

Objektorientierte Programmierung

Prof. Dr. Uwe Kastens

WS 2013 / 2014

Ziele

Die Studierenden sollen lernen,

- Konzepte und Konstrukte objektorientierter Sprachen **planvoll in der Programmentwicklung einzusetzen**,
- **höhere Paradigmen** zur objektorientierten Programmierung anzuwenden und
- **Probleme und Grenzen** objektorientierter Programmierung zu erkennen.

Durchführung

- Die in der Vorlesung vermittelten Methoden und Techniken werden in **Übungen praktisch erprobt**.
- Als Programmiersprache wird **Java** verwendet.
- Es werden **Fallstudien** durchgeführt und vorgegebene Programme untersucht und weiterentwickelt.
- Es wird **unter Anleitung in kleinen Gruppen** an vorbereiteten Aufgaben gearbeitet.

Inhalt

Thema	Semesterwoche	Buch
1. Grundlagen Allgemeine OO-Konzepte Statische Typisierung in OO-Sprachen, Generik	1, 2 3 - 5	1, 2, 3, 12 [Bruce]
2. Einsatz von Vererbung Spezialisierung, Klassifikation, Rollen, Eigenschaften, inkrementelle Weiterentwicklung	6 - 8	8 - 11
3. Entwurfsmuster zur Entkopplung von Modulen Factory Method, Bridge, Observer, Strategy	9	Gamma
4. Programmbausteine, Bibliotheken, Programmgerüste Kopplung von Bausteinen, Strukturkonzepte von Bibliotheken	10	Budd[2] P-H
5. Entwurfsfehler Missbrauch der Vererbung, Antipatterns, OO-Überraschungen	11	
6. Jenseits von Java Mehrfachvererbung, konsequent OO, kontrollierte Vererbung, prototypbasiert	12, 13	Budd[2]

Literatur

Elektronisches Skript:

- <http://ag-kastens.upb.de/lehre/material/oop>

Buch zur Vorlesung:

- **Timothy Budd: Understanding Object-Oriented Programming with Java, Updated Edition, Addison-Wesley, 2000**
- Timothy Budd: An Introduction to Object-Oriented Programming, Third Edition, Addison-Wesley, 2002
- **Kim B. Bruce: Foundations of Object-Oriented Languages, MIT Press, 2002**

Weitere Bücher zu Thema:

- Timothy Budd: Object-Oriented Programming, Addison-Wesley, 1991
- Arnd Poetzsch-Heffter: Konzepte Objektorientierter Programmierung, Springer, 2000
- E.Gamma, R. Helm, R. Johnson, J. Vlissides: Design Patterns, Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995
- Arnold, Ken / Gosling, James: The Java programming language, Addison-Wesley, 1996
- Antero Taivalsaari: On the Notion of Inheritance, ACM Computing Surveys, Vol. 28, No. 3, September 1996
- Peter Coad, David North, Mark Mayfield: Object Models: Strategies, Patterns & Applications, 2nd ed., Yourdon Press, Prentice Hall, 1997

Weitere Hinweise im Vorlesungsmaterial unter *Internet*

Elektronisches Skript

Vorlesung Objektorientierte Programmierung WS 2013/2014

ag-kastens.upb.de/lehre/material/oop/

UNIVERSITÄT PADERBORN
Die Universität der Informationsgesellschaft

Fachgruppe Kastens > Lehre > Objektorientierte Programmierung WS 2013/2014

Vorlesung Objektorientierte Programmierung WS 2013/2014

<p>Vorlesungsfolien</p> <ul style="list-style-type: none"> • Kapitelübersicht • Folienverzeichnis • Drucken 	<p>Übungsaufgaben</p> <ul style="list-style-type: none"> • Aufgabenblätter • Drucken
<p>Organisation</p> <ul style="list-style-type: none"> • Personen, Termine, Regeln • Aktuelles <p>06.10.2013 Vorlesungsbeginn 15.10.2013 um 09:15 in F0.530</p>	<p>Wissenswertes</p> <ul style="list-style-type: none"> • Ziele • Literatur • Inhalt Budd: Understanding OOP with Java • Internet-Links

Veranstaltungs-Nummer: L.079.05700

Generiert mit Camelot | Probleme mit Camelot? | Geändert am: 06.10.2013

© 2013 bei Prof. Dr. Uwe Kastens

Organisation

Personen
<p>Sprechstunde Uwe Kastens:</p> <ul style="list-style-type: none"> • Mi 16:00 – 17:00 Uhr • Die 11:00 – 12:00 Uhr <p>Übungsbetreuer:</p> <ul style="list-style-type: none"> • Peter Pfahler
Termine
<p>Vorlesung</p> <ul style="list-style-type: none"> • Di, 9:15 – 10:45 Uhr F0.530 <p>Beginn: Di, 15. Oktober 2013 um 9:15 Uhr</p>
<p>Übungen</p> <p>Die Übungen werden im 14-tägigen Abstand 2-stündig angeboten. Das Vorlesungsverzeichnis sieht 4 Übungsgruppen vor:</p> <ul style="list-style-type: none"> • G1: Dienstag 11:00 Uhr, <i>ungerade Wochen</i>, Beginn 22.10.2013, erst in F0.530, dann im Rechner-Pool F1 (hinterer Teil) • G2: Dienstag 11:00 Uhr, <i>gerade Wochen</i>, Beginn 15.10.2013, erst in F0.530, dann im Rechner-Pool F1 (hinterer Teil) • G3: Donnerstag 09:15 Uhr, <i>ungerade Wochen</i>, Beginn 24.10.2013, erst in F2.211, dann im Rechner-Pool F1 (hinterer Teil) • G4: Freitag 09:15 Uhr, <i>gerade Wochen</i>, Beginn 18.10.2013, erst in F2.211, dann im Rechner-Pool F1 (hinterer Teil)
<p>Prüfungstermine</p> <p>Mündliche Prüfungen von ca 30 min Dauer im Rahmen von Modulprüfungen; für Studierende anderer Studiengänge als Informatik auch Einzelprüfungen. Es werden zwei Prüfungszeiträume angeboten:</p> <ol style="list-style-type: none"> 1. 12.-14. Februar 2014 2. 01.-03. April 2014 <p>Zu Anmeldung in PAUL und Terminvergabe siehe http://www.cs.uni-paderborn.de/studierende/pruefungswesen/pruefungsanmeldung.html</p>

© 2013 bei Prof. Dr. Uwe Kastens

1. Grundlagen, 1.1 Allgemeine OO-Konzepte

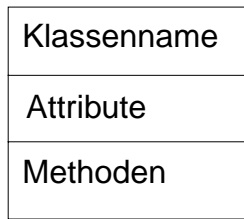
Objektorientierte Denkweise

- Modellierung und Entwurf von **Systemen interagierender Objekte**
statt Programmabläufe mit passiven Daten
- handelnde Objekte mit **Eigenschaften**, veränderlichem **Zustand** und eigenen **Operationen** darauf
statt Funktionen auf passiven Daten
- Objekt bestimmt, wie es einen Methodenaufruf ausführt:
Aufrufer schickt Nachricht (message) an Empfänger-Objekt (receiver)
Objekt führt Methode zu der Nachricht im Objektzustand aus
Eigenverantwortlichkeit, **Selbständigkeit der Objekte**
- **Schnittstelle** (Protokoll):
von außen beobachtbare Eigenschaften, benutzbare Methoden des Objektes
Schnittstelle **beschreibt das Was und verbirgt das Wie.**

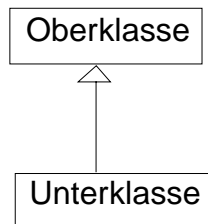
Klassen-basierter Entwurf

- Jedes **Objekt** gehört einer **Klasse** an.
- Klasse **definiert das Verhalten** ihrer Objekte
- Klasse als **abstrakter Datentyp** mit Implementierung
- **Klassen-Hierarchie:**
Unterklasse **erbt** Eigenschaften und Methoden von ihrer Oberklasse,
erweitert und verändert sie, z. B. zur Spezialisierung
- **Polymorphie:**
allgemeine Variable enthält spezielles Objekt;
Typ der Variable erlaubt Methodenaufruf mit bestimmter Signatur,
Klassenzugehörigkeit des Objektes bestimmt wie der Aufruf ausgeführt
wird: dynamische Methodenbindung (zur Laufzeit)
- **dynamische Methodenbindung:**
Software-Module können **unabhängig entwickelt** werden,
zur Laufzeit umkonfiguriert werden

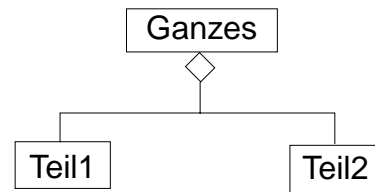
Entwurfsnotation UML Klassendiagramme



Klasse



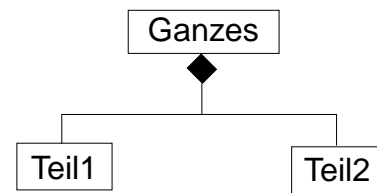
Vererbung



Aggregation
Teile existieren auch unabhängig



Assoziation



Komposition
Teile existieren nicht unabhängig

Objektorientierte Programmiersprachen

1966	Simula	Klassenhierarchie mit Einfachvererbung, statisch typisiert, dynamische Methodenbindung, diskrete Simulation, Algol 60 ist Teilsprache
1980	Smalltalk	Klassenhierarchie mit Einfachvererbung, dynamisch typisiert, dynamische Methodenbindung, konsequent objektorientiert, interpretierte Zwischensprache
1986	C++	Klassenhierarchie mit Mehrfachvererbung, statisch typisiert, dynamische Methodenbindung, generische Klassen, ANSI-C ist Teilsprache
1988	Eiffel	Klassenhierarchie mit Mehrfachvererbung, statisch typisiert, dynamische Methodenbindung, generische Klassen, explizite Vererbung
1994	Java	Klassenhierarchie mit Einfachvererbung, Interfaces, statisch typisiert, dynamische Methodenbindung, Internet-Bezüge, Prozesse, interpretierte Zwischensprache, umfassende Bibliotheken
1995	PHP	Skriptsprache; Server-seitige Web-Progr.; Klassen-basierte Vererbung
1995	JavaScript	Skriptsprache; Client-seitige Web-Progr.; Objekt-basierte Vererbung
2001	C#	Microsoft; Eigenschaften wie Java

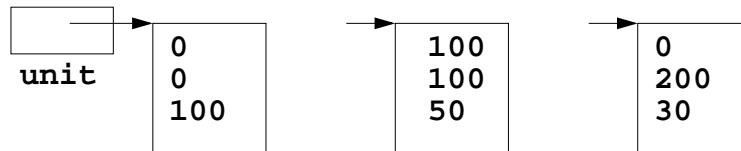
Grundlegende objektorientierte Sprachkonstrukte

Klassen und Objekte

Eine Klasse definiert **gleichartige Objekte**: Daten mit Operationen (**Methoden**) darauf. Operationen abstrahieren von der Implementierung.

Klasse	Ball
Attribute	Rectangle location double dx, dy Color color
Methoden	getX () getY () getRadius () move () paint (...)
Konstruktormethode	Ball (int x, int y, int r)

3 Objekte der Klasse **Ball**



```
Ball unit = new Ball (0, 0, 100);
```

```
unit.move();
```

Verwendung von Attributen

Objektbezogene Attribute: Variable oder Konstante eines jeden Objektes

- Eigenschaft des Objektes, unveränderlich oder explizit veränderlich `Color color;`
- Zustand des Objektes, veränderlich, auch durch andere Operationen `boolean visible;`
- Teil des Objektes `Flap leftFlap, rightFlap;`
- Bezug zu anderem Objekt `Monitor registers;`

Klassenbezogene Attribute: Information für alle Objekte der Klasse gemeinsam

- feste Codierung von Werten `final static int CLUBS = 1;`
- globale Information für oder über Objekte der Klasse `static int ballCount = 0;`

Zugriffsrechte für Attribute und Methoden

Regeln schränken ein, **wo** Zugriffe auf Attribute/Methoden **welcher Klasse** stehen dürfen:

private: `class C { private int x; ... }`

Zugriff nur **im Rumpf der Klasse C**,
 unqualifiziert (**x**) auf **x** des „eigenen“ C-Objektes und
 qualifiziert (**c.x**) auf **x** eines beliebigen C-Objektes

Package-weit (default): `class C { int x; ... }`

Zugriff in jeder Klasse des **Package**, in dem C deklariert ist;
 unqualifiziert in C und seinen Unterklassen oder
 qualifiziert auf Attribute/Methoden beliebiger C-Objekte

protected: `class C { protected int x; ... }`

Zugriff **Package-weit und zusätzlich auch in Unterklassen** von C,
 die nicht zum Package von C gehören

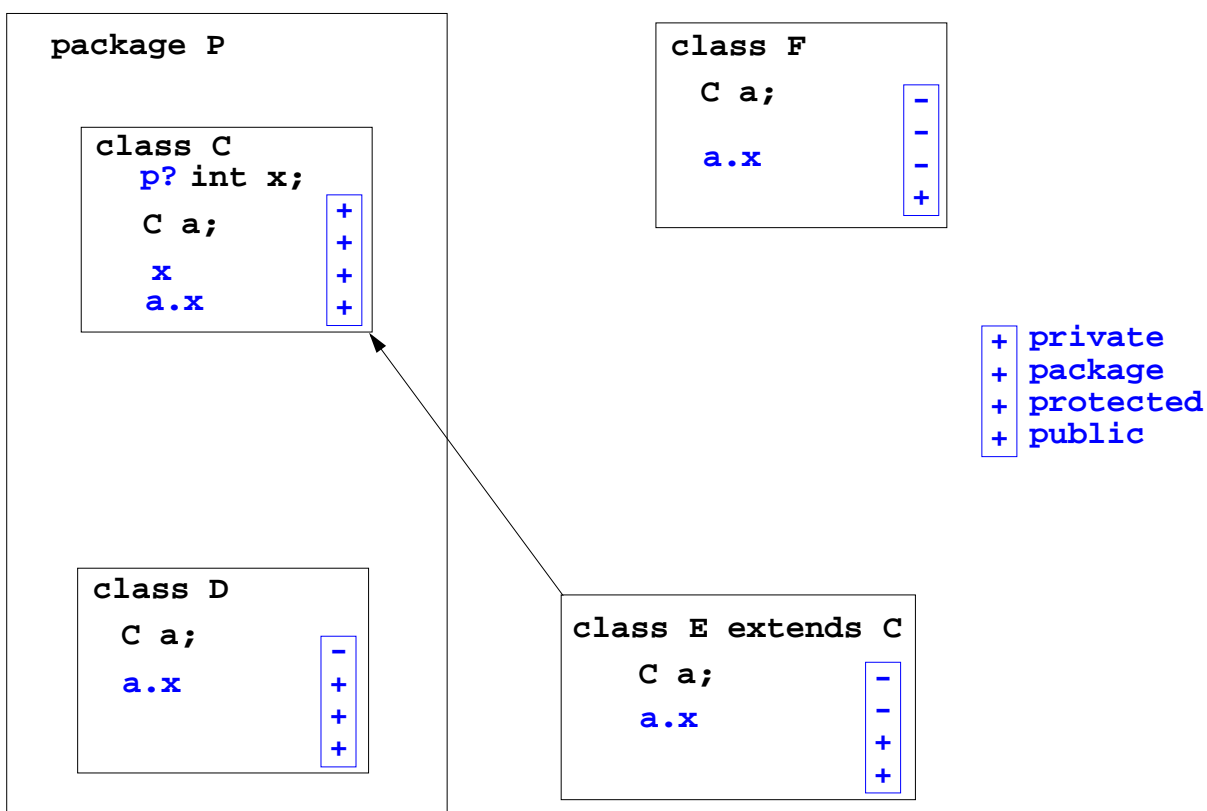
public: `class C { public int x; ... }`

unbeschränkter Zugriff

qualifizierter Name: `a.x` mit **a** als Ausdruck vom Typ C oder einer Unterklasse von C

unqualifizierter Name: `x` im Rumpf der Klasse C oder einer Unterklasse von C,
 entspricht `this.x`

Beispiel zu Zugriffsrechten in Java



Zugriff auf Attribute

Attribute in der Regel **private**:

- **keine unkontrollierte Zustandsänderung**,
- keine Verpflichtung, das Attribut nicht zu ändern

Vorsicht mit **protected**: Package-weit sichtbar, Verpflichtung gegen Unterklassen

Attribute lesen und zuweisen nur mit **get- und set-Methoden**
diese mit passenden Rechten versehen

Zustandsattribute meist nicht von außen zugänglich,
werden von Methoden benutzt und zugewiesen

unveränderliche Objekte

kein Attribut wird nach der Initialisierung verändert (auch nicht in Unterklassen),
z.B. `java.lang.String`;
Kopien sind nicht unterscheidbar, referenzielle Transparenz

Konstruktor-Methoden

Konstruktor-Methode dient der **Initialisierung der Attribute** bei der Objekterzeugung.
Danach muss das Objekt in einem benutzbaren Zustand sein.

Der **Hauptkonstruktor** hat alle einstellbaren Werte als Parameter.

```
public Ball (int x, int y, int r, Color c)
{
    location = new Rectangle (x-r, y-r, 2*r, 2*r);
    color = c;
    dx = 0; dy = 0; // initially no motion
}
```

Weitere Konstruktoren setzen einige Attribute mit default-Werten (überladene Konstruktoren):

```
public Ball (int x, int y, int r)
{
    this (x, y, r, Color.blue);
}
```

Alle Konstruktoren rufen den Hauptkonstruktor auf:
zentrale Stelle, wo jedes Objekt initialisiert wird.

Klasse und Objekte - Typ und Werte

- | | |
|---|---|
| <ol style="list-style-type: none"> 1. Klasse definiert die Eigenschaften ihrer Objekte 2. Objekte existieren im Speicher 3. Objekte haben Identität, ihre Objektreferenz 4. new erzeugt ein neues Objekt verschieden von allen anderen 5. Objekte werden nicht kopiert sondern geklont. 6. Klasse als Typ von Variablen: Variable kann eine Referenz auf ein Objekt der Klasse aufnehmen 7. Klasse als Typ von Ausdrücken: Auswertung des Ausdrucks liefert eine Referenz auf ein Objekt der Klasse 8. Typanpassung verändert weder das Objekt noch seine Referenz; erlaubt dem Übersetzer, die Referenz, als Referenz eines anderen Typs zu behandeln | <ol style="list-style-type: none"> 1. Typ definiert die Eigenschaften seiner Werte 2. Werte können Speicherinhalte sein 3. Werte haben keinen Speicher, keine Identität 4. zwei gleiche Werte sind nicht unterscheidbar 5. Werte werden kopiert. 6. Typ von Variablen: Variable kann Wert des Typs aufnehmen 7. Typ von Ausdrücken: Auswertung des Ausdrucks liefert einen Wert eines Typs. 8. Typanpassung Verändert ggf. die Repräsentation des Wertes, z.B. von int nach float |
|---|---|

Methoden

Operationen einer Klasse:

```
public void setMotion (double ndx, double ndy)
    {dx = ndx; dy = ndy;}
```

Typen der Parameter und des Ergebnis ist die **Signatur** der Methode (in Java: nur Typen der Parameter - ohne Ergebnistyp)

```
setMotion: double x double -> void
```

Aufruf kann als Seiteneffekt den **Objektzustand ändern**;
hier: Geschwindigkeit zuweisen.

Aufruf kann als Seiteneffekt den **Zustand von Parameterobjekten ändern**,
z. B. `Wand.reflektiere (ball);`

Zugriffsrechte für Methoden wie für Attribute

Methodenaufrufe

Notation von Methodenaufrufen

in der Klasse, in der die Methode deklariert ist, **unqualifiziert**

```
setMotion (1.0, 1.0);
```

qualifiziert für „fremdes“ Objekt

```
theBall.setMotion (1.0, 1.0);
```

Ausführung eines Aufrufs **a.m(p)**:

1. Ausdruck **a** auswerten, Ergebnis ist eine Referenz auf ein Objekt **obj**;
mit dem (dynamischen) Typ des **obj** die aufzurufende Methode **m** bestimmen
(dynamische Methodenbindung)
(Überladung wird anhand der Signatur schon zur Übersetzungszeit aufgelöst)
2. In der Umgebung von **obj** eine Schachtel **s** für einen Aufruf der Methode **m**
von **obj** bilden;
statischer Vorgänger braucht nicht auf dem Laufzeitkeller zu liegen;
Werte der aktuellen Parameter **p** berechnen und an formale Parameter in **s**
zuweisen;
3. Rumpf der Methode mit der Schachtel **s** ausführen;
ggf Ergebnis an die Aufrufstelle liefern; hinter die Aufrufstelle zurückkehren

Schnittstelle und Implementierung

Abstraktion: Außensicht **Was**; Innensicht **Wie**

Schnittstelle einer Methode:

Signatur: `setMotion: double x double -> void`

Sprachkonstrukt:

abstrakte Methode, die in Unterklassen implementiert wird:

```
abstract void setMotion (double, double);
```

Schnittstelle einer Klasse:

Menge der Signaturen der von außen aufrufbaren Methoden (**public**)

Implementierung einer Klasse:

Implementierung der Methoden der Schnittstelle

mit Methodenrümpfen und

dazu notwendige Attribute und Hilfsmethoden

Schnittstelle als selbständiges Sprachkonstrukt

in Java: **interface**,
 in C++: rein abstrakte Klasse,
 in SML und Haskell: **signature**

benannte Menge von Methoden-Signaturen

```
public interface Enumeration
{
    boolean hasMoreElements();
    Object nextElement()
        throws NoSuchElementException;
}
```

Klasse **C** implementiert ein Interface **Intf**,
 d. h. C implementiert alle Methoden aus **Intf**

Schnittstelle Intf als Abstraktion für Objekte von Klassen,
 die **Intf** implementieren:

Schnittstelle als Typ, z. B.

```
Enumeration e = new A();
while (e.hasMoreElements())
{
    Object x = e.nextElement();
    ...
}
```

Oberklassen und Vererbung

Klasse als **Unterklasse einer Oberklasse** definieren.
 (verschiedene Zwecke siehe Kapitel 2)

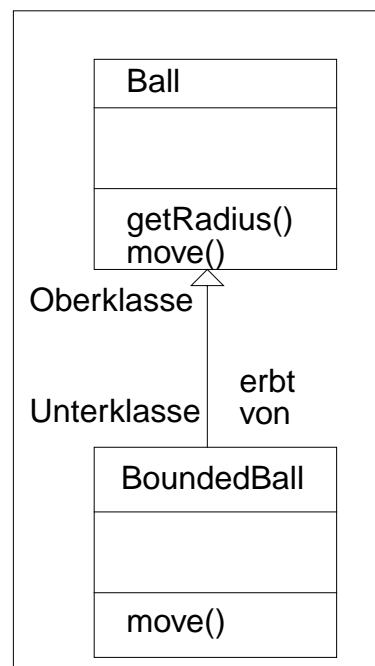
Unterklasse **erbt** von der Oberklasse
 Attribute, Methoden,
 Implementierungsverpflichtungen (d. h. abstrakte Methode)

Unterklasse **definiert zusätzliche**
 Attribute, Methoden, Implementierungsverpflichtungen

Unterklasse **überschreibt** Methoden der Oberklasse
 bei gleichem Methodennamen und -signatur.
 Unterlassenobjekt führt Methodenaufruf mit seiner
 statt der überschriebenen Methode aus
 (dynamische Methodenbindung)

Klasse als **Typabstraktion**:
 Variable vom Oberklassentyp können Referenzen von
 Unterlassenobjekten aufnehmen

```
Ball b = new BoundedBall (...); ...b.move();
```



Vererbung und verwandte Konzepte

Sei **A** eine Oberklasse von **B** und in **A** sei eine Methode **m** implementiert, z. B.

```
class A { C m (String s) {...} ...}   class B extends A {...}
```

1. Wenn **m** in **B** nicht definiert ist, **erbt** **B** die Methode **m**.
2. Wenn in **B** eine Methode **m** mit gleichen Parametertypen definiert ist, **überschreibt** sie das **m** aus **A**. Seit Java 5 muss der Ergebnistyp gleich **C** oder eine Unterklasse von **C** sein (Java: substitutable); kovariante Ergebnistypen.
3. Wenn in **B** eine Methode **m** definiert ist, deren Parametertypen sich von denen von **m** in **A** unterscheiden, dann sind die beiden Methoden in **B** **überladen**.
Ebenso, wenn in **A** eine weitere Methode **m** mit anderen Parametertypen definiert wird.
4. Wenn in **B** eine Klassenmethode (**static**) **m** definiert ist, deren Parametertypen mit denen von **m** in **A** übereinstimmen und die Ergebnistypen kovariant sind, dann **verdeckt** **m** in **B** die Methode **m** in **A**, d. h. **m** aus **A** ist in **B** nicht sichtbar. **m** in **A** muss dann auch **static** sein.

