Software Engineering

Implementierungsphase

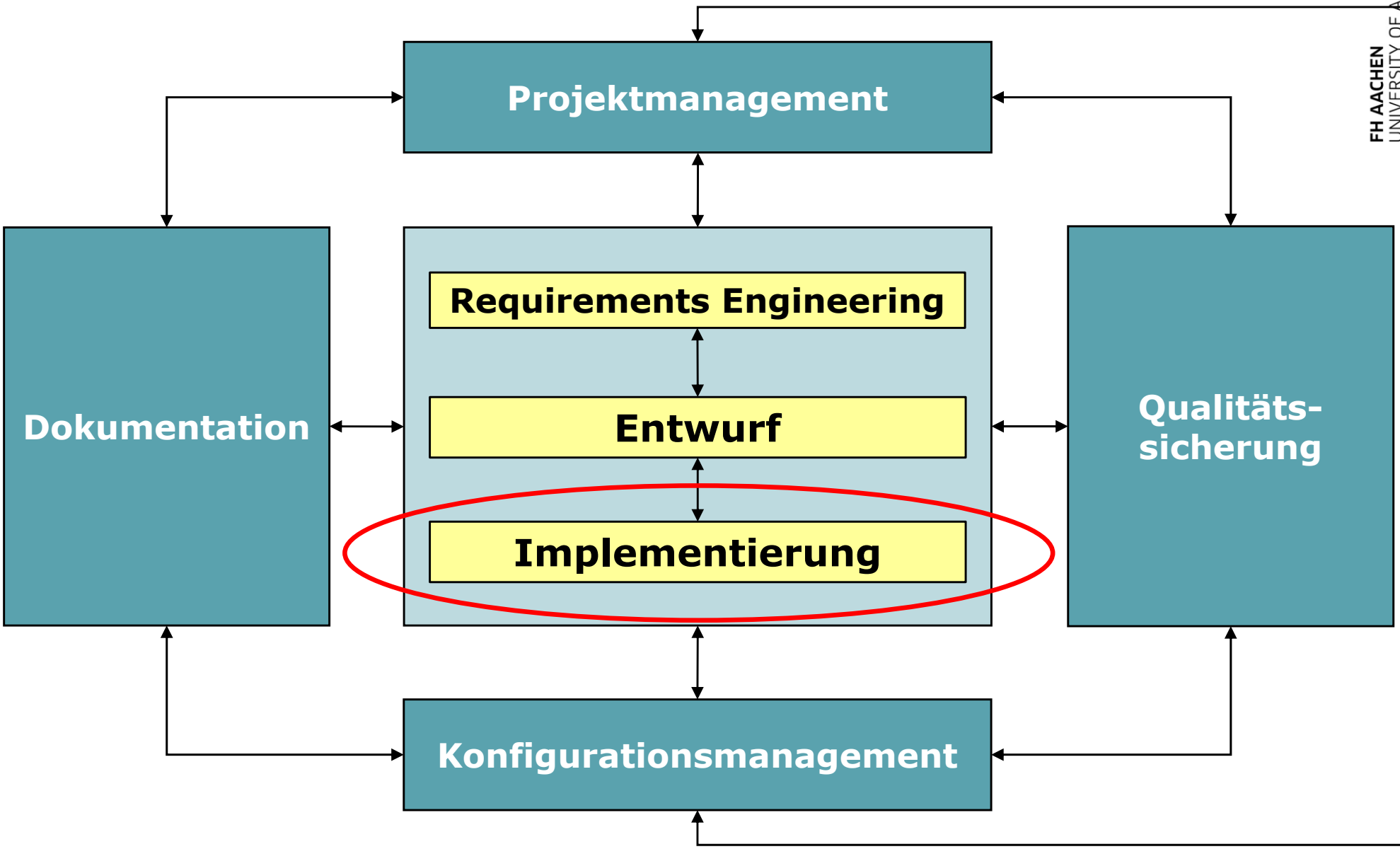
Prof. Dr. Bodo Kraft

FH AACHEN
UNIVERSITY OF APPLIED SCIENCES

Lernziele

- Wie wird das SW-System konkret hergestellt?
- Wie kann man Programmcode bewerten, unterschiedliche Implementierungen vergleichen?
- *Best-Practices* zur Fehlervermeidung

Motivation und Einordnung



Komplexität der Implementierungsphase
Motivation und Einordnung

Das Thema Programmiertechnik ist vergleichsweise weit fortgeschritten

Trotz oft guter Ausbildung der Entwickler der arbeits- und kostenintensivste Bereich der Softwareentwicklung

