLEMENTATION MODULE Benutzer;
  IMPORT MathFunc;
  VAR
    ergebnis, wert : CARDINAL;
  BEGIN
    ...
    ergebnis := MathFunc.fac(wert);
    ...
  END Benutzer.

```

© K. Murmann, H. Neumann, Fakultät für Informatik, Universität Ulm, 2001

## Wesentliche übertragbare Modulkonzepte



- Das **Modulkonzept** leistet vorrangig zweierlei:
  - Es führt einen **Namensraum** (engl.: name space) ein, um Programmeinheiten zu gliedern und zu benennen:
    - Block
    - Prozedur,
    - Klasse (Interface),
    - Paket (in Java),
    - Programm.
  - Es definiert einen **Zugriffsschutz** (engl.: access control) für die enthaltenen Elemente:
    - Elemente innerhalb eines Moduls sind zunächst verborgen.
    - Sie können durch gezielten Export für die Benutzung zugreifbar gemacht werden.

SE2 – OOPM – Teil 1

201

## Klassen sind auch Module

- Die Klassen von objektorientierten Programmiersprachen können auch als Module angesehen werden:
  - Sie sind logische Einheiten des statischen Quelltextes, um große Programme handhabbarer zu machen.
  - Sie lassen sich meist einzeln (und damit getrennt) übersetzen.
  - Sie definieren eine Schnittstelle mit den Dienstleistungen, die sie ihren Klienten anbieten.
  - Sie definieren einen Namensraum für ihre Dienstleistungen.
  - Sie können Teile ihrer Implementation verbergen.
- Wir wissen aber auch, dass Klassen deutlich mehr können als Module: Sie definieren Typen mit Exemplaren, die polymorph verwendbar sind, und können in Vererbungsbeziehungen zueinander stehen.
- Wenn wir die Klassen in Java betrachten, erkennen wir ein weiteres wichtiges Element: **Klassen** können ineinander **geschachtelt** werden.

SE2 – OOPM – Teil 1

202

## Geschachtelte Klassen in Java

- In Java können Klassen in Klassen (beliebig oft) geschachtelt werden. Eine Klasse, die in einer anderen Klasse enthalten ist, heißt **geschachtelte Klasse** (engl.: nested class).
- „A nested class is any class whose declaration occurs within the body of another class or interface.“ (Gosling 2005)
- Diese sind abzugrenzen von den uns bisher bekannten **Top-Level Klassen** (engl.: top level class), die in Übersetzungseinheiten „ganz außen“ (nicht geschachtelt) aufgeführt sind.
- Geschachtelte Klassen ermöglichen:
  - Bessere Strukturierung (Verstecken von Details)
  - Bequemes Programmieren (Beispiel: Listener für GUIs)



## Geschachtelte und innere Klassen in Java

Java unterscheidet vier Arten von geschachtelten Klassen:

- |                                 |                    |
|---------------------------------|--------------------|
| 1. <b>Static Member Classes</b> | } - Innere Klassen |
| 2. <b>Member Classes</b>        |                    |
| 3. <b>Local Classes</b>         |                    |
| 4. <b>Anonymous Classes</b>     |                    |
- Innere Klassen  
(engl.: inner classes)

- this-Referenz  
- keine static-Felder erlaubt

Die drei Arten von **inneren Klassen** (engl.: inner classes) definieren eine spezielle Untermenge aller geschachtelten Klassen; ihre Exemplare halten implizit immer eine Referenz auf ein Exemplar der umgebenden Klasse; sie definieren also spezielle Eigenschaften auch für die Laufzeit. Wir werden innere Klassen hier nicht näher betrachten.

Die **statischen geschachtelten Klassen** (static member classes) hingegen kommen dem Konzept von geschachtelten Modulen am nächsten.



## Statische geschachtelte Klassen in Java

- Deklariert mit dem Schlüsselwort `static` im Rumpf einer umgebenden Klasse:

```
class A
{
    public void operation()
    { ...