Sei **A** eine Oberklasse von **B** und in **A** sei eine Methode **m** spezifiziert, z. B.

```
abstract class A { abstract C m (String s); ...}
class B extends A {...}
```

5. Wenn **B** nicht abstract ist, muss **m** in **B** mit denselben Parametertypen und einem kovarianten Ergebnistyp **implementiert** werden.

Objektorientierte Polymorphie

Polymorph: vielgestaltig; hier im Sinne der Typen von Sprachkonstrukten

Referenzen auf **Objekte unterschiedlicher Klassen** können
in einer Variable enthalten sein oder
Ergebnis der Auswertung eines Ausdrucks sein.

Wird darauf eine Methode aufgerufen, so bestimmt
die **Klassenzugehörigkeit des Objektes (zur Laufzeit)**
welche Methode ausgeführt wird: **dynamische Methodenbindung**.

Der **statische Typ der Variable** (des Ausdrucks) garantiert zur
Übersetzungszeit, dass eine Methode mit passender Signatur existiert.

Einsatz zur

- Abstraktion
- Entkopplung von Programmmodulen
- Umkonfigurierung während der Ausführung
(z. B. Erscheinungsbild der GUI-Komponenten)

Taxonomy of type systems

[Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. ACM Computing Surveys, 17(4):471–523, 1985.]

monomorphism: Every entity has a unique type. Consequence: different operators for similar operations (e.g. for `int` and `float` addition)

polymorphism: An operand may belong to several types.

ad hoc polymorphism:

overloading: a construct may have different meanings depending on the context in which it appears (e.g. `+` with 4 different signatures in Algol 60)

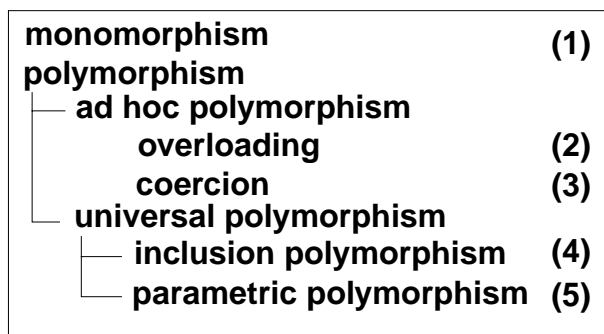
coercion: implicit conversion of a value into a corresponding value of a different type, which the compiler can insert wherever it is appropriate (only 2 add operators)

universal polymorphism: operations work uniformly on a range of types that have a common structure

inclusion polymorphism: sub-typing as in object-oriented languages

parametric polymorphism: a type denotation may have formal type parameters, e.g. `('a x 'a)`; they are substituted by type **inference** (e.g. in SML) or by **generic instantiation** (C++, Java)

Monomorphism and ad hoc polymorphism



monomorphism (1):
4 different names for addition:

```
addII: int    x int    -> int
addIF: int    x float  -> float
addFI: float  x int    -> float
addFF: float  x float  -> float
```

overloading (2):
1 name for addition `+`;
4 signatures are distinguished by actual operand and result types:

```
+: int    x int    -> int
+: int    x float  -> float
+: float  x int    -> float
+: float  x float  -> float
```

coercion (3):
`int` is acceptable as `float`,
2 names for two signatures:

```
addII: int    x int    -> int
addFF: float  x float  -> float
```

Dynamische Methodenbindung in Java

3 verschiedene Situationen für dynamische Methodenbindung in Java:

1. Schnittstellentyp:

```
interface Enumeration { boolean hasMoreElements(); ... }
Enumeration enum; .... enum.hasMoreElements() ...
```

2. Überschriebene Methode:

```
class Ball { public void move (){...} }
class BoundedBall extends Ball { public void move (){...} }
Ball b; .... b = new BoundedBall (...); ... b.move(); ...
```

3. Implementierung abstrakter Methode:

```
Class Animal { abstract void makeNoise(); ...}
Class Frog extends Animal
    {void makeNoise(){ System.out.print("QuakQuak");} }
Animal which; ... which.makeNoise();
```

Überladene Methoden

Überladene (overloaded) Methoden:

Mehrere Methoden mit **gleichem Namen**, aber **unterschiedlicher Anzahl** oder **Typen der formalen Parameter** sind an einer Aufrufstelle gültig.

Die **Typen der aktuellen Parameter** im Aufruf bestimmen, welche Methode aufgerufen wird.

Methoden einer Klasse können untereinander überladen sein.

Methoden einer Unterklasse können mit geerbten Methoden überladen sein (ohne sie zu überschreiben!).

Konstruktoren können überladen sein.

Operatoren (+, /, usw.) sind meist überladen.

Überladen wird häufig auch als **Variante der Polymorphie** bezeichnet.

1.2 Statische Typisierung in OO-Sprachen

Dieser Abschnitt folgt dem Buch

F.O.O.L von Kim B. Bruce, Kap 2, 3, 5, 6.

Darin werden Eigenschaften von OO-Sprachen, insbesondere die Typisierung, allgemeiner und konsequenter dargestellt als sie aus gängigen Sprachen wie C++, Java, C# bekannt sind. Diese Darstellung soll das Verständnis und die kritische Beurteilung von OO-Konzepten vertiefen.

Folgende Begriffe werden hier gegenüber dem vorigen Abschnitt verfeinert oder verschärft:

- Typen getrennt von Klassen
- Untertypen (subtyping) getrennt von Unterklassen (subclassing)
- Typisierungsdefekte in gängigen OO-Sprachen rigoros hergeleitet

Begriffe zu Klassen am Beispiel

Es sollte verschiedene Namen geben für

- Klasse `CellClass`
- Typ der Objekte `CellType`
- Konstruktor

Zunächst werden alle

- Instanzvariablen als `private`
- Methoden als `public`

angesehen.

Bruce verwendet eine Notation, die daran erinnern soll, dass es sich nicht um Java, C++, etc handelt

```
class CellClass {
  x: Integer := 0;

  function get (): Integer is
  { return self.x }

  function set (nuVal: Integer): Void is
  { self.x := nuVal }

  function bump (): Void is
  { self <= set(self <= get()+1) }
}
```

Um die besondere Operation **Senden einer Botschaft an ein Objekt** (d.h. Aufruf einer Methode) hervorzuheben notiert Bruce `o <= m(x)` statt `o.m(x)`

`self` benennt das Objekt, für das ein Aufruf der Methode ausgeführt wird - `this` in Java.

In den meisten Sprachen kann `x` statt `self.x` und `get()` statt `self<=get()` geschrieben werden

Objekttypen

Ein **Typ** ist eine Menge von Werten und den darauf anwendbaren Operationen.

Alle **Objekte**, auf die die **gleichen Operationen anwendbar** sind, sollten **denselben Typ** haben.

Anwendbarkeit einer Methode wird durch die Signatur (= Typ) der Methode bestimmt

```
get: Void -> Integer
```

Der Typ der Objekte, die zur Klasse `CellClass` erzeugt werden, ist die Menge der Methoden-Signaturen. (Instanzvariablen tragen nicht zum Objekttyp bei, da sie außen nicht sichtbar sind):

```
CellType = ObjectType {get: Void -> Integer,
                      set: Integer -> Void,
                      bump: Void -> Void }
```

Konsequenz:

Verschiedene Klassen können **Objekte desselben (bzw. äquivalenten) Typs** erzeugen, sofern die **Methoden paarweise dieselbe Signatur** haben!
(Strukturäquivalenz der Typen)

In **verbreiteten OO-Sprachen** werden **Klassen als Typen** aufgefasst: Alle Objekte einer Klasse C haben denselben Typ - verschieden vom Typ der Objekte jeder anderen Klasse D (Namensäquivalenz) - auch wenn die Mengen der Methodensignaturen übereinstimmen.

Unterklassen und Vererbung

Eine Unterklasse erbt Instanzvariablen und Methoden der Oberklasse und kann Methoden zufügen und überschreiben, z. B.

modifies set
verhindert
versehentliches
Überschreiben.

super macht die
überschriebene
Methode der
Oberklasse
zugänglich

```
class ClrCellClass inherits CellClass modifies set {
  color: ColorType := blue;

  function getColor (): ColorType is
  { return self.color }

  function set (nuVal: Integer): Void is
  { super <= set (nuVal);
    self.color := red }
}
```

Typ der Objekte von `ClrCellClass`:

```
ClrCellType = ObjectType {get: Void -> Integer,
                          set: Integer -> Void,
                          bump: Void -> Void,
                          getColor: Void -> ColorType }
```

Wird **bump** für ein Objekt der Klasse `ClrCellClass` aufgerufen, dann führt **self <= set (self <= get ()+1)** im Rumpf von **bump** zum Aufruf von **set** aus `ClrCellClass`.

Untertypen (Subtyping)

S ist ein **Untertyp (subtype)** von T, notiert als $S <: T$, wenn ein Wert vom Typ S in jedem Kontext verwendet werden kann, wo ein Wert vom Typ T erwartet wird. T heißt dann auch Obertyp von S.

Z.B. `ClrCellType <: CellType`

$<:$ braucht nicht für die Objekte jedes Paares von Unter- und Oberklasse zu gelten, z.B. wenn die Unterklasse Methoden der Oberklasse löschen oder umbenennen kann (wie in Eiffel).

Hier sind **Untertypen anhand der Typeigenschaften definiert** (welche Methoden-Signaturen enthalten sie). Das ist **structural subtyping**. In den meisten OO-Sprachen wird die **Untertyprelation durch die Unterklassenrelation** festgelegt; diese wird durch die Klassennamen bestimmt.

In Sprachen mit **Subtyping** kann ein Wert zu mehreren Typen gehören, die in Untertyprelation stehen. Deshalb spricht man von **subtype polymorphism**.

Untertypen von Record-Typen

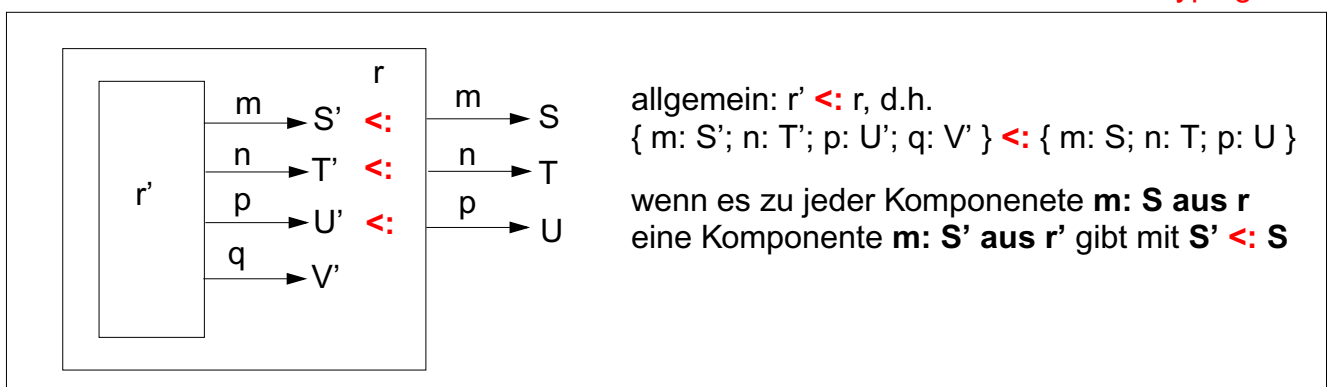
Betrachten wir die **Typen von unveränderlichen Record-Werten**: Komponenten werden **nur gelesen**, d.h. Typen der Werte müssen **paarweise die $<:$ Relation** erfüllen

Sei `CheeseType <: FoodType`, dann ist `CheeseSandwichType <: SandwichType`. Werte vom Typ `CheeseSandwichType` können die Rolle von Werten des Typs `SandwichType` spielen:

```
SandwichType = {      bread: BreadType;
                    filling: FoodType }
```

```
CheeseSandwichType = { bread: BreadType;
                       filling: CheeseType;
                       sauce: SauceType }
```

← tiefes
← breites
Subtyping



Untertypen von Funktionstypen

Betrachten wir die **Typen mit Signaturen** $P \rightarrow R$ (Parametertyp P und Resultattyp R).
 Hat f die Signatur $P \rightarrow R$, dann kann $g: P' \rightarrow R'$ die Rolle von f spielen, falls $P' \rightarrow R' \prec: P \rightarrow R$.
 Die Parameterübergabe sei call-by-value.

Sei **CheeseType** $\prec:$ **FoodType**, und **Integer** $\prec:$ **Long**:

Integer \rightarrow **CheeseType** $\prec:$ **Integer** \rightarrow **FoodType**

**kovariante
Ergebnistypen**

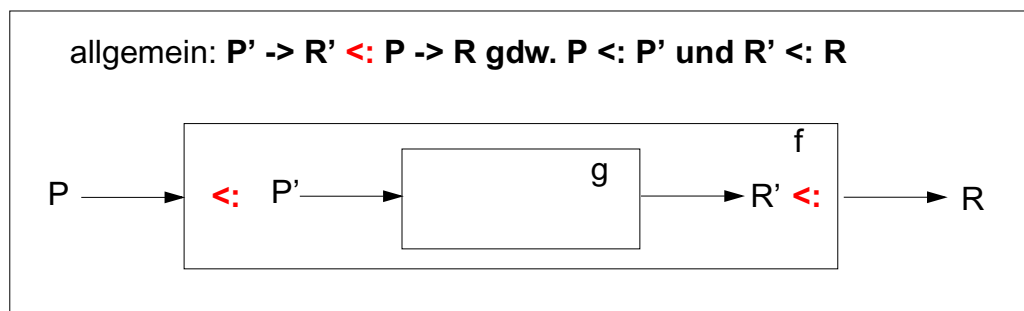
speziellere Resultate der ersetzenden Funktion sind als
 allgemeinere Resultate akzeptabel

FoodType \rightarrow **Integer** $\prec:$ **CheeseType** \rightarrow **Integer**

CheeseType \rightarrow **Integer** $\prec/:$ **FoodType** \rightarrow **Integer**

**kontravariante
Parametertypen**

Die ersetzende Funktion muss **mindestens so allgemeine
Parameter** akzeptieren wie die ersetzte.



Untertypen von Typen von Variablen

Unter welchen Umständen kann eine **Variable vom Typ T'** die Rolle einer **Variablen vom Typ T** spielen?

Mit einer Variable können **Lese- und Schreiboperationen** ausgeführt werden.

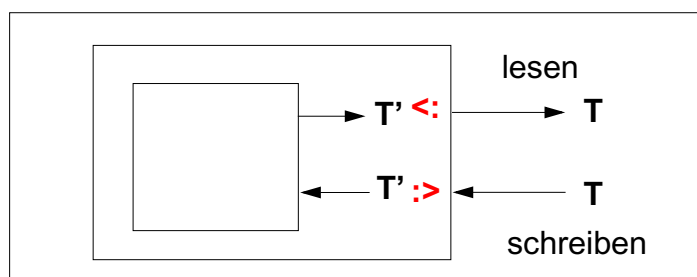
Sei v' vom Typ T' und v vom Typ T und $T' \prec: T$.

Für die **Leseoperation** kann v' die Rolle von v spielen: Jeder Wert aus v' ist auch ein Wert des Typs von v . Für das **Lesen wird Kovarianz** benötigt, $T' \prec: T$.

Für die **Schreiboperation** kann v' nicht die Rolle von v spielen: Nicht alle Werte des Typs von v können an v' zugewiesen werden. Es wird **Kontravarianz für das Schreiben** benötigt!

Insgesamt: $T' \prec: T$ und $T \prec: T'$, d.h. $T' \sim T$.

Es gibt **keine nicht-trivialen Untertypen von Variablen**. Sie können einander nur dann ersetzen, wenn ihre **Typen äquivalent** sind.



Keine nicht-trivialen Untertypen von Variablen: Konsequenzen

Es gibt **keine nicht-trivialen Untertypen von Variablen**.

Sie können einander nur ersetzen, wenn ihre **Typen äquivalent** sind.

Untertyprelation für **Records mit veränderlichen Komponenten**:

Für jede Komponente **n: T des Obertyps** muss es eine Komponente **n: T'** mit **T ~ T'** im Untertyp geben, d.h. nur **breite nicht tiefe Untertypen**.

Untertyprelation für **Arrays mit unveränderlichen Elementen [Ind] of EI**:

Ind ist der Indextyp, **EI** der Elementtyp.

[Ind'] of EI' <: [Ind] of EI wenn **Ind <: Ind'** (kontravariant) und **EI' <: EI** (kovariant)

Untertyprelation für **Arrays mit veränderlichen Elementen [Ind] of EI**:

[Ind'] of EI' <: [Ind] of EI wenn **Ind <: Ind'** (kontravariant) und **EI' ~ EI** (äquivalent)

Missachtung hat in Java eine Lücke in der statischen Typisierung verursacht, die durch dynamische Prüfung geschlossen wurde:

```
class Node { ... }
class Leaf extends Node { ... }
...
Node[] arr = new Leaf[42];

arr[0] = new Node();
```

Erlaubt in Java, obwohl nicht gilt
Leaf-Array <: Node-Array

statisch korrekte Zuweisung
verursacht **Laufzeitfehler**:
java.lang.ArrayStoreException: Node

Untertyprelation für Objekttypen: Präzisierung

- **Keine Änderung der Typen von Instanzvariablen** durch Überschreiben.
- **Zufügen von Methoden im Untertyp**: breite Untertyprelation wie bei Records.
- **Überschreiben von Methoden**:
Signaturen brauchen nicht äquivalent zu sein (wie in Java):
kontravariante Parametertypen und **kovariante Resultattypen** - wie bei Funktionen.
(Eiffel erlaubt kovariante Parametertypen: Lücke in der Typsicherheit.)

```
A a; X x; P p; C c; B b;

a = x.m (p); Übersetzer prüft Typen gegen Signatur von m in X und
Methoden-Untertyp zwischen m in Y und m in X

class X { C m (Q q) { use of q;... return c; }

    v̇ v̇ ^

class Y { B m (R r) { use of r;... return b; } }
```

Beispiel für Änderung des Resultattyps

Überschreiben der einer Methode `deepClone`, die einen Klon des Unterklassenobjektes bildet, benötigt **Änderung des Resultattyps**:

```
class CClass {
  var Cvar: XType;
  function deepClone (): CType is
  { var newClone: CType := self<=clone (); -- shallow clone
    newClone<=setCVars ();
    return newClone;
  }
  function setCVars (): void is {self.Cvar<=deepClone();}
}

class SClass inherits CClass modifies deepClone {
  var SCvar: YType;
  function deepClone (): SType is
  { var newClone: SType := self<=clone (); -- shallow clone
    newClone<=setSCVars ();
    return newClone;
  }
  function setSCVars (): void is
  { self<=setCVars (); self.SCvar<=deepClone(); }
}
```

Typisierungsproblem: binäre Methode

Methode soll Operation mit Operanden gleichen Typs realisieren, z.B. `equals`.
 Überschreiben in Unterklasse würde **kovariante Änderung des Parametertyps** erfordern:

```
class CClass {
  function equals (other: CType): Bool is
  { ... }
}

class SClass inherits CClass modifies equals {
  function equals (other: SType): Bool is
  { ... (SType) other <= ... }
}
```

Wird teilweise durch parametrisierte Typen gelöst.

1.3 Generik

Generik: zusätzliche Ebene der **Parametrisierung zur Übersetzungszeit**

Abgrenzung:

- **Funktionsaufruf:** Funktionen mit formalen Parametern werden mit aktuellen Parameterwerten zur **Laufzeit** aufgerufen.
- **Generik:** Aus **Schemata** für Definitionen von Programmkonstrukten (z. B. Klassen, Module, Funktionen) werden zur **Übersetzungszeit** Definitionen erzeugt.
Ein Schema hat **formale generische Parameter** für z. B. Typen, Klassen, Funktionen.
Dafür werden bei der **Instanziierung** **aktuelle generische Parameter** eingesetzt.
- **Makro-Substitution:** Makros sind **Symbolfolgen** mit formalen Parametern. An den Anwendungsstellen werden sie nach **Substitution** der aktuellen Parameter (auch Symbolfolgen) eingesetzt.
Substitution durch den **Präprozessor** ohne Beachtung der syntaktischen Struktur und der statischen Semantik.

```
int ggt (int a, int b)
{ ... }
ggt (m, n)
```

Java 1.5:

```
class Stack <E>
{ void push (E e)
  { ... }
  ...
}
Stack<Kreis> kreisStk;
```

```
#define PSEL(var, fld) \
((var)->fld)
PSEL (a[i], next)
```

Zweck der Generik

Abstraktion durch Programmschemata, z. B.

- Kellerimplementierung mit Elementtyp als generischem Parameter
- Sortierfunktion mit Elementtyp und Vergleichsfunktion als generische Parameter

Ermöglicht **Parametrisierung** mit Programmobjekten, die **nicht als Laufzeitdaten** existieren, nicht „first class“ sind; je nach Sprache: Typen, Klassen, Module, Funktionen.

Ergebnis der Instanziierung wird auf **Einhaltung der statischen Sprachregeln** überprüft, insbes. Typregeln;
z. B. Übersetzer garantiert homogene Behälter

Konsistenzbedingungen für aktuelle generische Parameter sind formulierbar und prüfbar, z. B. generische Sortierfunktion: Elementtyp mit passender Vergleichsfunktion

Insgesamt:

zusätzliche **Abstraktionsebene**

zusätzliche **Programmsicherheit** durch schärfere Regeln, schärfere statische Prüfungen

In Sprachen, die kaum statische Regeln haben, ist Generik nicht sinnvoll, z. B. Smalltalk.

Generische Definitionen (seit Java 1.5)

Eine **Generische Definition** hat **formale generische Parameter**. Sie ist eine **abstrakte Definition einer Klasse** oder eines Interfaces. Für jeden generischen Parameter kann ein **Typ (Klasse oder Interface)** eingesetzt werden. (Er kann auf Untertypen eines angegebenen Typs eingeschränkt werden.)

Beispiel in Java:

Generische Definition einer Klasse `Stack` mit generischem Parameter für den **Elementtyp**

```
class Stack<Elem>
{   private Elem [] store ;
    void push (Elem e1) {... store[top]= e1;...}
    ...
};
```

Eine **generische Definition wird instanziiert** durch Einsetzen von **aktuellen generischen Parametern**. Dadurch entsteht zur Übersetzungszeit eine Klassendefinition. Z. B.

```
Stack<Float> taschenRechner = new Stack<Float>();
Stack<Frame> windowMgr = new Stack<Frame>();
```

Generische Instanziierung kann im Prinzip durch **Textersetzung** erklärt werden: Kopieren der generischen Definition mit Einsetzen der generischen Parameter im Programmtext.

Der Java-Übersetzer erzeugt für jede generische Definition eine Klasse im ByteCode, in der `Object` für die generischen Typparameter verwendet wird. Er setzt Laufzeitprüfungen ein, um zu prüfen, dass die ursprünglich generischen Typen korrekt verwendet wurden.

Parametrisiertes Interface für binäre Methoden

Ein **formaler Typparameter im Interface** bestimmt den Typ des zweiten Operanden. Die implementierende Klasse bindet den **eigenen Typ an den Parameter**.

```
interface Comparable <T> {
    function equals (other: T): Bool;
}

class CClass implements Comparable<CType> {
    function equals (other: CType): Bool is
    { ... }
}

class SClass implements Comparable<SType> {
    function equals (other: SType): Bool is
    { ... other <= ... }
}
```

In diesem Modell kann leider nicht `SClass` Unterklasse von `CClass` sein und `equals` überschreiben.

Generik in C++

Generische Programmschemata heißen in C++ **Templates**.

Templates für Klassen, deren separat definierte **Methoden**, für eigenständige **Funktionen**.

Generische Parameter können sein:

Klassen, Typen, Templates, Funktionen, Variable, Zahlkonstante
auch Default-Werte für formale generische Parameter

Generische Parameter **nur ohne Restriktionen**:

```
template<class Elem>
class Stack
{
    ...
    void push (Elem e);
    ...
}

template<class Elem>
Stack::push (Elem e)
{...}
```

Funktion als generischer Parameter:

Klasse:

```
template<class T, void(*err_fct)()>
class List {...}
```

Funktion:

```
template<class T>
void sort(Array<T>& a) {...}
```

Instanziierungen:

```
Stack<int> s1;
List<Book, error_handl> lb;
Array<Book> ab;
sort(ab);
```

Generik in Eiffel

Eiffel hat ein **einheitliches Typkonzept**: Alle Typen sind durch Klassen definiert.

Generik: Klassendeklarationen können **Typen als generische Parameter** haben.

Instanziierung einer generischen Klasse liefert einen Typ.

Generischer Parameter **ohne Restriktionen**, Generische Parameter **mit Restriktionen**:
z. B. für Behälterklassen:

```
class Stack [ELEM] feature
    ...
    push (e: ELEM) is ...
    ...
end -- class Stack
```

Instanziierung als Typangabe:

```
si: Stack[INTEGER];
d: Dictionary[Book, STRING];
```

```
class Dictionary [G, KEY->Hashable]
    feature ...
end;
```

Der generische Parameter KEY ist auf
Unterklassen von Hashable eingeschränkt.
In der Klasse Dictionary können Hashable-
Methoden für KEY-Objekte benutzt werden.

