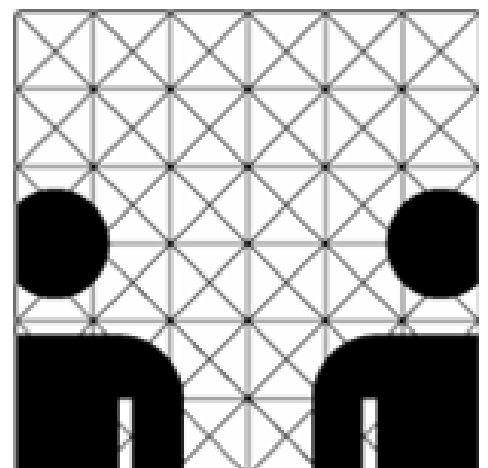


Prüfungsunterlagen
zur Lehrveranstaltung



Teil 3

Universität Hamburg
Fachbereich Informatik
SoSe 2012



Softwareentwicklung II

SE2

Objektorientierte Programmierung und Modellierung

Guido Gryczan
Axel Schmolitzky
Heinz Züllighoven
et al.

Teil 3

Verzeichnis der Folien

1. **Werte und Objekte in Programmiersprachen**
2. Literaturhinweise
3. Motivation
4. Wir erinnern uns: Menschen arbeiten mit Werkzeugen an Materialien
5. Bei genauer Betrachtung: Materialien enthalten fachlich motivierte Werte
6. These aus Sicht des Werkzeug & Material-Ansatzes
7. Mathematische Werte in der Programmierung
8. Wann verwenden wir Zahlen?
9. Was unterscheidet fachliche Werte von Zahlen?
10. Vergleich: Wert versus Objekt
11. Wert: Abstrakt
12. Objekt: Erzeugbar und zerstörbar
13. Wert: Zeitlos
14. Objekt: Potenziell veränderbar
15. Wert: Unveränderlich
16. Objekt: Zustandsbehaftet
17. Wert: Zustandslos
18. Objekt: Zur Kommunikation benutzbar
19. Objekt: Unterschied zwischen Gleichheit und Identität
20. Objekt: Potenziell kopierbar
21. Bis hierher: Wert versus Objekt
22. Zwischenergebnis
23. Objektorientierte Sprachen und Werte
24. Puristisch: Alles nur Objekte
25. Werte und Objekte in Programmiersprachen: Spektrum
26. Alles nur Objekte...
27. Von Objekten zu Typen...
28. ...und von Werten zu Typen
29. Zwischenstand: Wert, Objekt und der Typbegriff
30. Die Frage der Gleichheit
31. Gleichheit für Objekte
32. Technische vs. semantische Gleichheit
33. Aber es gibt doch benutzerdefinierte Gleichheit...
34. Gleichheit in Theorie und Praxis
35. Fazit: Wert, Objekt und der Typbegriff
36. Aufgepasst: Unveränderliche Objekte vs. Werte
37. In ferner Zukunft: benutzerdefinierte Werttypen
38. Werttypen in Java mit Wertklassen
39. Das Typesafe Enum Pattern
40. Sechs Richtlinien zu Wertklassen in Java
41. 1. Unveränderlicher Zustand
42. 2. Wertobjekte als Blätter im Objektbaum
43. 3. Keine bestehenden Objekte verändern
44. 4. Erzeugung verbergen
45. 5. equals und hashCode implementieren
46. Zusammenfassung Wert und Objekt

47. Refactoring

48. Literaturhinweise
49. Schon mal gehört: Software verändert sich
50. „Üble Gerüche“ in Software
51. Begriffsdefinition Refactoring
52. Refactoring, differenziert
53. Eine schlechte Metapher für Refactoring
54. Eine bessere Metapher für Refactoring
55. Hintergrund: Broken Window Theory
56. Übler Geruch: Code-Duplizierung
57. Übler Geruch: Große Einheiten
58. Übler Geruch: Verletzung des Law of Demeter
59. Mechanik von Refactorings
60. Mal wieder ein Zitat
61. Mechanik von „Methode umbenennen“ (vereinfacht)
62. Mechanik von „Methode extrahieren“ (vereinfacht)
63. Typen von Refactorings (Auswahl)
64. Refactorings zu Werten und Objekten
65. Werkzeugunterstützung beim Refactoring
66. Vorgehen beim Refactoring
67. Ausblick: große Refactorings
68. Zusammenfassung Refactoring

69. Formale Korrektheit, Abstrakte Datentypen

70. Literaturhinweise
71. Zuverlässigkeit als Merkmal für Softwarequalität
72. Aus SE1: Wann ist Software überhaupt „korrekt“?
73. Aus SE1: Positives und negatives Testen
74. Zuverlässigkeit als Voraussetzung der Wiederverwendbarkeit
75. Aus SE1: Ein Interface beschreibt nur Signaturen
76. Spezifikation der Semantik von Programmen
77. Ansätze der formalen Semantik (1)
78. Ansätze der formalen Semantik (2)
79. Ansätze der formalen Semantik (3)
80. Einschätzung der formalen Spezifikation von Semantik
81. Abstrakter Datentyp: ein Vorläufer des Klassenkonzepts
82. Abstrakter Datentyp: die Grundidee
83. Der Begriff "abstrakter Datentyp"
84. Formale Beschreibung eines ADT: algebraische Spezifikation der natürlichen Zahlen
85. Der Stack als ADT
86. Diskussion des Zwischenergebnisses
87. Die Liste als ADT
88. Die Schlange als ADT
89. Die Menge als ADT
90. Der Binärbaum als ADT
91. Diskussion des Zwischenergebnisses

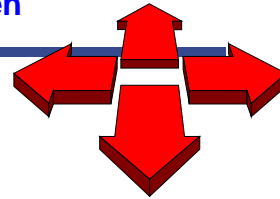
92. Metasprachliche Ansätze in der Programmierung

- 93. Literaturhinweise
- 94. Motivation
- 95. Metaobjekte
- 96. Gegenstandsbereiche "Anwendung" und "Software" in einem ausführbaren Programm
- 97. Metamodelle und Metasprachen: eine Analogie
- 98. Einteilung von Metaobjekten
- 99. Klasseninformationen oder Runtime Type Information (RTTI)
- 100. Weitere Metaobjekte zur Beschreibung von Programmelementen
- 101. Beispiel Klassenobjekt
- 102. Information über den Objektzustand
- 103. Introspektion von Objekten in Java
- 104. Veränderung des Zugriffsschutzes für Objekte in Java
- 105. Der dynamische Methodenaufruf
- 106. Der dynamische Methodenaufruf in Java
- 107. Weitergehende Konzepte des MOP
- 108. Zusammenfassung: Wesentliche Merkmale des Java MOP

109. Noch einmal: Testen objektorientierter Software

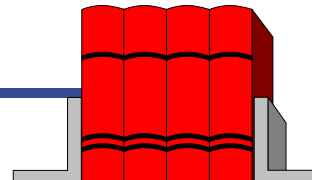
- 110. Literaturhinweise
- 111. OO-Testen: Die Klasse als kleinste Testeinheit
- 112. Probleme des oo Testens Revisited: Kapselung
- 113. Ansätze des oo Testens: Umgang mit Kapselung (1)
- 114. Ansätze des oo Testens: Umgang mit Kapselung (2)
- 115. Welche Tests legt dieses Zustandsdiagramm nahe?
- 116. Probleme des oo Testens: Vererbung (1)
- 117. Probleme des oo Testens: Vererbung (2)
- 118. Ansätze des oo Testens: Umgang mit Subtyping
- 119. Ansätze des oo Testens: Umgang mit Implementationsvererbung
- 120. Probleme des oo Testens: Polymorphie
- 121. Ansätze des oo Testens: Umgang mit Polymorphie
- 122. Tests und das Vertragsmodell
- 123. Welches Protokoll kann sinnvoll anhand dieses Zustandsmodells getestet werden?
- 124. Tests und das Vertragsmodell
- 125. Integrationstest
- 126. Weshalb ist diese zyklische Benutzt-Beziehung problematisch?
- 127. Probleme zyklischer Beziehungen und Testen
- 128. Probleme des oo Testens: Komplexe Abhängigkeiten
- 129. Ansätze des oo Testens: Umgang mit komplexen Abhängigkeiten (1)
- 130. Ansätze des oo Testens: Umgang mit komplexen Abhängigkeiten (2)
- 131. Zusammenfassung

Werte und Objekte in Programmiersprachen



- Warum Werte?
- Wie erkennen wir Werte?
- Wie unterstützen uns Programmiersprachen bei Werten?
- Wie programmieren wir Werttypen in Java?

Literaturhinweise



B.J. **MacLennan**, *Values and Objects in Programming Languages*,
ACM SIGPLAN Notices, Vol. 17, No. 12, Dec. 1982.

[Grundlage für diesen Teil]

J. **Bloch**, *Effective Java Programming Language Guide, 2nd Ed.*,
Addison Wesley, 2008.

[Kenntnisreiche Darstellung der Fußangeln von Java; ein Muss
für professionelle Java-Entwickler]

Motivation

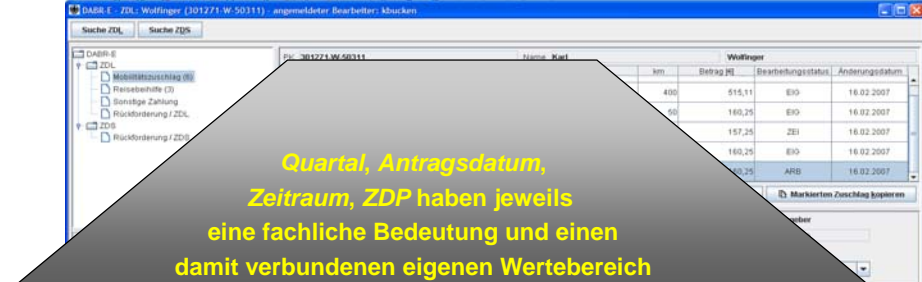
Programmieren ist Modellieren.

Wenn wir Software entwickeln, modellieren wir einen Ausschnitt aus der realen oder einer imaginären Welt.



Wir erinnern uns: Menschen arbeiten mit Werkzeugen an Materialien

Bei genauer Betrachtung: Materialien enthalten fachlich motivierte Werte



Quartal, Antragsdatum, Zeitraum, ZDP haben jeweils eine fachliche Bedeutung und einen damit verbundenen eigenen Wertebereich

Quartal	Antragsdatum	Zeitraum	ZDP
3/2005	15.10.2005	01.08.2005 - 12.10.2005	O/1234/56-001
4/2005	31.12.2005	13.11.2005 - 20.11.2005	O/76543/21-001
4/2005	31.12.2005	21.11.2005 - 31.12.2005	O/76543/21-001
4/2005	31.12.2005	13.11.2005 - 20.11.2005	O/76543/21-001
1/2007		01.10.2006 - 31.12.2006	O/76543/21-001

5

These aus Sicht des Werkzeug & Material-Ansatzes

In jedem Anwendungsbereich können wir fachlich motivierte **Werte** erkennen.

Zumindest fällt es schwer, die folgenden Phänomene als **Objekte** zu modellieren:

- einen Zeitpunkt
- einen Zeitraum
- einen Geldbetrag
- eine Kontonummer
- eine Sozialversicherungsnummer (SVN)

Zeitpunkte werden nicht erzeugt!

Wenn wir den Zeitraum für unsere Klausur „ändern“, dann wählen wir eigentlich nur einen anderen Zeitraum.

„100 Euro“ sind sogar in doppelter Hinsicht ein Wert.

Wenn ein neuer Mitbürger eine SVN bekommt, dann wird diese nicht frisch erzeugt, sondern es wird lediglich einer Person eine gültige SVN zugeordnet.

Wenn sich bei einem Bankkunden die Kontonummer „ändert“, dann ändert sich lediglich die Zuordnung zwischen Kunde und Nummer – nicht die Nummer selbst.

Mathematische Werte in der Programmierung

- Ganze Zahlen → Zweierkomplement (**int**)
- Rationale Zahlen → Gleitkommazahlen (**float**)
- Boolesche Wahrheitswerte (**boolean**)

- Sind das Objekte?
 - Wann haben wir je eine „4“ erzeugt? Können wir die „4“ verändern?
 - Wurde die Zahl Pi entdeckt oder erzeugt?
 - Wurde **true** vor **false** erzeugt? Ist das wichtig?

- Die primitiven Typen in Sprachen wie Java scheinen eindeutig eher Werte zu modellieren...

Wann verwenden wir Zahlen?

- Wir **verwenden Zahlen als eine Form von Werten**:
 - zur **Identifikation** (Bezeichnung) von modellierten Gegenständen (Bsp.: Kontonummer, Personalnummer),
 - zum **Abzählen** und **Ordnen** von Gegenständen außerhalb und im Rechner,
 - zur **Repräsentation** von messbaren **Größen**.
- Im Bereich **Naturwissenschaften und Technik**:
 - zur **numerischen Analyse**, d.h. zur Lösung mathematischer Probleme durch Operationen auf Zahlen.

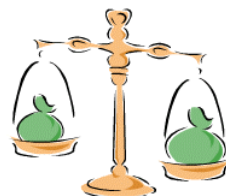
- Alle Zahlen sind Werte – aber sind alle fachlichen Werte auch Zahlen?

Was unterscheidet fachliche Werte von Zahlen?

- **Fachwerte** gehören oft zu einem Anwendungsbereich und haben eine fachliche Bedeutung.
 - Beispiele: **Postleitzahl**, **Bankleitzahl**
- Fachliche Werte haben Operationen, die oft von den allgemeinen numerischen Operationen abweichen. Beispiele:
 - **Kontonummern** werden manchmal **addiert** (Prüfsumme für Überweisungen), aber nicht **subtrahiert**.
 - **Geldbeträge** können wir **addieren** u. **subtrahieren**, aber nicht miteinander **multiplizieren**.
- Nicht alle fachlichen Werte lassen sich geeignet durch Zahlen modellieren – wie beispielsweise die Grundfarben: { **Rot**, **Grün**, **Blau** }.
- In **Programmen** sollten (programmiersprachliche) Werte bestimmte Eigenschaften besitzen, die sie klar von **Objekten** unterscheiden.