Besonderheiten:

- Kenntnisse über **Algorithmen und Datenstrukturen** notwendig
- Viele Notationen und **Programmier-Sprachen** mit unterschiedlichen Ansätzen verfügbar mit
- Viele **Werkzeuge** zur SDL-Unterstützung
- Zunehmend auch **Security** Aspekte
- Technologie sehr schnelllebig, oft komplex

Aufgaben der Implementierungsphase
Vom Entwurf zur Implementierung

1) Komponenten implementieren

- Geeignete Datenstrukturen wählen
- Algorithmen wählen
- In Programmiersprache formulieren /codieren

2) Komponenten dokumentieren

- Wie erledigt die Komponente ihre Aufgabe (Problemlösung)
- Implementierungsentscheidungen begründen
- Angaben zu Zeit- und Speicherkomplexität

3) Komponenten prüfen (vs. Entwurf)

- Testumgebung einrichten
- Testdaten erfassen
- Testläufe durchführen
- Verifizieren

Auch „Programmieren im Kleinen“ genannt

Die Systematische Implementierung
Grundprinzipien zur Erstellung guten Programmcodes

Das Ziel:

- Bekannte Probleme bei der Implementierung vermeiden
- hohe Codequalität erreichen und erhalten

Bekannte Probleme von Quellcode:

- Lesbar- & Verständlichkeit (äußert sich in Einarbeitungszeit)
- Wartbarkeit, Erweiterbarkeit, Wiederverwendbarkeit
- Robustheit
- Effizienz
- Eleganz, …

Idee: Fehler nicht beheben, sondern umgehen

Vorgehen beim Erstellen von Code:

1. Regeln und Richtlinien als Vorgaben nutzen (Code Conventions)
2. Prinzipien der systematischen Implementierung [Bal2011]

Motivation zur Verwendung von Code Conventions
Code-Conventions

1.1 Why Have Code Conventions [oracle.com]

Code conventions are important to programmers for a number of reasons:

- 80% of the lifetime cost of a piece of software goes to maintenance.
- Hardly any software is maintained for its whole life by the original author.
- Code conventions improve the readability of the software, allowing engineers to understand new code more quickly and thoroughly.

If you ship your source code as a product, you need to make sure it is as well packaged and clean as any other product you create.[…]

Motivation
Code-Conventions

Vorgaben sind oft unbeliebt, da als Einschränkung empfunden

- **Beispiel:** „Eingerückt wird mit 4 Leerzeichen.“

Regeln können jedoch viel unnötigen Aufwand ersparen!

- **Beispiel Prettyprinter:** überall automatisch 4 Leerzeichen

Gute Regeln

- Wählen eine von gleichwertigen Alternativen aus
- Verursachen keinen unnötigen Aufwand
- Machen den Kopf frei für interessantere Fragen
- Können im Konsens gepflegt werden

Übersicht Kategorien
Code-Conventions

Es gibt u.a. Vorgaben zu:

- **Formatierung**
  - Klammern
  - Tabs,
  - Blockeinrückung
- **Namenskonventionen**
  - Allg. Bezeichner-Konventionen
  - Typgebunden
- **Dokumentierung**
  - Inline
  - Javadoc
- **Stil**
  - Reihenfolge: Member, Operations

Siehe auch:
Google's coding standards for Java code

- **Annotationen**
  - Reihenfolge
  - Optionale Angaben
- **Ausnahmebehandlung**
  - Handling
  - Checked vs. Unchecked

Lesbarkeit, Verständlichkeit
Formatierungsstandards

Wenn man Entwickler frei laufen lässt …

… oder wie man es nicht machen sollte…

Was macht folgendes C-Programm?