Generik in Ada 95

Ada 95 hat ein **umfangreiches Typkonzept**;

Ableitung von Typen ist mit Vererbung zwischen Klassen vergleichbar;
weitere OO-Konzepte begründen den Anspruch OO-Sprache zu sein.

Generisch deklarierbar: Funktionen, Prozeduren und Packages (Module).

Generische Parameter können sein

Typen, auch mit vielfältigen Restriktionen, z. B. Obertyp oder Array-Typ,
Funktionen, Variable, Konstante

Beispiel:

```
generic Type T is private
procedure Swap (X, Y: in out T);

procedure Swap (X, Y: in out T) is
  tmp: constant T := X;
begin ... end;
```

Instanziierungen:

```
procedure Swap_Int is new Swap (Integer);
procedure Swap_Vect is new Swap (Vect);
```

2. Einsatz von Vererbung, Übersicht

Vererbung ist ein **Konzept und Mechanismus** in objektorientierten Programmiersprachen
(Abschnitt 2.1)

Sie wird eingesetzt für unterschiedliche **Zwecke des konzeptionellen Entwurfs** und
für unterschiedliche **Zwecke der Programmentwicklung**.

Welchem Zweck eine Vererbungsrelation dient, ist **schwierig am Programmtext zu erkennen**.

Einsatzzweck Entwurfskonzepte:

- 2.2 Abstraktion
- 2.3 Spezialisierung
- 2.4 Rollen, Eigenschaften
- 2.5 Spezifikation

Einsatzzweck Programmentwicklung:

- 2.6 Inkrementelle Weiterentwicklung
- 2.7 Komposition statt Vererbung
- 2.8 Weitere Techniken

siehe auch [Antero Taivalsaari: On the Notion of Inheritance, ACM Computing Surveys, Vol. 28, No. 3, September 1996]

2.1 Mechanismen und Konzepte Mechanismen der Sprache

- **Unterklasse S** zur Oberklasse A bilden: **subclassing**
- transitive Vererbungsrelation bildet **Klassenhierarchie**
- Unterklasse kann **Methoden und Datenattribute** ihrer Oberklassen **erben**
- Unterklasse kann **zusätzliche Methoden und Datenattribute definieren**
- Unterklasse kann **Methoden** ihrer Oberklassen **überschreiben**
- **Schnittstelle für Methoden spezifizieren**: abstrakte Methode
- **Methodenschnittstelle** durch konkrete Methode **implementieren**
- **Schnittstelle für Klassen spezifizieren**: Interface
- **Klasse implementiert** die für sie spezifizierten **Schnittstellen**
- **polymorphe Variable** haben statischen Typ T und können Objektreferenzen anderer Typen S aufnehmen.
S ist Unterklasse von T oder S implementiert die Schnittstelle T

Konzeptionelle Begriffe

Vererbung bewirkt gleichzeitig **Erweiterung** der Funktionalität und **Einschränkung** der Verwendbarkeit

```
class PinBallGame extends Frame { ... }
```

Überschreiben einer Methode mit einer anderen kann die Funktionalität **verändern**.

Erschwert es, die konzeptionelle Bedeutung der Methode zu verstehen.
Missbrauch ist möglich.

Ersetzbarkeit (substitutability):

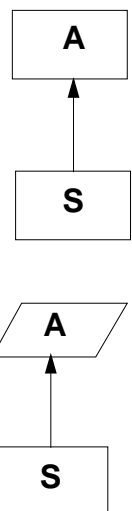
Objekte einer Klasse **s** sind in Kontexten verwendbar, wo Objekte der Klasse **A** benötigt werden.
Abweichungen vom Verhalten, das für **A** definiert ist, sind nicht beobachtbar.

Untertypen (subtyping):

wie Ersetzbarkeit, aber ausgedrückt durch Typen:
Eine Variable vom Typ **A** kann **s**-Werte aufnehmen.
Verhaltensabweichungen sind nicht beobachtbar.

Untertypeneigenschaft ist **technisch immer erfüllt** für
Unterklasse **s** zur Oberklasse **A**, Klasse **s** zum Interface **A**

Untertypeneigenschaft ist **konzeptionell** in diesen Fällen **nicht immer erfüllt**,
z. B. wenn die Definition von **s** eine Methode aus **A** durch Überschreiben „eliminiert“.



Kriterien zur Anwendung von Vererbung

Nach [Coad, North, Mayfield], rigorose Kriterien **gegen freizügige Vererbung**:

1. **is-a Beziehung** muss gelten: Unterklasse S ist eine besondere Art der Oberklasse A führt zum **Paradigma Abstraktion** (siehe 2.2)
Beispiel: Kaufvertrag, Mietvertrag, Darlehensvertrag sind **Arten von Verträgen**
2. Objekte **gehören der Unterklasse für immer an**, sie wechseln nicht in eine andere Klasse
Gegenbeispiel: Kaulquappe wird erwachsener Frosch; führt zu Rollen (2.4) statt Unterklassen
3. **Unterklasse erweitert** die Fähigkeiten der Oberklasse, sie **entfernt keine** und **ändert keine**
Gegenbeispiel: Stack durch Vektor implementiert; besser: Implementierung mit Komposition
4. Die Relation „A spielt eine **Rolle S**“ **nicht** durch Unterklasse S modellieren sondern durch Rollen (siehe 2.4)
Beispiel: Person spielt Rolle Student, oder Rolle Assistent - oder beide!
5. **Nicht** Unterklasse S aus A bilden, nur weil die **Funktionalität von A gut nutzbar** ist
Beispiel: Stack durch Vektor implementiert
6. **has-a Beziehung nicht** durch Unterklasse implementieren sondern durch **Komposition**
Beispiel: Auto hat ein Antriebsaggregat
7. **Komposition** bewahrt die **Kapselung** der Klasse, **Vererbung öffnet** sie zur Unterklasse hin

2.2 Abstraktion

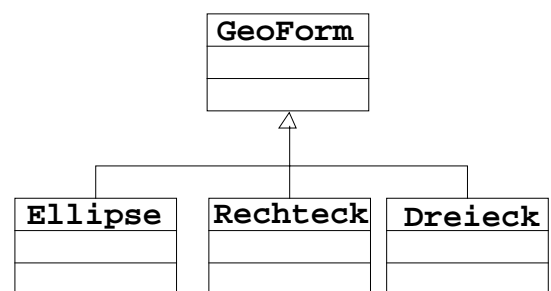
Unterklassen modellieren

verschiedene Arten eines abstrakten Konzeptes

(„ist spezielle Art von“, „is-a“)

Beispiele:

verschiedene Arten von geometrischen Formen,
verschiedene Arten von Dokumenten,
Strukturbäume in Übersetzern,
Design Pattern **Composite**



Einteilung in **disjunkte Objektmen**gen der Unterklassen,
kein Objekt ist zugleich in mehreren der Unterklassen
führt zu **Hierarchiebaum**.

Man muss jede Ebene vollständig kennen!

vergleiche: Klassifikation in der biologischen Systematik

Gegenbeispiele:

Fahrzeuge <- Landfahrzeuge, Wasserfahrzeuge <- Amphibienfahrzeuge

Schnabeltier: eierlegendes Säugetier

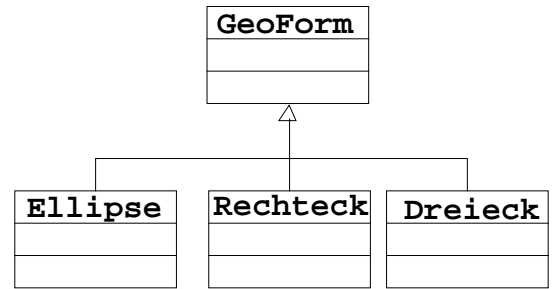
signalisieren Entwurfsfehler:

nicht durch Mehrfachvererbung lösen, sondern durch Rollen

alle **inneren Klassen der Hierarchie sind konzeptionell abstrakt**,
Objekte nur zu den Blattklassen

Entwurf mit dem Paradigma Abstraktion

1. **Operationen, die allen Arten gemeinsam sind**, als Methoden mit ihren Datenattributen in der Oberklasse zusammenfassen, z. B. Operationen auf Koordinaten der Formen
 2. **Operationen, die konzeptionell Gleiches leisten**, aber unterschiedliche Ausprägungen haben, als **Methoden-Schnittstellen** (abstrakte Methoden) in die Oberklasse, z. B. Flächenberechnung
 3. Bei strenger Anwendung des Paradigmas **kommt Überschreiben nicht vor**, wohl aber **Implementierung abstrakter Methoden**; Ausnahmen z. B. Überschreiben zur Spezialisierung von „Service-Methoden“
 4. Beim Entwurf muss **eine Hierarchieebene hinreichend vollständig** bekannt sein; sonst müssen die Abstraktionsentscheidungen beim Zufügen neuer Klassen revidiert werden.
Beispiel: wenn zu den Formen noch Linien hinzukommen, ist Flächenberechnung fragwürdig
 5. Entwurf meist **bottom-up, um die Funktionalität zu explorieren**
- Achtung:** Spezialisierung in **eine** Unterklasse ist ein anderes Paradigma (siehe 2.4)



Strukturbäume zu Grammatiken

Kontext-freie Grammatik definiert Baumstrukturen (abstrakte Syntax)

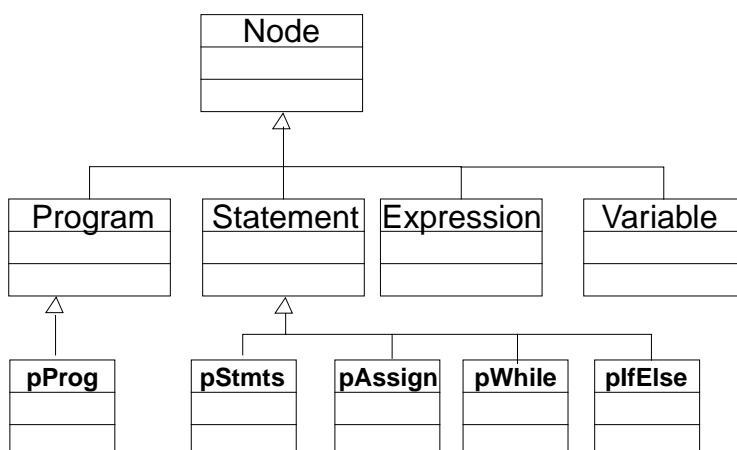
Beispiel:

pProg: Program ::= Statement
 pStmts: Statement ::= Statement Statement
 pAssign: Statement ::= Variable Expression
 pWhile: Statement ::= Expression Statement
 plfElse: Statement ::= Expression Statement Statement

2-mal Paradigma Abstraktion:

1. alle Nichtterminalklassen abstrahiert zu Node
2. die alternativen Produktionen eines Nichtterminals N abstrahiert zu N

3-stufige Klassenhierarchie:



allgemeine Knotenklasse

abstrakte Methode zum Baumdurchlauf

Nichtterminalklassen

Methoden zum Setzen/Lesen spezieller Attribute, z. B. Typ von Expression

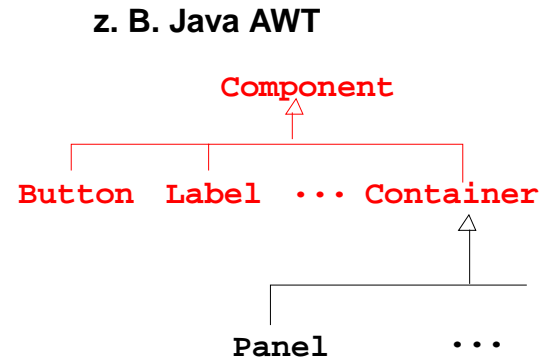
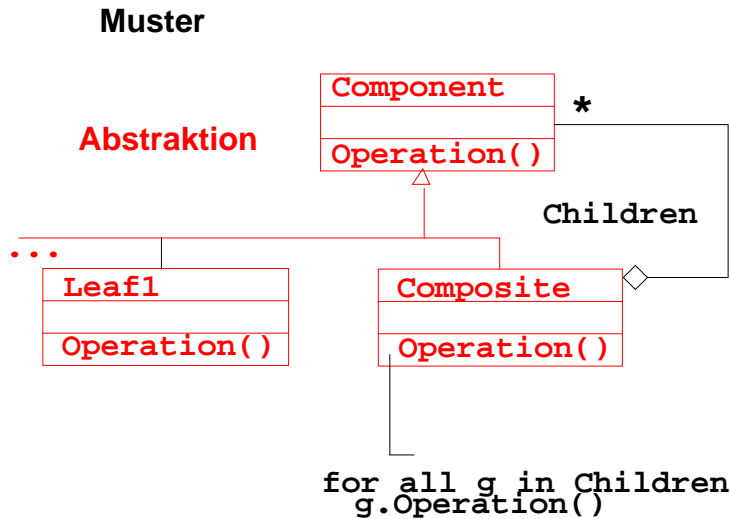
Produktionenklassen

Unterbäume, Methoden zum Durchlauf, Attributberechnung

Design Pattern Composite

Komposition zusammengesetzter Strukturen aus einfachen Komponenten unterschiedlicher Art, auch rekursiv.

Abstraktion der Blattklassen und Composite-Klasse zur Klasse Component



2.3 Spezialisierung

Zu einer gegebenen Oberklasse wird **eine spezialisierte Unterklasse** gebildet

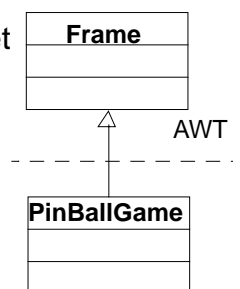
- Echte Erweiterung** der Oberklasse im Sinne der **Ersetzbarkeit**; die Unterklasse fügt spezifische Operationen hinzu. Die **is-a-Relation** gilt.
- Die Oberklasse realisiert ein **komplexes Konzept**.
- Es kommt in **anwendungsspezifischen Ausprägungen** vor.
- Die Oberklasse liefert umfangreiche Funktionalität

Wiederverwendbarkeit ist sorgfältig **geplant**, sehr **aufwändig**, sehr **wirksam**

- Balance zwischen **mächtiger Funktionalität** und **vielfältiger Verwendbarkeit**,
- vorbereitete **Erweiterungsstellen**: Unterklasse impl. Methoden, Oberklasse ruft sie auf Techniken z.B. abstr. Methoden, Reaktion auf Ereignis, Verfahren an Schnittstelle einsetzen
- Die Oberklasse ist häufig zusammen mit **verwandten Konzepten** Teil eines **objektorientierten Programmgerüsts (Framework)**, z. B. Java's AWT

Abgrenzung:

- zu *Abstraktion*: dort bottom-up, hier top-down; dort ganze Hierarchieebene, hier einzeln
- zu *inkrementeller Weiterentwicklung*: dort nicht Ersetzbarkeit, nicht is-a, Erweiterung ad hoc



Frame Beispiel für Spezialisierung

Oberklasse **Frame** aus Java's AWT-Framework für Benutzungsoberflächen

Konzept:

1. Bildschirmfenster mit Bedienungsrahmen,
2. Fläche zum Aufnehmen und Anordnen von Bedienungselementen und
3. zum Zeichnen.

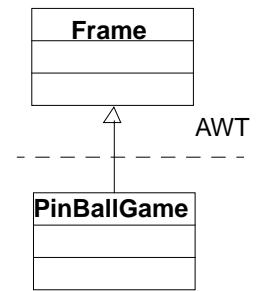
Umfangreiche Funktionalität:

4. Zeichnen von Fenster mit Inhalt,
5. Verändern des Fensters und des Rahmens,
6. Anordnen von Bedienungskomponenten

Erweiterungsstellen:

7. leere Methode `paint` zum Zeichnen, wird in Unterklasse überschrieben und aus dem Framework aufgerufen,
8. Einstellen der Strategie zur Anordnung der Bedienelemente (**LayoutManager**)
9. Ereignisbehandlung ist vorbereitet, ausfüllen mit **EventListener**

Frame in der AWT-Framework **zusammen mit anderen Container-Klassen** (**Panel**, **ScrollPane**, ...)



Thread als Beispiel für Spezialisierung

Oberklasse **Thread** im Paket `java.lang`

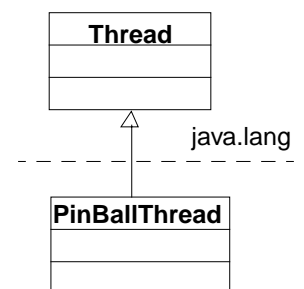
Konzept: parallel ausführbarer, leichtgewichtiger **Prozess**

umfangreiche **Funktionalität:**

- Prozesse starten, anhalten, synchronisieren,
- Prozessverwaltung mit Scheduler und Prozessumschaltung

Erweiterungsstelle:

leere Methode `run` für den Prozess-Code, wird von der Unterklasse überschrieben und vom System aufgerufen.



2.4 Rollen, Eigenschaften

2.4.1 Werkzeug-Material-Paradigma (statisch)

Rolle:

Fähigkeit in bestimmter Weise zu (re)agieren.

Verkäufer: anbieten, liefern, kassieren

Eigenschaft:

Fähigkeit in bestimmter Weise zu (re)agieren.

Steuerbar: starten, anhalten, wenden

Rollen und Eigenschaften als **Schnittstellen:**

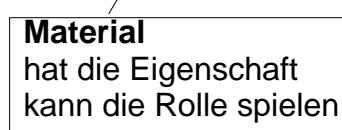
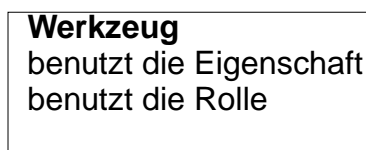
Spezifikation von Rollen oder Eigenschaften **entkoppelt die Entwicklung** von Werkzeugen und Material:

Zusammenfassen charakteristischer Fähigkeiten oder Operationen

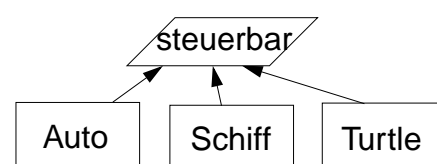
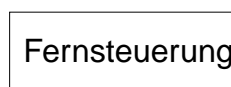
Werkzeug sieht Objekte nur in Ihrer Rolle

Rollen entkoppeln Software-Module:

Neue Werkzeuge, neues Material können bei der Entwicklung zugefügt werden



Beispiel:



© 2009 bei Prof. Dr. Uwe Kastens

Rollen, Eigenschaften spezifiziert durch Interfaces

Die **Fähigkeiten** einer Rolle, einer Eigenschaft werden als **Methodenspezifikationen** in einem **Interface** zusammengefasst:

```
interface Movable
{ boolean start (); void stop (); boolean turn (int degrees); }
```

Material-Klassen haben Eigenschaften, können Rollen spielen; sie implementieren das Interface und ggf. weitere Interfaces:

```
class Turtle implements Movable
{ public boolean start () {...}
  // Implementierung von stop und turn sowie weiterer Methoden ...
}
class Car implements Movable, Motorized
{ // Implementierung der Methoden aus Movable und Motorized, sowie weiterer ...
}
```

In **Werkzeug-Klassen** werden Interfaces als **Typabstraktion** verwendet und werden die spezifizierten Fähigkeiten benutzt, z. B. innerhalb der Klasse zur Fernsteuerung:

```
Movable vehicle;... vehicle = new Car();... vehicle.turn(20);...
void drive (Movable vehicle, Coordinate from, Coordinate to)
{ vehicle.start (); ... vehicle.turn (45); ... vehicle.stop (); }
```

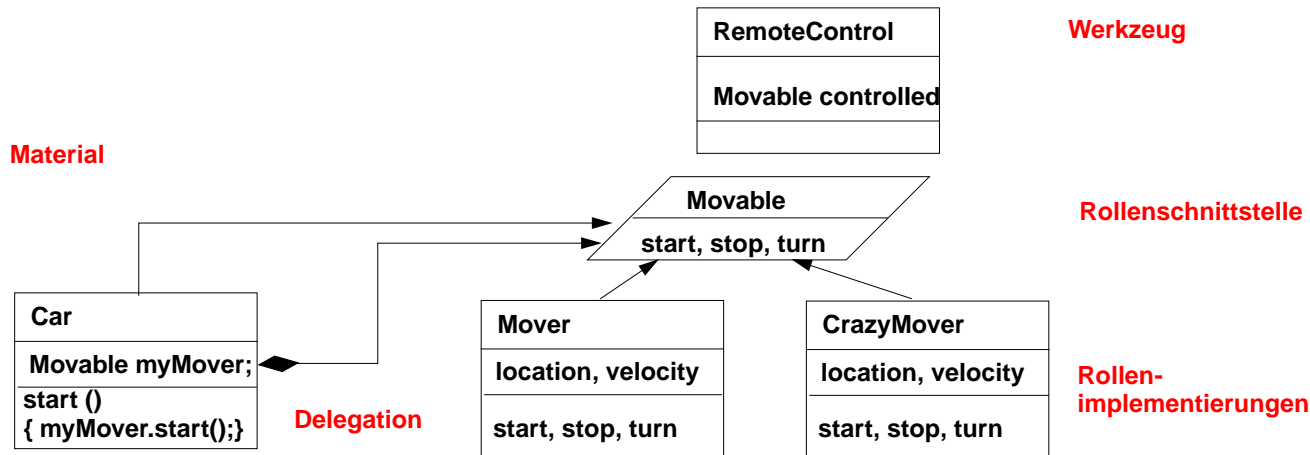
© 2005 bei Prof. Dr. Uwe Kastens

2.4.2 Delegation an implementierte Rolle (dynamisch)

Manche **Rolle** kann man **implementieren** - unabhängig von den Klassen, die die Rolle spielen.

Delegation vermeidet die wiederholte Implementierung der Rollenschnittstelle in jeder der *Material-Klassen*.

Rollenschnittstelle wird weiterhin für die Werkzeuge benötigt.



Dynamisch wechselnde Rollen sind so realisierbar, z. B.
 Nach Zusammenstoß wird das Mover-Objekt durch ein CrazyMover-Objekt ersetzt.
 In der Frog-Klasse wird das Kaulquappen-Verhalten durch ein Frosch-Verhalten ersetzt.

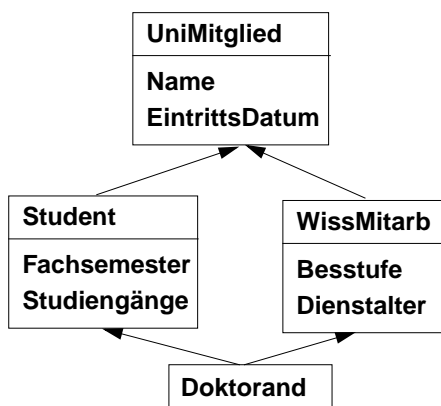
Rollen statt Abstraktion

Häufiger **Entwurfsfehler**: „hat-Rollen“ als nicht-disjunkte Abstraktion modelliert

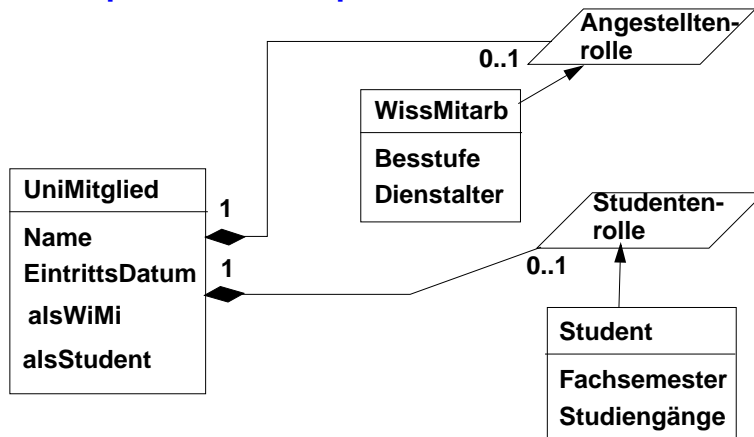
Symptome: „is-a“ gilt nicht; mehrere Rollen können zugleich oder nacheinander gespielt werden

Einsatz von Mehrfachvererbung verschlimmert das Problem

schlecht:
nicht-disjunkte Abstraktion



gut:
Komposition mit implementierten Rollen

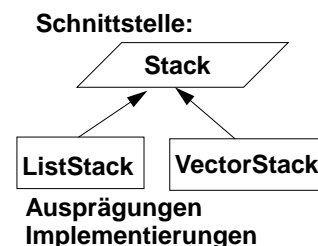


schlecht „Abstraktion für hat-Rollen“:
 bei mehreren „Rollen-Klassen“ wird eine große Anzahl von „Kombinationsklassen“ benötigt,
 mehrere Rollen sind nicht zugleich möglich,
 dynamisch zugeordnete Rollen sind nicht möglich.