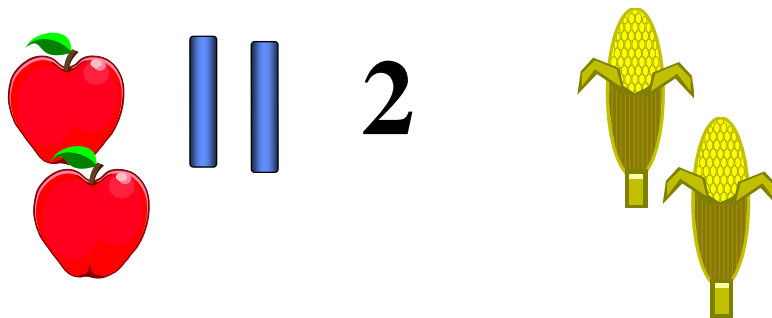
Vergleich: Wert versus Objekt

- Auf den folgenden Folien stellen wir die Begriffe **Wert** und **Objekt** einander gegenüber:
 - Was sind die zentralen Eigenschaften von Werten und Objekten?
 - Worin unterscheiden sich Objekte und Werte voneinander?
- Ziel ist ein klareres Verständnis dieser fundamentalen Begriffe.



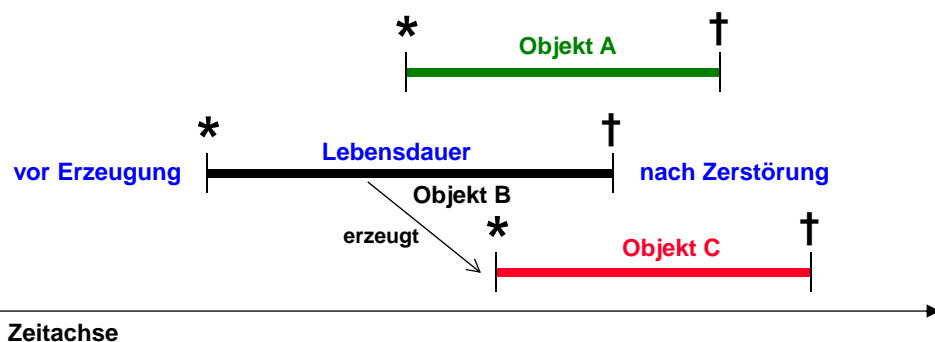
Wert: Abstrakt

- Ein **Wert** ist **abstrakt**:
 - Werte sind immer immateriell.
 - Werte abstrahieren von **konkreten Kontexten**.
 - Fachliche Werte** sind häufig Abstraktionen von Dingen, um diese zu identifizieren (Kontonummer, Postleitzahl).



Objekt: Erzeugbar und zerstörbar

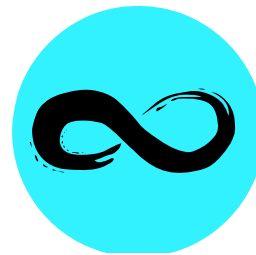
- Ein **Objekt** kann **erzeugt** und **zerstört** werden:
 - Da Objekte in der Zeit existieren, haben sie einen **zeitlichen Anfang** und ein **Ende**, also eine **Lebensdauer**.
 - Aus Sicht eines Objektes sind drei Zeitabschnitte relevant: Vor seiner Erzeugung, während seiner Lebenszeit und nach seiner Zerstörung.



Wert: Zeitlos

- Ein **Wert** ist **zeitlos**:
 - Begriffe wie **Zeit** und **Dauer** sind nicht anwendbar.
 - Werte werden nicht **erzeugt** oder **zerstört**.
 - In Ausdrücken **entstehen** keine Werte und sie werden nicht **verbraucht**.

$$40 + 2 = 42$$

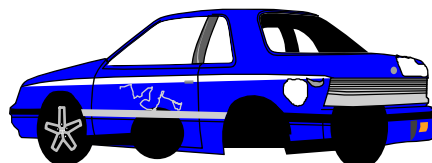


Objekt: Potenziell veränderbar

- Ein **Objekt** kann **veränderbar** sein, d.h.:
 - die **erkennbaren Merkmale** können sich ändern,
 - trotz **Veränderung** bleibt die **Identität** erhalten.



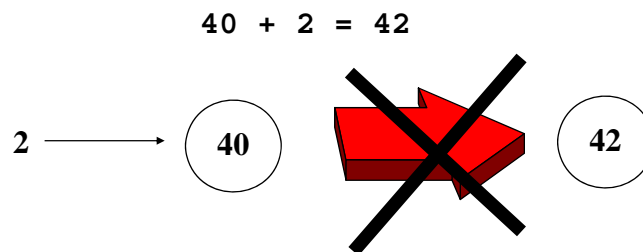
Mein Auto 1995



Mein Auto 2004

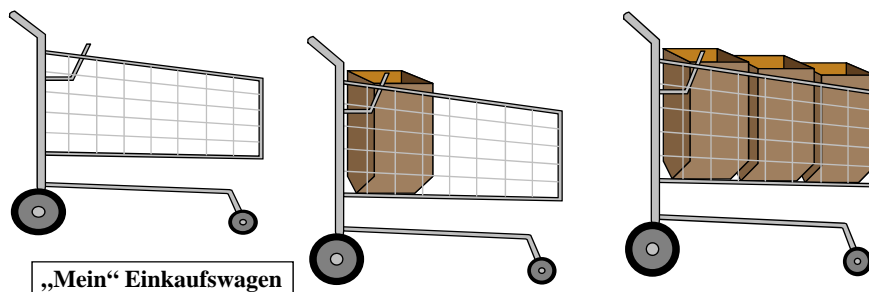
Wert: Unveränderlich

- Ein **Wert** ist **unveränderlich**:
 - Er kann **berechnet** und **auf andere Werte bezogen** werden, aber **nicht verändert** werden.
 - Funktionen** können (applikativ) auf Werte angewandt werden, um **andere Werte** zu berechnen.



Objekt: Zustandsbehaftet

- Ein **Objekt** hat einen (inneren) **Zustand**:
 - Veränderung** eines Objekts bedeutet die Veränderung seines **Zustandes**.
 - Veränderungen** finden „in der **Zeit**“ statt.



Es gibt auch **unveränderliche Objekte**:
Diese erhalten ihren Zustand einmalig
bei ihrer Erzeugung.

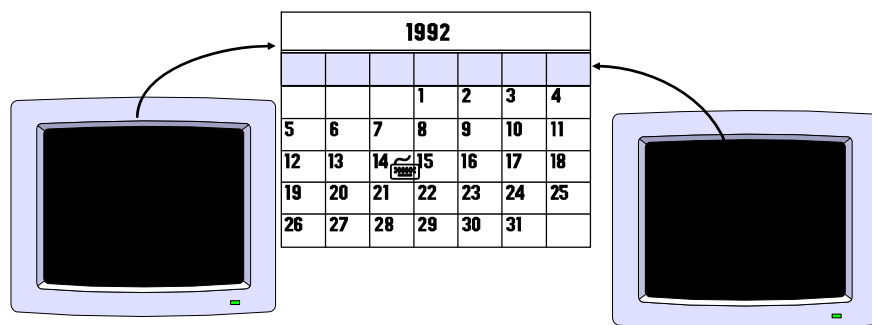
Wert: Zustandslos

- Ein Wert ist **zustandslos**:
 - Operationen berechnen Werte; sie **verändern** sie aber **nicht**.
 - Da der Zeitbegriff auf Werte nicht zutrifft, können sie sich auch nicht „mit der Zeit“ verändern.
 - Zustände lassen sich zwar mit Hilfe von Werten modellieren (Kontostand), aber ein Wert selbst hat keinen Zustand.



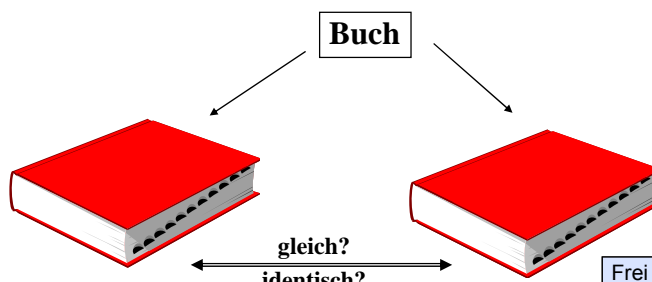
Objekt: Zur Kommunikation benutzbar

- Ein **Objekt** kann zur **Kommunikation** und **Kooperation** benutzt werden:
 - **Verändert** ein Benutzer den Zustand eines Objekts, ist dies für andere Benutzer erkennbar.
 - Dieser „**Seiteneffekt**“ kann gewünscht oder problematisch sein.



Objekt: Unterschied zwischen Gleichheit und Identität

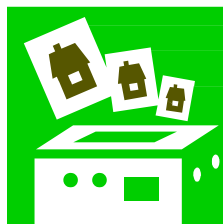
- Zwei **gleiche** Objekte sind nicht notwendig **identisch**:
 - **Gleichheit** bezieht sich auf die **erkennbaren Merkmale**,
 - **Identität** auf die (äußeren) **Zusammenhänge** eines Objekts (Position in Raum und Zeit).



Frei nach Frege:
Identität ist die Beziehung, in der ein Objekt mit sich selbst und mit keinem anderen steht.

Objekt: Potenziell kopierbar

- Begriffe wie „**Original**“ und „**Kopie**“ sind bei Objekten fachlich oft sinnvoll.



- Von Werten gibt es **keine Kopien**.
- Der Begriff **Anzahl** ist auf einen Wert nicht anwendbar.

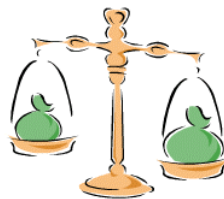
Bis hierher: Wert versus Objekt

- Ein **Wert** ist:

- abstrakt
- zeitlos
- unveränderlich

- Ein **Objekt** ist:

- zustandsbehaftet (potenziell veränderbar)
- erzeugbar und zerstörbar (existent in Raum und Zeit)



Zwischenergebnis

- **Werte** und **Objekte** sind **Grundkonzepte** der Softwareentwicklung.
- Das Verständnis dieser Konzepte hilft beim **Entwurf** und bei der **Konstruktion** von Software.
- Wir sollten beim Modellieren eines Anwendungsbereichs die **Unterschiede** zwischen Werten und Objekten sehr gut kennen.
- Die **Programmiersprache** als wichtigstes Software-Werkzeug sollte uns dabei **unterstützen, sowohl Werte als auch Objekte** in unseren Programmen geeignet abzubilden.
- Frage: Wie gut machen die aktuell verwendeten **objektorientierten Sprachen** dies?



Objektorientierte Sprachen und Werte

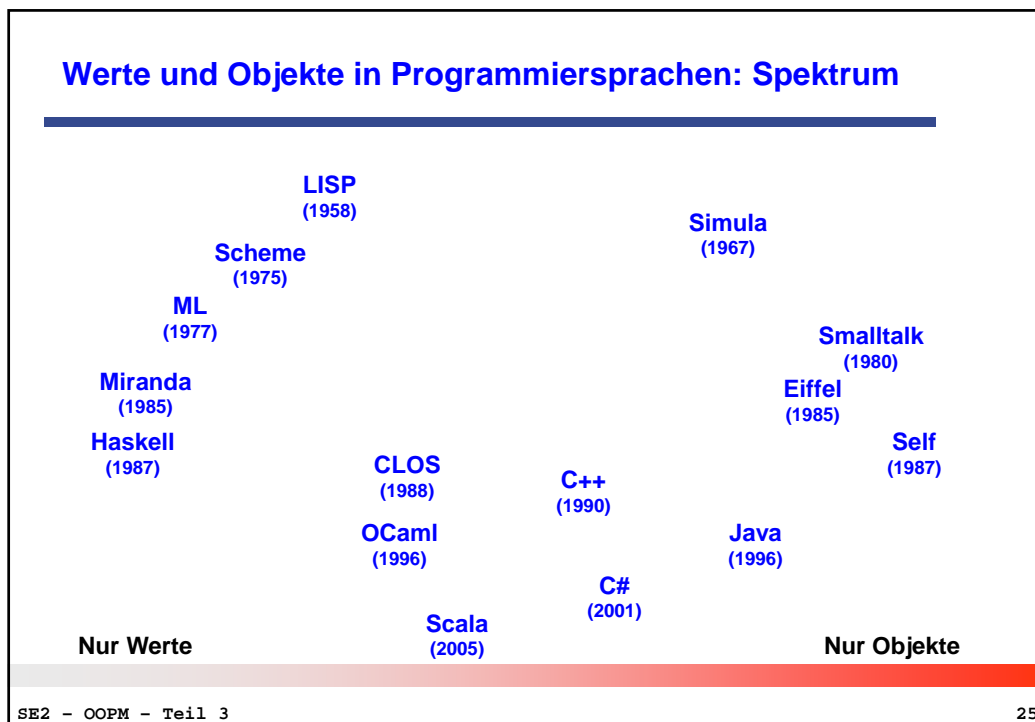
- Objektorientierte Sprachen erlauben mit dem Klassenkonstrukt die Definition beliebiger **Objekttypen**.
 - Da mit dem Schlüsselwort **class** ein neuer Typ konstruiert wird, sprechen wir auch von einem **Typkonstruktor**.
- Bei den **Werttypen** hingegen ist die Menge innerhalb der Sprache üblicherweise festgelegt.
 - **Beispiel Java**: `int, byte, short, long, float, double, boolean, char`
- Wenn in vielen Anwendungszusammenhängen fachlich motivierte Werttypen nützlich sind, warum bieten objektorientierte Sprachen dann so wenig Unterstützung für benutzerdefinierte Werte?
 - Vermutlich wegen des Strebens nach **Purismus**...