    }

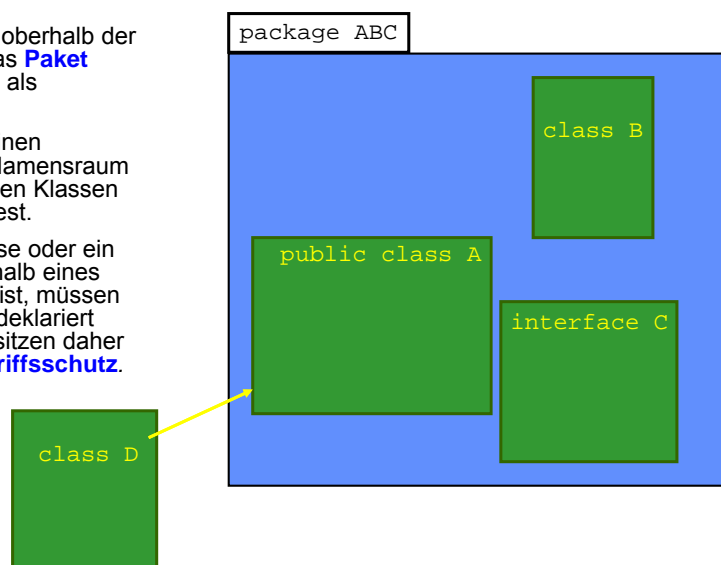
    private static class MyStaticMemberClass()
    { ... // beliebige Felder und Methoden
    }
}
```



- Reine textuelle Schachtelung, ohne Auswirkungen zur Laufzeit.
- Ermöglicht u.a. das Verstecken von Hilfsklassen vor Klienten der umgebenden Klasse (im Beispiel durch den Modifikator `private`).
- Die Klassen sind sehr eng miteinander vertraut: Beide haben **Zugriff auf die privaten Eigenschaften** der Exemplare der jeweils anderen Klasse!

## Auch sehr ähnlich zu Modulen: Pakete in Java

- In Java existiert oberhalb der Klassen noch das **Paket** (engl.: package) als **Namensraum**.
- Ein Paket legt einen gemeinsamen Namensraum für die enthaltenen Klassen und Interfaces fest.
- Damit eine Klasse oder ein Interface außerhalb eines Pakets sichtbar ist, müssen sie als `public` deklariert sein. Pakete besitzen daher auch einen **Zugriffsschutz**.



## Pakete als Namensräume in Java

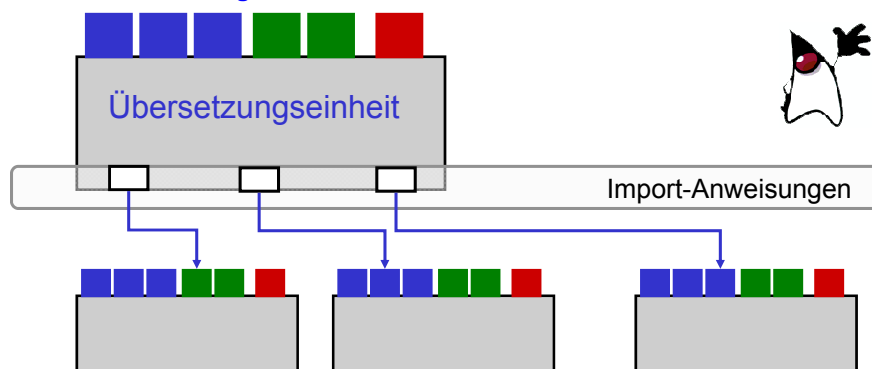
- Pakete als eigene Namensräume ermöglichen, allgemein übliche Namen (wie **List** oder **File**) in einem begrenzten Kontext ohne Konflikte zu verwenden.