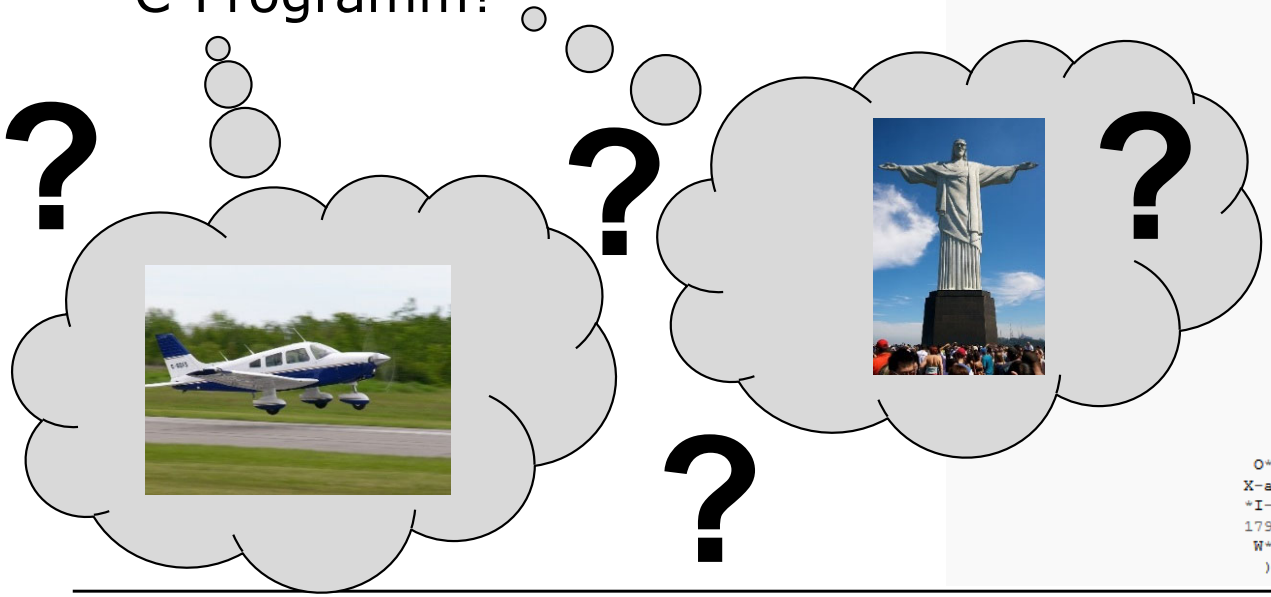
```
#include                                  <math.h>
#include                                <sys/time.h>
#include                                <X11/Xlib.h>
#include                               <X11/keysym.h>
                                        double L ,o ,P
                                       ,_=dt,T,Z,D=1,d,
                                       s[999],E,h= 8,I,
                                       J,K,w[999],M,m,O
                                      ,n[999],j=33e-3,i=
                                      1E3,r,t, u,v ,W,S=
                                      74.5,l=221,X=7.26,
                                      a,B,A=32.2,c, F,H;
                                      int N,q, C, y,p,U;
                                     Window z; char f[52]
                                   ; GC k; main(){ Display*e=
 XOpenDisplay( 0); z=RootWindow(e,0); for (XSetForeground(e,k=XCreateGC (e,z,0,0),BlackPixel(e,0))
; scanf("%lf%lf%lf",y +n,w+y, y+s)+1; y ++); XSelectInput(e,z= XCreateSimpleWindow(e,z,0,0,400,400,
0,0,WhitePixel(e,0) ),KeyPressMask); for(XMapWindow(e,z); ; T=sin(O)){ struct timeval G={ 0,dt*1e6}
; K= cos(j); N=1e4; M+= H*_; Z=D*K; F+=_*P; r=E*K; W=cos( O); m=K*W; H=K*T; O+=D*_*F/ K+d/K*E*_; B=
sin(j); a=B*T*D-E*W; XClearWindow(e,z); t=T*E+ D*B*W; j+=d*_*D-_*F*E; P=W*E*B-T*D; for (o+=(I=D*W+E
*T*B,E*d/K *B+v+B/K*F*D)*_; p<y; ){ T=p[s]+i; E=c-p[w]; D=n[p]-L; K=D*m-B*T-H*E; if(p [n]+w[ p]+p[s
]== 0|K <fabs(W=T*r-I*E +D*P) |fabs(D=t *D+Z *T-a *E)> K)N=1e4; else{ q=W/K *4E2+2e2; C= 2E2+4e2/ K
 *D; N-1E4&& XDrawLine(e ,z,k,N ,U,q,C); N=q; U=C; } ++p; } L+=_* (X*t +P*M+m*l); T=X*X+ l*l+M *M;
  XDrawString(e,z,k ,20,380,f,17); D=v/l*15; i+=(B *l-M*r -X*Z)*_; for(; XPending(e); u *=CS!=N){
                                   XEvent z; XNextEvent(e ,&z);
                                       ++*((N=XLookupKeysym
                                         (&z.xkey,0))-IT?
                                         N-LT? UP-N?& E:&
                                         J:& u: &h); --*(
                                         DN -N? N-DT ?N==
                                         RT?&u: & W:&h:&J
                                          ); } m=15*F/l;
                                          c+=(I=M/ l,l*H
                                          +I*M+a*X)*_; H
                                          =A*r+v*X-F*l+(
                                          E=.1+X*4.9/l,t
                                          =T*m/32-I*T/24
                                           )/S; K=F*M+(
                                           h* 1e4/l-(T+
                                           E*5*T*E)/3e2
                                           )/S-X*d-B*A;
                                           a=2.63 /l*d;
                                           X+=( d*l-T/S
                                            *(.19*E +a
                                            *.64+J/1e3
                                            )-M* v +A*
                                            Z)*_; l +=
                                            K *_; W=d;
                                            sprintf(f,
                                            "%5d  %3d"
                                            "%7d",p =l
                                           /1.7,(C=9E3+
                                 O*57.3)%0550,(int)i); d+=T*(.45-14/l*
                                X-a*130-J* .14)*_/125e2+F*_*v; P=(T*(47
                                *I-m* 52+E*94 *D-t*.38+u*.21*E) /1e2+W*
                                179*v)/2312; select(p=0,0,0,0,&G); v-=(
                                 W*F-T*(.63*m-I*.086+m*E*19-D*25-.11*u
                                  )/107e2)*_; D=cos(o); E=sin(o); } }
```