Puristisch: Alles nur Objekte

„Pure“ objektorientierte Programmiersprachen:

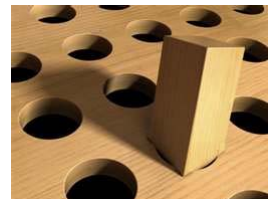
- **Smalltalk**
 - Maxime: **Everything is an object!** Klassen, Werte, Blöcke...
- **Eiffel**
 - Zumindest zur Laufzeit nur Objekte
- **Self**
 - Keine Klassen, nur noch Objekte
 - neue Objekte entstehen durch Klonen





Alles nur Objekte...

- ... was spricht dafür?
 - Einfaches Sprachmodell
 - Konsistente Semantik (insbesondere bei Variablen)
 - Einheitliche Generizität
 - ...
- Dies sind primär Argumente für die Sprachdesigner, aber nicht unbedingt für Programmierer; auf keinen Fall jedoch für Modellierer.
-
- **Wenn wir gezwungen werden, alle relevanten Abstraktionen eines Anwendungsbereiches mit Objekten zu modellieren, dann verbiegen wir uns bei den Werten!**



Von Objekten zu Typen...

- In objektorientierten Programmiersprachen modellieren wir **Objekte**, indem wir **Klassen** definieren, von denen Exemplare erzeugt werden können.
- Jede Klasse definiert einen **Typ**, die Operationen der Klasse sind auch die **Operationen** des Typs.
- Die Exemplare einer Klasse bilden (extensional) die **Elementmenge** ihres **Objekttyps**.
- Wir nennen diese Menge explizit Elementmenge, da der sonst übliche Begriff „Wertemenge“ gerade bei der Diskussion über Werte und Objekte verwirrend sein kann. Sowohl Werte als auch Objekte bezeichnen wir im folgenden als **Elemente** ihres Typs.

Wikipedia: Die **Extension** eines **Begriffs** (z.B. „Mensch“) [...] ist sein **Umfang**, das heißt die **Menge** aller Objekte („Erfüllungsgegenstände“), die unter diesen Begriff fallen.

...und von Werten zu Typen



- **Werte** wie die ganzen Zahlen oder die Wahrheitswerte werden in Programmiersprachen durch vordefinierte **Werttypen** (Java: **int**, **boolean**) modelliert.
- Die **Elementmenge** dieser Typen ist **unveränderlich**: Der Werttyp **int** verfügt über eine Wertemenge mit 2^{32} Elementen, während **boolean** zwei Elemente hat.
- Wir stellen verallgemeinernd fest: Wenn Werte konzeptuell nicht erzeugt und vernichtet werden, dann ist zwangsläufig die **Elementmenge bei Werttypen allgemein unveränderlich**.
- Wir sprechen deshalb bei Werttypen ungern von „Konstruktoren“, denn Werte werden ja nicht erzeugt; stattdessen nennen wir Operationen, die uns die Werte eines Werttyps liefern, **Selektoren**. Sie selektieren einen Wert aus der Menge der bestehenden Werte.

Zwischenstand: Wert, Objekt und der Typbegriff

- Ein **Wert** ist:
 - abstrakt
 - zeitlos
 - unveränderlich

- Ein **Objekt** ist:
 - zustandsbehaftet (potenziell veränderbar)
 - erzeugbar und zerstörbar (existent in Raum und Zeit)

↓ in Programmiersprachen definiert über ↓

- **Werttypen:**
 - haben eine **unveränderliche Elementmenge**
 - bieten **Selektoren** an

- **Objekttypen:**
 - haben eine **dynamische Elementmenge**
 - bieten **Konstruktoren** an

- **Gemeinsam:**
 - Typ = Wertemenge + Operationen
 - Variablen des Typs mit veränderbarer Belegung

Die Frage der Gleichheit

- Beim **Gleichheitsbegriff** bestehen in Programmiersprachen deutliche Unterschiede zwischen Werten und Objekten.
- Beispiel Java: Was bedeutet `a == b` ?
- Betrachte:

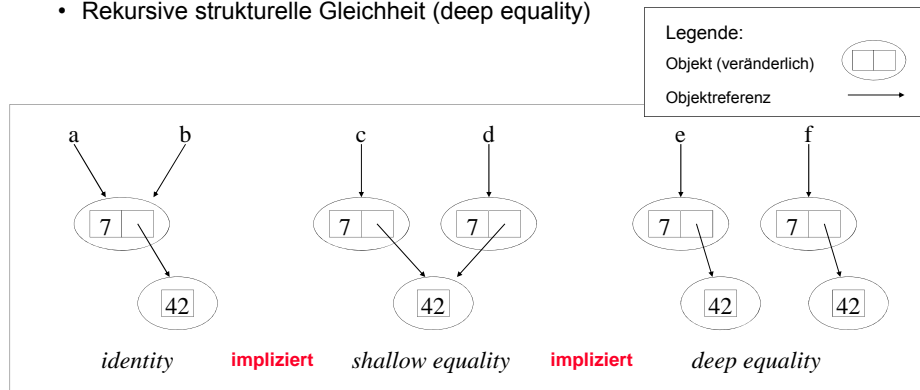

```
a = b;
if (a == b) { ... }
a = 42;
b = 42;
if (a == b) { ... }
```
- Wir würden uns wünschen, dass `a` und `b` gleich sind!
- Was ist dann mit Strings?
- Betrachte:


```
name = "Elling";
if (name.toLowerCase() == "elling") {
    ...
}
```

Gleichheit für Objekte

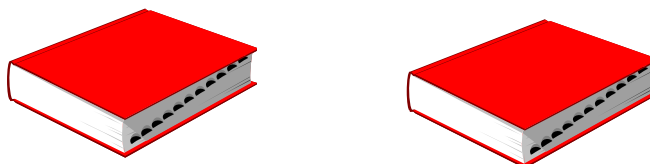
Technisch sind drei Formen unterscheidbar:

- Referenzgleichheit, Identität (identity)
- Einfache strukturelle Gleichheit (shallow equality)
- Rekursive strukturelle Gleichheit (deep equality)



Technische vs. semantische Gleichheit

- Alles einfach zu implementieren, aber nicht ausreichend.
- Betrachte:
Bibliotheksbuch:
Titel Autor ISBN RefNummer
 - kann als gleich angesehen werden, auch wenn die Referenznummer unterschiedlich ist...
- Gebraucht wird deshalb häufig auch ein Konzept von **semantischer Gleichheit**.



Aber es gibt doch benutzerdefinierte Gleichheit...

- ... wie beispielsweise über das `equals` in Java. Damit können wir für jeden Objekttyp definieren, wann Objekte als gleich gelten sollen!
- Ist das die Lösung?
- Die Prüfung auf Gleichheit ist dann ein Aufruf einer Operation:
 - asymmetrisch
 - Vergleich mit `null`?
 - Welchen Typ hat der Parameter?
 - `Object` wie in Java? Das führt zu Downcasts...
 - `likeCurrent` wie in Eiffel? Ist nicht statisch typsicher...

Gleichheit in Theorie und Praxis

Mathematisch formuliert:

- Reflexiv
 $a = a$ (Identität impliziert Gleichheit)
- Symmetrisch
 $a = b \Rightarrow b = a$
- Transitiv
 $a = b \wedge b = c \Rightarrow a = c$

Wir halten fest:

- Bei Objekten besteht ein **Unterschied** zwischen **Gleichheit** und **Identität**; bei Werten hingegen ist diese Unterscheidung unüblich.

Transitivität seit Java 1.5...

Auto-Boxing und -Unboxing:

```
Integer a = 128;  
int b = 128;  
Integer c = 128;
```

Es gilt:

```
a == b
```

```
b == c
```

Aber:

```
a != c
```

Upps!

Fazit: Wert, Objekt und der Typbegriff

- Ein **Wert** ist:

- abstrakt
- zeitlos
- unveränderlich

- Ein **Objekt** ist:

- zustandsbehaftet (potenziell veränderbar)
- erzeugbar und zerstörbar (existent in Raum und Zeit)



in Programmiersprachen definiert über



- **Werttypen:**

- haben eine unveränderliche Elementmenge
- bieten Selektoren an

- **Objekttypen:**

- haben eine dynamische Elementmenge
- bieten Konstruktoren an
- unterscheiden Gleichheit und Identität

- **Gemeinsam:**

- Typ = Wertemenge + Operationen
- Variablen des Typs mit veränderbarer Belegung

Aufgepasst: Unveränderliche Objekte vs. Werte

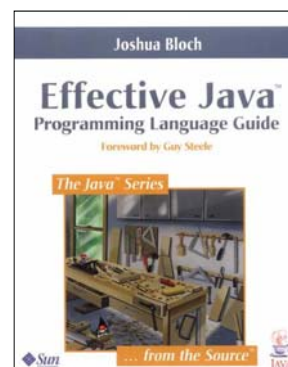
- In vielen Anwendungskontexten ist es sinnvoll, **unveränderliche Objekte** zu definieren.
- Beispiel: Bei einem Film in einer Filmdatenbank sind **Titel**, **Regisseur** und **Filmlänge** unveränderlich, ein Film kann somit durch eine Objektklasse **Film** definiert werden, deren Exemplare diese Eigenschaften bei der Erzeugung eines Filmobjektes übergeben bekommen und sie anschließend nur über lesende Operationen zur Verfügung stellen.
- Wenn ein **Objekt ausschließlich sondierende Operationen** anbietet – ist es dann nicht quasi ein Wert?
- Die Antwort lautet **Nein**: Denn für einen Wert müssen **alle konzeptionellen Eigenschaften** zutreffen. Ein Film ist sicher etwas Abstraktes und kann auch als etwas Unveränderliches angesehen werden; aber ein Film ist nicht zeitlos, denn es gibt einen Zeitpunkt, bis zu dem der Film nicht existierte und ab dem der Film dann (konzeptionell) für immer existiert.

In ferner Zukunft: benutzerdefinierte Werttypen

- Zukünftige objektorientierte Programmiersprachen sollten ein dediziertes **Konstrukt** anbieten, mit dem eigene **Werttypen** definiert werden können.
- Anforderungen an die **Werttypen**, die durch einen solchen Typkonstruktor definiert werden können, sind nach der vorangegangenen Diskussion:
 - Ihre **Wertemenge** sollte **fixiert** sein; da Werte nicht erzeugt und zerstört werden können, kann es beispielsweise keine Konstruktoren für Werte geben.
 - Ihre **Elemente** müssen **unveränderlich** sein (am besten garantiert durch den Compiler).
 - Ihre **Operationen** sollten **referentiell transparent** sein und **keine beobachtbaren Seiteneffekte** zeigen.
 - Sie sollten **ausschließlich auf der Basis anderer Werttypen definiert** werden (Werte können sich nicht auf Objekte beziehen).

Werttypen in Java mit Wertklassen

- Da Java keine explizite Unterstützung für benutzerdefinierte Werttypen bietet, müssen wir mit den vorhandenen „Bordmitteln“ auskommen.
- Wir müssen deshalb Werttypen **mit Objektklassen definieren** (im Folgenden **Wertklassen** genannt) und dabei darauf achten, dass die Werteigenschaften eingehalten werden.
- Es gibt etliche **Programmiermuster**, die uns dabei unterstützen.
- Prominentes Beispiel:
 - Das **Typesafe Enum Pattern** von Bloch.



Das Typesafe Enum Pattern

- Beschreibt, wie **Aufzählungstypen** typsicher in Java modelliert werden können.

Grundmuster:

- Biete ein **public static final** Feld für jede Konstante der Aufzählung.
- Verstecke den Konstruktor.

```
public class Farbe {
    private final String _name;

    private Farbe(String name) { _name = name; }

    public String toString() { return _name; }

    public static final Farbe ROT = new Farbe("rot");
    public static final Farbe GRUEN = new Farbe("gruen");
    public static final Farbe BLAU = new Farbe("blau");
    ...
}
```

- Ist inzwischen in die Sprache eingeflossen (seit Java 1.5): Hinter den Kulissen der **Enums** in Java wird dieses Muster realisiert.
- Die Enums in Java sind allerdings keine „waschechten“ Werttypen, da an den Exemplaren eines Enums ein **veränderlicher Zustand** modelliert werden kann.
- Es entstehen **Wertobjekte** – fachlich motivierte Werte, die technisch durch Objekte repräsentiert werden müssen.

Sechs Richtlinien zu Wertklassen in Java

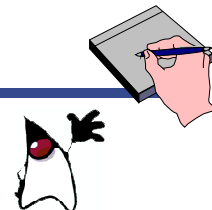
Konstruktion von Wertklassen

Stelle sicher,

1. dass Wertobjekte keinen veränderbaren Zustand haben;
2. dass Werte und Wertobjekte sich nicht auf Objekte beziehen;
3. dass im Quelltext einer Wertklasse keine (bestehenden) Objekte verändert werden.
4. Verberge die (technisch notwendige) Erzeugung von Wertobjekten.
5. Implementiere **equals** und dazu passend **hashCode**.

Benutzung von Wertklassen

6. Verwende immer **equals** statt **==** bei der Prüfung auf Gleichheit zweier Werte.



1. Unveränderlicher Zustand

- Die Wertobjekte von Wertklassen sollten unveränderlich sein.
- **Sprachunterstützung**
 - Mit dem Schlüsselwort **final** kann für **Zustandsfelder** festgelegt werden, dass sie unveränderlich sein sollen.
 - Einem Zustandsfeld, das als **final** gekennzeichnet ist, darf nur innerhalb des Konstruktors ein Wert zugewiesen werden.
 - Mit dem Schlüsselwort **final** kann für die gesamte **Wertklasse** festgelegt werden, dass es keine Subklassen geben darf.
 - Auf diese Weise wird verhindert, dass Unterklassen mit veränderlichem Zustand definiert werden können, deren Exemplare polymorph verwendet werden könnten.