- Pakete werden in Übersetzungseinheiten deklariert. Beispiel:  
`package graphics;`
- Alle Klassen und Interfaces eines Pakets haben implizit den Paketnamen als Prefix. Der **voll qualifizierte Name** setzt sich aus dem Paketnamen und dem Typnamen zusammen. Beispiel:  
`graphics.Punkt`
- Jede Klasse (und jedes Interface) kann **nur zu einem** Package gehören. Damit können Operationen einer exportierten Klasse aus einem anderen Paket eindeutig über den voll qualifizierten Bezeichner angesprochen werden. Beispiel:  
`graphics.Punkt.ursprung();`



Package-Deklarationen stehen am Anfang einer Übersetzungseinheit. Dateien ohne explizite Deklaration gehören zu einem Default-Package

## Import von Paketelementen

- Java verfügt über einen **Import-Mechanismus**, der wenig mit der klassischen Import-Schnittstelle zu tun hat:
  - Alle von anderen Paketen exportierten Klassen und Interfaces können durch eine import-Anweisung in einer Übersetzungseinheit einfacher benutzbar gemacht werden. Die importierten Klassen und Interfaces müssen dann nicht mehr voll qualifiziert werden; damit ist der Import nur eine **syntaktische Vereinfachung**.



## Bereits bekannt: Import von Paketelementen

- Die **import-Anweisung** hat zwei Formen:

```
import somepackage.SomeClass;
```

Die Klasse **SomeClass** von **somepackage** ist als Typ verfügbar.

```
import somepackage.*;
```

Alle von **somepackage** exportierten Klassen und Interfaces sind als Typ verfügbar.

- Die importierten Typen sind anschließend in verkürzter Form benutzbar. Z.B.:

```
import graphics.Punkt;  
...  
Punkt pkt = new Punkt();
```

- Ohne Package-Import würde das Beispiel so aussehen:

```
graphics.Punkt pkt = new graphics.Punkt();
```



Softwaretechnisch ist es sinnvoll, nur die tatsächlich benötigten Typen zu importieren; dies verdeutlicht Abhängigkeiten.

## Namensvergabe für Pakete in Java

- Pakete können ineinander geschachtelt werden. Auf diese Weise entstehen Paketnamen, die aus mehreren Teilen bestehen können, die mit einem Punkt voneinander getrennt werden.
- Da Java-Anwendungen leicht über das Internet ausgetauscht werden können, müssen Namenskonflikte vermieden werden. Daher hat sich eine **Namenskonvention** für Pakete durchgesetzt:
  - Die Konvention der Internet Domännennamen wird als Basis der Paketnamen genommen, wobei die Reihenfolge umgedreht wird. Nach Konvention werden die Bestandteile von Paketnamen aus Kleinbuchstaben gebildet.
  - Die „top-level“ Bezeichner **java** und **sun** sind per Konvention reserviert.
  - Beispiel:  
**de.uni-hamburg.informatik.swt.graphics**



## Beschränkter Zugriff: Modifikatoren in Java

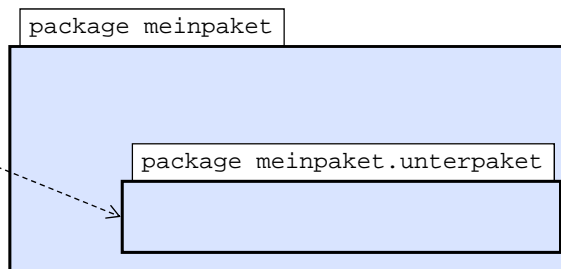
- Objektorientierte Programmiersprachen bieten die Möglichkeit, die **Sichtbarkeit**, genauer den Zugriff, auf Klassen, Interfaces und ihre Bestandteile zu **steuern**.
- Dazu dienen unterschiedliche Zugriffsrechte. Diese werden meist durch reservierte Schlüsselwörter (in Java „**Modifikatoren**“ z.B. **private**, **public**) notiert.
- Dabei müssen wir in Java unterscheiden:
  - Sichtbarkeit einer Klasse oder eines Interfaces selbst, **außerhalb des Packages**.
  - Zugriff auf **Bestandteile einer Klasse**.



## Zugriffsrechte in Java: nicht einschränkbar zwischen Paketen

- Ein **Paket** selbst ist immer zugreifbar; es gibt keine Schutz-Mechanismen zwischen Paketen.
- Auch die **Schachtelung** von Paketen bewirkt in Java **keinerlei Einschränkung der Zugreifbarkeit**; die Schachtelung dient ausschließlich der Strukturierung von Namen.