2.5 Spezifikation

Ein **abstraktes Konzept** wird durch eine **Schnittstelle** spezifiziert;
z. B. das Konzept Keller

```
public interface Stack
{ void push (Object x); void pop();
  Object top (); boolean isEmpty();
}
```



Konkrete Ausprägungen oder **Implementierungen** des Konzeptes durch Klassen,
die die Schnittstelle implementieren.

Ziel: **Entkopplung** der Entwicklung von Anwendungen und Realisierungen

Gleiches Prinzip:

abstrakte Klasse statt Interface, mit Methodenschnittstellen und implementierten Methoden

Vergleich zu **Spezialisierung**:

Spezifikation betont die Schnittstelle (keine oder wenige implementierte Methoden),
Spezialisierung betont das Wiederverwenden der Implementierung in der Oberklasse

Abgrenzung gegen **Abstraktion**:

Spezifikation: top-down, Ausprägungen werden **einzeln** entwickelt

Abgrenzung gegen **Rollen**:

Spezifikation: eine konkrete Ausprägung gehört zu nur **einem** abstrakten Konzept
Rollen: eine Materialklasse kann **mehrere** Rollen spielen

Beispiele für Spezifikation

1. **Spezifikation einer Datenstruktur** (z. B. Stack),
um ihre Implementierungen zu entkoppeln

2. **Ereignisbehandlung** in der AWT-Bibliothek für
Benutzungsoberflächen:

```
interface ActionListener
{ void actionPerformed (ActionEvent e); }
```

Bei komplexeren Schnittstellen (z. B. **MouseListener**) auch
abstrakte Klasse (**MouseAdapter**) mit Default-Implementierungen der Methoden.

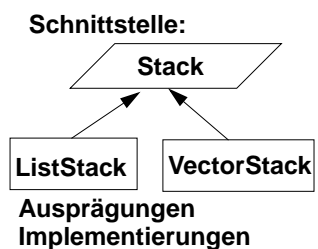
3. **Abstraktes Konzept „arithmetische Typen“**
mit arithmetischen Operationen +, -, *, /, zero, one, ...:

Einige konkrete Ausprägungen:

Integer, Long, Float, Double, Polynom, Matrix, ...

Number in der Bibliothek java.lang könnte so definiert sein.

Grenzfall zum Paradigma „Abstraktion“



2.6 Inkrementelle Weiterentwicklung

Die **Funktionalität** einer Oberklasse wird **genutzt**, um sie zu einer neuen Unterklasse **weiterzuentwickeln**.

Mit **geringem Entwicklungsaufwand** wird etwas Neues geschaffen.

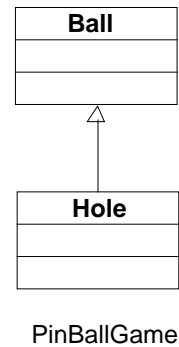
Die **Anforderungen der Spezialisierung** (2.3) brauchen **nicht alle erfüllt** zu werden:

1. Das **Konzept** der Oberklasse (sie ist meist konkret) braucht für die Unterklasse nicht mehr zu gelten.
2. Die **is-a-Relation** braucht nicht zu gelten.
3. Konzeptionell braucht „**subtyping**“ nicht erfüllt zu sein.
4. Die Unterklasse braucht sich **nicht auf Erweiterungen zu beschränken**; durch Überschreiben können **Methoden konzeptionell verändert oder gelöscht** werden.
5. Diese **Wiederverwendung** der Oberklasse ist **so nicht geplant** worden.

Meist zeigt die inkrementelle Weiterentwicklung **Probleme wegen unpassender** Konzepte, Methoden, Methodennamen, Methodensignaturen, Implementierungen, ...

Meist ist die bessere Lösung **Komposition statt Vererbung**:

Ein Objekt der neuen Klasse nutzt ein Objekt der gegebenen Klasse zur Implementierung seiner Funktionalität.



Beispiel für inkrementelle Weiterentwicklung

Die Klasse **Ball** in den Fallstudien des Buches von Budd realisiert das **Konzept von Bällen, die sich in der Ebene bewegen**. Sie hat Methoden, die Eigenschaften (Ort, Farbe, Geschwindigkeitsvektor) lesen und schreiben und den Ball zeichnen.

Im **PinBallGame** werden **Löcher als Hindernisse** benötigt.

Man kann sie einfach aus der Klasse **Ball** durch **Vererbung entwickeln**:

```

class Hole extends Ball implements PinBallTarget
{
    public Hole (int x, int y)
    {
        super (x, y, 12); setColor (Color.Black);
    }

    public boolean intersects (Ball aBall)
    {
        return location.intersects (aBall.location);
    }

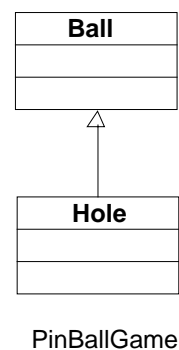
    public void hitBy (Ball aBall)
    {
        aBall.moveTo (0, PinBallGame.FrameHeight+30);
        aBall.setMotion (0, 0);
    }
}
  
```

Die Implementierung ist **beeindruckend kurz und einfach**.

Sie erbt **move**, **paint**, Zugriffsmethoden, ...

Das **Konzept** „Loch“ unterscheidet sich vom Konzept „beweglicher Ball“; **is-a** gilt nicht, **subtyping** gilt nicht konzeptionell.

Es wird **nur erweitert**; die **unnötige „Beweglichkeit** von Löchern“ wird in Kauf genommen.



T. Budd: *Inheritance for Construction*

T. Budd: Understanding Object-Oriented Programming with Java; updated ed; Addison Wesley, 2000
pp. 129, 130, 164ff

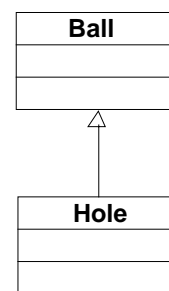
„A class can often **inherit almost all of its desired functionality** from a parent class, perhaps changing only the names of the methods used to interface to the class, or modifying the arguments. This may be true even if **the new class and the parent class fail to share any relationship as abstract concepts**.

An example of subclassification for construction occurred in the pinball game application described in Chapter 7. In that program, the class `Hole` was declared as a subclass of `Ball`. There is **no logical relationship between the concepts of a `Ball` and a `Hole`**, but from a practical point of view much of the behavior needed for the `Hole` abstraction matches the behavior of the class `Ball`. Thus using inheritance in this situation **reduces the amount of work** necessary to develop the class `Hole`.“

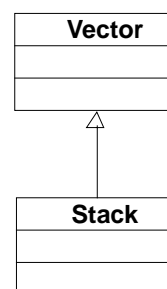
Similar explanations are given on p. 130 for class `Stack` derived from `Vector`.

In Sect. 10.2, 10.3 (pp. 164) Budd discusses **composition** versus **inheritance**:

„But **composition** makes no explicit or implicit claims about substitutability. When formed in this fashion, the data types `Stack` and `Vector` are entirely distinct and neither can be substituted in situations where the other is required.“



PinBallGame



java.util

Schlechte inkrementelle Weiterentwicklung

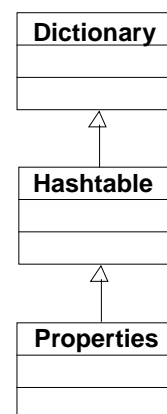
`java.util.Dictionary` spezifiziert das **abstrakte Konzept** einer als **Abbildung** organisierten **Menge von Paaren** (Key, Element).
Key und Element sind **beliebige Objekte**. Signatur zweier Methoden:

```
Object put (Object key, Object elem);
Object get (Object key);
```

`Hashtable` implementiert das abstrakte Konzept `Dictionary`.

`Properties` ist eine **inkrementelle Weiterentwicklung** aus `Hashtable`;
realisiert dasselbe Konzept wie `Dictionary`,

aber - Key und Element sind beide **string-Objekte**,
- hat **zusätzliches Konzept**: vielstufige Suche in **Default-Properties**



java.util

Probleme:

1. Mit `Hashtable-put` können auch nicht `string`-Paare eingefügt werden; in Java 1.2 wird eine `Properties`-Methode `setProperty` für `String`-Paare nachgeliefert.
2. Ergebnisse von `Hashtable-get` sind nicht notwendig `Strings`; deshalb gibt es eine `Properties`-Methode `getProperty` für `String`-Paare, seit Java 1.2 zeigt sie auch nicht-`string`-Paare nicht an.
3. `getProperty` sucht ggf. auch in der `Default-Menge`, `get` tut das nicht.
4. **Geerbte Methoden passen nicht. Chaos!!**
Bei **Komposition statt Vererbung** könnte man sie anpassen!

Eigene und geerbte Methoden von Properties

Method Summary	
String	getProperty (String key) Searches for the property with the specified key in this property list.
String	getProperty (String key, String defaultValue) Searches for the property with the specified key in this property list.
void	list (PrintStream out) Prints this property list out to the specified output stream.
void	list (PrintWriter out) Prints this property list out to the specified output stream.
void	load (InputStream inStream) Reads a property list (key and element pairs) from the input stream.
Enumeration	propertyNames () Returns an enumeration of all the keys in this property list, including the keys in the default property list.
void	save (OutputStream out, String header) Deprecated. This method does not throw an IOException if an I/O error occurs while saving the property list. As of JDK 1.2, the preferred way to save a properties list is via the <code>store(OutputStream out, String header)</code> method.
Object	setProperty (String key, String value) Calls the hashtable method put.
void	store (OutputStream out, String header) Writes this property list (key and element pairs) in this Properties table to the output stream in a format suitable for loading into a Properties table using the load method.

Methods inherited from class java.util.Hashtable
clear, clone, contains, containsKey, containsValue, elements, entrySet, equals, get, hashCode, isEmpty, keys, keySet, put, putAll, rehash, remove, size, toString, values

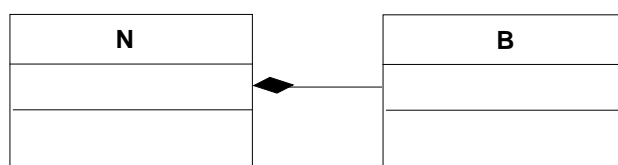
aus Version 1.2 of Java Platform API Specification, Copyright 1993-1998 Sun Microsystems

© 2005 bei Prof. Dr. Uwe Kastens

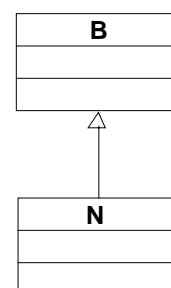
2.7 Komposition statt Vererbung

Gründe für die Bildung einer **neuen Klasse N** aus einer **bestehenden Klasse B** durch **Komposition** statt durch Vererbung:

1. Objekt der Klasse N **enthält** Objekt(e) der Klasse B;
has-a-Relation statt is-a-Relation
2. N-Objekte **spielen B-Rollen**.
3. N realisiert ein **anderes Konzept** als B (kein subtyping).
4. Einige **Methoden aus B passen schlecht** für N („entfernen“, kein subtyping)
5. Die **Implementierung** von N durch B soll **nicht öffentlich** gemacht werden.
6. **Delegation** von Methoden der Klasse N an B-Objekte **entkoppelt** die Implementierungen.



Komposition



Vererbung

© 2010 bei Prof. Dr. Uwe Kastens

Vergleich: Stack als Unterklasse von Vector

Beide Klassen sind im Package `java.util` deklariert:

```
class Vector
{
    public boolean isEmpty () { ... }
    public int size () { ... }
    public void addElement (Object value) { ... }
    public Object lastElement () { ... }
    public Object removeElementAt (int index) { ... }
    ... viele weitere Methoden (-> OOP-2.23a)
}

class Stack extends Vector
{
    public Object push (Object item)
    {addElement(item); return item;}

    public Object peek ()
    { return elementAt (size() - 1); }

    public Object pop ()
    { Object obj = peek ();
      removeElementAt (size () - 1);
      return obj;
    }
}
```

© 2005 bei Prof. Dr. Uwe Kastens

Stack erbt viele Methoden von Vector

Fields inherited from class java.util.Vector
capacityIncrement, elementCount, elementData

protected

Fields inherited from class java.util.AbstractList
modCount

Method Summary	
boolean	empty() Tests if this stack is empty.
Object	peek() Looks at the object at the top of this stack without removing it from the stack.
Object	pop() Removes the object at the top of this stack and returns that object as the value of this function.
Object	push(Object item) Pushes an item onto the top of this stack.
int	search(Object o) Returns the 1-based position where an object is on this stack.

Methods inherited from class java.util.Vector	
add, add, addAll, addAll, addElement, capacity, clear, clone, contains, containsAll, copyInto, elementAt, elements, ensureCapacity, equals, firstElement, get, hashCode, indexOf, indexOf, insertElementAt, isEmpty, lastElement, lastIndexOf, lastIndexOf, remove, remove, removeAll, removeAllElements, removeElement, removeElementAt, removeRange, retainAll, set, setElementAt, setSize, size, sublist, toArray, toArray, toString, trimToSize	

public!!

© 2005 bei Prof. Dr. Uwe Kastens

Vergleich: Stack implementiert mit Vektor-Objekt

```

class Stack
{ private Vector theData;

  public Stack ()
  { theData = new Vector (); }

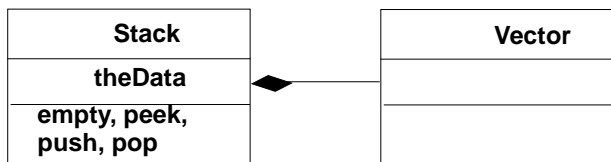
  public boolean empty ()
  { theData.isEmpty (); }

  public Object push (Object item)
  {theData.addElement(item); return item;}

  public Object peek ()
  { return theData.lastElement (); }

  public Object pop ()
  { Object result = theData.lastElement ();
    theData.removeElementAt (theData.size () - 1);
    return result;
  }
}

```



Komposition

Vergleich: Stack - Vector Vererbung - Komposition

- Stack und Vector sind **unterschiedliche Konzepte**; Vererbung verursacht subtyping-Problem
- Die **Kompositionslösung** ist **einfacher**.
Vollständige Definition, knapp und **zusammenhängend** an einer Stelle.
- Die **Vererbungslösung** ist **schwerer zu durchschauen**; auf zwei Klassen verteilt; man muss hin- und herblättern.
Viele **unnötige und unpassende Methoden** in der Klasse Vektor.
- Die **Vererbungslösung** ist **kürzer**.
- In der **Vererbungslösung** wird nicht verhindert, dass **ungeeignete Vector-Methoden** auf Stack-Objekte angewandt werden, z. B. `insertElementAt`.
Die **Schnittstelle** der Stack-Klasse ist **viel zu groß**.
- Die **Kompositionslösung verbirgt die Implementierungstechnik**.
Wir könnten sie ändern (z. B. lineare Liste statt Vector), ohne dass Anwendungen davon betroffen wären.
- Die Vererbungslösung hat **Zugriff auf protected Attribute** der Vector-Klasse (`elementData`, `elementCount`, `capacityIncrement`) und gibt den Zugriff an Unterklassen weiter. Komposition nutzt nur `public`-Zugriff.
- Die Vererbungsimplementierung ist etwas schneller.

2.8 Weitere Techniken, Varianten in mehreren Dimensionen

Beispiel `java.io.InputStream`:

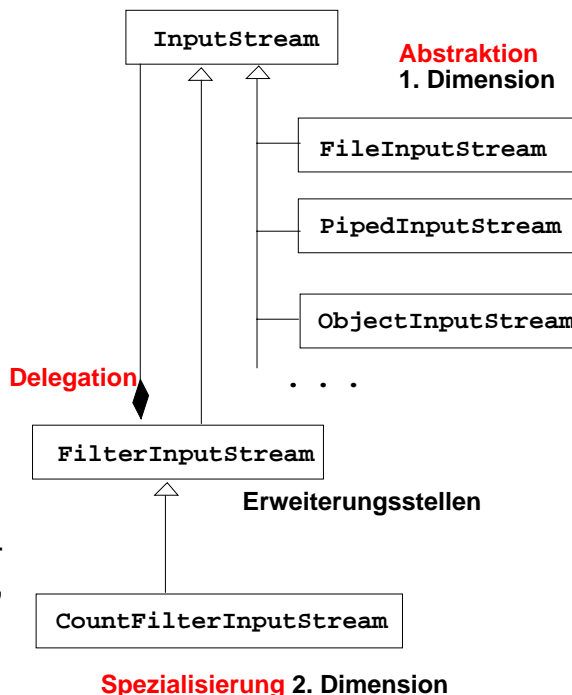
`InputStream` ist ein **abstraktes Konzept** zu verschiedenen Arten von Eingabequellen;
Abstraktion zur 1. Varianten-Dimension

`FilterInputStream` bereitet **Erweiterungsstellen** für die 2. Dimension von Varianten vor:
Delegation an ein `InputStream`-Objekt,
 Unterklasse von `InputStream`
 wegen **Spezialisierung**

Spezialisierung zu Varianten der 2. Dimension: Spezielle Filter;
 Wiederverwendung der delegierten Methoden

Bei Generierung eines `CountFilterInputStream`-Objektes wird ein **Delegationsobjekt übergeben**, das die Eingabequelle festlegt (1. Dimension):

```
new CountFilterInputStream
    (new FileInputStream ("myFile"))
```



Eingebettete Objekte als Agenten

Schema:

Ein **Agent** handelt für ein Institut;
 er ist untrennbar **an sein Institut gebunden**;
 Kunden **sehen nur den Agenten** - nicht das Institut
 Institut - Agent - Kunde

Beispiel: Versicherung - Vertreter - Kunde

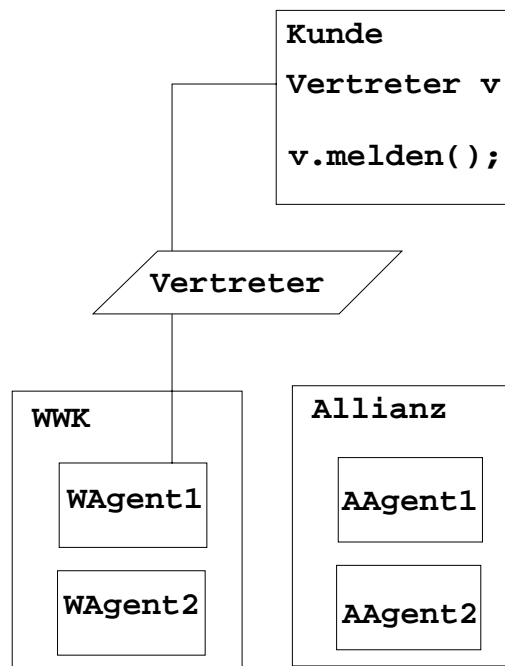
Technik: **eingebettete Klassen**

```
interface Vertreter
{ void abschließen(); void melden(); }

class Versicherung
{ void regulieren() {...}

    class Agent implements Vertreter
    { void abschließen () { ... }
      void melden ()
        {... regulieren() ... }
    }
}
```

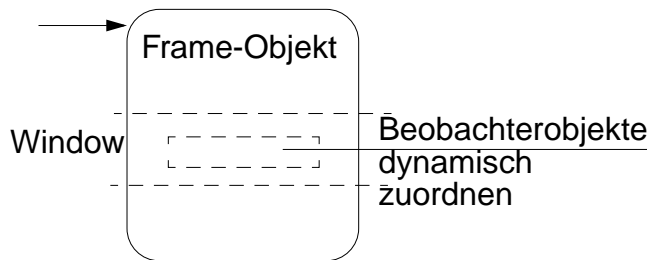
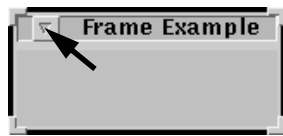
Objektbild:



Schachtelung der Klassen: enge Einbettung der Objekte
Innere Objekte erfüllen global sichtbare Schnittstelle,
 sind also außerhalb ihrer Umgebung verwendbar.

Beispiel: Listener der Ereignisbehandlung

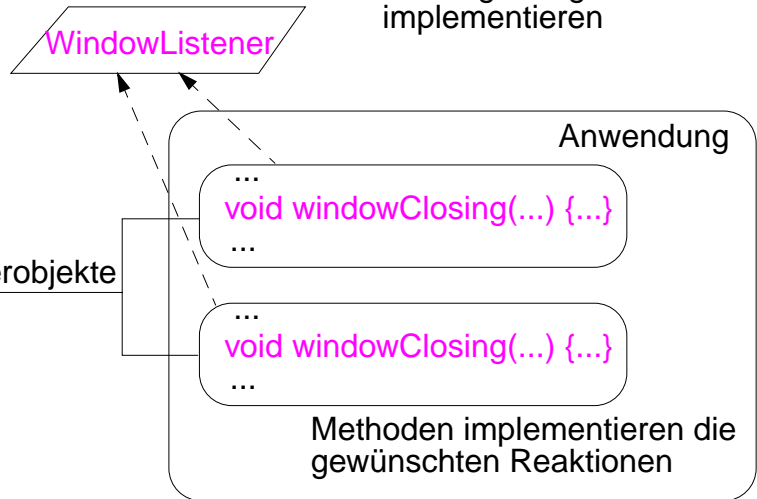
An AWT-Komponenten werden Ereignisse ausgelöst, z. B. ein WindowEvent an einem Frame-Objekt:



Ereignis auslösen: zugehörige Methode in jedem Beobachter-Objekt aufrufen

„eingebettete Agenten“

Beobachter für bestimmte Ereignistypen: Objekte von Klassen, die das zugehörige Interface implementieren



Entwurfsmuster „Observer“: Unabhängigkeit zwischen den Beobachtern und dem Gegenstand wegen Interface und dynamischem Zufügen von Beobachtern.

3. Entwurfsmuster zur Entkopplung von Modulen

Entwurfsmuster (Design Patterns):

Software-**Entwicklungsaufgaben**, die in vielen **Ausprägungen** häufig auftreten.

Objektorientierte Schemata, die als Lösungen **wiederverwendet** werden können.

Software-Qualitäten: erprobte Strukturen, wartbar, flexibel, adaptierbar.

Präsentation der Entwurfsmuster:

1. **Name:** Aufgabe treffend beschreiben, gut kommunizierbar
2. **Aufgabe:** typische Situation, gut wiedererkennbar, Struktur, Umgebung, Anwendungsbedingungen
3. **Lösung:** objektorientierte Strukturen, abstrakte Beschreibung variierbar, instanzierbar, Anwendungsbeispiele
4. **Folgerungen:** Nutzen, Kosten, Varianten; Implementierbeispiele

[E. Gamma, R. Helm, R. Johnson, J. Vlissides: Design Patterns, Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995]

Übersicht zu Entwurfsmustern

Purpose	Design Pattern	Aspect(s) That Can Vary
Creational	Abstract Factory (87) ●	families of product objects
	Builder (97)	how a composite object gets created
	Factory Method (107) ●	subclass of object that is instantiated
	Prototype (117)	class of object that is instantiated
	Singleton (127)	the sole instance of a class
Structural	Adapter (139)	interface to an object
	Bridge (151) ●	implementation of an object
	Composite (163)	structure and composition of an object
	Decorator (175)	responsibilities of an object without subclassing
	Facade (185)	interface to a subsystem
	Flyweight (195)	storage costs of objects
	Proxy (207)	how an object is accessed; its location
	Behavioral	Chain of Responsibility (223)
Command (233)		when and how a request is fulfilled
Interpreter (243)		grammar and interpretation of a language
Iterator (257)		how an aggregate's elements are accessed, traversed
Mediator (273)		how and which objects interact with each other
Memento (283)		what private information is stored outside an object, and when
Observer (293) ●		number of objects that depend on another object; how the dependent objects stay up to date
State (305)		states of an object
Strategy (315) ●		an algorithm
Template Method (325)		steps of an algorithm
Visitor (331)		operations that can be applied to object(s) without changing their class(es)

[Gamma, u.a.: Design Patterns, S. 30]

behandelt

Adaptierbare Strukturen

Spezielles Ziel in dieser Vorlesung:

Software-Strukturen für zukünftige Änderungen vorbereiten; „Designing for Change“

Einige Gründe für **schlechte Adaptierbarkeit** und Entwurfsmuster, die sie beheben (Auswahl aus [Gamma u.a.]):

1. **Objekte einer fest benannten Klasse erzeugen.**
Abstract Factory, Factory Method
2. **Abhängigkeit von spezieller Software-Plattform**
Abstract Factory, Bridge
3. **Abhängigkeit von speziellen Implementierungen**
Abstract Factory, Bridge
4. **Abhängigkeit von speziellen Algorithmen**
Strategy
5. **Zu enge Kopplung**
Abstract Factory, Bridge, Observer
6. **Funktionalität erweitern durch Vererbung**
Bridge, Observer, Strategy

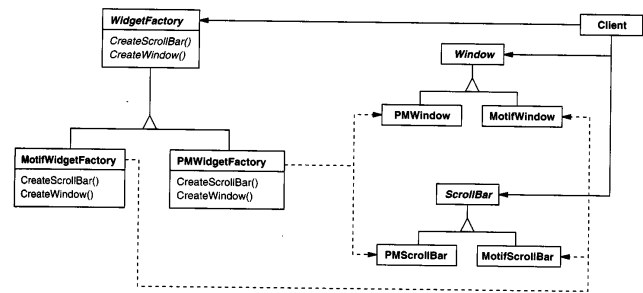
Abstract Factory (Beispiel)

Ziel: Objekte zu einer Gruppe zusammengehöriger Klassen erzeugen, ohne die Klassen konkret zu benennen

Beispiel:

Verschiedene GUI-Werkzeuge bieten unterschiedliche Ausprägungen von Sätzen von GUI-Komponenten an

Die Anwendung soll nicht durch **Klassennamen** auf eine bestimmte Ausprägung festgelegt werden



Eigenschaften des Beispiels:

- verschiedene Sätze von GUI-Klassen verfügbar machen
- Klassennamen nicht „fest verdrahten“, da schwer änderbar
- 2 Schnittstellen bereitstellen:
 - abstrakte GUI-Factory mit Ausprägungen für verschiedene GUI-Werkzeuge zur Generierung aller Produkte
 - Schnittstelle für jedes Produkt mit Ausprägungen für die GUI-Werkzeuge

Abstract Factory (Muster)

Ziel: Objekte zu einer Gruppe zusammengehöriger Klassen erzeugen, ohne die Klassen konkret zu benennen

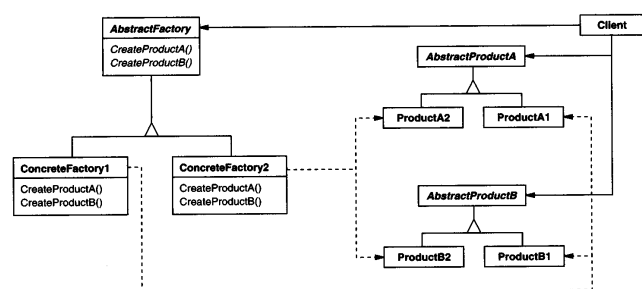
Muster:

AbstractFactory:
Schnittstelle zur Objekterzeugung

ConcreteFactory:
eine Ausprägung der Objekterzeugung

AbstractProduct:
Schnittstelle jeweils einer Produktklasse

ConcreteProduct:
Ausprägung jeweils einer Produktklasse



E. Gamma, R. Helm, R. Johnson, J. Vlissides: Design Patterns, Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995

Einsatzziele:

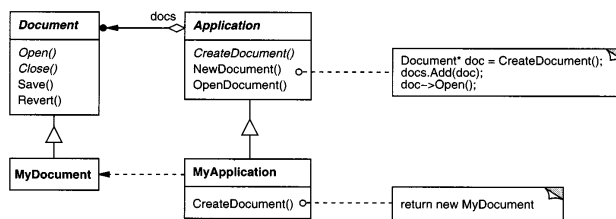
- Ein **System** soll mit einer von mehreren Produktgruppen **konfiguriert werden**.
- **Unabhängigkeit** von der Wahl der Produktgruppe.
- **Nur die Schnittstellen** nicht die Implementierungen der Produktgruppen **freigeben**.