2. Wertobjekte als Blätter im Objektbaum

- Die primitiven Werte in Java bilden bereits die Blätter im Objektbaum. Darüber hinaus sollten Wertobjekte keine Referenzen auf (echte) Objekte enthalten.
- **Sprachunterstützung**
 - keine
- **Selbst beachten**
 - In einer Wertklasse sollten die Typen aller Zustandsfelder nur Werttypen (primitive Typen oder Wertklassen) sein.
- String kann als eine Wertklasse angesehen werden.



3. Keine bestehenden Objekte verändern

- Wertobjekte sollten keine (echten) Objekte verändern.
- **Sprachunterstützung**
 - keine
- **Selbst beachten**
 - Im Quelltext einer Wertklasse sollten keine (bereits bestehenden) Objekte verändert werden.
 - In der Schnittstelle einer Wertklasse sollten deshalb keine Objekttypen als Parametertypen erscheinen.
 - Pragmatisch kann zugelassen werden, dass in einer Methode lokal Objekte erzeugt werden (**StringBuffer**, **Formatter**, etc.).



4. Erzeugung verbergen

- Die Kontrolle über die Erzeugung von Exemplaren einer Wertklasse sollte in der Wertklasse selbst liegen.
- **Sprachunterstützung**
 - Die Konstruktoren einer Klasse können als **private** deklariert werden. Klienten können dann nicht mehr direkt Exemplare der Klasse erzeugen.
 - Als Ersatz kann eine Klasse **Fabrikmethoden** anbieten: Klassenmethoden (mit **static** deklariert), die Exemplare der Wertklasse liefern. Diese Methoden dienen dann als **Selektoren**.
- Auf diese Weise ist es teilweise möglich, auf Regel 6 zu verzichten:
 - Wenn in der Wertklasse garantiert wird, dass für jeden Wert nur ein Wertobjekt erzeugt wird, dann können Klienten Referenzgleichheit verwenden (siehe die Enumerations in Java).



5. equals und hashCode implementieren

- Die Gleichheit für Wertobjekte sollte explizit definiert werden. Da verschiedene Wertobjekte denselben Wert repräsentieren können (keine Referenzgleichheit), muss die Gleichheit mit der Operation **equals** definiert werden.

- Sprachunterstützung**

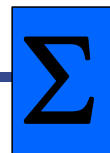
- keine



- Selbst beachten**

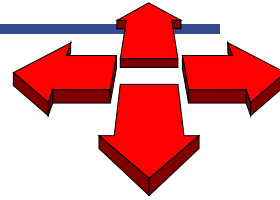
- Der Vertrag des Typs **Object** für seine Operationen **equals** und **hashCode** muss eingehalten. U.a. gilt:
 - Zwei Wertobjekte, die als gleich gelten, müssen auch denselben Wert als Hash-Code liefern.

Zusammenfassung Wert und Objekt



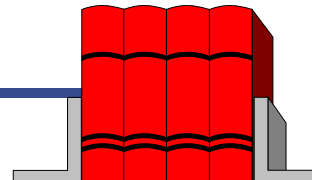
- Werte** und **Objekte** sind fundamental verschieden.
- Zustandsbehaftete, vergängliche Gegenstände und Konzepte** lassen sich gut als **Objekte** modellieren.
- Zeitlose und unveränderliche Größen** wie Zahlen und Zeiträume werden sinnvoll als **Werte** dargestellt.
- Fachliche Werte** spielen in vielen Anwendungsbereichen eine große Rolle. Sie haben oft Operationen, die nicht den mathematischen Operationen auf Zahlen entsprechen.
- Objektorientierte Programmiersprachen** stellen neben Objekten auch elementare Werte zur Verfügung. Sie bieten aber keine einfachen Möglichkeiten, **benutzerdefinierte fachliche Werte** einzuführen.

Refactoring



- Motivation: Software Aging
- Definition des Begriffs Refactoring
- „Üble Gerüche“ in Software
- Mechanik von Refactorings
- Werkzeugunterstützung für Refactorings
- Ausblick: große Refactorings

Literaturhinweise



- Martin **Fowler**, *Refactoring – Improving the Design of Existing Code*.
430 Seiten - Addison-Wesley, 2000.
[Das Standard-Buch über Refactoring]
- Stefan **Roock**, Martin **Lippert**, *Refactorings in großen Softwareprojekten*. 348 Seiten - dpunkt Verlag 2004.
[Weiterführend: große Refactorings, DB-Refactorings, u.ä.]

Schon mal gehört: Software verändert sich

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

© Barnes, Kölling



**„Any fool can write code that a computer can understand.
Good programmers write code that humans can understand.“**
(Fowler, 2000)

„Üble Gerüche“ in Software

- Software, die sich ändert, muss auch intern restrukturiert werden.
- Ein Entwurf, der heute gut war, ist morgen aufgrund neuer Anforderungen eventuell nicht mehr angemessen.
- Ein Entwurf, der uns Änderungen erschwert, sollte überdacht werden.
- Wann werden Änderungen erschwert? Hinweise sind so genannte **„üble Gerüche“** (engl.: bad smells):
 - Eine Methode ist zu lang, um sie auf dem Bildschirm überblicken zu können.
 - Dieselbe Fallunterscheidung wird an mehreren Stellen in einem System vorgenommen.
- Generell sind Verstöße gegen bekannte Entwurfsregeln meist Kandidaten für üble Gerüche.

Begriffsdefinition Refactoring



Refactoring bedeutet, die interne Struktur einer Software zu verbessern, ohne ihr beobachtbares Verhalten zu verändern.

Was ist Refactoring demnach **nicht**?

- Das Einfügen zusätzlicher Funktionalität ist kein Refactoring.
- Das Beheben von Fehlern ist kein Refactoring.
- Auch eine Änderung am Layout der GUI ohne Änderung der Funktionalität ist kein Refactoring.

Refactoring, differenziert

Wir differenzieren den Begriff **Refactoring** etwas genauer:

- Als Bezeichnung für die **eine bestimmte Restrukturierung** eines Systems:
 - Konkreter Zusammenhang, konkrete Tätigkeit
 - Bsp.: „Umbenennen der Methode **fuegeEin** der Klasse **Liste** in **einfüegen**“, „Extrahieren des Interfaces **Melder** aus der Klasse **Hauptfenster**“
- Als Bezeichnung für eine **Klasse von Restrukturierungen**
 - Abstrakte Beschreibung, katalogisierbar
 - Bsp.: **Methode extrahieren** (engl.: Extract Method), **Interface extrahieren** (Extract Interface)
- Als Bezeichnung der **Tätigkeit des Restrukturierens**:
 - Im Sinne von „Beim Refactoring (sprich: beim Restrukturieren der Klassen) haben wir festgestellt, dass wir noch einige Leichen im Keller haben.“

Eine schlechte Metapher für Refactoring

Nicht gut: Refactoring ist wie „Bilder aufhängen“ (Dekorieren)

- „Unser Quelltext ist wie eine Wohnung. Wir wollen uns als Entwickler wohl fühlen und den Quelltext gut lesen und warten können. Wir halten deshalb Namenskonventionen ein, halten unsere Methoden kurz und die Klassen übersichtlich. Wir hübschen alles auf, damit es schöner ist. Bilder machen einen Raum freundlicher.“
- Diese Metapher wird dem Anspruch von Refactoring nicht gerecht; es geht nicht um Schönheitspreise oder Eleganz, sondern um Notwendigkeit.



Eine bessere Metapher für Refactoring

Refactoring ist wie „Müll raustragen“ (Aufräumen)

- „Unser Quelltext ist wie eine Küche. Wie in einer Küche verrichten wir sehr viele alltägliche Dinge an unserem Quelltext. Wir kochen, indem wir ändern, anpassen, erweitern. In der Hitze des Gefechts räumen wir nicht immer alles gleich weg, es bleibt eine Menge liegen. So entsteht mit der Zeit viel Müll. Wenn wir den Müll nicht wegräumen, werden wir mit der Zeit keinen Spaß mehr in der Küche haben und in unserem eigenen Müll ersticken.“



Hintergrund: Broken Window Theory

- Nach der Broken-Window-Theory zieht Schmutz und Zerstörung weitere Verschmutzung und Zerstörungen an. In einem Versuch in den USA wurde dazu ein intaktes Auto mehrere Wochen auf einem Parkplatz abgestellt. Der Wagen wurde nicht beschädigt. Dann hat man gezielt eine Scheibe des Autos eingeschlagen. Diese kaputte Scheibe zog weitere Beschädigungen an: in sehr kurzer Zeit wurden die anderen Scheiben auch eingeschlagen, die Reifen gestohlen etc.
- In New York werden daher Graffitis sofort entfernt, wenn sie entdeckt werden. Man geht davon aus, dass es Sprayern leichter fällt, eine Wand mit einem Graffiti zu „verschönern“, wenn sie nicht die ersten sind.



SE2 – OOPM – Teil 3

55

Übler Geruch: Code-Duplizierung

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

SE2 – OOPM – Teil 3

56

Übler Geruch: Große Einheiten

- Ein bekanntes Anti-Pattern - die „**Gott-Klasse**“:
 - Sie ist für alles zuständig und hält 90% des Quelltextes.
- Methoden, die **mehrere Bildschirmseiten** lang sind, sind schlecht wartbar.
- Klassen mit mehreren hundert Methoden oder Exemplarvariablen sind meist zu groß!

Übler Geruch: Verletzung des Law of Demeter

Verletzung des Law of Demeter:

```
student.getRecord().getExamEntry("SE2/2006").addResult(93);
```

besser:

```
sc = new ScoreCard("SE2/2006");
```

...

<Eintragen der Noten in die Score-Card, bspw. interaktiv>

...

```
student.addScore(sc);
```

Mechanik von Refactorings

- Als die **Mechanik** (engl.: mechanics) eines Refactorings versteht man „eine **knappe und präzise schrittweise Beschreibung, wie das Refactoring ausgeführt werden soll.**“
- Der Anspruch bei den Schritten dieser Beschreibung ist meist, dass das System **nach jedem Schritt übersetzbar** ist.
- Die Schritte sollen deshalb **möglichst klein** gewählt werden.
- Bei Fowler sind bei jedem Refactoring die Mechanics angegeben.
- Schritte einer solchen Mechanik können beispielsweise sein:
 - „Erzeuge eine neue Klasse mit einem bestimmten Namen.“
 - „Füge eine neue Methode hinzu.“
 - „Kopiere den Quelltext einer Methode in eine andere.“
 - „Entferne eine Methode.“



Mal wieder ein Zitat

- „*Some [...] of the refactorings [...] may seem obvious. But don't be fooled. Understanding the mechanics of such refactorings is the key to refactoring in a disciplined way.*“

Erich Gamma, aus dem Vorwort zu Fowlers Buch



Mechanik von „Methode umbenennen“ (vereinfacht)

Ein häufig benutztes Refactoring ist „**Methode umbenennen**“ (engl.: rename method).

Mechanik-Beschreibung:

- Deklare eine neue Methode mit dem neuen Namen. Kopiere den alten Quelltext in die neue Methode und mache die notwendigen Anpassungen.
- **Übersetze.**
- Ändere den Rumpf der alten Methode so, dass er die neue Methode ruft.
- **Übersetze und teste.**
- Finde alle Referenzen auf die alte Methode und ändere sie so, dass sie auf die neue verweisen. **Übersetze und teste nach jeder Änderung.**
- Entferne die alte Methode.
- **Übersetze und teste.**

Vereinfachung hier: keine Berücksichtigung von Ober- und Unterklassen

Mechanik von „Methode extrahieren“ (vereinfacht)

Ein Refactoring zum Zergliedern von großen Methoden ist „**Methode extrahieren**“ (engl.: extract method). Dabei wird ein Teil einer Methode in eine neue Methode ausgelagert.

Mechanik-Beschreibung:

- Deklare eine neue Methode mit einem sprechenden Namen.
- Kopiere den zu extrahierenden Abschnitt in die neue Methode.
- Suche in der neuen Methode nach Referenzen auf lokale Variablen der alten Methode. Mache diese entweder zu Parametern oder zu lokalen Variablen der neuen Methode.
- **Übersetze und teste.**
- Ersetze in der alten Methode den extrahierten Quelltext mit dem Aufruf der neuen Methode.
- **Übersetze und teste.**

Typen von Refactorings (Auswahl)

Es gibt eine ganze Reihe von etablierten Refactorings (mit unterschiedlicher Komplexität):

- **Rename <Element>**: Ein Element (Paket, Klasse, Methode, Variable, Konstante) im Quelltext wird umbenannt.
- **Move <Element>**: Ein Element im Quelltext wird verschoben.
- **Change Signature**: Eine Methodensignatur wird um einen Parameter ergänzt/reduziert.
- **Bewege** Methode in Super-/Subtyp
- **Extrahiere** ein Interface, eine Methode, ...
- **Ersetze** Subtyp durch Supertyp, Vererbung durch Delegation, ...

Refactorings zu Werten und Objekten

- Fowler beschreibt auch zwei Refactorings zum Thema **Werte und Objekte**:
 - "Change Value to Reference"
 - "Change Reference to Value"
- Auszug aus "**Change Value to Reference**":
 - "You can make a useful classification of objects in many systems: reference objects and value objects. **Reference object** are things like customer or account. Each object stands for one object in the real world, and you use the object identity to test whether they are equal. **Value objects** are things like date or money. They are defined entirely through their data values. You don't mind that copies exist; you may have hundreds of "1/1/2000" objects around your system. You do need to tell whether two of the objects are equal, so you need to override the equals methods (and the hashCode method too).