Lesbarkeit, Verständlichkeit
Formatierungsstandards

Wenn man Entwickler
frei laufen lässt …

… oder wie man es
nicht machen sollte…

Was macht folgendes
C-Programm?



```
#include                                  <math.h>
#include                                <sys/time.h>
#include                                <X11/Xlib.h>
#include                               <X11/keysym.h>
                                       double L ,o ,P
                                      ,_=dt,T,Z,D=1,d,
                                      s[999],E,h= 8,I,
                                      J,K,w[999],M,m,O
                                     ,n[999],j=33e-3,i=
                                     1E3,r,t, u,v ,W,S=
                                     74.5,l=221,X=7.26,
                                     a,B,A=32.2,c, F,H;
                                     int N,q, C, y,p,U;
                                    Window z; char f[52]
                                  ; GC k; main(){ Display*e=
 XOpenDisplay( 0); z=RootWindow(e,0); for (XSetForeground(e,k=XCreateGC (e,z,0,0),BlackPixel(e,0))
; scanf("%lf%lf%lf",y +n,w+y, y+s)+1; y ++); XSelectInput(e,z= XCreateSimpleWindow(e,z,0,0,400,400,
0,0,WhitePixel(e,0) ),KeyPressMask); for(XMapWindow(e,z); ; T=sin(O)){ struct timeval G={ 0,dt*1e6}
; K= cos(j); N=1e4; M+= H*_; Z=D*K; F+=_*P; r=E*K; W=cos( O); m=K*W; H=K*T; O+=D*_*F/ K+d/K*E*_; B=
sin(j); a=B*T*D-E*W; XClearWindow(e,z); t=T*E+ D*B*W; j+=d*_*D-_*F*E; P=W*E*B-T*D; for (o+=(I=D*W+E
*T*B,E*d/K *B+v+B/K*F*D)*_; p<y; ){ T=p[s]+i; E=c-p[w]; D=n[p]-L; K=D*m-B*T-H*E; if(p [n]+w[ p]+p[s
]== 0|K <fabs(W=T*r-I*E +D*P) |fabs(D=t *D+Z *T-a *E)> K)N=1e4; else{ q=W/K *4E2+2e2; C= 2E2+4e2/ K
 *D; N-1E4&& XDrawLine(e ,z,k,N ,U,q,C); N=q; U=C; } ++p; } L+=_* (X*t +P*M+m*l); T=X*X+ l*l+M *M;
  XDrawString(e,z,k ,20,380,f,17); D=v/l*15; i+=(B *l-M*r -X*Z)*_; for(; XPending(e); u *=CS!=N){
                                  XEvent z; XNextEvent(e ,&z);
                                      ++*((N=XLookupKeysym
                                        (&z.xkey,0))-IT?
                                        N-LT? UP-N?& E:&
                                        J:& u: &h); --*(
                                        DN -N? N-DT ?N==
                                        RT?&u: & W:&h:&J
                                         ); } m=15*F/l;
                                         c+=(I=M/ l,l*H
                                         +I*M+a*X)*_; H
                                         =A*r+v*X-F*l+(
                                         E=.1+X*4.9/l,t
                                         =T*m/32-I*T/24
                                          )/S; K=F*M+(
                                          h* 1e4/l-(T+
                                          E*5*T*E)/3e2
                                          )/S-X*d-B*A;
                                          a=2.63 /l*d;
                                          X+=( d*l-T/S
                                           *(.19*E +a
                                           *.64+J/1e3
                                           )-M* v +A*
                                           Z)*_; l +=
                                           K *_; W=d;
                                           sprintf(f,
                                           "%5d  %3d"
                                           "%7d",p =l
                                          /1.7,(C=9E3+
                                O*57.3)%0550,(int)i); d+=T*(.45-14/l*
                               X-a*130-J* .14)*_/125e2+F*_*v; P=(T*(47
                               *I-m* 52+E*94 *D-t*.38+u*.21*E) /1e2+W*
                               179*v)/2312; select(p=0,0,0,0,&G); v-=(
                                W*F-T*(.63*m-I*.086+m*E*19-D*25-.11*u
                                 )/107e2)*_; D=cos(o); E=sin(o); } }
```

