Grundlagen der Informatik für Ingenieure I

3. Einführung in das objektorientierte Programmier-Paradigma

3.1 Software Design-Methoden
3.1.1 Top-down structured design
3.1.2 Data-driven design
3.1.3 Object-oriented design
3.2 Historische Entwicklung
3.3 OO-Programmierung - Grundlagen
3.3.1 Klassen, Objekte, Methoden
3.3.2 Eigenschaften(Attribute) und Verhalten(Methoden)
3.3.3 Kreieren einer Klasse
3.3.4 Diskussion des dynamischen Ablaufs eines Programm anhand eines Beispiels

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

3.1 Software Design-Methoden

- ***Top-down structured design (composite design)***
  - Traditionelle Software-Design-Methode
- ***Data-driven design***
  - Orientiert sich an der Abbildung von Eingabedaten auf Ausgabedaten
- ***Object-oriented design***
  - die bisher “modernste” Methode.

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

3.1.1 Top-down structured design (composite design)

Wesentlich durch die traditionellen Programmiersprachen (Fortran, Cobol, C, ...) beeinflußt.

- Einheit für Dekomposition: **Unterprogramm**
  - Resultierendes Programm hat die Form eines Baumes, in dem Unterprogramme ihre Aufgaben durch den Aufruf weiterer Unterprogramme erledigen.
- **Algorithmische Dekomposition** zur Zerlegung größerer Probleme
  - Zentrale Sicht:
    Ausführung einer Problemlösung zerlegt in Teilproblemlösungen und Einzelschritten;
    durch Vorgabe einer Reihenfolge --> Gesamtlösung
- Eignung für die Strukturierung heutiger, sehr großer Softwaresysteme wird bezweifelt, trotzdem existieren große Softwaresysteme auf der Basis dieser Methode. (Welcher Preis?)

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

3.1.1 Top-down structured design (composite design)

- Diese Methode erfaßt nicht:
  - Datenabstraktion & *Information Hiding*
  - Nebenläufigkeit
- Teilproblemlösungen operieren auf globalen Datenbestand, lokale Daten sind im allgemeinen Zwischenspeicher.
- Bislang am häufigsten eingesetzte Design-Methode

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

3.1.2 Data-driven design

- Grundlegende Arbeiten u. a. von Jackson
- Softwarestruktur beruht auf Abbildung von Eingabe in Ausgabe
  - Anwendung vor allem im Bereich *Information Management*
- Probleme vor allem mit zeitkritischen Ereignissen
- hat sich nicht durchsetzen können.

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

3.1.3 Object-oriented design

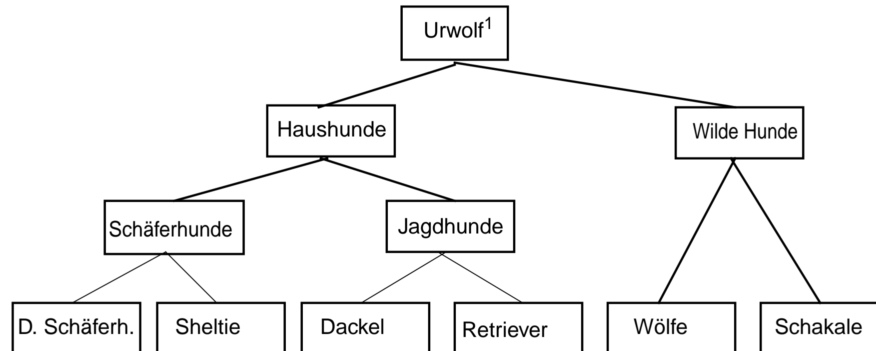
Bertrand Meyer:
"Rechner führen Operationen auf bestimmten Objekten aus. Um flexiblere und wiederverwendbare Systeme zu erhalten, ist es daher sinnvoller, die Software-Struktur an diesen Objekten statt an den Operationen zu orientieren."

- Softwaresystem wird als Sammlung kooperierender Objekte modelliert
- Einzelne **Objekte** sind **Instanz** einer **Klasse** in einer **Hierarchie** von **Klassen**
  - Eine Klasse ist die Beschreibung der Eigenschaften eines Objekts.
  - Hierachie entsteht durch **Oberklassen** und **Unterklassen**, wobei jeweils die Unterklasse eine Spezialisierung der Oberklasse ist.
  - Objekte/Instanzen werden zur Laufzeit aus dem *template* (= Muster) "Klasse" erzeugt.

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

3.1.3 Object-oriented design

- Klassenhierachie (verbreitet in der Wissenschaft):



1) Biologen mögen mir diese Taxonomie nachsehen

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

3.1.3 Object-oriented design

- in der Struktur modernerer, höherer Programmiersprachen reflektiert
  - Smalltalk
  - C++
  - ADA
  - Java

◆ Grundlage: objektorientierte Dekomposition
Vorteile:

- Wiederverwendung gemeinsamer Mechanismen;
  - Oberklassen beschreiben gemeinsame Eigenschaften, die von Unterklassen übernommen werden;
  - Software wird kompakter
- Software ist leichter zu ändern und weiterzuentwickeln
  - Änderungen sind örtlich begrenzt; Seiteneffekte werden vermieden; Änderungen einer Oberklasse haben eine einheitliche Auswirkung auf die Unterklassen
- Ergebnisse weniger komplex, da inkrementelle Entwicklung begünstigt wird.