The decision between reference and value is not always clear. ..."

Werkzeugunterstützung beim Refactoring

- Aufgrund der starken Mechanisierung von Refactorings ist häufig eine Unterstützung durch Werkzeuge möglich.
- Seit einigen Jahren bieten Entwicklungsumgebungen wie Eclipse oder IntelliJ IDEA umfangreiche Unterstützung an. Diese wird kontinuierlich weiter entwickelt.
- Der Grad der Unterstützung ist ebenfalls von der Programmiersprache abhängig:
 - Bei Java ist aufgrund des einfacheren Sprachmodells eine bessere Unterstützung möglich als beispielsweise in C++.



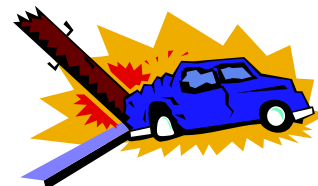
SE2 – OOPM – Teil 3

65

Vorgehen beim Refactoring

Aus (Fowler 2000), S. 7+8:

- „When you find you have to add a feature to a program, and the program's code is not structured in a convenient way to add the feature, first refactor the program to make it easy to add the feature, then add the feature.
- Before you start refactoring, **check that you have a solid suite of tests**. These tests must be self-checking.“
- Beim Refactoring besteht immer die Gefahr, dass wir neue Fehler in den Quelltext einbauen.
- Metapher: Refactoring ohne ausreichende Testabdeckung ist wie Autofahren ohne Sicherheitsgurt!



SE2 – OOPM – Teil 3

66

Ausblick: große Refactorings

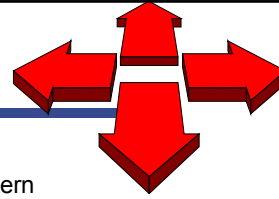
- Die Definition des Begriffs Refactoring sagt nichts über den **Umfang** der Umstrukturierungen:
 - Automatisierte Refactorings (werkzeugunterstützt) werden meist innerhalb von Sekunden ausgeführt.
 - Bei händischen Refactorings gilt: Ein Entwickler(-paar) sollte ein konkretes Refactoring innerhalb von zwei bis vier Stunden durchführen können, damit der Rest des Teams die Zwischenstände gar nicht erst zu sehen bekommt.
- In großen Projekten kann es jedoch vorkommen, dass ein **Refactoring über mehrere Tage** oder sogar **Wochen** durchgeführt werden muss.
 - Beispiel: Die Typhierarchie der fachlichen Materialien muss vollständig reorganisiert werden.
- In solchen Fällen müssen **Refactoring-Pläne** erstellt werden, die allen Projektmitgliedern kommunizieren, was bereits geändert wurde und was noch geplant ist. Sonst sind die Baustellen teilweise nicht nachvollziehbar.

Zusammenfassung Refactoring



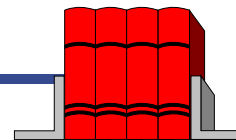
- Software wird fortlaufend verändert; dadurch entstehen leicht „üble Gerüche“.
- Ein Refactoring ist eine Restrukturierung von Software, die deren Funktionalität erhält.
- Es gibt eine Reihe von etablierten Refactorings.
- Refactorings können in ihrer Mechanik beschrieben werden.
- Refactoring gehört zum Handwerkszeug der Softwareentwicklung.
- Eine gute Entwicklungsumgebung kann für eine gute Programmiersprache bei vielen Refactorings automatisierte Unterstützung anbieten.

Formale Korrektheit, Abstrakte Datentypen



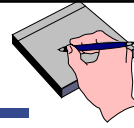
- Wir greifen die Forderung nach Softwarequalität auf und erläutern das Merkmal Zuverlässigkeit.
- Wir stellen die Grundlagen zur Zusicherung formaler Korrektheit vor.
- Wir erläutern Abstrakte Datentypen als wesentlichen Abstraktionsmechanismus.
- Wir lernen das Vertragsmodell mit seinen Zusicherungen und Abstraktionsmechanismen kennen; dadurch wird das Schnittstellenkonzept für Klassen semantisch erweitert.

Literaturhinweise



- Bertrand [Meyer](#): *Object-oriented Software Construction*. Second Edition. Prentice Hall, 1997.
- Peter [Rechenberg](#), Gustav [Pomberger](#), *Informatik-Handbuch*. Hanser-Verlag, 1977.
- Gunter [Saake](#), Kai-Uwe [Sattler](#): *Algorithmen und Datenstrukturen*. dpunkt-Verlag, 3. Auflage, 2006.

Zuverlässigkeit als Merkmal für Softwarequalität



- Wir wollen nicht nur wiederverwendbare, erweiterbare und änderbare, sondern auch **zuverlässige Software** entwickeln.
- **Zuverlässigkeit** ist nach Meyer gekennzeichnet durch:
 - **Korrektheit**
bezeichnet die Fähigkeit von Software, ihre Aufgabe genau so zu erfüllen, wie es die Spezifikation vorschreibt.
 - **Robustheit**
bezeichnet die Fähigkeit von Software, angemessen auf Fälle zu reagieren, die nicht in der Spezifikation definiert sind.

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.

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**.



SE2 – OOPM – Teil 3

73

Zuverlässigkeit als Voraussetzung der Wiederverwendbarkeit

- Eine Grundidee der Objektorientierung ist **Wiederverwendung** von Entwurfs- und Konstruktionseinheiten (z.B. Klassen)
- Voraussetzung der Wiederverwendung ist:
 - **Verständlichkeit**,
Sinn und Zweck der Komponenten müssen verstanden werden,
 - **Zuverlässigkeit**,
die Komponente muss als korrekt und robust eingeschätzt werden.
- Das „reine“ objektorientierte Modell liefert:
 - den Klassennamen,
 - die Signaturen von Operationen.
- Es fehlt:
 - Spezifikation der **Semantik**.



Wir erinnern uns:

- Der Kunde einer Klasse sieht *nur* die Schnittstelle. Diese lässt sich durch die Signaturen ihrer Operationen beschreiben.

SE2 – OOPM – Teil 3

74

Aus SE1: Ein Interface beschreibt nur Signaturen

```
interface Stack<E>
{
    void push(E element);
    E pop();
    boolean isEmpty();
}
```

```
interface Queue<E>
{
    void enqueue(E element);
    E dequeue();
    boolean isEmpty();
}
```

```
interface Geblubber<E>
{
    void bla(E element);
    E bloe();
    boolean blub();
}
```

Interfaces sind nur **semi-formale Spezifikationen**. Die Signaturen sind formal festgelegt, aber es fehlt die **Semantik** der Implementationen. Wir können die Semantik lediglich in Form von **Kommentaren** angeben (also nur informell).



Spezifikation der Semantik von Programmen



Um zu verstehen, wie sich Programme verhalten, betrachtet die theoretische Informatik ihre **formale Semantik**:

- Ist die formale Semantik eines Programmes definiert, lässt sich das Verhalten eines Programms statisch feststellen (beweisen), also ohne es auszuführen.
- Wenn dies für alle gültigen Eingaben eines Programms möglich ist, dann ist der Nachweis der Korrektheit des Computerprogramms erbracht (vgl. Verifikation).
- Die formale Semantik benutzt ausschließlich mathematische Methoden, um die Bedeutung von Computerprogrammen in einer formalen Sprache auszudrücken. Die Semantik eines Computerprogramms wird also syntaktisch beschrieben. Auf diese formalen Ausdrücke lassen sich **Ableitungsregeln (Kalküle)** anwenden, um so Aussagen über das Programm zu beweisen.

Ansätze der formalen Semantik (1)



Ein Ansatz der formalen Semantik betrachtet die Änderungen im Speicher, die ein Programm bewirkt:

- **Denotationale Semantik**
Die Wirkungsweise eines Programms wird als partielle Abbildung eines Speicherzustandes in einen anderen verstanden.
- Sei Z die Menge aller möglichen Speicherzustände. Die Wirkung, die eine Anweisung A auf einen Zustand hat, ist formal gesprochen eine Abbildung

$$f[A]: Z \rightarrow Z$$

die einem Zustand Z einen Folgezustand Z' zuordnet.

- Die denotationale Semantik definiert die Funktion f , die den Ausgangszustand eines Programms in seinen Endzustand überführt.

Ansätze der formalen Semantik (2)



Ein weiterer Ansatz der formalen Semantik betrachtet die schrittweise Veränderung des Zustands einer abstrakten Maschine durch ein Programm:

- **Operationale Semantik**
ersetzt meist ein konkretes Programm durch ein semantisch gleiches aber abstraktes Programm. Dieses Programm wird von einer abstrakten Maschine interpretiert. Die Wirkung eines Programms ist die schrittweise Veränderung dieser abstrakten Maschine, d.h. ihrer Variablenbelegung. Dabei ist die Annahme, dass die Veränderung der abstrakten Maschine gleich der eines konkreten Computers ist.

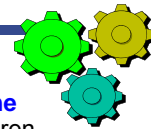
Ansätze der formalen Semantik (3)



Im Kontext der objektorientierten Programmierung hat ein 3. Ansatz bisher den größten praktischen Einfluss:

- **Axiomatische Semantik**
Sie legt für die einzelnen Programmelemente **Vor- und Nachbedingungen** fest. Betrachtet werden dabei die Veränderungen von Programmvariablen auf logischer Ebene. Jeder Operation der Sprache auf den entsprechenden Sprachelementen entspricht ein Axiom, das festlegt, welche Vorbedingung für Variablen bei der Programmausführung in welche Nachbedingung überführt wird.
- Ein klassischer Ansatz der axiomatischen Semantik für die **imperative Programmierung** ist der sog. **Hoare Kalkül**. (siehe "An Axiomatic Basis for Computer Programming" in Comm. of the ACM, 12/10, S. 576-580, 583).

Einschätzung der formalen Spezifikation von Semantik



- Mit Ansätzen wie dem Hoare Kalkül lassen sich **imperative Programme** aus Anweisungsfolgen mit elementaren Datentypen und Kontrollstrukturen als (partiell) korrekt beweisen.
- Für **objektorientierte Programme** mit Referenzvariablen und Methodenaufrufen ist ein entsprechender Beweis sehr aufwendig.
- Manuelle Korrektheitsbeweise sind zudem sehr fehleranfällig (wer beweist die Korrektheit des Beweises?).
- Heute existieren maschinelle Theorembeweiser, die aber nicht vollautomatisch arbeiten und hohes Expertenwissen für ihren Einsatz benötigen.
- Daher:
 - Die formale Korrektheit ganzer Programme wird aktuell nur für ausgewählte Programmteile in sicherheitskritischen Bereichen bewiesen.

Abstrakter Datentyp: ein Vorläufer des Klassenkonzepts

- **Abstrakte Datentypen (ADTs)** sind ein weiterer Ansatz, zuverlässige (korrekte) Software zu entwickeln.
- In der traditionellen imperativen Programmierung waren Daten und die Operationen auf ihnen getrennt. Damit war für Variablen oder Speicherplätze unklar, was mit ihnen gemacht wurde, d.h. ihre Semantik war unklar.
- Das galt auch noch für einfache Typkonzepte, weil dort die Definition der Typen losgelöst war von der Definition der darauf arbeitenden Operationen.
- Mit den abstrakten Datentypen (engl.: abstract data type) wurden erstmals Datentypen und ihre zulässigen Operationen gemeinsam definiert.

Abstrakter Datentyp: die Grundidee

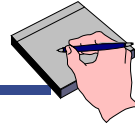
- Die Grundidee ist, einen (dynamischen) **Datentyp** nur durch die **Operationen**, die auf ihn anwendbar sind und durch die **Bedingungen** ihrer Anwendung zu beschreiben.
- Die zulässigen Operationen eines Datentyps sollen abstrakt, d.h. nicht durch eine Implementation definiert sein; die Implementation soll verborgen und damit austauschbar bleiben.
- Ein abstrakter Datentyp ist von der Idee her also eine Spezifikation, die unabhängig von einer konkreten Programmiersprache definiert wird.

Wir haben wichtige Grundlagen für die Umsetzung abstrakter Datentypen schon kennen gelernt:

- Zugriff auf die Dienstleistungen einer Klasse nur über ihre Schnittstelle,
- Kapselung der Exemplarvariablen,
- Verbergen und potenzielles Austauschen der Implementation von Operationen.



Der Begriff "abstrakter Datentyp"



Nach Informatik-Duden:

- Unter einem **Datentyp** versteht man die Zusammenfassung von Wertebereichen und Operationen zu einer Einheit.
- Liegt der Schwerpunkt auf den *Eigenschaften*, die die Operationen und Wertebereiche besitzen, dann spricht man von einem **abstrakten Datentyp (ADT)**.

Die formale Definition eines ADT besteht üblicherweise aus:

- **Typen**,
- **Funktionen**,
- **Axiomen**,
- **Vorbedingungen**.

© nach Meyer

Formale Beschreibung eines ADT: algebraische Spezifikation der natürlichen Zahlen

•TYPE

Nat

•FUNCTIONS

$0: \rightarrow \text{Nat}$

$\text{suc}: \text{Nat} \rightarrow \text{Nat}$

$\text{add}: \text{Nat} \times \text{Nat} \rightarrow \text{Nat}$

•AXIOMS

For any $i, j: \text{Nat}$

$\text{add}(i, 0) = i$

$\text{add}(i, \text{suc}(j)) = \text{suc}(\text{add}(i, j))$

•Der Typ **Nat** mit seinen möglichen Werten (genannt: Sorten) und Funktionen wird definiert

•Die Signatures der Funktionen legen die Schnittstelle des Typs **Nat** fest

•Die Axiome sind Gleichungen über Terme, die mit den Signatures gebildet werden. Sie beschreiben Einschränkungen der Verwendung der Funktionen. (formal: Einschränkungen möglicher Algebren zu diesem Typ)

Eine Algebra ist mathematisch eine Wertemenge und die Operationen auf diesen Werten. Eine mehrsortige Algebra verwendet mehrere Sorten als Wertebereiche.