1)Das Prinzip der Verbalisierung

2)Das Prinzip der problemadäquaten Datentypen

3)Das Prinzip der Verfeinerung

4)Das Prinzip der strukturierten Programmierung

5)Das Prinzip des defensiven Programmierens

6)Das Prinzip der integrierten Dokumentation

Das Prinzip der Verbalisierung
Systematische Programmierung

Ziel:

Ideen und Konzepte des Programmierens im Programm möglichst deutlich, gut sichtbar zu machen und zu dokumentieren

Verbalisierung bedeutet:

Gedanken und Vorstellungen in Worten ausdrücken und sich (und insb. anderen) bewusst zu machen

Wörter in Programmen:

(1) Bezeichner von
- Variablen
- Konstanten
- Methodennamen/Prozeduren
- Modulnamen

(2) Kommentare
- Inline
- Javadoc etc.

(3) Schlüsselwörter der Programmiersprache

Das Prinzip der Verbalisierung
systematische Programmierung

Das Prinzip der Verbalisierung bringt folgende Vorteile:

- Die Lesbarkeit wird deutlich verbessert
- Das Programm wird änderungsfreundlicher
- Die Einarbeitung in fremde Programme wird erleichtert
- Die (Wieder-)Einarbeitung in eigene Programme wird erleichtert
- QS, Wartung & Pflege wird deutlich erleichtert

Für die Verständlichkeit eines Programmes ist insbesondere eine geeignete Bezeichnerwahl entscheidend.

Motivation Bezeichnerwahl
Prinzip der Verbalisierung

Was macht folgendes Programm?

```
public class Z {
  public static void main(String[] args) {
    double x;
    double z;
    int l;

    x =     Console.readDouble(„X:");
    z =     Console.readDouble(„Z:") / 100;
    l =     Console.readInt("L:");

    for (double y = z-0.01; y <= z+0.01; y += 0.00125) {
      double zM = y/12;
      double p = x * zM / (1 – (Math.pow(1/(1 + zM), l*12)));

      System.out.println(100 * y + " : " + p);
    }
  }
}
```

Beispiele für gute/schlechte Bezeichner
Prinzip der Verbalisierung

Beispiel 1

schlecht: problemfreie, technische Bezeichner

```
feld1, feld2, zaehler;
```

besser: problembezogene Bezeichner:

```
messreihe1, messreihe2, anzahlZeichen;
```

Beispiel 2

schlecht: unverständliche Konstanten, hart codiert

```
Rabatt = 20.0 + (2.5 * 4)
```

besser: sprechende Namenskonstanten

```
final double WSV = 20.0,
FRAUENBONUS = 2.5, KINDER = 4;
Rabatt = WSV + (FRAUENBONUS * KINDER);
```

Beispiele für gute/schlechte Bezeichnerwahl
Prinzip der Verbalisierung

Beispiel 3

schlecht: kryptische Bezeichner ohne Aussagekraft (zu kurz)

```
p = g + z * d
```

besser:

```
praemie = grundpraemie + zulage * dienstjahre
```

Beispiel 4

schlecht: sehr lange über-aussagekräftige Bezeichner

```
void berechneGesamtschuldenUndVerschickeMahnungAn(Nutzer n)
```

besser:

```
void berechneGesamtschulden(Nutzer n);
und/oder
void verschickeMahnungAn(Nutzer n);
```

Weist auf anderes Problem hin, Verstoß SRP

Regeln für geeignete Bezeichnerwahl
Prinzip der Verbalisierung

Regel 1:
Ein geeigneter Name sollte die semantisch funktionale Rolle des Bezeichners wiederspiegeln.