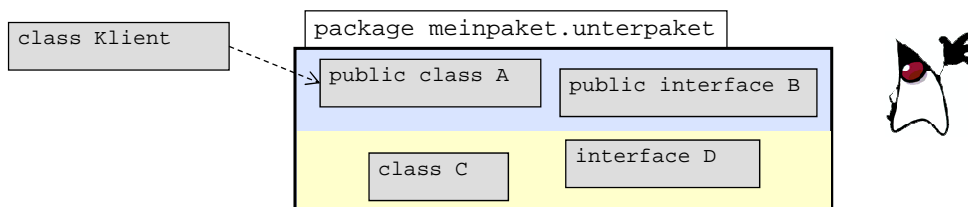
```
import meinpaket.unterpaket.*;  
class Klient  
{ ... }
```





## Zugriffsrechte in Java: Elemente von Paketen

- Die **Klassen** (u. Interfaces) innerhalb eines Paketes sind für andere Klassen (Interfaces) je nach Zugriffsrecht zugreifbar:
  - <Standardzugriff>**: Eine Klasse (Interface) ohne Zugriffsmodifikator besitzt Standardzugriff, d.h. ist ausschließlich zugreifbar für Klassen (Interfaces) im selben Paket.
  - public**: Eine als **public** deklarierte Klasse (Interface) ist zusätzlich zugreifbar für Klassen (Interfaces) in (allen!) anderen Paketen.
- Pakete haben somit eine implizite **Export-Schnittstelle**: Alle als **public** deklarierten Klassen und Interfaces gehören zur Schnittstelle eines Paketes.



SE2 – OOPM – Teil 1

213

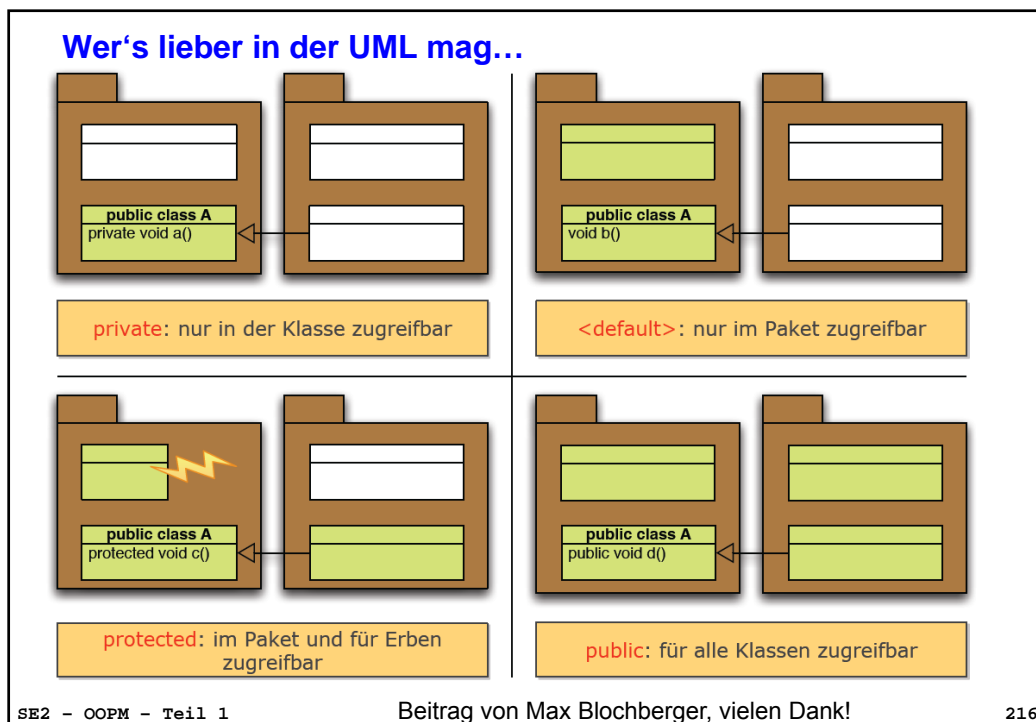
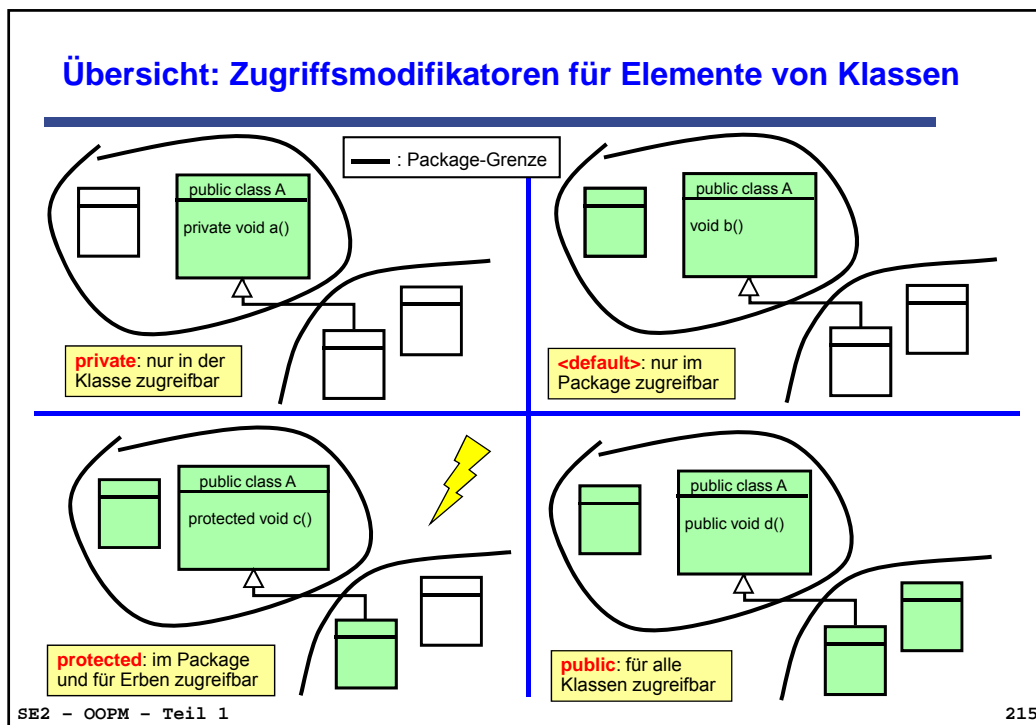
## Zugriffsrechte in Java: Elemente von Klassen (vollständig)

- private**: Ein als **private** deklariertes Element (Feld, Methode oder Klasse) einer Klasse kann nicht von anderen (Top-Level) Klassen zugegriffen werden. Dies gilt unabhängig von der Paketzugehörigkeit oder der Vererbungsbeziehung, auch wenn die Klasse selbst **public** ist.
- <Standardzugriff>**: Ein Element ohne deklarierten Zugriffsschutz ist zugreifbar für alle Klassen im selben Paket.
- protected**: Ein als **protected** deklariertes Element einer **public** Klasse ist zugreifbar für Klassen im selben Paket und für Subklassen in anderen Paketen.
- public**: Ein als **public** deklariertes Element einer **public** Klasse ist zugreifbar für Klassen im selben Paket und in anderen Paketen.



SE2 – OOPM – Teil 1

214



Beitrag von Max Blochberger, vielen Dank!

### Jenseits von Packages: SuperPackages?

com.sun.myModule

Computational Theology  
Gilad Bracha's Sun Weblog\*

```
super package com.sun.myModule {
  export com.sun.myModule.myStuff.*;
  export com.sun.myModule.yourStuff.Interface;
  com.sun.myModule.myStuff;
  com.sun.myModule.yourStuff;
  com.sun.SomeOtherModule.theirStuff;
  org.someOpenSource.someCoolStuff;
}
```


JSR 294: super packages auf dem Weg in die nächste Version von Java?