Der Stack als ADT

TYPE

`STACK [T],`
`imports BOOLEAN`

FUNCTIONS

`empty: → STACK [T]`
`push: STACK [T] × T → STACK [T]`
`pop: STACK [T] → STACK [T]`
`top: STACK [T] → T`
`is_empty: STACK [T] → BOOLEAN`

AXIOMS

For any s: STACK[T], x: T
`top (push (s, x)) = x`
`pop (push (s, x)) = s`
`is_empty (empty) = true`
`is_empty (push (s, x)) = false`

PRECONDITIONS

`pop (s) require not is_empty (s)`
`top (s) require not is_empty (s)`

Es wird ein Stack mit dem Elementtyp **T** definiert. **T** ist ein generischer Typparameter, an den ein konkreter Typ (z.B.: Int) gebunden werden kann

Der Typ **BOOLEAN** wird importiert und kann genutzt werden.

Die Funktionen **push** und **pop** sind partiell, d.h. sie sind für den leeren Stack nicht definiert.

Die Axiome definieren, wie die Funktionen des Stacks wirken, ohne ihre Implementation anzugeben.

Die Vorbedingungen definieren die Einschränkungen für die Verwendung der Funktionen **push** und **pop**.

SE2 - 85

Diskussion des Zwischenergebnisses

- Die *algebraische Spezifikation* eines abstrakten Datentyps ist *funktional* oder *applikativ* ausgerichtet. Sie lässt sich so nicht einfach in ein imperatives Programm übertragen.
- Trotzdem bleibt die *Grundidee*, einen *Datentyp* durch sein *Verhalten* und nicht durch seine *Repräsentation* zu definieren.
- Wir halten für die weitere Diskussion fest:
 - Eine ADT-Spezifikation ist funktional und nicht an Objekten und ihren Operationen orientiert. Beachte den Unterschied:
`push: STACK [T] × T → STACK [T] /* funktional`
`einStack.push(e1) /* objektorientiert`
 - Die Axiome beschreiben implementationsunabhängig das (LIFO) Verhalten des Datentyps.
 - Die Vorbedingungen definieren wichtige Einschränkungen für die Verwendung des Datentyps

`top (pop (push (pop (push (push (pop (push (push (push (new, x1), x2), x3))), top (pop (push (push (new, x4), x5))))), x6)), x7)))`

Funktionale Verwendung des **STACK**.

SE2 - Teil 3 86

TYPES

LIST [T],
 ■ imports BOOLEAN

FUNCTIONS

empty: \rightarrow LIST [T]
 first: LIST [T] \rightarrow T
 rest: LIST [T] \rightarrow LIST [T]
 append: $T \times$ LIST [T] \rightarrow LIST [T]
 concat: LIST [T] \times LIST [T] \rightarrow LIST [T]
 is_empty: LIST [T] \rightarrow BOOLEAN

AXIOMS

For any $x: T, l, ll: LIST [T]$
 first (append (x, l)) = x
 rest (append (x, l)) = l
 is_empty (empty()) = true
 is_empty (append (x, l)) = false
 concat (append (x, l) , ll) = append (x, concat (l, ll))

PRECONDITIONS

first (l) **require not** is_empty (l)
 rest (l) **require not** is_empty (l)

Die Liste als ADT

nach © Goos in: Rechenberg, Pomberger

SE2

87

TYPES

QUEUE [T],
 ■ imports BOOLEAN

FUNCTIONS

empty: \rightarrow QUEUE [T]
 enqueue: QUEUE [T] \times T \rightarrow QUEUE [T]
 dequeue : QUEUE [T] \rightarrow QUEUE [T]
 front: QUEUE [T] \rightarrow T
 is_empty: QUEUE [T] \rightarrow BOOLEAN

AXIOMS

For any $x: T, q: QUEUE [T]$
 dequeue (enqueue(empty(), x)) = empty()
 dequeue (enqueue (q, x)) = enqueue (dequeue (q), x)
 front (enqueue(empty(), x)) = x
 front (enqueue(q, x)) = front (q)
 is_empty (empty()) = true
 is_empty (enqueue(q, x)) = false

PRECONDITIONS

front (q) **require not** is_empty (q)
 dequeue (q) **require not** is_empty (q)

Die Schlange als ADT

nach © Goos in: Rechenberg, Pomberger

SE2 – OC

88

TYPES

SET [T]
imports BOOLEAN

FUNCTIONS

empty: SET [T]
 insert: SET [T] \times T \rightarrow SET [T]
 delete: SET [T] \times T \rightarrow SET [T]
 union : SET [T] \times SET [T] \rightarrow SET [T]
 intersection: SET [T] \times SET [T] \rightarrow SET [T]
 difference: SET [T] \times SET [T] \rightarrow SET [T]
 has_element: T \times SET [T] \rightarrow BOOLEAN
 is_empty: SET [T] \rightarrow BOOLEAN

AXIOMS

For any x: T, s: SET [T]

has_element(x, empty()) = false
 delete(s, x) = difference(s, insert(empty(), x))
 insert(s, x) = union(insert(empty(), x), s)
 union(empty(), s) = s

Die Menge als ADT

intersection (empty(), s) = empty()
 difference (empty(), s) = empty()
 insert(x, empty()) = false
 is_empty (empty()) = true

PRECONDITIONS

delete (s, x) **require not** is_empty (s)

TYPES

BINTREE [T],
imports BOOLEAN

FUNCTIONS

empty: \rightarrow BINTREE [T]
 maketree: BINTREE [T] \times T \times BINTREE [T] \rightarrow BINTREE [T]
 value : BINTREE [T] \rightarrow T
 left: BINTREE [T] \rightarrow BINTREE [T]
 right: BINTREE [T] \rightarrow BINTREE [T]
 is_empty: BINTREE [T] \rightarrow BOOLEAN

AXIOMS

For any x: T, t1, t2: BINTREE [T]

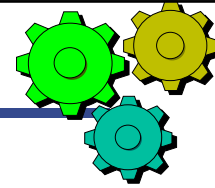
left (maketree(t1, x, t2)) = t1
 right (maketree(t1, x, t2)) = t2
 value (maketree(t1, x, t2)) = x
 is_empty (empty()) = true
 is_empty (maketree(t1, x, t2)) = false

PRECONDITIONS

value (t1) **require not** is_empty (t1)
 left (t1) **require not** is_empty (t1)
 right (t1) **require not** is_empty (t1)

Der Binärbaum als ADT

Diskussion des Zwischenergebnisses



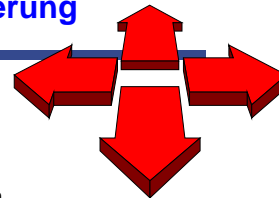
- Die **algebraische Spezifikation** eines abstrakten Datentyps ist *funktional* oder applikativ ausgerichtet. Sie lässt sich so nicht einfach in ein imperatives Programm übertragen.
- Trotzdem bleibt die Grundidee, einen Datentyp durch sein **Verhalten** und nicht durch seine **Repräsentation** zu definieren.
- Wir halten für die weitere Diskussion fest:
 - Eine ADT-Spezifikation ist funktional und nicht an Objekten und ihren Operationen orientiert.

Beachte den Unterschied:

```
push: STACK [T] × T → STACK [T] /* funktional  
einStack.push(el) /* objektorientiert
```

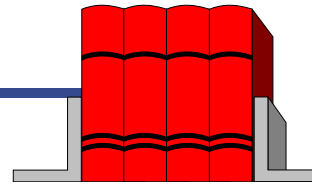
- Die Axiome beschreiben implementationsunabhängig das Verhalten des Datentyps (LIFO beim Beispiel Stack).
- Die Vorbedingungen definieren wichtige Einschränkungen für die Verwendung des Datentyps.

Metasprachliche Ansätze in der Programmierung



- Metaobjekte und Metaobjekt-Protokoll (MOP)
- Metaobjekte zur Beschreibung von Programmelementen
- Introspektion und Veränderung von Objekten
- Dynamische Objekterzeugung
- weitergehende metasprachliche Ansätze

Literaturhinweise



Ken **Arnold**, James **Gosling**, David **Holmes**, *The Java Programming Language*. 2006, Addison-Wesley, Amsterdam.
[Das Standardwerk von den Designern der Programmiersprache Java]

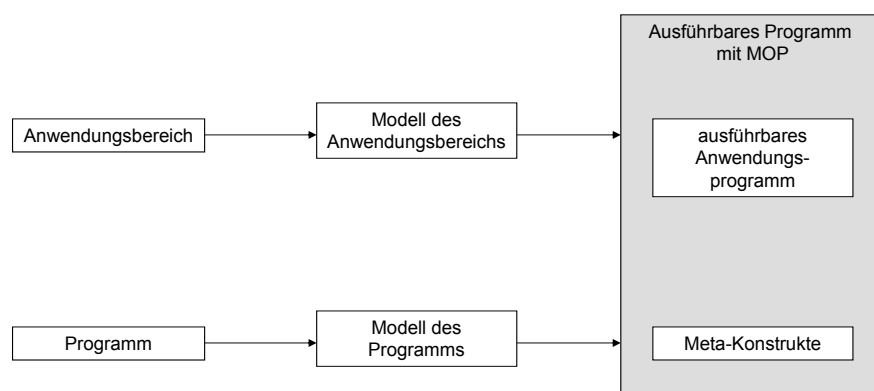
Motivation

- Für interaktive Systeme reicht es oft nicht aus, dass wir fachliche Logik und Oberflächen implementieren. Wir müssen uns auch mit den programmiersprachlichen Elementen unserer Programme beschäftigen.
- Während des Programmierens werden wir dabei von unserer Programmierungsumgebung unterstützt. Es gibt Fälle, in denen wir zur Laufzeit Informationen über unser Programm benötigen oder ein Programm sogar modifizieren müssen. Beispiele sind:
 - Laden von fremdgeschriebenen Komponenten (von welchem Typ sind sie?, wie sieht ihre Schnittstelle aus?).
 - Suchen und laden von abgespeicherten Objekten in einer Datenbank (dynamische Erzeugung eines vorher unbekannten Objekts, Aufruf seiner Operationen).
- Wir wollen also Programmteile schreiben, die das eigentliche Anwendungsprogramm und seine Elemente (Klassen, Interfaces, Methoden, Attribute) analysieren und ggf. verändern. Dazu muss eine Programmiersprache sog. **meta-sprachliche Eigenschaften** besitzen.
- Im weiteren erläutern wir entsprechende Grundkonzepte.

Metaobjekte

- Bisher haben wir mit Objekten Gegenstände eines Anwendungsbereichs modelliert.
- Wir können auch ein objektorientiertes Programm zum Gegenstandsbereich unserer Modellierung machen.
- In unseren Modellen wollen wir dann die Konstrukte eines Programms, die uns die jeweilige Programmiersprache bietet, darstellen.
- In der Objektorientierung ist es üblich, die Konstrukte eines Programms (und seiner Programmiersprache) selbst als Objekte zu repräsentieren.
- Solche Objekte werden **Metaobjekte** genannt. Wie jedes Objekt sind auch Metaobjekte Exemplare von Klassen, den sog. Metaklassen.
- Die Menge der Schnittstellen dieser Metaklassen werden auch das **Metaobjekt-Protokoll (MOP)** genannt.

Gegenstandsbereiche "Anwendung" und "Software" in einem ausführbaren Programm



Metamodelle und Metasprachen: eine Analogie

- Mit unserer Sprache machen wir Aussagen über unsere Umwelt:
Das Kind spielt mit dem Ball.
- Wir können mit unserer Sprache auch Aussagen über unsere sprachlichen Äußerungen machen (metasprachliche Aussagen):
"das Kind" ist Subjekt des Satzes.
- Wir können mit unserer Sprache auch Aussagen über die Regeln unserer Sprache machen (Metamodell):
Sätze bestehen oft aus Subjekt, Prädikat und Objekt.
- Randbemerkung:
Da wir alle Ebenen mit den gleichen Sprachmitteln ausdrücken, kann es zu Verwirrungen (Paradoxien) kommen:
Bitte beachten Sie diesen Satz nicht!

Einteilung von Metaobjekten

- **Metaobjekte** lassen sich in zwei Kategorien einteilen:
 - Objekte, die das (statische) Programm repräsentieren (z.B.: Welche Klassen gibt es; welche Schnittstelle hat eine Klasse).
 - Objekte, die das Laufzeitsystem repräsentieren (z.B.: Ausführung einer Methode; aktuelle Belegung der Attribute eines Objekts).
- Üblich sind folgende Bezeichnungen des Metaobjekt-Protokolls:
 - **Reflexion** (in Java: Reflection oder Core Reflection):
alle metasprachlichen Eigenschaften eines Programms
 - **Introspektion** (Introspection):
Analyse der aktuellen Eigenschaften eines Objekts zur Laufzeit.