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

3.2 Historische Entwicklung

- Generationen von Programmiersprachen
  - Erste Generation (1954 - 1958)
    - Mathematische Ausdrücke (FORTAN I, ALGOL58)
  - Zweite Generation (1959 - 1961)
    - Unterprogramme, getrennte Übersetzung (FORTAN II)
    - Blockstruktur, Datentypen (ALGOL60)
    - Datenbeschreibung, Dateibehandlung (COBOL)
    - Listenverarbeitung, Zeiger (Lisp)
  - Dritte Generation (1962 - 1970)
    - verschiedene ALGOL-Nachfolger (ALGOL68, Pascal)
    - Klassen, Datenabstraktion (Simula)
  - Vierte Generation (1970 - 1995)
    - Viele neue Sprachen, nur wenige haben Bedeutung erlangt
      - C, Pascal, Modula, Smalltalk, C++, Eiffel, Java

© Claus-Uwe Linster, Universität Erlangen-Nürnberg, Verteilte Systeme und Betriebssysteme,
SS 2001kap03_ooparadigma.fm 2002-04-08 14.59
Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

3.3 OO-Programmierung - Grundlagen

- Definition

  OOP ist eine Methode der Implementierung, in der Programme in Form von

  - **Mengen kooperierender Objekte**

    organisiert sind, wobei jedes Objekt
  - **Instanz einer Klasse**

    ist und die Klassen Bestandteil einer über
  - **Vererbungsbeziehungen**

    definierten Hierarchie von Klassen sind.

© Claus-Uwe Linster, Universität Erlangen-Nürnberg, Verteilte Systeme und Betriebssysteme,
SS 2001kap03_ooparadigma.fm 2002-04-08 14.59
Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

3.3.1 Klassen, Objekte, Methoden

- **Klasse *(Class)*** = Objektbeschreibung (Template; Muster; Vorschrift)
  - Beschreibung der Eigenschaften eines Objekts; jedes Objekt ist Instanz einer Klasse.
  - Eine Klasse beschreibt die Implementierung der Operationen von Objekten (→**Methoden *(methods)***) der Klasse
  - und deren Zustandsstruktur (→**Attribute**).
    Die “Hülle” der Variablenzustände eines Objekts ist der **Objektzustand.**
  - **Vererbung *(inheritance)***: Beziehungsrelation zwischen Klassen.
- **Objekt/Instanz** = aus einer Klasse (dynamisch = zur Laufzeit) erzeugtes Objekt
  - Objektorientierte Programmierung verwendet Objekte und nicht Algorithmen als die grundlegenden Bausteine der Problemlösung.
  - Die Begriffe Instanz und Objekt werden synonym verwendet. Es handelt sich um die konkrete Repräsentation einer Klasse, gemäß der “generischen” Beschreibung der Klasse.

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

3.3.1 Klassen, Objekte, Methoden

- **Instantiierung** = Erzeugung eines Objekts(einer Instanz) einer Klasse.
  - Alle Objekte können den in der Klasse implementierten Code der Methoden gemeinsam nutzen, haben aber jeweils eine eigene Version des Objektzustandes (Datenbasis), gemäß der Strukturbeschreibung in der Klasse.
- **Instanzvariablen (Attribute)** = Variablen eines Objekts (einer Instanz) = Zustand

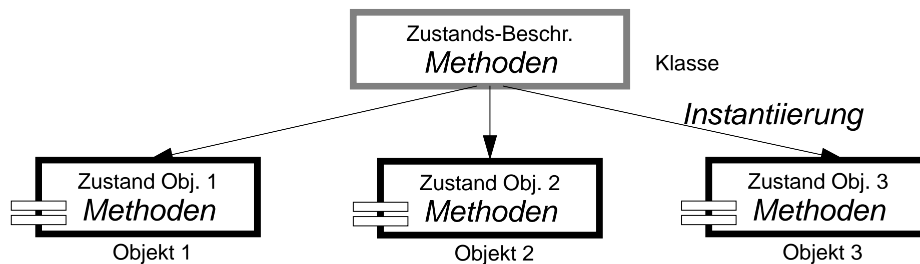
Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

3.3.1 Klassen, Objekte, Methoden

◆ **Methoden** = Operationen eines Objekts (Schnittstelle)

- Die Methoden eines Objekts definieren das Verhalten (die Zustandsübergänge) eines Objekts.