Bezeichnertyp	beschreibt [Rolle]	Beispiel
Variable	Inhalt	`nextState` `brake_coeffizient`
Methode	Aufgabe	`printPage()` `calculateDelay()`
Symbolische Konstante	Wert	`DEFAULT_SPEED` `MaxOpenWindow`
Grundtyp (atomar, Basistyp)	Gegenstand oder Begriff ➢ **Einfache** Namen	`File, Table, Reader`
Abgeleitet & Komponententyp	Gegenstand oder Begriff ➢ **Zusammengesetzte** Namen	`SequentialFile` `InputStreamReader`

Regeln für Bezeichnerwahl
Prinzip der Verbalisierung

Regel 2:
Ein geeigneter Name sollte einheitlicher Stil- und Namens-konvention folgen.

Konstanten:

- Großschreibung
- Zusammengesetzte Wörter mit Underscores erlaubt (in C)
- Standardpräfixe verwenden: `MIN_, MAX_, DEFAULT_, …`
- Bsp.: `NORTH, BLUE, LARGE, MAX_WIDTH, DEFAULT_SIZE`

Variablen/Attribute:

- Mit/Ohne führendem Underscore (in C/Java)
- Erster Buchstabe klein
- Bsp.: _available, _date,

Regeln für Bezeichnerwahl
Prinzip der Verbalisierung

Klassen:

- Substantiv, erster Buchstabe groß, Rest klein
- Ganze Worte, Zusammensetzung durch Großschreibung
- Bsp.: Account, StandardTemplate

Methoden/Variablen

- Beginnen klein
- camelCase-Schreibweise
- Unterscheidung nicht alleine durch CaseSensitivität
- Beginnen mit Verb (Imperativ), optional gefolgt von Substantiv

  `checkAvailability(), doMaintenance(), getDate()`
- Sonderbehandlung präfixe:

  `set/get<Attributname>, is/has<Attributvalue>`

[Siehe Java Code Conventions]

Regeln für Bezeichnerwahl
Prinzip der Verbalisierung

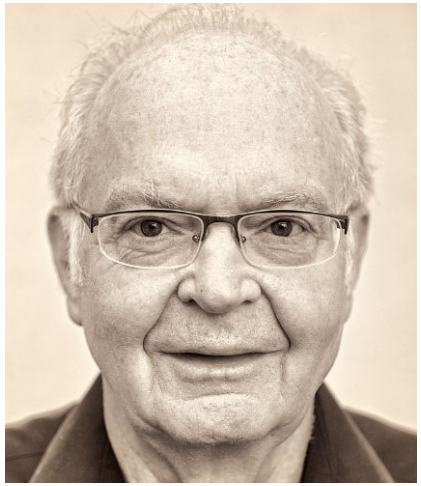
Regel 3:
Ein geeigneter Name sollte leicht zu merken sein.

- Nach Möglichkeit **natürlicher Sprache** entnommen
  - Vorzugsweise Englisch, kein Kauderwelsch
  - Keine kryptischen Bezeichner
  - Ausnahme:
    Schleifenvariablen oder ggfs. physikalische Größen in Formeln
- Nicht zu lang
  - Schlechte Lesbarkeit insbes. bei häufiger Verwendung
- Nicht zu kurz
  - Nur bei sehr kleinem Scope wie Schleifen-Iteration
  - Keine Abkürzungen

Als Faustregel:
- 8-20 Zeichen für Variablen
- 15-30 Zeichen für Prozeduren/Methoden

Motivation Kommentartechnik

Prinzip der Verbalisierung



(Aufnahme 2010)

"Let us change our traditional attitude to the construction of programs:

Instead of imagining that our main task

is to instruct a computer what to do, let us concentrate rather on

explaining to human beings what we want a computer to do."

(Donald Knuth, 1984)

Chat: Warum ist der obere Kommentar so schlecht?

Motivation Kommentartechnik
Prinzip der Verbalisierung

Beispiel (schlechter Kommentar durch Umgebung):

```
//calculates square root of given number
public void abc(int a) {
    r = a/2;
    while (abs(r - (a/r)) > t) {
        r = 0.5 * (r + (a/r));
    }
    System.out.println("r = " + r);
}
```

Beispiel (geeigneter Kommentar):

```
// uses Newton-Raphson method
public void squareRoot(int num){
    root = num/2;
    while (abs(root - (num/root)) > t) {
        r = 0.5 * (root + (num/root));
    }
    System.out.println("root = " + root);
}
```

- Kommentare tragen maßgeblich zur Lesbarkeit eines Programmes bei
- Daher macht es Sinn sich über eine systematische Kommentartechnik Gedanken zu machen

Regeln für Kommentare
Prinzip der Verbalisierung

Oberste Grundregel: guter Code dokumentiert sich selbst

- D.h. der beste Kommentar ist einer, der nicht geschrieben werden braucht.