Klasseninformationen oder Runtime Type Information (RTTI)

- Polymorphie bedeutet, dass der statische Typ einer Variablen, an die ein konkretes Objekt gebunden ist, oft nur ein Supertyp des dynamischen Objekttyps ist.
- Polymorphie tritt häufig bei den Elementen einer Sammlung auf, da Sammlungen ihre Elemente oft als `Object` abspeichern.
- Um die aktuelle Typinformation wiederzugewinnen, bieten fast alle objektorientierten Programmiersprachen die Möglichkeit, den Typ eines Objekts zur Laufzeit zu erfragen (genannt **runtime type information, RTTI**).
- In Java können wir uns zu jedem Objekt sein Klassenobjekt liefern lassen, das gleichzeitig seinen Typ repräsentiert.
`Class c = obj.getClass();`
- Oft werden im Programm Annahmen darüber gemacht, welcher dynamische Objekttyp vorliegt. In Java gibt es dazu einen Typtest als Prädikat:
`obj instanceof Class`

Weitere Metaobjekte zur Beschreibung von Programmelementen

- Java besitzt exemplarisch für eine Sprache mit MOP u.a. folgende Metaobjekte:
 - `Class`
für Klassen- und Typinformationen
 - `Package`
beschreibt ein Package
 - `Field`
liefert Informationen über Attribute und erlaubt Introspektion
 - `Method`
beschreibt eine Methode und erlaubt ihren Aufruf

Beispiel Klassenobjekt

- In Java können von einem Klassenobjekt u. a. folgende Informationen abgerufen werden:
 - die Superklasse
 - die implementierten Interfaces
 - die Namen und Typen aller Felder
 - ein Array aller Methoden mit Namen, Parameter- und Ergebnis-Typen
 - Die Parameter aller Konstruktoren
- Beispiel:
`Method[] getDeclaredMethods()`

Information über den Objektzustand

- Der direkte Zugriff auf den Zustand (die Belegung der Attribute) von Objekten unter Umgehung ihrer Schnittstelle ein massiver softwaretechnischer Verstoß.
- Wir sollten immer ausreichend viele sondierenden und verändernde Operationen an der Schnittstelle einer Klasse anbieten, die den Zugriff auf den Implementationszustand kapseln.
- Es gibt aber technische Aufgaben, für die wir direkt auf den Implementationszustand zugreifen müssen.
- Beispiele sind:
 - Generische Vergleichs- und Kopieroperationen für unterschiedliche Objekte,
 - allgemeine Objekt-Ein- und -Ausgabe, z.B. zum Speichern in Dateien, Anschluss an relationale Datenbanken, Transportieren von Objekten in einem verteilten System,
 - Garbage Collection,
 - Debugging.

Introspektion von Objekten in Java

- Das Klassenobjekt eines Exemplars liefert die notwendigen Informationen für die Analyse des Objektzustands, z.B. über die Attribute einer Klasse:

```
Class c = obj.getClass();  
Field f = c.getDeclaredField(name);
```

- Die Attribute (z.B. Exemplarvariablen) einer Klasse sind im MOP als Objekte der Klasse `Field` repräsentiert.
- Der aktuelle Wert eines Attributs lässt sich auslesen:

```
Object value = f.get(obj);
```
- Ein Attribut lässt sich auch setzen:

```
f.set(obj, newvalue);
```
- Voraussetzung für die Introspektion des Objektzustands ist ein entsprechend reduzierter Zugriffsschutz.

Veränderung des Zugriffsschutzes für Objekte in Java

- Bei der Introspektion von Objekten ist zunächst nur erlaubt, was auch bei einem normalen Zugriff auf eine Klasse, ihre Methoden oder Felder möglich ist. Dies wird im wesentlichen durch die Zugriffsmodifikatoren bestimmt.
- Verletzt ein MOP-Aufruf dieses Prinzip, wird eine `IllegalAccessException` ausgelöst.
- Die Schutzmechanismen können über einen speziellen `AccessController` ausgeschaltet werden:

```
java.security.AccessController.doPrivileged(...)
```
- Dann können auch private Felder über Reflection gelesen und geschrieben werden! Doch selbst die Designer von Java, K. Arnold und J. Gosling, schreiben:
"You should use reflection only when you have exhausted all other object-oriented design mechanisms."

Der dynamische Methodenaufruf

- Häufig soll bei der fortgeschrittenen Programmierung erst zur Laufzeit einer Anwendung eine beliebige öffentliche Methode eines Objekts ausgewählt und aufgerufen werden.
- So kann z.B. ein selbst-geschriebener Interpreter häufig vorkommende Abläufe in einem Script zusammenfassen und ausführen. Auch beim Testen können so generell Methoden gerufen werden (z.B. von JUnit).
- Dazu muss ein MOP ein Methodenverzeichnis liefern, das Abfragen über die vorhandenen Methoden und ihren Aufruf erlaubt. Ein solches Verzeichnis enthält für jede Methode ein Methoden-Metaobjekt mit folgenden Informationen:
 - Name der Methode
 - Signatur der Methode
 - sonstige Merkmale (wie Exceptions etc.).
- Der Aufruf der Methode erfolgt entweder über das Methoden-Metaobjekt oder über eine spezielle Methode (Meta-Call) am Objekt selbst.

Der dynamische Methodenaufruf in Java

- Das Klassenobjekt eines Exemplars liefert die notwendigen Informationen für den Aufruf einer Methode, z.B. ein Methodenobjekt:

```
Class c = obj.getClass();
Method m = c.getDeclaredMethod(name, argType.class);
```
- Ein Methodenobjekt wird über seinen Namen und die Klassenobjekte seiner Parameter ausgewählt.
- Die Methode wird gerufen, indem das aktuelle Objekt und die aktuellen Parameter mitgegeben werden:

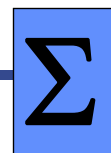
```
m.invoke(obj, argValue);
```
- Es ist auch möglich, Exemplare einer Klasse, z.B. mit dem Default-Konstruktor, dynamisch zu erzeugen:

```
Class c = obj.getClass();
Object obj2 = c.newInstance();
```

Weitergehende Konzepte des MOP

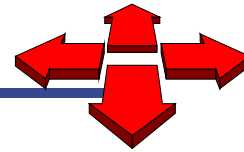
- In einigen Sprachen (wie Smalltalk) stellt das MOP eine Schnittstelle zur Verfügung, um die Methodenausführung zur Laufzeitsystem zu verändern:
 - **Veränderung des Methodenaufrufs**
Der Zugriff auf den Methodenaufruf-Mechanismus wird durch eine Metamethode ermöglicht, an die jeder Methodenaufruf umgelenkt wird. Diese Metamethode kann sowohl vor als auch nach der eigentlichen Ausführung beliebige Aktionen ausführen.
 - **Ersetzen der Methodenimplementation**
Der ausführbare Code einer Methode kann analysiert und modifiziert werden. Damit lässt sich die Implementation einer Methode vollständig verändern.

Zusammenfassung: Wesentliche Merkmale des Java MOP



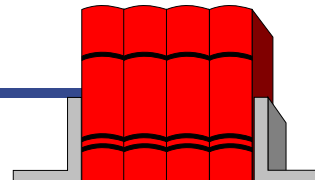
- Ein Programm lässt sich zur Laufzeit inspizieren:
 - Welchen Typ hat ein Objekt?
 - Von welcher Klassen erbt eine Klasse?
 - Welche Interfaces implementiert eine Klasse?
 - Welche Konstruktoren hat eine Klasse?
 - Welche Operationen hat eine Klasse?
 - Welche Attribute hat eine Klasse?
- Manipulationen sind in Grenzen möglich:
 - Objekte können über Klassenobjekte erzeugt werden.
 - Operationen können über ihren Namen aufgerufen werden.
 - Attribute können über ihren Namen ausgelesen und ggf. gesetzt werden.

Noch einmal: Testen objektorientierter Software



- Die **objektorientierte Programmierung** besitzt gegenüber der klassischen imperativen Programmierung **strukturelle und dynamische Besonderheiten**, welche **beim Testen zu berücksichtigen** sind.
- Wir haben bereits zu Beginn von SE2 gesagt: Das Thema Testen von objektorientierten Programmen ist sehr umfangreich und wird in SE2 nur im Überblick behandelt.
- Im Folgenden werden einige Aspekte des Testens objektorientierter Software näher betrachtet, die sich erst aus den im Laufe von SE2 behandelten Inhalten ergeben, wie Vererbung, Vertragsmodell, Entwurf, etc.

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]

OO-Testen: Die Klasse als kleinste Testeinheit

- Die kleinsten konstruktiven Einheiten eines objektorientierten Programms sind **Klassen**.
- Die einzelnen **Operationen** einer Klasse **existieren nicht isoliert**, sondern immer im Kontext einer Klasse oder ihrer Exemplare.
- **Operationen** (Methoden) haben meist eine **höhere Bindung** (Kohäsion) als klassische Unterprogramme und Prozeduren, da sie sich untereinander rufen und auf das gemeinsame „aktuelle“ Objekt beziehen. Damit ist die klasseninterne Komplexität durch die Zahl der abhängigen Elemente größer als bei typischen Prozeduren oder Unterprogrammen.
- Der **Zustand eines Objekts** ist durch die Belegung seiner Felder bestimmt. Oft sind diese Belegungen nicht elementar, sondern Referenzen auf andere Objekte. Polymorphie ermöglicht zudem, dass diese Objekte zur Laufzeit substituierbar sind. Die Folge ist ein **sehr großer Zustandsraum**.
- Die Konsequenz daraus ist, dass die kleinste sinnvoll testbare Einheit in der objektorientierten Programmierung die **Operation im Kontext einer Klasse** ist.

SE2 – OOPM – Teil 3

111

Probleme des oo Testens Revisited: 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.

SE2 – OOPM – Teil 3

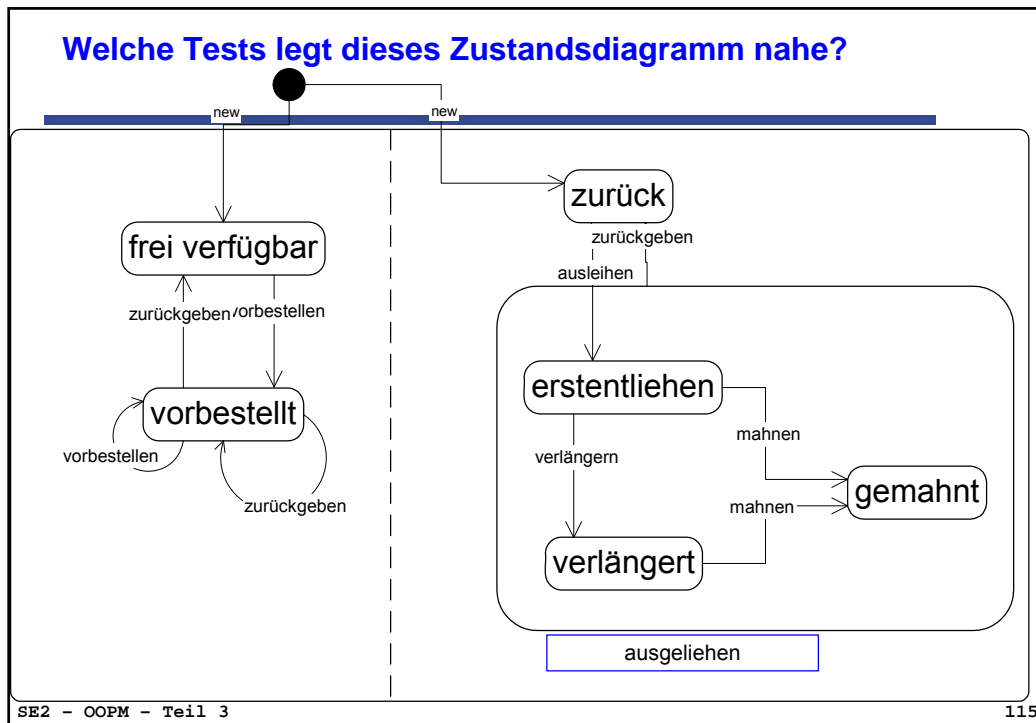
112

Ansätze des oo Testens: Umgang mit Kapselung (1)

- Um auf den gekapselten Zustand (der Felder) eines Objektes zuzugreifen, können entsprechende **sondierende Operationen an der Schnittstelle** angeboten werden, damit ein Test den Zustand eines Objektes ermitteln kann.
 - **Nachteil:**
Die fachliche Schnittstelle der Klasse wird eigentlich nur für den Test erweitert. Dies kann Klienten verwirren.
- Statt privater (Hilfs-) Methoden nur **default-Sichtbarkeit auf Package-Ebene**. Dann können Testklassen im selben Package durch Grey-Box Tests alle Methoden testen.
 - **Nachteil:**
Die Kapselung wird aufgeweicht. Nur Programmierdisziplin verhindert die Package-weite Verwendung von Hilfsmethoden außerhalb von Tests. Testklassen und fachliche Klassen in einem Package.
- **Statisch geschachtelte (innere) Testklassen** können auf alle Merkmale der äußeren Klasse zugreifen.
 - **Nachteil:**
Starke Kopplung einer fachlichen Klasse mit ihrer Testklasse. Unterschiedliche Testklassen führen zur "Aufblähung" der äußeren Klasse.
- Zugriff über Introspektion (Core Reflection) wird weiter hinten besprochen.

Ansätze des oo Testens: Umgang mit Kapselung (2)

- Bei der Konstruktion von komplexen Klassen ist es sinnvoll, das **Verhalten anhand eines Zustandsdiagramms** zu spezifizieren und die möglichen Zustandsübergänge durch Testfälle zu überprüfen. Hierfür sind Grey-Box-Tests notwendig, welche die internen Zustände kennen.
 - **Nachteil:**
Handhabbare Zustandsdiagramme sind nicht einfach zu entwickeln. Es müssen geeignete fachliche Abstraktionen über die technisch möglichen Zustände jeder Variablen (z.B. alle Integer-Werte) gefunden werden.