Factory Method (Aufgabe)

Ziel: Objekterzeugung zu einer Klasse von der Anwendung entkoppeln;
 Anwendung realisiert ein **komplexes Konzept (Spezialisierung)**;
 Objekterzeugung ist darin eine **Erweiterungsstelle**

Beispiel:

Ein Framework zur Präsentation von Dokumenten.
 Die spezielle Ausprägung der Dokumente wird erst mit der speziellen Anwendung festgelegt.



E. Gamma, R. Helm, R. Johnson, J. Vlissides: Design Patterns, Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995

Eigenschaften des Beispiels:

- Anwendung (**Application**) benutzt ein abstraktes Produkt (**Document**), spezialisierte Anwendung (**MyApplication**) erzeugt das konkrete Produkt (**MyDocument**)
- **Application**: allgemeine Dokumentverwaltung, Dokument erzeugen, öffnen, etc. bestimmt **wann Objekte erzeugt** werden - **nicht welcher Klasse** sie angehören
- **konkrete Implementierung** der abstrakten Factory Method in **MyApplication** bestimmt die Klasse, z. B. Graphik-Objekte im Zeichenprogramm

Factory Method (Muster)

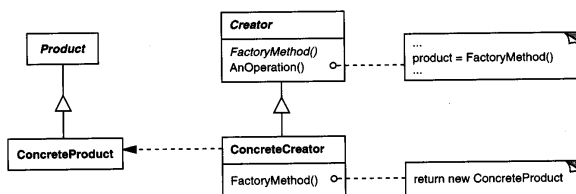
Muster:

Product:
 Schnittstelle der zu erzeugenden Objekte

ConcreteProduct:
 spezielle Ausprägung der Produkte

Creator:
 Kontext, in dem die Objekte erzeugt werden;
 hat Schnittstelle oder Default-Implementierung der FactoryMethod

ConcreteCreator:
 implementiert oder überschreibt die FactoryMethod,
 erzeugt ein ConcreteProduct-Objekt



E. Gamma, R. Helm, R. Johnson, J. Vlissides: Design Patterns, Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995

Einsatzziele:

- **Entscheidung** über Objekterzeugung (**welche Klasse?** ConcreteProduct) **verschieben** und in einer Klasse (ConcreteCreator) **kapseln**.
- Art von Delegationsobjekten (ConcreteProduct) in einer Klasse (ConcreteCreator) kapseln.

Bridge (Aufgabe)

Ziel: Eine **Spezifikation** wird von ihren **Implementierungen entkoppelt**.

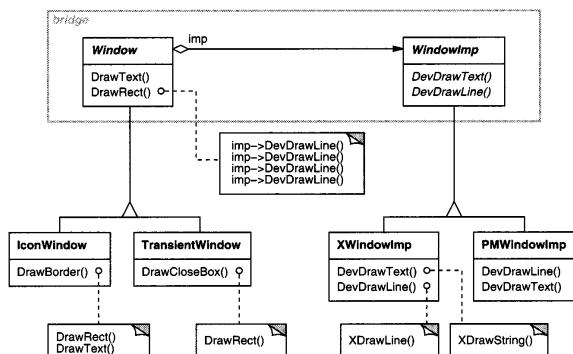
Verfeinerte Spezifikationen und **weitere Implementierungen** unabhängig zufügen.

2 Aspekte unabhängig variieren benötigt 2 Hierarchien, durch Delegation verbunden

Beispiel:

In einem GUI-Paket zu einer Window-Schnittstelle 2 Aspekte:

- a. Implementierungen für verschiedene Plattformen (XWindowImp, PMWindowImp)
- b. spezialisierte Schnittstellen (IconWindow, TransientWindow)



E. Gamma, R. Helm, R. Johnson, J. Vlissides: Design Patterns, Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995

VP Spezialisierung
(von Schnittstellen)

VP Spezifikation
(von Implementierungen)

Erweiterungen der Schnittstelle werden über **Delegation implementiert (Bridge)** (z. B. DrawRect)

Bridge (Muster)

Muster:

Abstraction:

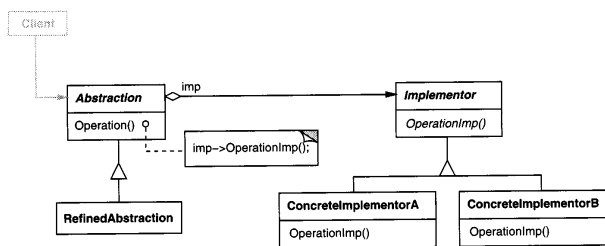
definiert Schnittstelle, hat Referenz auf ein Implementor-Objekt, implementiert Methoden damit

RefinedAbstraction: spezialisierte Schnittste

Implementor:

Schnittstelle für Implementierung, ist i. a. elementarer als die der **Abstraction**

ConcreteImplementor: Implementierungen



E. Gamma, R. Helm, R. Johnson, J. Vlissides: Design Patterns, Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995

Einsatzkriterien:

- **Feste Bindung** zwischen Abstraktion und Implementierung **vermeiden**
- benutzte **Implementierung zur Laufzeit wechseln**
- **2 Dimensionen der Variation** nicht in 1 Hierarchie!

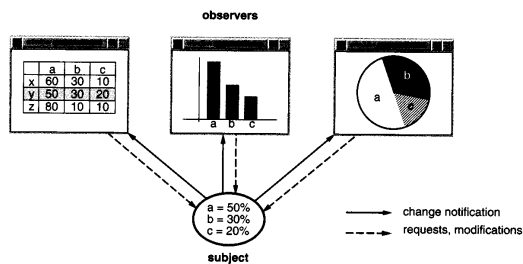
Observer (Aufgabe)

**Ziel: Observer-Objekte werden über Zustandsänderungen eines Subjekts informiert.
Dynamisch veränderliche Zuordnung von Observern zum Subjekt.**

Beispiel:

In GUI-Paketen: Trennung verschiedener Varianten der Präsentation von der Anwendung, die die präsentierten Daten liefert.

MVC: Model / View / Controller
[Krasner, Pope, 1988, Smalltalk]



E. Gamma, R. Helm, R. Johnson, J. Vlissides: Design Patterns, Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995

Eigenschaften des Beispiels:

- **Entkoppeln der Anwendung** (Model, Concrete Subject) von den **Präsentationen** (View, Observer)
- **Software-Aufgaben trennen**
- Präsentations-Software **separat variieren** (statisch)
- Präsentatoren **zur Laufzeit zufügen** und entfernen (dynamisch)
- Auch **andere** als GUI-Anwendungen:
z. B. eine Uhr und Prozesse, die die Zeit beobachten

© 2014 bei Prof. Dr. Uwe Kastens

Observer (Muster)

Muster:

Subject:

Abstraktion; verwaltet Referenzen auf alle Observer
Operationen zu Benachrichtigen

Observer:

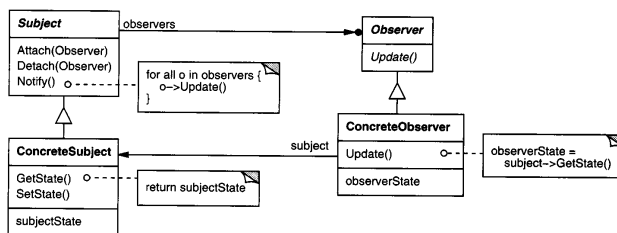
Update-Schnittstelle

ConcreteSubject:

Anwendung mit dem beobachteten Zustand

ConcreteObserver:

hat eine Referenz auf sein **concreteSubject**-Objekt,
hat einen Zustand, der mit dem **subject** konsistent gehalten wird,
implementiert die **update**-Schnittstelle



E. Gamma, R. Helm, R. Johnson, J. Vlissides: Design Patterns, Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995

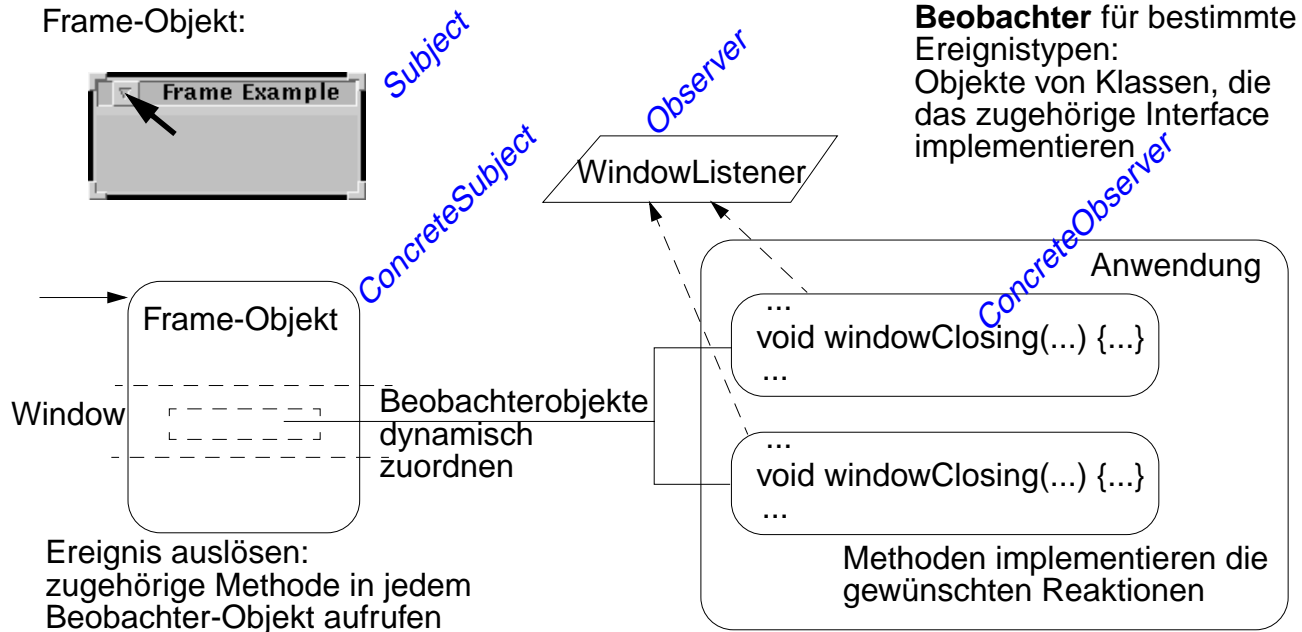
Einsatzkriterien:

- **abhängige Aspekte** entkoppeln (Model -> View)
- Verbindung zu Beobachtern **dynamisch änderbar**
- **keine Annahmen** beim Benachrichtigen

© 2005 bei Prof. Dr. Uwe Kastens

Entwurfsmuster Observer im AWT-Paket von Java

An AWT-Komponenten werden Ereignisse ausgelöst, z. B. ein WindowEvent an einem Frame-Objekt:



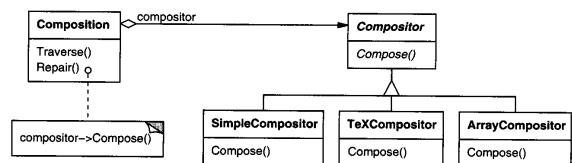
Entwurfsmuster „Observer“: Unabhängigkeit zwischen den Beobachtern und dem Gegenstand wegen Interface und dynamischem Zufügen von Beobachtern.

Strategy (Aufgabe)

Ziel: Für eine Gruppe unterschiedlicher Algorithmen für den gleichen Zweck Entwicklung entkoppeln und dynamisch austauschbar machen

Beispiel:

Algorithmen zum Zeilenumbruch:
Composition zeigt veränderbaren Text an. Den Zeilenumbruch delegiert sie an ein **Compositor**-Objekt.
 Eines mit den gewünschten Fähigkeiten wird installiert und ggf. ausgetauscht.



E. Gamma, R. Helm, R. Johnson, J. Vlissides: Design Patterns, Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995

weitere Beispiele:

- **LayoutManager** im AWT-Paket von Java
- Algorithmen zur Speicherzuteilung nach unterschiedlichen Verfahren
- Algorithmen zur Registerzuteilung nach unterschiedlichen Verfahren

Strategy (Muster)

Muster:

Strategy:

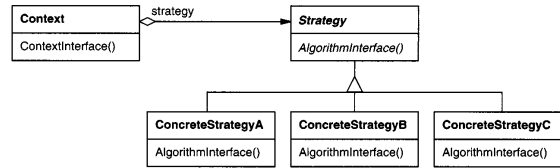
gemeinsame **Algorithmen-Schnittstelle (Spezifikation)**

ConcreteStrategy:

Varianten von **Algorithmen-Implementierungen**

Context:

wird mit einem **ConcreteStrategy**-Objekt konfiguriert und hat Referenz darauf (**Delegation**, wie bei Bridge); kann eine Schnittstelle anbieten, über die Algorithmen ihre Daten beziehen



E. Gamma, R. Helm, R. Johnson, J. Vlissides: Design Patterns, Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995

VP Spezifikation

Einsatzkriterien:

- Algorithmen separieren macht auch den Kontext einfacher, **Aufgabenzerlegung, Wiederverwendung**
- **nachträglich separieren ist aufwändig**
- Daten **kapseln**

4. Programmbausteine, Bibliotheken und Programmgerüste

Konzept **Programmbaustein:**

Programme aus **vorgefertigten Teilen** zusammensetzen - statt ganz neu schreiben „Baukasten“, industrielle Produktion

Gute Bausteine, Programmgerüste: geplante Wiederverwendung, kostbar, aufwändig

Wichtige Fragen:

- Wie stellt man gute Bausteine her?
- Wie allgemein bzw. anpassbar sind die Bausteine?
- Sind die Bausteine direkt einsetzbar oder realisieren sie nur Teilaspekte und müssen vor der Anwendung komplettiert werden?
- Sind die Bausteine unabhängig voneinander oder wechselseitig aufeinander angewiesen?
- Wie werden Bausteine zusammengesetzt?

nach [Arnd Poetzsch-Heffter: Konzepte Objektorientierter Programmierung, Springer, 2000]

Beziehungen zwischen Programmbausteinen

Unabhängige Bausteine

Schnittstelle verwendet **nur vom Baustein deklarierte Typen** oder Standardtypen (Grundtypen und die aus java.lang).

Beispiele:
Datentypen

- Listen
- Mengen
- Hashtabellen

Typisch:
Elementanpassung durch generische Typparameter (z. B. in C++: Leda, STL)
lokal verstehen, einfach
isolierte Aufgaben

Eigenständige Bausteine

Bausteinfamilie ist hierarchisch strukturiert. Schnittstelle verwendet zusätzlich Supertypen (oft abstrakt)

Beispiele:
- die Typen zur Ausnahmebehandlung in Java
- die Stromklassen aus java.io

Typisch: Trotz der hierarchischen Beziehung sind die Bausteine eigenständig verwendbar.
keine Querbeziehungen, nur Abstraktion

Eng kooperierende Bausteine

Bausteinfamilie dient **komplexer**, spezieller softwaretechnischer **Aufgabenstellung**.

Enge Kooperation führt zu **komplexen Abhängigkeiten**, z. B. verschränkt rekursiver Beziehung von Typen und Methoden.

Beispiele:
Programmgerüste, wie

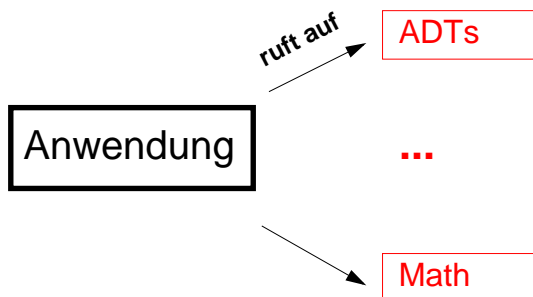
- Java AWT
- San Francisco (IBM): Abwicklung von Geschäftsprozessen

Typisch: wg. Abhängigkeiten anspruchsvoller

Unterschied zwischen Programmgerüst und Bausteinbibliothek

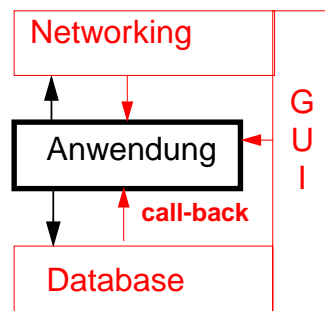
Bausteinbibliotheken

- mit unabhängigen oder eigenständigen Bausteinen
- anwendungsunabhängig
- Ablauf unter Kontrolle des Anwendungsprogramms

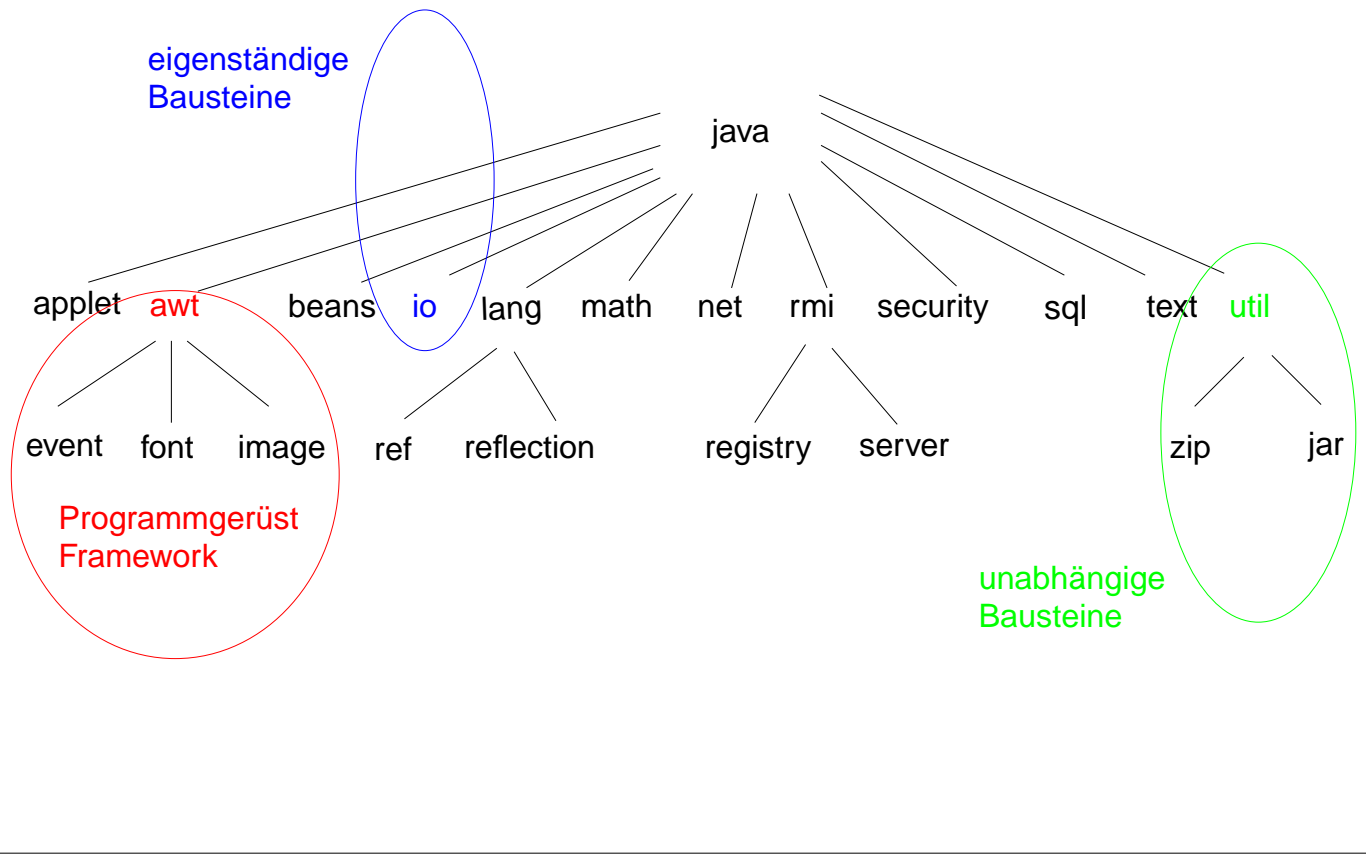


Programmgerüst (Framework)

- „fast-fertige“ Anwendung
- anwendungsspezifisch
- Umkehr der Ablaufsteuerung, call-back-Routinen, „don't call us, we call you“



Die Java-Bibliothek



Anwendung von Bausteinen: Ausnahmebehandlung in Java

Ausnahmebehandlung als einfaches Beispiel für eine **erweiterbare Bausteinhierarchie**

Zeigt das enge Zusammenspiel zwischen **Sprache und Bibliothek**

Trend bei modernen OO-Sprachen: **Anzahl der Sprachkonstrukte klein** halten, mit Mitteln der Sprache formulierbare Konzepte in die **Bibliothek**;
 extrem: In Smalltalk sind `ifTrue`, `ifFalse` Methoden der Bibliotheksklasse `Boolean`

Ausnahmebehandlung in Java:

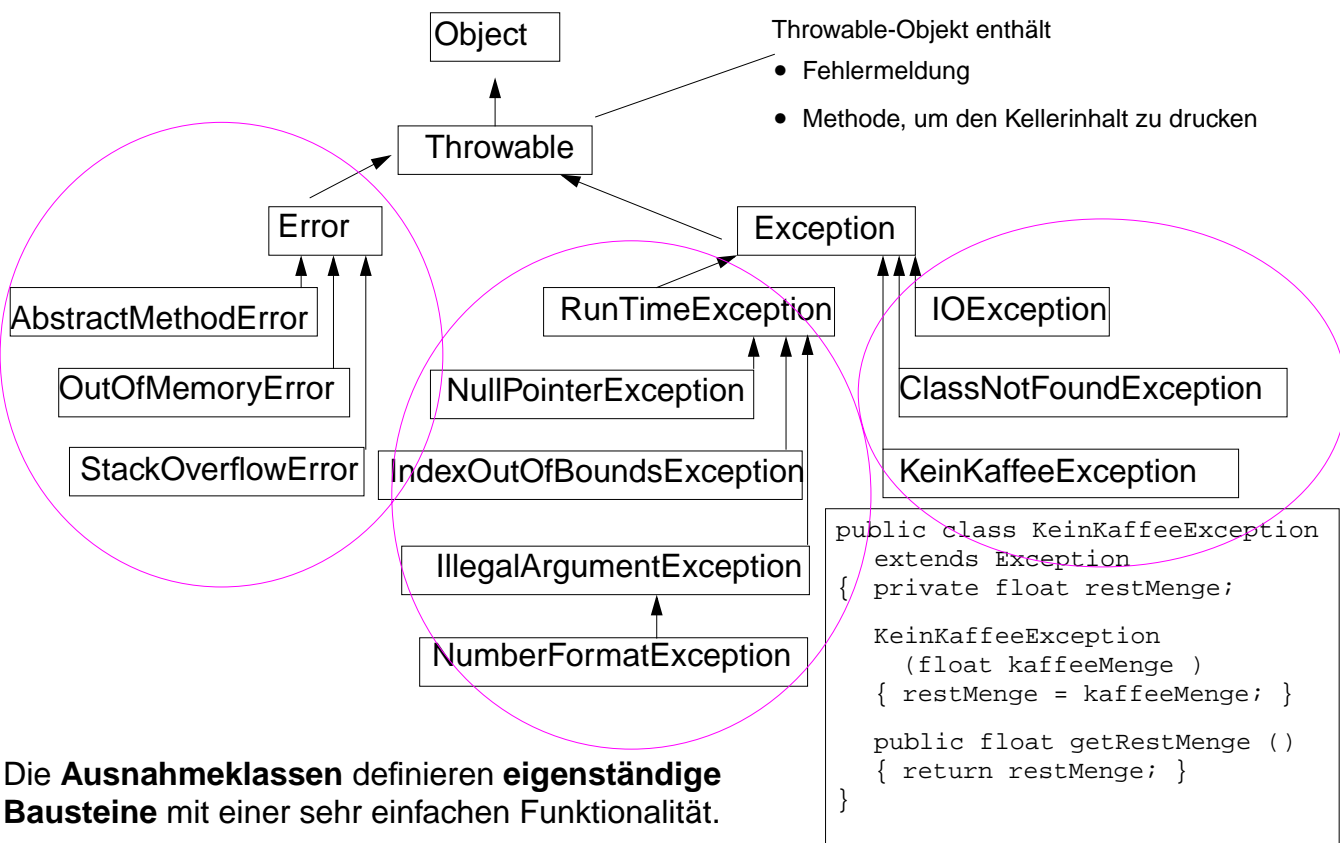
Bei „abrupter Terminierung“ der Auswertung von Ausdrücken oder Ausführung von Anweisungen wird

- ein Ausnahmeobjekt `e` erzeugt und
- die in der **dynamischen Umgebung** (Laufzeitkeller) nächste, zur Klasse von `e` passende Ausnahmebehandlung mit `e` als Parameter aufgerufen.