Für alle anderen Fälle gilt:

1) Kommentare aus externer Sicht möglicher Leser verfassen
2) Metadaten in Kommentaren wenn möglich vermeiden (Autor, last-modified-date, …)
3) Kommentare liefern Informationen auf höherer Abstraktionsebene als Quellcode
   - nicht das Offensichtliche beschreiben
4) Verfügbare Dokumentationsgeneratoren verwenden
   - Javadoc, doxygen, …

Dokumentationsgeneratoren (Bsp. javadoc)
Prinzip der Verbalisierung

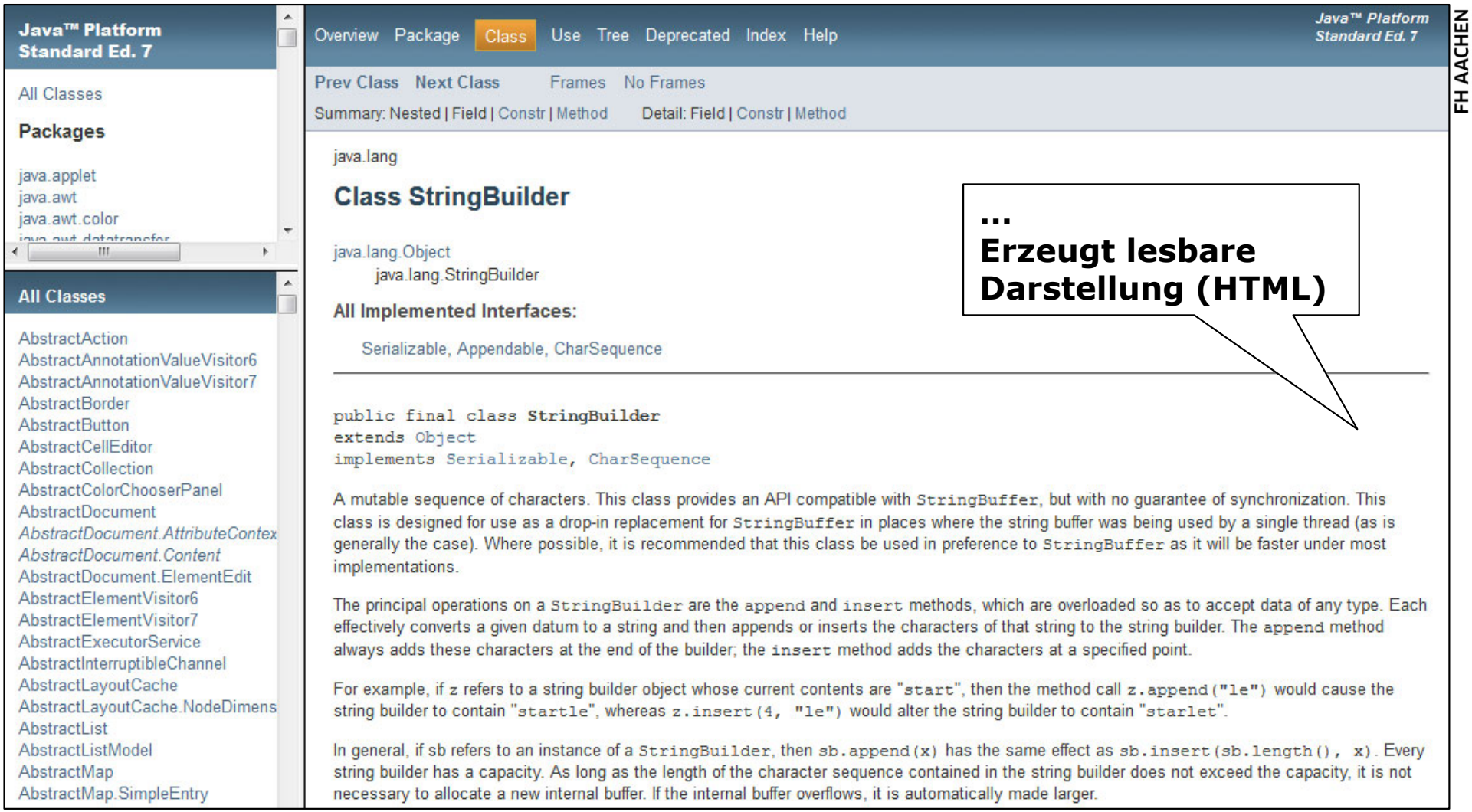
```
/*
 * Copyright (c) 2003, 2013, Oracle and/or its affiliates. All rights reserved.
 * ORACLE PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
 */
package java.lang;

/**
 * A mutable sequence of characters.  This class provides an API compatible with
 * {@code StringBuffer}, but with no guarantee of synchronization.
 * This class is designed for use as a drop-in replacement for * {@code StringBuffer}
 * in places where the string buffer was being used by a single thread (as is
 * generally the case). […]
 *
 * <p>The principal operations on a {@code StringBuilder} are the {@code append} and
 * {@code insert} methods, which are overloaded so as to accept data of any type.
 * Each effectively converts a given datum to a string and then appends or inserts
 * the characters of that string to the string builder.  […]

 * @author      Michael McCloskey
 * @see         java.lang.StringBuffer
 * @see         java.lang.String
 * @since       1.5
 */
public final class StringBuilder extends AbstractStringBuilder
          implements java.io.Serializable, CharSequence { […] }
```