SomeOtherModule.theirStuff

someCoolStuff

myModule.yourStuff

myModule.myStuff




\*[http://blogs.sun.com/gbracha/entry/developing\\_modules\\_for\\_development](http://blogs.sun.com/gbracha/entry/developing_modules_for_development)

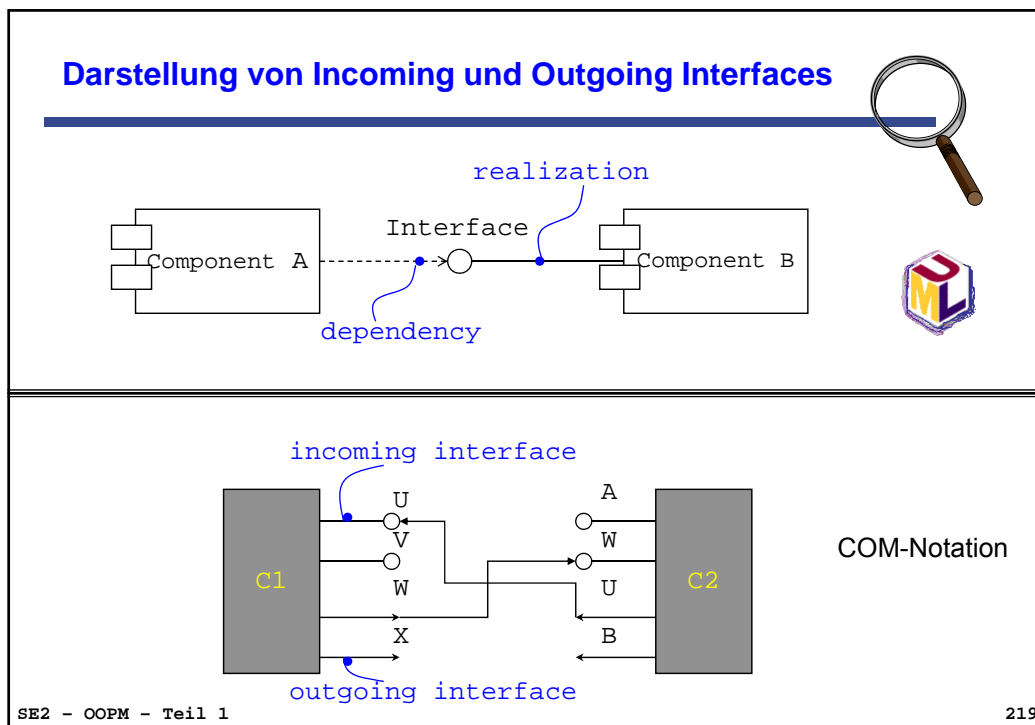
SE2 – OOPM – Teil 1
217

### Ausblick: Incoming und Outgoing Interfaces

- Die Einheiten einer Softwarearchitektur werden häufig auch als **Komponenten** bezeichnet. Der Komponentenbegriff ist jedoch reicher als der Modulbegriff; u.a. sind Komponenten auch zur Laufzeit erkennbar (im Gegensatz zu klassischen Modulen).
- In der **Komponententechnologie** spricht man häufig auch von **Incoming** und **Outgoing Interfaces** statt von Export- und Import-Schnittstellen.
- Ein **Incoming Interface** ist die Schnittstelle, die an einer Komponente **von außen gerufen** werden kann.
- Ein **Outgoing Interface** ist die Schnittstelle, die eine Komponenten **von anderen Komponenten erwartet**, um sie aufzurufen.



SE2 – OOPM – Teil 1
218



### Zusammenfassung: Modul und Klasse

- **Klassen** sind wie **Module** Einheiten der (statischen) Softwarearchitektur. Große Systeme werden aus solchen Einheiten zusammengesetzt. Sie sind üblicherweise auch **Übersetzungseinheiten**.
- Klassen und Pakete bilden wie Module **Namensräume**, deren Elemente vor äußerem Zugriff geschützt werden können.
- Im Gegensatz zu (klassischen) Modulen definieren **Klassen** gleichzeitig einen **Typ**. So lassen sich zur Laufzeit Exemplare von Klassen erzeugen.
- **Module** besitzen verbindliche **Export- und Import-Schnittstellen**. Klassen und Pakete in Java deklarieren ihre Export- und Import-Schnittstellen implizit im Quelltext; sie müssen aus dem Programmtext „extrahiert“ werden.

SE2 – OOPM – Teil 1 220