3 Arten von Ausnahmen (siehe Klassifikation, OOP-4.7):

- Benutzer hat kaum Einfluss darauf, z. B. erschöpfte Maschinenressourcen (**Error**),
- vermeidbare Ausnahmen wegen Programmierfehlern, z. B. der Versuch, die null-Referenz zu dereferenzieren (**RuntimeException**),
- nicht vermeidbare Ausnahmen, die im Ablauf des Programms vorgesehen sein sollten, z. B. Zugriff auf nicht vorhandene Eingabe-Datei (weitere Unterklassen von **Exception**)

Ausnahmhierarchie



Die **Ausnahmeklassen** definieren **eigenständige Bausteine** mit einer sehr einfachen Funktionalität.

Kriterium für Entwurfsqualität: Kohärenz

Kohärenz (Bindung): Stärke des **funktionalen Zusammenhangs** zwischen den Bestandteilen eines Bausteins

Ziel: So **stark** wie möglich

Kohärenz einer Klasse:

- Klasse realisiert **nur ein semantisch bedeutungsvolles Konzept**
- ihre **Methoden kooperieren** für diese Aufgabe
- Menge der Methoden ist **vollständig**
- **keine Methode ist überflüssig** für das Konzept

starke Kohärenz weil Komponenten

funktional zusammenhängen

schwache oder keine Kohärenz weil Komponenten

zur gleichen Zeit gebraucht werden (z.B. Initialisierung)

in der Ablaufstruktur zusammenhängen

zufällig zusammengekommen sind

Kriterien Entwurfsqualität: Kopplung

Kopplung: Stärke der Beziehung, Interaktion zwischen Bausteinen

- Kopplung **zwischen Methoden**, zwischen **Klassen**, zwischen **Paketen**
- Kopplung entsteht durch **Benutzen von Daten** oder durch **Aufrufe** (siehe OOP-4.11)

Ziel: Kopplung so schwach wie möglich

- schwache Kopplung entspricht schmaler Schnittstelle

dadurch bessere

- Zerlegbarkeit
- Komponierbarkeit
- Verständlichkeit
- Sicherheit

Maßnahmen zur Entkopplung

Unnötige Kopplung von Klassen **vermindert** grundlos die **Wiederverwendbarkeit**

Anwendung von Vererbungsparadigmen:

- **Rollen und Eigenschaften:**
Material entkoppelt von Werkzeugen, Verhalten entkoppelt vom Subjekt
- **Spezifikation:**
Implementierung entkoppelt von Anwendungen

Entwurfsmuster:

- **Bridge:** verfeinerte Spezifikation entkoppelt von Implementierungen
- **Strategy:** Implementierung entkoppelt von Anwendungen
- **Abstract Factory:** konkrete Erzeuger der Objekte entkoppelt von Anwendungen
- **Observer:** veränderliches Subjekt entkoppelt von Präsentatoren des Zustandes

Beispiel zur Entkopplung: Sortieren von Personen-Listen

```
class Person
{
    Name n;
    long PersNr;
    ...
}
```

```
SortedPersonList oop =
    new SortedPersonList();
Person neu;
...
oop.add (neu);
```

Was ist hieran schlecht?

```
class SortedPersonList
{
    Object[] elems;
    ...
    public void add (Person x)
    {
        ...
        Name a = x.getName();
        Name b =
            (Name)elems [k].getName();
        if (a.lessThan(b))
            ...
        ...
    }
}
```

Diese Klasse ist für nichts
anderes zu gebrauchen!

nicht wiederverwendbar!

Entkoppeln durch Interfaces

```
interface Comparable
{
    public boolean lessThan (Object compareMe);
}

class Person implements Comparable
{
    Name n;
    long PersNr;

    public boolean lessThan(Object compareMe)
    {
        return n.lessThan(((Person)compareMe).n);
    }
}

class SortedList
{
    Object[] elems;
    ...
    public void add (Comparable x)
    {
        if (x.lessThan(elems[k]))
            ...
        ...
    }
}
```

**Schnittstelle
entkoppelt
Bausteine**

**wiederverwendbare
Klasse!**

unabhängig von den
sortierten Elementen

Entkoppeln durch Funktoren

Die **Vergleichsfunktion** wird erst bei **Objekterzeugung** bestimmt.
Variable **Funktionen** werden in **Objekte** verpackt [Functor Pattern von Coplien]

```
import java.util.Comparator; // seit Java 1.2

class Person {Name n; long PersNr; ...}

class PersonComparator implements Comparator
{ public int compare (Object links, Object rechts)
  { return
    ((Person)links).n.lessThan (((Person)rechts).n)?
    -1:((Person)links).n.equal (((Person)rechts).n)?
      0 : 1;
  } }

class SortedList
{ private Comparator cmp; ...
  public SortedList (Comparator c) { cmp = c; }
  ...
  public void add (Object o)
  { ... leq = cmp.compare (o, elements[k]); ...
  } }
```

Funktor:
Verpackung von **Vergleichsfunktion** für Person-Objekte

wiederverwendbar

wird mit Vergleichsfunktion initialisiert

Vorsicht: Passt nicht zum Grundsatz „Klasse beschreibt Zustand und Verhalten“.

Objektorientierte Programmgerüste

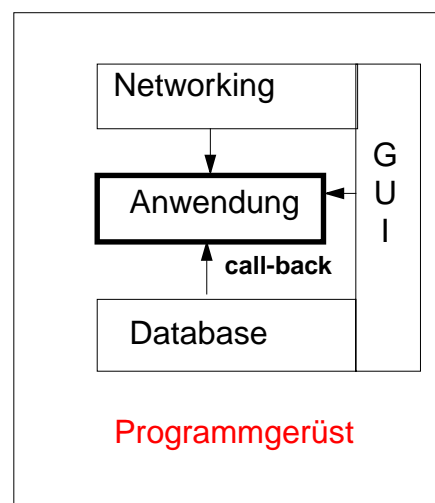
Programmgerüst (Framework):

Integrierte Sammlung von **Bausteinen**, die **zusammenarbeiten**, um eine **wiederverwendbare, erweiterbare Architektur** für eine **Familie verwandter Aufgaben** zur Verfügung zu stellen.

[siehe Folie 4.3]

Charakteristika:

- „fast-fertige“ Anwendung mit **geplanten Erweiterungsstellen**
- **anwendungsspezifisch**
- Umkehr der Ablaufsteuerung, **call-back-Routinen**, „don't call us, we call you“
- **komplexe Beziehungen** zwischen den Bausteinen werden **ohne Zutun des Benutzers** wiederverwendet



Entwicklung mit Programmgerüsten

Die Anwendung

- **spezialisiert Klassen** des Programmgerüstes,
- füllt **Erweiterungsstellen** des Programmgerüstes aus:
- implementiert **call-back-Routinen**, die vom Programmgerüst aufgerufen werden
- implementiert **Reaktionen auf Ereignisse**, die über das Programmgerüst ausgelöst werden
- setzt **Algorithmen, Klassen**, die das Programmgerüst anbietet, zu **Schnittstellen** ein, die das Programmgerüst vorgibt.

Programmgerüst am Beispiel des Java AWT

Was macht **Java AWT** zum typischen Beispiel für ein Programmgerüst?

1. Der **Anwendungsbereich** Benutzungsoberflächen ist **komplex und vielfältig variierbar**.
2. Java AWT **deckt den Anwendungsbereich vollständig ab**;
man kann damit vollständige Benutzungsoberflächen entwickeln
- nicht nur einzelne Teilaufgaben erledigen
3. Einzelne **Komponenten** des Java AWT sind **spezialisierbar**:
bieten umfangreiche wiederverwendbare Funktionalität;
Benutzer ergänzt sie durch vergleichsweise kleine spezifische Ausprägung;
siehe Frame als Beispiel für das Konzept **Spezialisierung** (OOP-2.9)
4. **komplexes Zusammenwirken** der Komponenten liegt fast vollständig in der **wiederverwendbaren Funktionalität**.
Voraussetzung: **wohl-definierte Software-Architektur** des Java AWT;
siehe OOP-4.23

Komplexes Zusammenwirken im Programmgerüst

Beispiele für komplexes Zusammenwirken der AWT-Komponenten:

- **Ereignissteuerung:**
Durch Maus, Tastatur ausgelöste Ereignisse verursachen Aufrufe von Call-back-Routinen. siehe: Konzept eingebettete Agenten OOP-2.27, Entwurfsmuster Observer OOP-3.7a
- **Anordnung von Komponenten (LayoutManager):**
Bedienelemente werden auf bestimmten Behälterflächen nach wählbaren Strategien angeordnet;
siehe: Entwurfsmuster Strategy OOP-3.8

Die **Mechanismen für das Zusammenwirken** liegen (fast) **vollständig** in der **wiederverwendbaren Funktionalität** der Komponenten.

Benutzer brauchen sie **nicht (tiefgehend) zu kennen oder zu verstehen**.

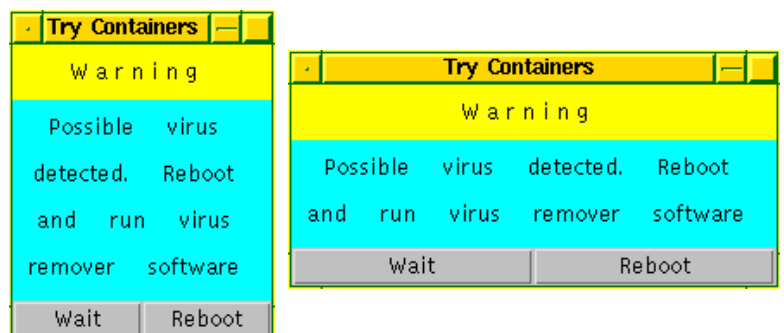
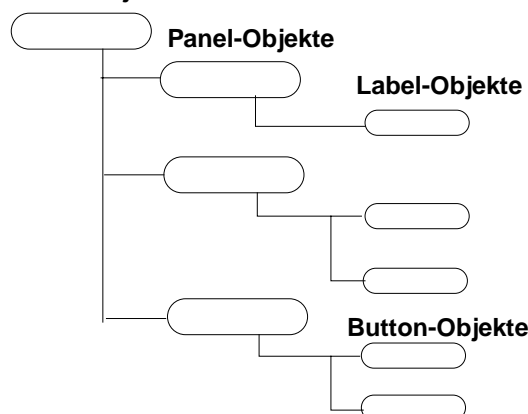
Software-Architektur des Programmgerüsts ist Voraussetzung dafür:

- **Architekturgerüst** ist festgelegt, so dass es für alle Anwendungsvarianten passt;
- Komponenten haben **unterschiedliche Aufgaben**, z. B. Container, Event, Listener, LayoutManager (OOP-4.24)
- **Regeln zum Zusammensetzen** einer Anwendungsvariante

Architekturkonzept: hierarchisch strukturierte Fensterinhalte

- Zusammengehörige Komponenten in einem Objekt einer **Container**-Unterklasse unterbringen (**Panel**, **Window**, **Frame** oder selbstdefinierte Unterklasse).
- Eigenschaften der Objekte im **Container** können dann gemeinsam bestimmt werden, z. B. Farbe, **LayoutManager**, usw. zuordnen.
- Mit **Container**-Objekten werden beliebig tiefe Baumstrukturen von AWT-Komponenten erzeugt. In der visuellen Darstellung sind sie ineinander geschachtelt.

Frame-Objekt



5. Entwurfsfehler, Übersicht

Hier: Fehler beim **Entwurf objektorientierter Strukturen**, bei **Anwendung objektorientierter Konzepte**.

nicht: Fehler in **anderen Entwicklungsphasen:** Anforderungsanalyse, Systemmodellierung, Grobstruktur, Projektorganisation, Implementierung, Validierung

Versuch und Anfang einer **Sammlung typischer Fehler**

5.1 **Missbrauch von Vererbung** (Zusammenfassung aus Abschnitt 3)

5.2 **Anti Patterns** (W.J. Brown et.al)

5.3 **Üble Gerüche im Code** - Signal zum Refaktorisieren (M. Fowler)

5.4 **Unangenehme OOP-Überraschungen** (nach E. Plödereder)

nicht erwähnt: die Negation positiver Regeln (z. B. „starke Kopplung statt schwacher“)

5.1 Missbrauch von Vererbung

siehe Abschnitt 2; Darstellung hier im Stil der AntiPattern

5.1.1 Verkannter Schauspieler

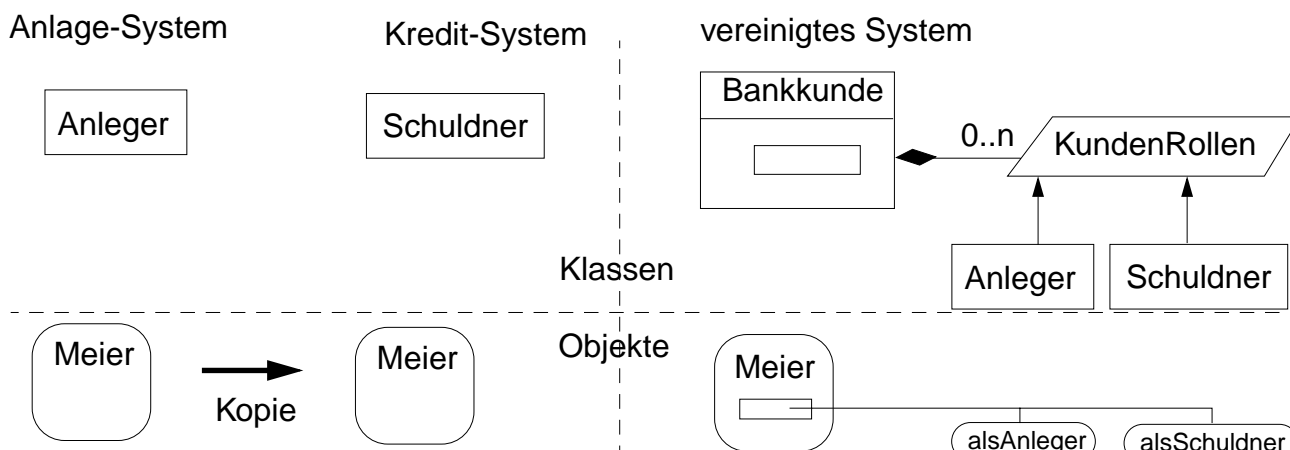
Beschreibung: Unterschied **Objekt und sein Verhalten** nicht modelliert

Ursache: evtl. wurden getrennt entworfene **Systeme zusammengelegt**

Symptom: ein **Objekt** soll vorübergehend oder auf Dauer seine **Klasse wechseln**.

Abhilfe: Objekt bleibt in allgemeiner Klasse und **spielt verschiedene Rollen**.

Beispiel:



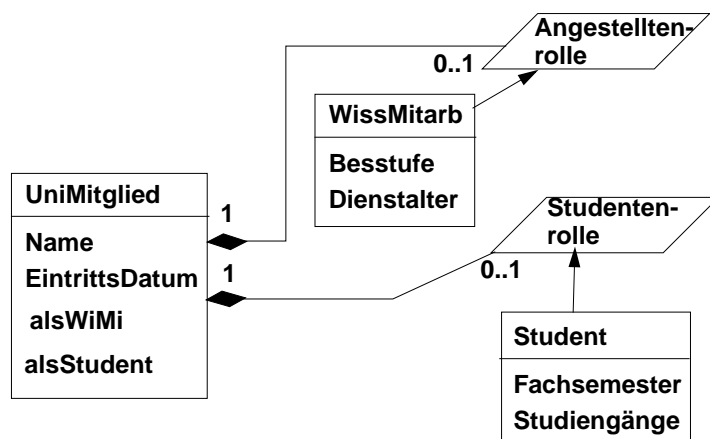
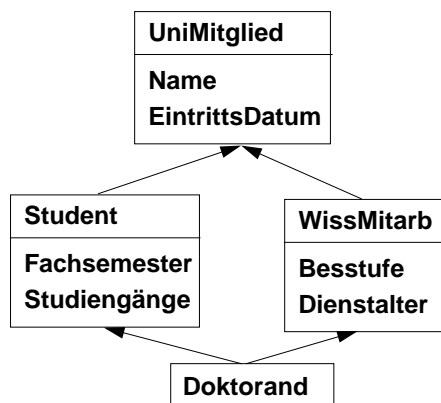
5.1.2 Januskopf

Beschreibung: Eine Klasse soll die Eigenschaften **mehrerer Oberklassen** vereinigen. Anwendung des Prinzips „**Abstraktion**“ (2.2) hat zu **überlappenden Unterklassen** geführt (2.16).

Ursache: evtl. **Rollenkonzept unbekannt**

Abhilfe: Abstraktionsebene aus der Klassenhierarchie entfernen
Komposition von Rollen
Abstraktion der Rollen

Beispiel:



Doktorand ist nun ein UniMitglied, das **zugleich beide Rollen** spielt: Student und WissMitarb.

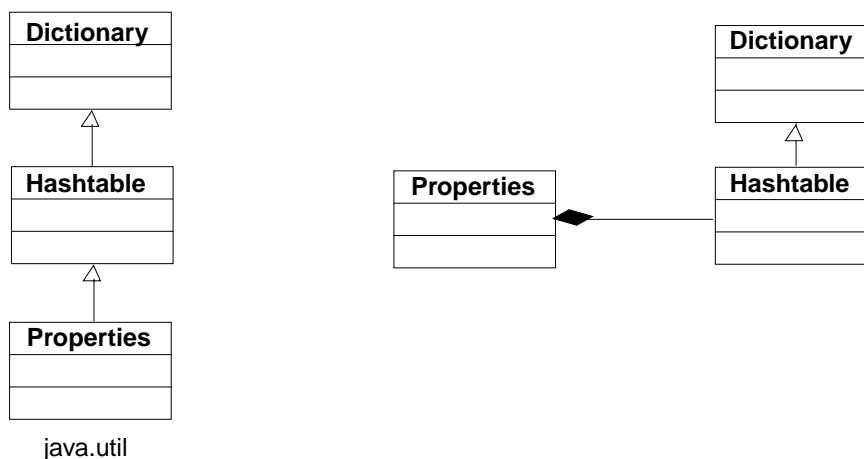
5.1.3 Ungeplanter Anbau

Beschreibung: Unterklasse **nutzt die Funktionalität der Oberklasse**, um ein **anderes Konzept** zu realisieren; schlechte **inkrementelle Weiterentwicklung** (Abschnitt 2.6)

Ursache: evtl. **Spezialisierung missverstanden**

Abhilfe: Funktionalität nicht durch Vererbung sondern durch **Delegation** nutzen.

Beispiel: siehe OOP-2.21



5.2 AntiPatterns

W.J. Brown, R.C. Malveau, H.W. McCormick III, T.J. Mowbray:
Anti Patterns, Refactoring Software, Architectures, and Projects in Crisis,
 John Wiley, 1998.

AntiPattern: beschreibt ein **Schema von Lösungen** das häufig angewandt wird,
 aber **mehr Probleme verursacht als es löst.**

Anti Patterns für **Software-Entwicklung, Architektur, Projektorganisation**

Darstellungsschema im Buch:

- Hintergrund
- Allgemeine Beschreibung
- Symptome und Konsequenzen
- Ausnahmen
- Verbesserte Lösung
- Varianten
- Beispiel
- Verwandte Lösungen

Kurzform hier:

- Beschreibung
- Symptome
- Konsequenzen
- Verbesserte Lösung
- Beispiel

5.2.1 The Blob

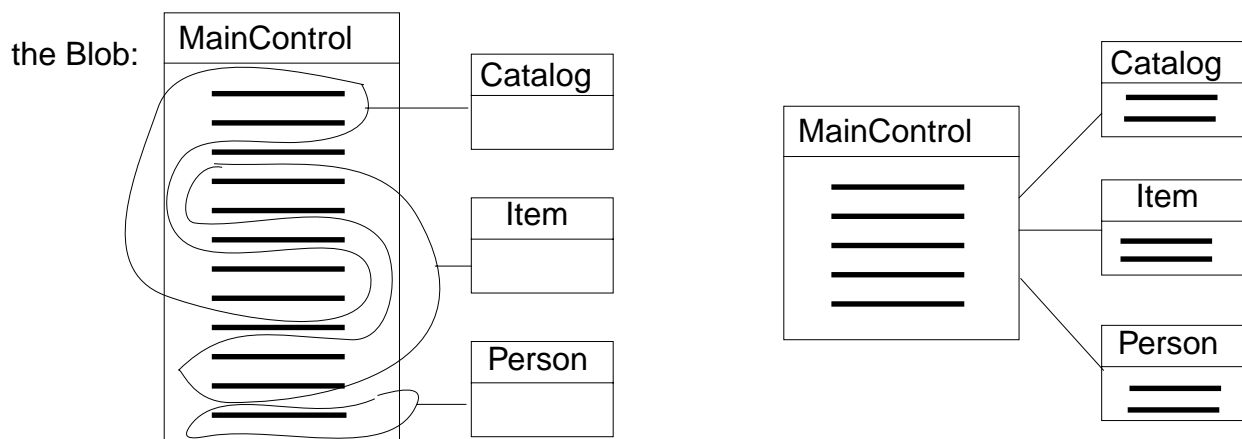
Beschreibung: Eine einzige **Klasse dominiert** den Ablauf.
 Darum herum sind **Datenklassen** angeordnet.
 Ablauf-orientierter Entwurf: **Daten von Operationen getrennt.**

Symptome: Eine **Klasse mit vielen Methoden und Attributen**,
geringe Kohärenz.