Probleme des oo Testens: Vererbung (1)

- Vererbung erschwert das Testen, weil **Spezifikation und Implementation über mehrere Klassen verteilt** sein können. Zudem spiegelt die **Struktur des Quelltextes nicht mehr den Kontrollfluss** wider. Dies verletzt eine der wichtigsten Forderungen der strukturierten Programmierung.
- Beim objektorientierten Testen stellen sich durch Vererbung die Fragen:
 - In welchen Unterklassen wird eine Methode einer Oberklasse benutzt?
 - Wurde eine implementierte Operation auf dem Vererbungspfad redefiniert?
- Innerhalb einer Vererbungshierarchie mit **Implementationsvererbung** wird das **Geheimnisprinzip weitgehend aufgehoben**. Auf Datenfelder und Methoden, die in Oberklassen als **public** oder **protected** deklariert sind, hat eine Unterklasse ungehinderten Zugriff. **Die Konsistenz einer Oberklasse kann durch die Unterklassen zerstört werden.**

Probleme des oo Testens: Vererbung (2)

- Umgekehrt haben wir festgestellt, dass **Operationsaufrufe in einer Oberklasse zur Laufzeit an redefinierende Methoden einer Unterklasse** gebunden werden können. **Die Semantik einer Operation kann in der Vererbungshierarchie verschoben werden (semantic shift).**
- Semantische Unterschiede zwischen geerbten und redefinierten Methoden einer Operation treten oft erst in bestimmten Kontexten zutage. Auch wenn die Vor- und Nachbedingungen textuell gleich sind, können sie unterschiedliche Bedeutung in der Ober- und der Unterklasse erhalten.
- Ein besonderer Fall sind die **abstrakten Klassen und Interfaces**. Diese können nicht instanziiert und damit nicht direkt dynamisch getestet werden.

Ansätze des oo Testens: Umgang mit Subtyping

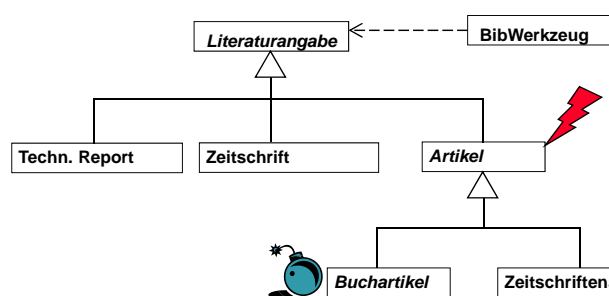
- **Vererbung im Sinne von Subtyping:**
 - vermeidet alle Probleme, die durch Redefinitionen auf dem Vererbungspfad und durch Zugriff von Methoden einer Unterklasse auf Attribute oder Methoden einer Oberklasse entstehen.
 - verhindert aber nicht, dass in den konkreten Unterklassen semantische Unterschiede bei der Implementation einer Operation auftreten.
- **Abstrakte Klassen und Interfaces:**
 - Um zu testen, ob abstrakte Klassen oder Interfaces konsistent im Kontext verwendet werden können, empfiehlt es sich, sie in **Mock-Up- oder Dummy-Klassen** (d.h. Wegwerf-Klassen zu Testzwecken) zu konkretisieren, damit sie getestet werden können.

Ansätze des oo Testens: Umgang mit Implementationsvererbung

- **Implementationsvererbung:**
 - Generell gilt bei Redefinitionen auf dem Vererbungspfad, dass nicht nur die Merkmale einer neuen Klasse, sondern **alle Merkmale ihrer Oberklassen getestet** werden müssen. Nur so ist sicherzustellen, dass im Kontext einer neuen Klasse keine unerwarteten Seiteneffekte auftreten können.
 - Wird in einer Klassenhierarchie eine neue Klasse eingeführt, welche Operationen redefiniert, so muss **in benutzenden Klassen nach allen Stellen gesucht werden, an denen der neue dynamische Typ auftreten kann**. Diese Klassen sind dann unter Verwendung des neuen dynamischen Typs zu testen.
 - Die möglichen Seiteneffekte der Implementationsvererbung können reduziert werden, indem in Unterklassen nicht direkt auf Exemplarvariablen der Oberklasse zugegriffen wird. Dazu stellen Oberklassen für ihre Exemplarvariablen **eigene protected Setter- und Getter-Methoden für die Unterklassen bereit**.

Probleme des oo Testens: Polymorphie

- **Parameter und Rückgabewert einer Operation** können zur Laufzeit auf **Objekte aller polymorph substituierbaren Klassen** (im Sinne von Subtypen) verweisen.
- So kann es zum Beispiel in der Klienten-Klasse **BibWerkzeug** nur dann zu einem Fehler kommen, wenn dort eine Operation der Klasse **Literaturangabe** benutzt wird **und** zur Laufzeit beim Aufruf ein Exemplar der Unterklasse **Buchartikel** gebunden ist. Der Fehler könnte daran liegen, dass in der Klasse **Artikel** eine Operation von **Literaturangabe** fehlerhaft redefiniert wurde.



Ansätze des oo Testens: Umgang mit Polymorphie

- Beim Testen ist sicherzustellen, dass eine Spezifikation für alle polymorph substituierbaren Klassen eingehalten wird. Dies bedeutet, dass **jede Kombination von möglichen Unterklasse-Objekten in redefinierten Methoden getestet** werden muss. Dies kann bei großen Klassenbäumen sehr aufwändig sein.
- In Unterklassen müssen **nicht nur die redefinierten und neuen Operationen** getestet werden, sondern **auch die unverändert geerbten**.
 - Wurde eine Operation nicht redefiniert, kann der Test aus der Oberklasse unverändert eingesetzt werden.
 - Eine redefinierte Operation muss zunächst die Tests der Oberklasse erfüllen.
- Der **Einsatz des Vertragsmodells kann diese Probleme reduzieren**. Auf praktikable Weise sind Probleme aber nur in Grenzen auszuschließen.
 - Soweit eine Reimplementation die Vorbedingung des Vertragsmodells abschwächt oder die Nachbedingung verschärft, müssen zusätzliche Tests diese neuen Grenzen prüfen.

SE2 – OOPM – Teil 3

121

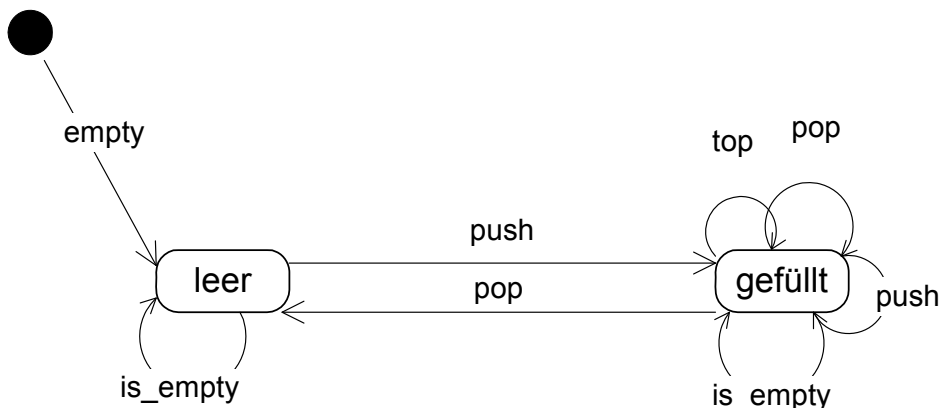
Tests und das Vertragsmodell

- **Testklassen** werden durch die systematische Verwendung des Vertragsmodells **nicht überflüssig**.
- Die **Zusicherungen** des Vertragsmodells sagen aus, welche Vorleistungen erwartet und welche Ergebnisse geliefert werden. Damit sind die **Randbedingungen für die Verwendung einer Operation** spezifiziert. **Testklassen prüfen herausgehobene Zustände und Parameterwerte im Gültigkeitsbereich** einer Operation.
- Im Vertragsmodell wird zudem **häufig implizit ein Protokoll zugrunde** gelegt, aber nicht explizit abgeprüft, da sich ein Vertrag immer nur auf eine Operation bezieht. Die Prüfung sinnvoller oder unerlaubter Aufrufsequenzen ist nur in Testklassen möglich.

SE2 – OOPM – Teil 3

122

Welches Protokoll kann sinnvoll anhand dieses Zustandsmodells getestet werden?



SE2 – OOPM – Teil 3

123

Tests und das Vertragsmodell

- Gerade bei Vererbung als Subtyping ist das Zusammenspiel von Vertragsmodell und Tests wichtig:
 - Explizite **Vor- oder Nachbedingungen** einer Methode werden **nicht** noch einmal **getestet**.
 - Getestet werden die „Ränder“ der Vor- und Nachbedingungen, d.h. die Werte, die **am Rande des Gültigkeitsbereichs** liegen.
 - **Alle nur in Kommentaren deklarierten Zusicherungen** (besonders die Invarianten) werden durch Tests abgesichert.

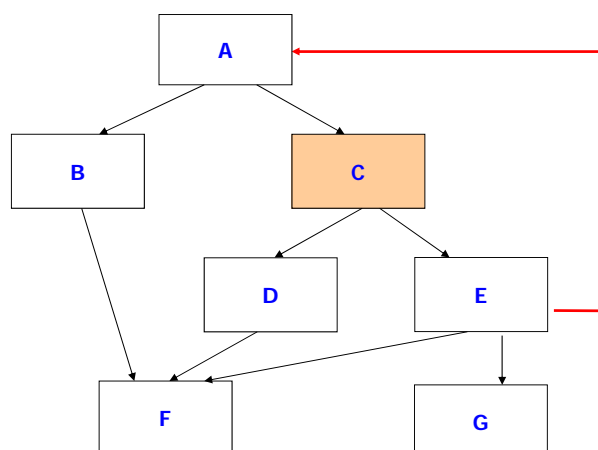
SE2 – OOPM – Teil 3

124

Integrationstest

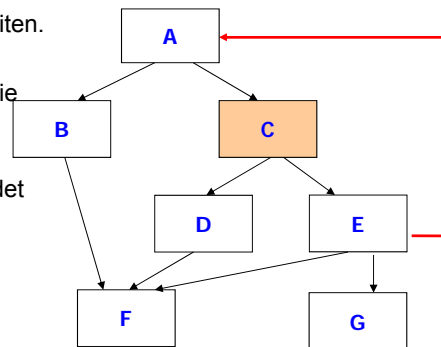
- Im **Integrationstest** werden einzelne Komponenten, die bereits einen Unit-Test bestanden haben, im Zusammenspiel getestet. Die Literatur spricht auch von **Interaktionstest**.
- Der Integrationstest „im Kleinen“ dient dazu, **Fehler im „benutzenden“ Code zu finden** und nicht im „benutzten“ Code. Typische Fehler, die durch Integrationstests auf der Ebene von Operationsaufrufen gesucht werden, sind:
 - Aufruf der falschen Operation,
 - falsche Benutzung der richtigen Operation
 - unerwartete Resultate.
- Die Vielzahl von Beziehungen in objektorientierten Programmen (Benutzung, Vererbung, Assoziationen, Polymorphismus) machen Integrationstest für objektorientierte Programme notwendig. Hier sind oft **zyklische Abhängigkeiten** zu beobachten.

Weshalb ist diese zyklische Benutzt-Beziehung problematisch?



Probleme zyklischer Beziehungen und Testen

- Meist ein Zeichen von unsauberer Konstruktion.
- Meist keine klare Aufteilung von Zuständigkeiten.
- Keine saubere Test- oder Integrationsstrategie möglich (bottom-up oder top-down).
- Nur der gesamte Zyklus kann wiederverwendet werden.
- Zerstört die meisten Architekturprinzipien (z.B. Schichtenarchitektur).



SE2 – OOPM – Teil 3

127

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?

SE2 – OOPM – Teil 3

128

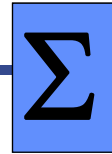
Ansätze des oo Testens: Umgang mit komplexen Abhängigkeiten (1)

- Erzeugen eines eigenen Testkontexts für die zu testenden Objekte, um nicht die gesamte Systemumgebung aufbauen zu müssen. Dieser Testkontext heisst oft **Testfixture**.
- Zur Minimierung des Testkontexts und während der Entwicklung eigene **Mock-** oder **Dummy-Objekte** bauen (siehe *Binder* oder *Link*), die einen vorläufigen und nur für Testzwecke geeigneten Kontext um das eigentlich zu testende Objekt bilden.
- Anhand von typischen Aktion-Reaktion-Sequenzen (*Akteur schickt Ereignis in das System – System reagiert*) lassen sich für interaktive Systeme Testkontexte definieren. Diese können aus technischen Use Cases (Anwendungsfällen) abgeleitet werden.
- Klassen sollten in Subsystemen organisiert werden, welche relativ autonom ihre Dienste erbringen können und lose mit anderen Subsystemen gekoppelt sind. Dadurch sinkt die Zahl der zu testenden Kontrollflüsse und möglichen Seiteneffekte.

Ansätze des oo Testens: Umgang mit komplexen Abhängigkeiten (2)

- Zusammengehörige Klassenteams führen zu komplexen Kontrollflüssen. Dabei werden unterschiedliche Entwurfsmuster eingesetzt, welche den Kontrollfluss regeln (z.B. Mediatoren, Events, Zuständigkeitsketten, Adapter, Template-Methoden). Diese beruhen oft auf abstrakten Klassen und ihrer Spezialisierung. Der Test auf die Korrektheit des Kontrollflusses ist dabei problematisch.
- Gleichzeitig sind Entwurfsmuster und ihr typische Verhalten aber Hilfen, um auf höherer Ebene Abhängigkeiten und Interaktionsbeziehungen zwischen Klassen und Objekten zu identifizieren und Teststrategien zu entwickeln.

Zusammenfassung



- Objektorientierte Software stellt erschwerte Anforderungen an das Testen.
 - Das zentrale Entwurfskonzept **Kapselung** kann White-Box-Tests erschweren.
 - **Vererbungsstrukturen** (Typ- und Implementationshierarchien) machen das Testen häufig komplizierter.
 - **Komplexe Abhängigkeiten**, insbesondere Zyklen, erschweren das Testen.