◆ Graphische Darstellung:

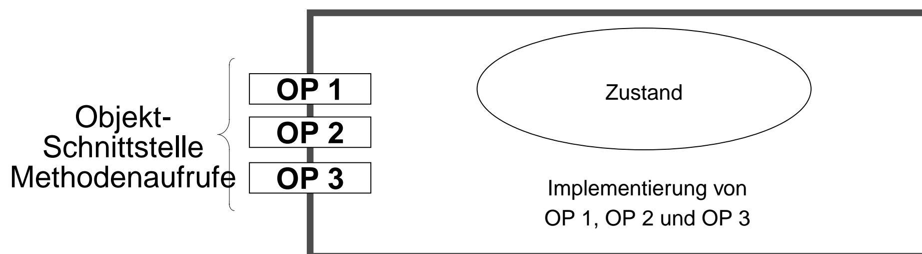


© Claus-Uwe Linster, Universität Erlangen-Nürnberg, Verteilte Systeme und Betriebssysteme,

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

3.3.1 Klassen, Objekte, Methoden

◆ **Objekt** = Variablen (Zustand) + Operationen



- Zusammenfassung von Datenstrukturen und Operationen auf diese Datenstrukturen zu einer Programmeinheit.

◆ Klassisches Beispiele: Warteschlange oder Stack (Kellerspeicher). Beide als verkettete Liste implementiert mit den Operationen “out” (ältestes Element) “in” (jüngstes Element) bei der Warteschlange und “push” und “pop” (jüngstes Element) beim Stack.

© Claus-Uwe Linster, Universität Erlangen-Nürnberg, Verteilte Systeme und Betriebssysteme,

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

3.3.2 Eigenschaften(Attribute) und Verhalten(Methoden)

◆ Anhand eines abstrakten Motorrades soll das bisher Besprochene vertieft werden:

- Zunächst kreieren wir eine Klasse *Motorcycle*
- diese Klasse hat einige Eigenschaften (Attribute), die wir mit Hilfe von ➔Variablen und deren ➔Wertebereiche beschreiben wollen, wie z. B.:

```
color: [red, green, silver, brown]
make: [Honda, BMW, Yamaha]
engineState: [ein, aus]
                                  [...]: Wertebereich
```

color, make, engineState sind die Namen der Instanz-Variablen. Der Inhalt aller Variablen beschreibt den Zustand des Objekts.

- Das Verhalten dieser Klasse wird von Methoden beschrieben, die die Inhalte der Variablen verändern können; d. h. also, den Zustand des Objekts verändern können.

```
Start the engine      --> startEngine
Stop the engine       --> stopEngine
Show Attributs        --> showAtts
```


Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

3.3.3 Kreieren einer Klasse

◆ Kreieren einer Klasse:

```
class Motorcycle {
}
```

◆ Vereinbarung der Instanzvariablen

```
String make;
String color;
boolean engineState;
```

◆ Definition einer Methode

```
void startEngine() {
     if (engineState == true)
         System.out.println("The engine is already on.");
     else {
          engineState = true;
          System.out.println("The engine is now on.");
     }
}
```


Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

3.3.3 Kreieren einer Klasse

- Wenn wir den bisher erstellten Programmcode übersetzen wollten, würde der Compiler mit einer Fehlermeldung “aussteigen”.
  Was fehlt, ist eine Methode, die unsere Klasse “benutzt”. Der Java-Interpreter muß wissen, mit welcher Anweisung er die Programmausführung beginnen soll.
  Deshalb muß es genau eine Methode geben, die den Namen “*main”* besitzt:

```
public static void main (String args[]) {
    Motorcycle m
        m = new Motorcycle();
        m.setmake("Yamaha RZ350");
        m.setcolor("yellow");
        System.out.println("Calling showAtts...");
        m.showAtts();
        .....
```

Für die Methode "main" können wir eine weitere Klasse definieren (die Regel!) oder aber, sie in die Klasse *Motorcycle* integrieren (bei kleinen Beispielen praktisch) . Bei unserem vollständigen Beispiel haben wir letzteres gemacht. In der Vorlesung wird die 1. Möglichkeit erläutert.

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

3.3.4 Diskussion des Ablaufs des Programms

```
class Motorcycle {

  private String make;
  private String color;
  private boolean engineState;

     void startEngine() {
       if (engineState == true)
         System.out.println("The engine is already on.");
       else {
         engineState = true;
         System.out.println("The engine is now on.");
       }
     }
     void showAtts() {
       System.out.println("This motorcycle is a "+ color + " " + make);
         if (engineState == true)
           System.out.println("The engine is on.");
         else
           System.out.println("The engine is off.");
      }
     void setmake(String comp) {
       make = comp;
     }
     void setcolor(String co) {
       color = co;
     }
```


Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

3.3.4 Diskussion des Ablaufs des Programms

```
public static void main (String args[]) {
    Motorcycle m;

    m = new Motorcycle();
    m.setMake("Yamaha RZ350");
    m.setColor("yellow");

    System.out.println("Calling showAtts...");
    m.showAtts();
    System.out.println("--------");
    System.out.println("Starting engine...");
    m.startEngine();
    System.out.println("--------");
    System.out.println("Calling showAtts...");
    m.showAtts();
    System.out.println("--------");
    System.out.println("Starting engine...");
    m.startEngine();
  }
}
```


© Claus-Uwe Linster, Universität Erlangen-Nürnberg, Verteilte Systeme und Betriebssysteme,

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

Notizen

© Claus-Uwe Linster, Universität Erlangen-Nürnberg, Verteilte Systeme und Betriebssysteme,

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.