Ursache: evtl. OO-Kenntnisse fehlen

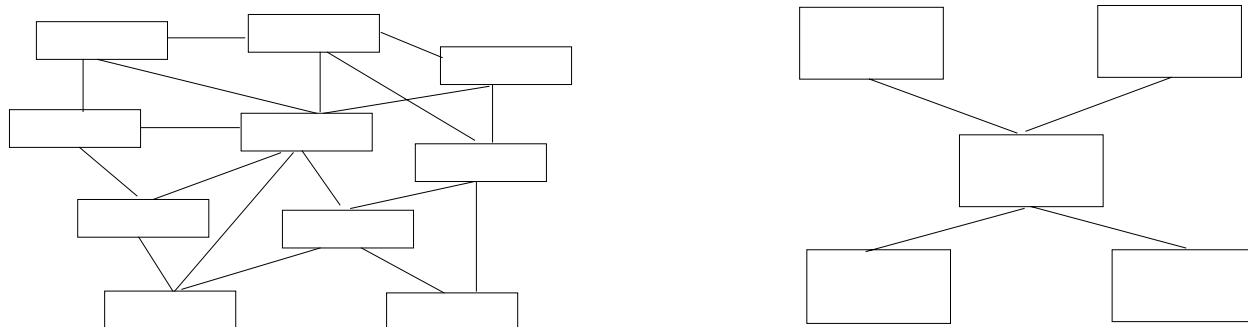
Konsequenzen: **Schlecht wartbar, änderbar; nicht wiederverwendbar**

Verbesserte Lösung: Zerlegen; Operationen und zugehörige Daten **zusammenfassen.**



5.2.2 Poltergeister

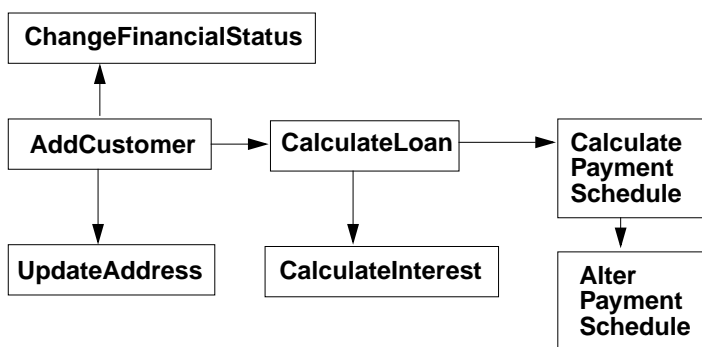
- Beschreibung:** **Übermäßig viele Klassen** und Beziehungen dazwischen, **unnötige Klassen** mit **winzigen Aufgaben**, kurzlebige Objekte ohne Zustand; **schwache Abstraktionen**, unnötig komplexe Struktur.
- Symptome:** s.o.
- Ursache:** evtl. übertriebener Einsatz von Klassen; Anfängerfehler
- Konsequenzen:** Unnötige Klassen und Komplexität **stören das Verständnis**, **verschlechtern die Wartbarkeit**, Änderbarkeit.
- Verbesserte Lösung:** **Irrelevante Klassen eliminieren**; winzige Aufgaben zusammenfassen zu **kohärenten größeren Klassen**.



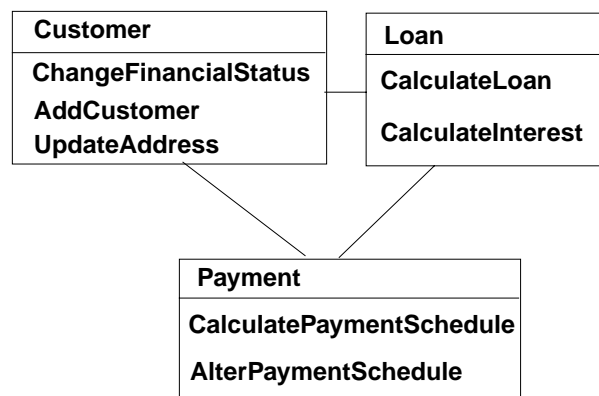
© 2005 bei Prof. Dr. Uwe Kastens

5.2.3 Funktionale Zerlegung

- Beschreibung:** Funktionen **schrittweise verfeinert**; **eine Klasse für jede Funktion**; **OO-Techniken nicht angewandt**.
- Symptome:** **Klassen haben Funktionsnamen** (z. B. CalculateInterest), **nur eine Methode** in jeder Klasse; **degenerierte Struktur**;
- Ursache:** evtl. OO-Kenntnisse fehlen, Programmierstil der 1970er Jahre
- Konsequenzen:** **schwer zu verstehen** und zu warten; **keine Wiederverwendung**.
- Verbesserte Lösung:** **Mit OO-Konzepten neu entwerfen**.



funktional



objektorientiert

© 2005 bei Prof. Dr. Uwe Kastens

5.2.3 Spaghetti Code

Spaghetti Code: geprägt als Begriff zur Charakterisierung schlechter FORTRAN-Programme mit **verschlungenen Abläufen**, nur durch **Sprünge und Marken** programmiert.

- Beschreibung:** **schwach strukturiertes System;**
Klassen mit **riesigen Methoden,**
viele Methoden ohne Parameter.
- Symptome:** Methoden sind sehr **Ablauf-geprägt,**
sehr lange Methodenrumpfe;
globale Variable statt Parameter.
- Ursache:** evtl. Programmier- und OO-Kenntnisse fehlen
- Konsequenzen:** **schlecht wartbar;**
kein Nutzen von OO, nicht wiederverwendbar.
- Verbesserte Lösung:** **Umstrukturieren;**
besser: **nicht entstehen lassen.**

Kurzbeschreibungen einiger Anti Patterns

5.2.5 Lava Flow (ausgeflossen, alt und hart geworden)

- Beschreibung:** **Unkontrollierte Auswüchse und Anbauten** des Systems;
veraltete, **schwierig zu entfernende Klassen;**
- Ursachen:** **verpasster Redesign,** mangelhafte Struktur
- Abhilfe:** **Redesign**

5.2.4 Ofenrohr-Struktur (zusammengefleckt und verrostet)

- Beschreibung:** **adhoc-Struktur;** alle Subsysteme sind **paarweise verbunden;**
Verbindungen müssen **immer wieder repariert** werden.
- Abhilfe:** **Redesign**

5.2.6 Swiss Army Knife (universell verwendbar)

- Beschreibung:** Klasse mit **riesiger Schnittstelle;** große Zahl von Methoden,
die **alle Möglichkeiten und Varianten** vorsehen sollen
- Abhilfe:** **Schnittstelle strukturieren,** reduzieren,
Schema (Profile) einführen zur Parametrisierung

5.3 Üble Gerüche im Code - Signal zum Refaktorisieren

Refaktorisieren:

Ziel: Code verbessern

- bessere Struktur,
- leichter verständlich,
- besser änderbar,
- besser korrigierbar

Methode:

Objektorientierten Code eines Software-Systems in kleinen Schritten systematisch verändern:

- Transformation aus Katalog anwenden (72 bei Fowler)
- beobachtbare Wirkung bleibt unverändert
- prüfen durch umfassende Tests (selbstprüfende Regressionstests)
- **üble Gerüche** (bad smells) als Hinweis auf schlechte Stellen im Code

Herkunft des Refaktorisierens (Refactoring):

Mitte 80er, Smalltalk-Community; Cunningham, Fowler, Beck, Opdyke

Martin Fowler: Refactoring, Addison-Wesley,

Kent Beck: eXtreme Programming, Addison-Wesley, 1999

Liste übler Gerüche (nach Fowler)

- | | |
|-------------------------------------|--|
| 1. Duplizierter Code | 12. Faule Klasse |
| 2. Lange Methode | 13. Spekulative Allgemeinheit |
| 3. Große Klasse | 14. Temporäre Felder |
| 4. Lange Parameterliste | 15. Nachrichtenketten |
| 5. Divergierende Änderungen | 16. Vermittler |
| 6. Schrotkugeln herausoperieren | 17. Unangebrachte Intimität |
| 7. Neid | 18. Alternative Klassen mit verschiedenen Schnittstellen |
| 8. Datenklumpen | 19. Unvollständige Bibliotheksklasse |
| 9. Neigung zu elementaren Typen | 20. Datenklassen |
| 10. Switch-Anweisungen | 21. Ausgeschlagenes Erbe |
| 11. Parallele Vererbungshierarchien | 22. Kommentare |

Beispiele für üble Gerüche im Code (1)

1. Duplizierter Code:

An mehreren Stellen tritt die gleiche (ähnliche) Codestruktur auf.

Refaktorisierungen zur Abhilfe:

Methode extrahieren, Klasse extrahieren

10. switch-Anweisung

In mehreren switch-Anweisungen wird über denselben Wertebereich verzweigt; Typschlüssel; Änderungen betreffen alle Auftreten; nicht objektorientiert

Refaktorisierungen zur Abhilfe:

Methode extrahieren, Methode verschieben, Typschlüssel durch Unterklassen ersetzen, Bedingten Ausdruck durch Polymorphismus ersetzen

17. Unangebrachte Intimität:

Methode befasst sich zu intensiv mit den Daten und Methoden einer anderen Klasse

Refaktorisierungen zur Abhilfe:

Methode verschieben, Feld verschieben, Klasse extrahieren, Delegation verbergen

20. Datenklassen:

Klasse hat nichts außer Feldern und get- und set-Methoden dafür; wird von anderen Klassen aus manipuliert

Refaktorisierungen zur Abhilfe:

Methode verschieben, Methode extrahieren, set-Methode entfernen, Methode verbergen

Beispiele für üble Gerüche im Code (2)

21. Ausgeschlagenes Erbe:

a. Unterklasse verwendet kaum Methoden und Felder aus der Oberklasse

Refaktorisierungen zur Abhilfe:

Neue Geschwisterklasse bilden, Methode/Feld nach unten schieben

b. Unterklasse verwendet Methoden und Felder der Oberklasse, hält aber deren Schnittstelle nicht ein (Ersetzbarkeit verletzt, inkrementelle Weiterentwicklung)

Refaktorisierungen zur Abhilfe:

Vererbung durch Delegation ersetzen

22. Kommentare:

Kommentare als Deodorant gegen übel riechenden Code; signalisieren häufig schwer verständlichen Code

Refaktorisierungen zur Abhilfe:

Methode extrahieren, Methode umbenennen, Zusicherung einführen

Gute Kommentare an gutem Code begründen z. B. Entwurfsentscheidungen - hilfreich für spätere Wartung

5.4 Unangenehme OOP-Überraschungen

Für die **Software-Wartung** wünscht man sich, dass folgendes nicht eintreten kann:

Zufügen einer Deklaration in einem Modul
 ändert die **Wirkung** von Funktionen eines **anderen Moduls** -
ohne Warnung durch den Übersetzer.

Man müsste sonst bei **Weiterentwicklungen und beim Testen**
immer das ganze System betrachten.

In nicht-OO-Sprachen weitgehend erfüllt.

In objektorientierte Sprachen:

dynamische Methodenbindung und Überschreiben
 erzeugen den **o.g. Effekt - beabsichtigt.**

Die folgenden Beispiele zu o.g.Effekt stammen aus einem Vortrag über OOP & Maintenance von E. Plödereder, Universität Stuttgart.

Versehentlich überschrieben (1)

```
class C1
{
    void doA () {... if (...) this.doB(); ...}
    void doB () {...}
}

class C2 extends C1
{
    // doA geerbt
    // doB geerbt, dann überschrieben
    void doB () {... this.doA(); ...}
}
```

Nach Einfügen von `doB` in `C2` wird in `doA` nicht mehr `C1.doB` sondern `C2.doB` aufgerufen.

Der Aufruf von `C2.doB` terminiert dann u. U. nicht.

Nach einem Beispiel aus einem Vortrag über OOP & Maintenance von E. Plödereder, Universität Stuttgart.

Versehentlich überschrieben (2)

```
class C1
{ public void doA () {...}
}

class C2 extends C1
{

}

class C3 extends C2
{

}
```

----- Ende der Bibliothek

```
class C4 extends C2
{ public void doC () {...}
}
```

Nach einem Beispiel aus einem Vortrag über OOP & Maintenance von E. Plödereder, Universität Stuttgart.

Versehentlich überschrieben (2)

```
class C1
{ public void doA () {...}
}
```

```
class C2 extends C1
{
    protected void doC () {...}
}
```

Ergänzung der Bibliothek

```
class C3 extends C2
{
    void foo (C2 v) { ... v.doC(); ...}
}
```

Ergänzung der Bibliothek
Aufruf führt in die

----- Ende der Bibliothek

schon vorher existierende

```
class C4 extends C2
{ public void doC () {...}
}
```

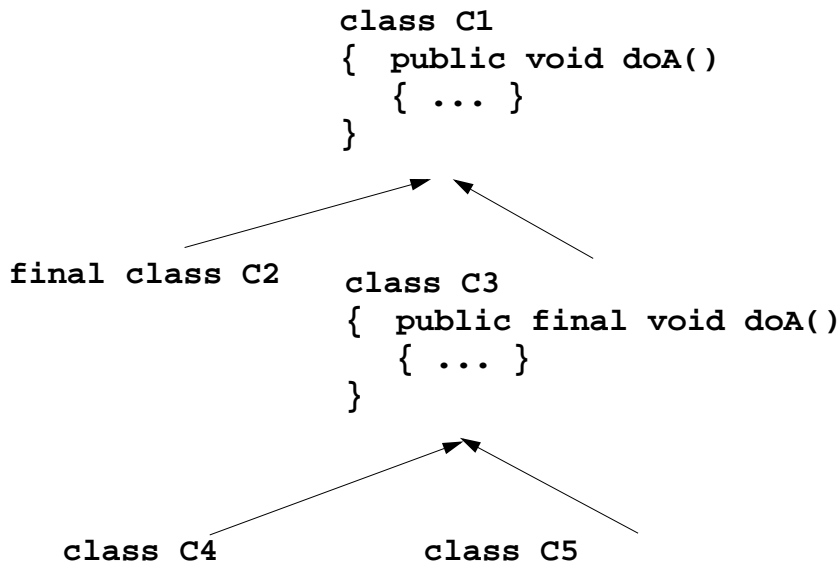
Benutzermethode
C4.doC

Programmierer sollten explizit machen können, ob Überschreiben gemeint ist.
Der Übersetzer dann prüfen und melden „accidental overriding“.

Nach einem Beispiel aus einem Vortrag über OOP & Maintenance von E. Plödereder, Universität Stuttgart.

final umgangen

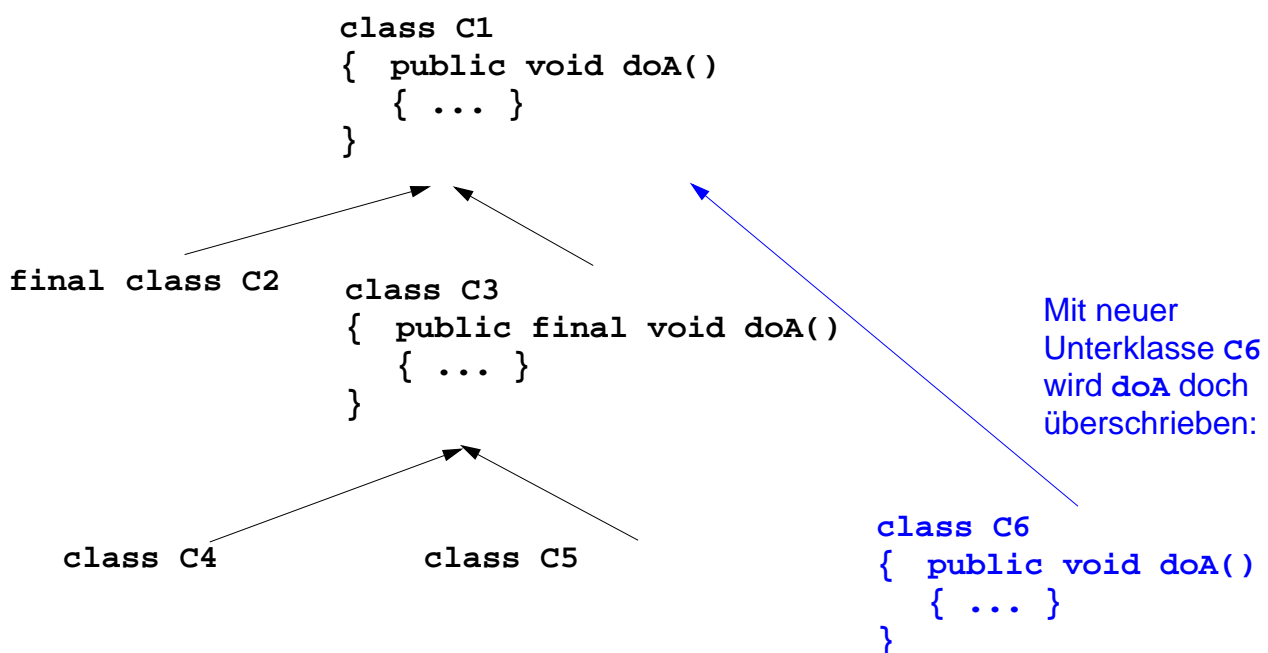
In der Klassenhierarchie C1, C2, C3 sollte weiteres Überschreiben von doA verhindert werden:



Nach einem Beispiel aus einem Vortrag über OOP & Maintenance von E. Plödereder, Universität Stuttgart.

final umgangen

In der Klassenhierarchie C1, C2, C3 sollte weiteres Überschreiben von doA verhindert werden:



Mit neuer
Unterklasse C6
wird doA doch
überschrieben:

Nach einem Beispiel aus einem Vortrag über OOP & Maintenance von E. Plödereder, Universität Stuttgart.

Überschreiben - Überladen

vorher:

```
class C1
{ void doM (C1 x, T1 B);
  {...}
}

class C2 extends C1 {...}

class C3 extends C2
{ void doM (C1 x, T1 B);
  {...}
}

class C4 extends C3 {...}
```

C3.doM **überschreibt** C1.doM

nachher:

```
class C1
{ void doM (C1 x, T2 B);
  {...}
}

class C2 extends C1 {...}

class C3 extends C2
{ void doM (C1 x, T1 B);
  {...}
}

class C4 extends C3 {...}
```

C3.doM **überlädt** C1.doM

Nach einem Beispiel aus einem Vortrag über OOP & Maintenance von E. Plödereder, Universität Stuttgart.

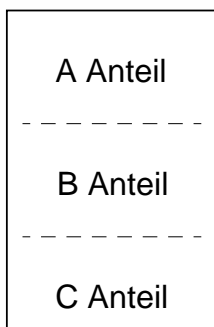
6 Jenseits von Java 6.1 Variationen von Inheritance 6.1.1 Konkatenation oder Delegation

Betrifft Implementierung von Objektspeicher und Methodenbindung

Konkatenation:

Objektspeicher:

zusammenhängender Speicher mit Anteilen für die Klasse und jede Oberklasse



Methodensuche:

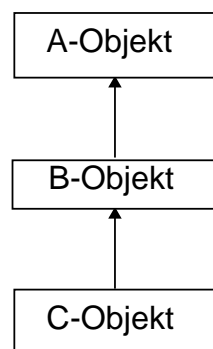
wird für C-Objekt vollständig erledigt

Sprachen: C++, Eiffel, Simula, Beta

Delegation:

Objektspeicher:

Speicherblock für das Objekt und Referenzen auf die Elternobjekte



Methodensuche:

Aufrufe werden in einem Objekt erledigt oder an Oberobjekt delegiert

Sprachen: Smalltalk, Self

6.1.2 Multiple Inheritance

Single Inheritance (Einfachvererbung):

Jede Klasse hat **höchstens oder genau eine direkte Oberklasse** (Base Class).
 Inheritance-Relation ist linear aus Sicht einer Unterklasse;
 Inheritance-Relation ist ein **Wald bzw. Baum** für alle Klassen zusammen.
 Sprachen: Smalltalk, Java (für Klassen, nicht für Interfaces).

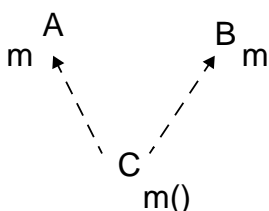
Multiple Inheritance (Mehrfachvererbung):

Jede Klasse hat **beliebig viele direkte Oberklassen**
 Inheritance-Relation ist DAG (aus Sicht einer Klasse und insgesamt)
 Sprachen: C++, Eiffel, CLOS

Java's Inheritance und Interfaces können in C++ mit Multiple Inheritance und rein abstrakten Klassen nachgebildet werden.

C++ ermöglicht Übergang von Interfaces zu Klassen.

Namensproblem: gleicher Name in verschiedenen Klassen erreichbar



C++: undifferenzierter Zugriff **m** statt **A::m** ist Fehler

Eiffel: Erben ohne Umbenennung ist Fehler

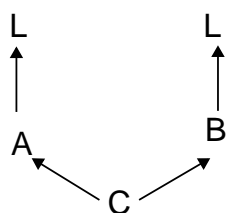
CLOS: Kanten sind geordnet, erster Weg zu **m** gilt

Mehrfach vorkommende Oberklasse

Eine Klasse **L** kann **mehrfach indirekte Oberklasse** einer Klasse **C** sein.

C++: zwei Varianten:

Konkatenation

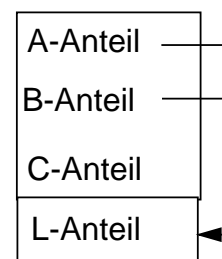
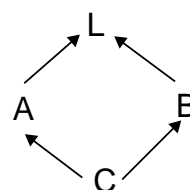


L-Anteil v. A.
A-Anteil
L-Anteil v. B
B-Anteil
C-Anteil

```

class A: public L {...}
class B: public L {...}
    
```

Objekte enthalten **L**-Anteile mehrfach
A und **B** benutzen **L** unabhängig
 (Attribute und Methoden)
C muss differenzieren



```

class A: public virtual L {...}
class B: public virtual L {...}
    
```

Objekte enthalten **L**-Anteil nur einmal;
L hat gemeinsame Daten für die Unterklassen

Eiffel: nur diese Variante mit expliziter Selektion

6.1.3 Selektives Erben in Eiffel

Einzelne Methoden und Datenattribute der Oberklasse können **beim Erben** explizit **überschrieben, umbenannt, gelöscht** werden:

- **Überschreiben** (`redefine`)
wird explizit gemacht!
- **Umbenennen** (`rename`),
z. B. um Namenskonflikte zu vermeiden;
ermöglicht auch, dieselbe Klasse direkt mehrfach zu erben
Problem: Ersetzbarkeit, Subtyping werden zerstört!
- **Löschen** - nicht erben
`undefine f end;`
Problem: Ersetzbarkeit, Subtyping werden zerstört!
- Für **bestimmte** Klassen **zugreifbar** machen
`export {A,B} f; ANY g; end`

```
class FIXED_TREE[T] inherit
  TREE[T]
  redefine
    attach_to_higher, higher
  end;

  CELL[T];

  LIST [like Current]
  rename
    off as child_off, ...
  redefine
    duplicate, first_child
  end

feature
  ....
end;
```

Weitere Eigenschaften von Eiffel

- **Generische Definitionen**
- **Multiple inheritance:**
`FIXED_TREE[T]` erbt von
`TREE[T], CELL[T], LIST[...]`
- **Current** entspricht `this` o. `self`
- **like x**: steht für den Typ von `x`,
z.B. einsetzbar zur Lösung des Problems
der binären Operationen:

Eine Methode `plus` in der Klasse `A` sei
deklariert als

`like Current plus (like Current)..`

dann hat sie in `A` die Signatur

`plus: A x A -> A`

In einer Unterklasse `B` wird sie mit der
Signatur geerbt

`plus: B x B -> B`

```
class FIXED_TREE[T] inherit
  TREE[T]
  redefine
    attach_to_higher, higher
  end;

  CELL[T];

  LIST [like Current]
  rename
    off as child_off, ...
  redefine
    duplicate, first_child
  end

feature
  ....
end;
```

6.1.4 Mixin Inheritance

Mixin:

Wiederverwendbare Zusammenfassung weniger Methoden und Datenattribute für bestimmten Zweck.

Mixins liegen **außerhalb der Klassenhierarchie**, werden von Klassen geerbt.

```

mixin Name
{
  String name;
  void setName (String n){.... }
  string getName () {.... }
}

mixin Adresse
{.... }
    
```

```

class Person inherits Name, Adresse
{
  ....
}

class Firma inherits Adresse
{
  ....
}
    
```

Nicht explizit als Konstrukt in verbreiteten Sprachen.

Kann mit Mehrfachvererbung implementiert werden (z. B. in C++, Eiffel).

6.2 Varianten der dynamischen Methodenbindung

Richtung der Methodensuche: aufwärts

Die Entscheidung, welche von **überschriebenen Methoden aufgerufen** wird, kann als **Suche entlang der Vererbungsrelation** beschrieben werden:

von der Klasse des Objekts **aufwärts durch die Oberklassen** **ersetzt** die überschreibende Methode die überschriebene **vollständig** - ggf. mit **anderer Bedeutung!**

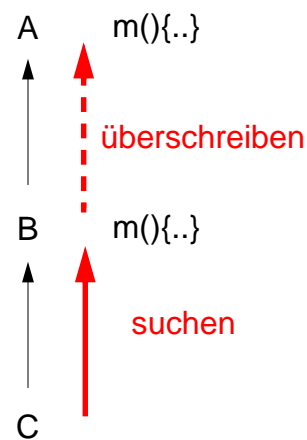
Die Oberklasse kann sich dagegen nicht absichern.

Sprachen: Java, C++ , Eiffel, Smalltalk

In Smalltalk ist die Suche so als Delegation implementiert.

In übersetzten Sprachen wird die Suche möglichst vermieden - durch Übersetzertabellen.

Wird auch **American semantics** genannt.



C c = new C(); c.m()

Richtung der Methodensuche: abwärts

Die Entscheidung, welche **Methoden aufgerufen** werden, kann als **Suche entlang der Vererbungsrelation** beschrieben werden:

Von der **höchsten Oberklasse abwärts zur Klasse des Objektes** werden **alle Methoden** gleichen Namens und passender Signatur **ausgeführt**.

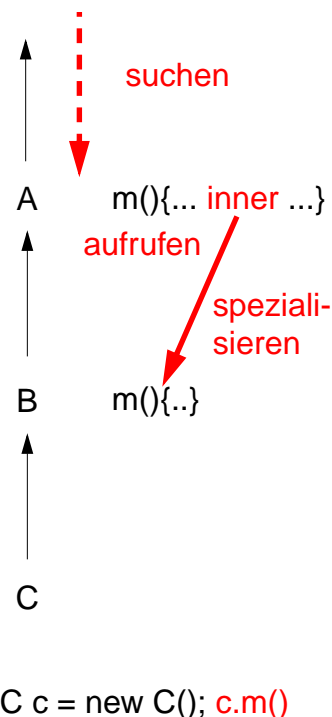
Im Rumpf der Methode kann durch **inner** gekennzeichnet werden, wo die Ausführung „tieferer“ Methoden eingebettet wird.

Die Oberklasse kann so **Erweiterungsstellen** definieren.

Die **Methoden** werden so in Unterklassen **spezialisiert** statt ersetzt.

Sprachen: Beta, Simula.

Wird auch **Scandinavian semantics** genannt.



Mehrfach-Dispatch

Einfach-Dispatch:

Methodenaufruf wird **auf ein Objekt** angewandt (Receiver): `a.plus (b)`

Die **Klassenzugehörigkeit des Objektes** in **a** bestimmt, welche Methode aufgerufen wird. Die Parameterobjekte tragen nicht zu der Entscheidung bei. Auch konzeptionell symmetrische Operationen werden **unsymmetrisch** ausgeführt.

Sprachen: fast alle objektorientierten Sprachen

Mehrfach-Dispatch:

Die **Klassenzugehörigkeiten aller Parameterobjekte zusammen** bestimmen, welche Methode aufgerufen wird. `plus (a, b)`

Symmetrische Behandlung der Parameter; kein hervorgehobener Receiver; z. B. `plus` für `int`, `float` und `vector` unterschieden anhand der Objekte in **a** und **b**.

Sprachen: CLOS, Lisp-Derivate

6.3 Prototyp-basierte Sprachen

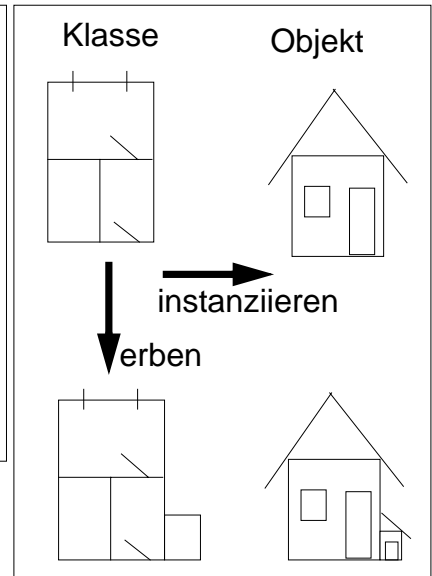
Klassen-basierte Sprachen (die meisten OO-Sprachen):

Klasse:

- Programmelement
- Plan zur Herstellung von Objekten
- vereinigt statische Eigenschaften der Objekte
- Vererbung zwischen Klassen

Objekt:

- zur Laufzeit erzeugt
- Exemplar einer Klasse
- speichert individuelle Daten, Zustand

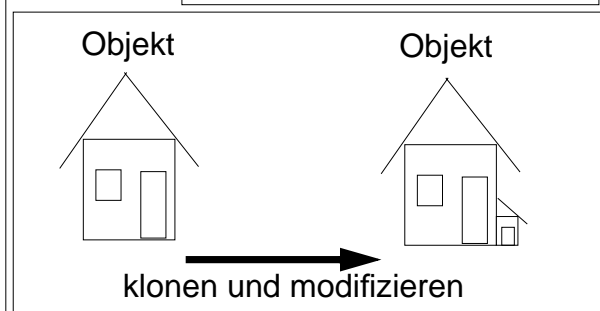


Prototyp-basierte Sprachen (Self, JavaScript):

Es gibt **keine Klassen** - nur **Objekte**.

Objekte werden

- **neu hergestellt** oder
- **aus Prototyp-Objekten kopiert** (geklont) und
- **individuell modifiziert**, weiterentwickelt
- **Vererbung zwischen Objekten**



JavaScript: Objektorientierte Scriptsprache

Scriptsprache:

- Sprache zum Aufruf von Programmen, ursprünglich Betriebssystem Kommandosprache, z. B. Unix Shell ähnlich: Perl, PHP, Python
- interpretiert
- dynamisch typisiert, einfache Datenstrukturen
- komfortable Operationen auf Strings

JavaScript:

- Scriptsprache zur Programmierung von Effekten auf Web-Seiten
- von Netscape entwickelt
- in Browser integriert
- kein Bezug zu Java
- objektorientiert, Prototyp-basiert
- in HTML (u.a.) integriert

```
<html>
  <head>
    <title>Beispiel mit JavaScript
    </title>
    <script type="text/javascript">
      function TestObjekt()
      { alert("Hello World!");
      }
    </script>
  </head>
  <form ...>
    <input type="submit" ...
      onclick="TestObjekt()">
  </form>
</body>
</html>
```

Objekterzeugung

Objekte werden durch **Konstruktorfunktionen** erzeugt:

- mit `new` aufgerufen,
- Funktionsrumpf greift über `this` auf Datenattribute zu
- **Datenattribute** werden durch **Zuweisung** eingeführt



```
function Farbe
  (Farbwert_R, Farbwert_G, Farbwert_B)
{
  this.Farbwert_R = Farbwert_R;
  this.Farbwert_G = Farbwert_G;
  this.Farbwert_B = Farbwert_B;
}

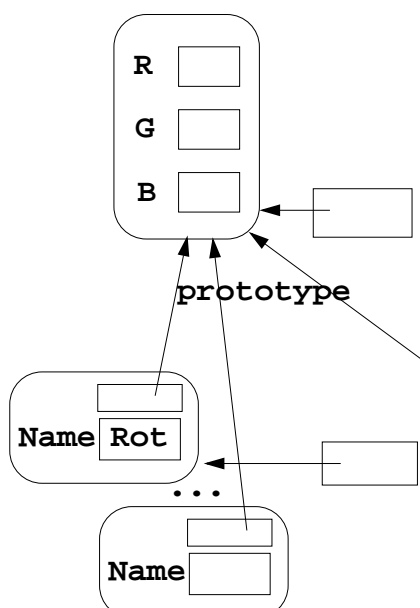
function TestObjekt()
{
  Test = new Farbe("A2", "FF", "74");
  alert("Der Rotwert meiner Farbe ist " +
    Test.Farbwert_R);
}
```

Objekt aus Prototyp erzeugen

Prototyp-Objekt der Konstruktorfunktion zuordnen.

Alle erzeugten **Objekte erben die Prototyp-Eigenschaften**.

Delegation: Ändern des Prototyps ändert alle Objekte dazu - auch existierende



```
function Farbe
  (Farbwert_R, Farbwert_G, Farbwert_B)
{
  this.Farbwert_R = Farbwert_R;
  this.Farbwert_G = Farbwert_G;
  this.Farbwert_B = Farbwert_B;
}

Rot = new Farbe("FF", "00", "00");

function FarbeMitNamen(Name)
{
  this.Name = Name;
}

FarbeMitNamen.prototype = Rot;

function TestObjekt()
{
  Test = new FarbeMitNamen("Rot");
  alert("Der Rotwert meiner Farbe ist " +
    Test.Farbwert_R +
    ". Der Name ist " + Test.Name);
}
```

Objekt-Hierarchie

Objekt-Hierarchie aus Prototypen; dynamische Zuordnung von Methoden:

Methode
zuweisen

**Konstruktor-
Funktionen** mit
fig als Prototyp

**dynamische
Bindung** für
Namen von
Attributen und
Methoden

**veränderlicher
Zustand** im
gemeinsamen
Prototyp

Überschreiben:
Suche entlang der
Prototyp-
Referenzen

```
fig = new function(){}(); // leeres Objekt erzeugen
fig.move =
  function (dx, dy) // lambda-Ausdruck
  { this.x += dx; this.y += dy};

function Kreis(r)
{ this.x = 0; this.y = 0; this.radius = r;}
Kreis.prototype = fig;

function Rechteck(l, b)
{ this.x = 0; this.y = 0; this.lg = l; this.br = b;}
Rechteck.prototype = fig;

function TestObjekt()
{ k1 = new Kreis(1); k1.move(5, 5);
  r1 = new Rechteck(3,7); r1.move(2,4);

  fig.printKoord =
    function ()
    { alert( "x = " + this.x + " y = " + this.y);}

  k1.printKoord(); r1.printKoord();
}
```

Prototypen gegenüber Klassen

positiv für Prototypen:

- Prototypen sind **konkreter**, näher an Objekten als Klassen es sind.
- **Einzige Objekte** einer Art sind einfacher herzustellen.
- **Einfache**, freizügig verwendbare Sprachkonzepte
- Wenn **Klassen** „**first class**“ sind braucht man **Meta-Klassen** und dafür Meta-Klassen ...

negativ für Prototypen:

- **Konstruktorfunktionen** sind ein halber Schritt in Richtung Klassen.
- **Individuelle Objektzustände** können nicht im gemeinsamen Prototyp gespeichert werden; Abstraktion wird inkonsequent
- keine **Typsicherheit** (bezüglich Klassen)
- schwächere Möglichkeiten zur **Abstraktion**
- zur **Modellierung** nicht geeignet (Ebene der Klassen und Typen fehlt)

6.4 Scala: objektorientierte und funktionale Sprache

Scala: Objektorientierte Sprache (wie Java, in kompakterer Notation) ergänzt um funktionale Konstrukte (wie in SML); objektorientiertes Ausführungsmodell (Java)

funktionale Konstrukte:

- geschachtelte Funktionen, Funktionen höherer Ordnung, Currying, Fallunterscheidung durch Pattern Matching
- Funktionen über Listen, Ströme, ..., in der umfangreichen Sprachbibliothek
- parametrische Polymorphie, eingeschränkte, lokale Typinferenz

objektorientierte Konstrukte:

- Klassen definieren alle Typen (Typen konsequent oo - auch Grundtypen), Subtyping, beschränkbare Typparameter, Case-Klassen zur Fallunterscheidung
- objektorientierte Mixins (Traits)

Allgemeines:

- statische Typisierung, parametrische Polymorphie und Subtyping-Polymorphie
- sehr kompakte funktionale Notation
- komplexe Sprache und recht komplexe Sprachbeschreibungen
- übersetzbar und ausführbar zusammen mit Java-Klassen
- seit 2003, Martin Odersky, www.scala.org

Übersetzung und Ausführung: Scala und Java

• Reines Scala-Programm:

ein Programm bestehend aus einigen Dateien `a.scala`, `b.scala`, ... mit Klassen- oder Objekt-Deklarationen in Scala,
eine davon hat eine `main`-Funktion;

übersetzt mit `scalac *.scala`
ausgeführt mit `scala MainKlasse`

```
// Klassendeklarationen
object MainKlasse {
// Funktionsdeklarationen
    def main(args: Array[String]) {
// Ein- und Ausgabe, Aufrufe
    }
}
```

• Java- und Scala-Programm:

ein Programm bestehend aus Scala-Dateien `a.scala`, `b.scala`, ... und Java-Dateien `j.java`, `k.java`, ...;
eine Java-Klasse hat eine `main`-Funktion;

übersetzt mit `scalac *.scala *.java`
dann mit `javac *.scala *.java`
(Pfad zur Bibliothek angeben)
ausgeführt mit `java MainKlasse`

• Reines Scala-Programm interaktiv: (siehe Übungen)

Benutzung von Listen

Die abstrakte **Bibliotheksklasse** `List[+A]` definiert Konstruktoren und Funktionen über **homogene Listen**

```
val li1 = List(1,2,3,4,5)

val li2 = 2 :: 4 :: -1 :: Nil
```

Verfügbare Funktionen:

`head`, `tail`, `isEmpty`, `map`, `filter`, `forall`, `exist`, `range`, `foldLeft`, `foldRight`, `range`, `take`, `reverse`, `:::` (`append`)

zwei Formen für Aufrufe:

```
li1.map (x=>x*x)// qualifizierter Bezeichner map

li1 map (x=>x*x)// infix-Operator map
```

Funktionsdefinitionen mit Fallunterscheidung:

```
def isort(xs: List[Int]): List[Int] = xs match {
  case List() => List()
  case x :: xs1 => insert(x, isort(xs1))
}

def insert(x: Int, xs: List[Int]): List[Int] = xs match {
  case List() => List(x)
  case y :: ys => if (x <= y) x :: xs else y :: insert(x, ys)
}
```

Case-Klassen: Typkonstruktoren mit Pattern Matching

Klassen können **Parameter** haben. Sie sind Instanzvariable der Klasse und Parameter des Konstruktors.

Die **Konstruktoren von Case-Klassen** können zur **Fallunterscheidung** und zum **Binden der Werte** dieser Instanzvariablen verwendet werden. Objekte können ohne `new` gebildet werden; Methoden für strukturellen Vergleich (`==`) und `toString` werden erzeugt.

```
abstract class Person
case class King    () extends Person
case class Peer    (degr: String, terr: String, number: Int )
                  extends Person
case class Knight  (name: String) extends Person
case class Peasant (name: String) extends Person

val guestList =
  Peer ("Earl", "Carlisle", 7) :: King () ::
  Knight ("Gawain") :: Peasant ("Jack Cade") :: Nil

def title (p: Person): String = p match {
  case King () => "His Majesty the King"
  case Peer (d, t, n) => "The " + d + " of " + t
  case Knight (n) => "Sir " + n
  case Peasant(n) => n }

println ( guestList map title )

List(His Majesty the King, The Earl of Carlisle, Sir Gawain, Jack Cade)
```

Definition polymorpher Typen

Polymorphe Typen werden durch **Klassen mit Typparameter** definiert, z.B. Container-Klassen.

Alternative Konstruktoren werden durch **Case-Klassen** formuliert, z.B. Binärbäume.

```
abstract class BinTree[A]
case class Lf[A] () extends BinTree[A]
case class Br[A] (v: A, left: BinTree[A], right: BinTree[A])
    extends BinTree[A]
```

Funktionen über Binärbäume:

```
def preorder[A] (p: BinTree[A]): List[A] = p match {
  case Lf() => Nil
  case Br(v,tl,tr) => v :: preorder (tl) ::: preorder (tr)
}

val tr: BinTree[Int] =
  Br (2, Br (1, Lf(), Lf()), Br (3, Lf(), Lf()))

println ( preorder (tr) )
```

Funktionen höherer Ordnung und Lambda-Ausdrücke

Ausdrucksmöglichkeiten in Scala entsprechen etwa denen in SML, aber die **Typinferenz polymorpher Signaturen** benötigt an vielen Stellen **explizite Typangaben**

Funktion höherer Ordnung: Faltung für Binärbäume

```
def treeFold[A,B] (f: (A, B, B)=>B, e: B, t: BinTree[A]): B =
  t match {
    case Lf () => e
    case Br (u,tl,tr) =>
      f (u, treeFold (f, e, tl), treeFold (f, e, tr))
  }
```

Lambda-Ausdrücke:

```
11.map ( x=>x*x )           Quadrat-Funktion
13.map ( _ => 5 )           konstante Funktion
12.map ( Math.sin _ )      Sinus-Funktion
14.map ( _ % 2 == 0 )      Modulo-Funktion

treefold ( ((_: Int, c1: Int, c2: Int) => 1 + c1 + c2) , 0, t)
```

Currying

Funktionen in **Curry-Form** werden durch mehrere **aufeinanderfolgende Parameterlisten** definiert:

```
def secl[A,B,C] (x: A) (f: (A, B) => C) (y: B) = f (x, y);
def secr[A,B,C] (f: (A, B) => C) (y: B) (x: A) = f (x, y);
def power (x: Int, k: Int): Int =
  if (k == 1) x else
  if (k%2 == 0) power (x*x, k/2) else
    x * power (x*x, k/2);
```

Im Aufruf einer Curry-Funktion müssen **weggelassene Parameter** durch **_** angegeben werden:

```
def twoPow = secl (2) (power) _ ; Funktion, die 2er-Potenzen berechnet
def pow3 = secr (power) (3) _ ; Funktion, die Kubik-Zahlen berechnet
println ( twoPow (6) )
println ( pow3 (5) )
println ( secl (2) (power) (3) )
```

Ströme in Scala

In Scala werden **Ströme** in der Klasse `Stream[A]` definiert.

Besonderheit: Der **zweite Parameter der cons-Funktion** ist als **lazy** definiert, d.h. ein aktueller **Parameterausdruck** dazu wird erst ausgewertet, wenn er benutzt wird, d.h. der Parameterausdruck wird in eine **parameterlose Funktion** umgewandelt und so übergeben. Diese Technik kann allgemein für Scala-Parameter angewandt werden.

```
def iterates[A] (f: A => A) (x: A): Stream[A] =
  Stream.cons(x, iterates (f) (f (x)))
def smap[A] (sq: Stream[A]) (f: A => A): Stream[A] =
  Stream.cons(f (sq.head), smap[A] (sq.tail) (f) )
val from = iterates[Int] (_ + 1) _
val sq = from (1)
val even = sq filter (_ % 2 == 0)
val ssq = from (7)
val msq = smap (ssq) (x=>x*x)
println( msq.take(10).mkString(",") )
```

Objektorientierte Mixins

Mixin ist ein Konzept in objektorientierten Sprachen: Kleine Einheiten von implementierter Funktionalität können Klassen zugeordnet werden (spezielle Form der Vererbung). Sie definieren nicht selbst einen Typ und liegen neben der Klassenhierarchie.

```
abstract class Bird { protected val name: String }

trait Flying extends Bird {
    protected val flyMessage: String
    def fly() = println(flyMessage)
}

trait Swimming extends Bird {
    def swim() = println(name+" is swimming")
}

class Frigatebird extends Bird with Flying {
    val name = "Frigatebird"
    val flyMessage = name + " is a great flyer"
}

class Hawk extends Bird with Flying with Swimming {
    val name = "Hawk"
    val flyMessage = name + " is flying around"
}

val hawk = (new Hawk).fly(); hawk.swim(); (new Frigatebird).fly();
```

Verschiedene Verhaltensweisen werden hier als **trait** definiert:

7. Zusammenfassung (1)

Typisierung in OO-Sprachen

- Subtyping vs. Subclassing
- Untertypen für Typkonstrukte
- Funktionsuntertypen und Überschreiben
- Generik

Einsatz von Vererbung konzeptioneller Entwurf:

- Abstraktion
- Spezialisierung
- Rollen, Eigenschaften
- Schnittstellenabstraktion
- Programmentwicklung:**
- Inkrementelle Weiterentwicklung
- Komposition statt Vererbung
- Varianten in mehreren Dimensionen
- Eingebettete Agenten

Entwurfsmuster zur Entkopplung von Software

- Abstract Factory
- Factory Method
- Bridge
- Observer
- Strategy

Bibliotheken

- Programmbausteine am Beispiel Ausnahmen
- Kopplung, Kohärenz
- Entkoppeln durch Interfaces, Funktoren
- Programmgerüste am Beispiel AWT

Zusammenfassung (2)

Entwurfsfehler

Missbrauch von Vererbung:

- Verkannter Schauspieler
- Januskopf
- Ungeplanter Anbau

AntiPatterns

- The Blob
- Poltergeister
- Funktionale Zerlegung
- Spaghetti Code

Üble Gerüche im Code

- Symptome für Refaktorisieren, Katalog

Unangenehme OOP-Überraschungen

- Ursache: meist Überschreiben

OO Sprachkonzepte und ihr Einsatz

- Klassen - Objekte vs. Typen - Werte
- Attribute, Methoden, Konstruktoren
- Zugriffsrechte
- Schnittstelle, Implementierung
- Vererbung, Subtyping
- Polymorphie, dyn. Methodenbindung

Jenseits von Java

- Konkatenation - Delegation
- Multiple Inheritance
- Selektives Erben
- Mixin Inheritance
- Varianten der Methodenbindung
- Prototypen statt Klassen

Verständnisfragen zur Objektorientierten Programmierung (1)

1. Erklären Sie, dass Subclassing und Subtyping unterschiedliche Relationen sind.
2. Erklären Sie Subtyping für Records, Funktionen, Variable, Arrays und Objekte.
3. Leiten Sie die Typregel für das Überschreiben aus dem Subtyping ab.
4. Vergleichen Sie Generik in C++ und Java.
5. Wie löst Generik das Typproblem für „binäre Methoden“?
6. Nennen Sie Paradigmen der Vererbung auf Entwurfsebene.
7. Was bedeuten „ Ersetzbarkeit“ und „Subtyping“ beim Einsatz von Vererbung?
8. Nennen Sie einige Kriterien gegen die Anwendung von Vererbung.
9. Erklären Sie Vererbung zum Zwecke der Abstraktion.
10. Grenzen Sie die Vererbungsparadigmen Abstraktion und Spezialisierung gegeneinander ab.
11. Warum muss man beim Paradigma Abstraktion mehr als 2 Klassen zugleich betrachten?
12. Welches Design Pattern basiert auf dem Abstraktions-Paradigma?
13. Erklären Sie Vererbung zum Zwecke der Spezialisierung.
14. Grenzen Sie die Paradigmen Spezialisierung und inkrementelle Weiterentwicklung gegeneinander ab.

Verständnisfragen zur Objektorientierten Programmierung (2)

15. Welche Rolle spielt Spezialisierung bei Programmgerüsten?
16. Erklären Sie die Vererbungsparadigmen Rollen, Eigenschaften.
17. Erklären Sie das Schema der Delegation an implementierten Rollen.
18. Grenzen Sie die Paradigmen Rollen und Abstraktion gegeneinander ab.
19. Erklären Sie das Vererbungsparadigma Spezifikation.
20. Grenzen Sie das Paradigma Spezifikation gegen Spezialisierung, Abstraktion und Rollen ab.
21. Erklären Sie die Technik inkrementelle Weiterentwicklung.
22. Wie kann man Probleme der inkrementellen Weiterentwicklung vermeiden?
23. Unter welchen Umständen ist Komposition der Vererbung vorzuziehen?
24. Erklären Sie die Technik der Varianten in mehreren Dimensionen.
25. Erklären Sie die Technik der eingebetteten Agenten.
26. Nennen Sie Entwurfsmuster, die speziell der Entkopplung von Software dienen.
27. Erläutern Sie das Entwurfsmuster Abstract Factory.
28. Erläutern Sie das Entwurfsmuster Factory Method.

Verständnisfragen zur Objektorientierten Programmierung (3)

29. Erläutern Sie das Entwurfsmuster Bridge.
30. Erläutern Sie das Entwurfsmuster Observer.
31. Erläutern Sie das Entwurfsmuster Strategy.
32. Charakterisieren Sie unabhängige, eigenständige und eng kooperierende Bausteine.
33. Zeigen Sie das Zusammenspiel von Sprache und Bibliothek an Ausnahmen in Java.
34. Erklären Sie die Arten der Kopplung von Bausteinen.
35. Erklären Sie Entkopplung durch Interfaces und durch Funktoren.
36. Erklären Sie den Begriff der Koheränz.
37. Was charakterisiert ein Programmgerüst?
38. Geben Sie Beispiele für komplexes Zusammenwirken von Komponenten des AWT Programmgerüsts.
39. Welche Rolle spielt die Software-Architektur eines Programmgerüsts?
40. Geben Sie Schemata für den Missbrauch von Vererbung an.
41. Charakterisieren Sie einige AntiPatterns.
42. Beschreiben Sie Ziele und Methoden des Refactoring.

Verständnisfragen zur Objektorientierten Programmierung (4)

43. Was bedeuten „üble Code-Gerüche“ im Refactoring? geben Sie Beispiele.
44. Geben Sie Beispiele für versehentliches Überschreiben. Wie könnte man es vermeiden?
45. Geben Sie Empfehlungen für den Zugriff auf Attribute.
46. Stellen Sie Klassen und Objekte sowie Typen und Werte gegenüber.
47. Nennen Sie die 3 Situationen, die in Java zur dynamischen Methodenbindung führen.
48. Erläutern Sie die beiden Varianten der mehrfach indirekten Oberklassen in C++.
49. Erläutern Sie das selektive Erben in Eiffel.
50. Was bedeutet Mixin-Inheritance?
51. Erklären Sie das inner-Prinzip aus Simula und Beta.
52. Was bedeutet Mehrfach-Dispatch?
53. Erklären Sie Prototyp-basierte Vererbung an JavaScript.