Code mit Javadoc-Kommentaren …

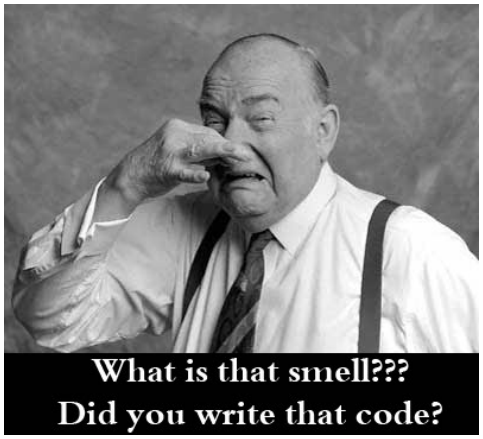
Dokumentationsgeneratoren
Prinzip der Verbalisierung



Clean Code & Code Smells
Entwickler-Richtlinien

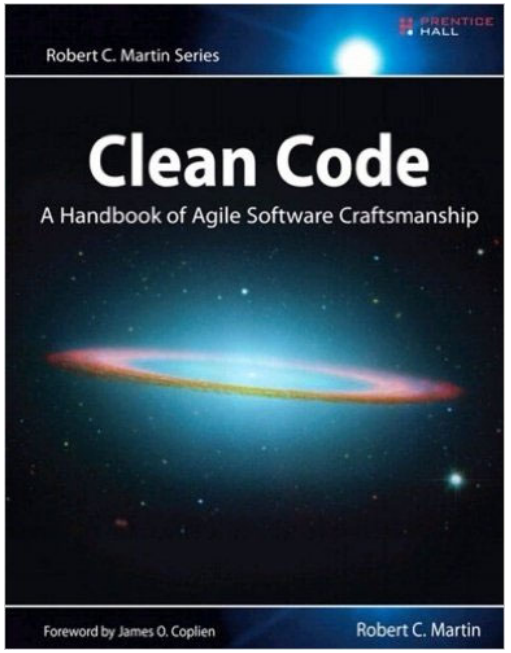
Bewegung der CleanCode-Developer

- Name in Anlehnung an Buch „Clean Code“ von Uncle Bob
- Listet gute/schlechte Programmier-Praktiken auf
  - ➢ Prägt Begriff „Code-Smell“



[Roland Golla]

- Entwickler, die sich auf gemeinsame Standards für saubere, professionelle Code-Entwicklung geeinigt haben
- qualitativ hochwertige Software als Berufung & Tugend
- Kontinuierliche Verbesserung
- Siehe: [Clean-Code-Developer]
- Siehe: [Clean-Code-Handbook]



Übersicht Aspekte des Clean Code

Entwickler-Richtlinien

Wir können im Rahmen der Vorlesung nur auf einzelne Aspekte eingehen…
Es bietet sich an dies im Selbststudium zu vertiefen

FH AACHEN
UNIVERSITY OF APPLIED SCIENCES

•**Kommentare**
•C1: Ungeeignete Informationen
•C2: **Überholte Kommentare**
•C3: Redundante Kommentare
•C4: Schlecht geschriebene Kommentare
•C5: **Auskommentierter Code**

•**Umgebung**
•E1: Ein Build erfordert mehr als einen Schritt
•E2: Tests erfordern mehr als einen Schritt

•**Funktionen**
•F1: **Zu viele Argumente**
•F2: Output-Argumente
•F3: **Flag Argumente**
•F4: Tote Funktionen

Allgemein:
•G1: Mehrere Sprachen in einer Quelldatei
•G2: Offensichtliches Verhalten ist nicht implementiert
•G3: Falsches Verhalten an den Grenzen
•G4: Übergangene Sicherungen
•G5: **Duplizierung**
•G6: Auf der falschen Abstarktionsebene codieren
•G7: Basisklasse hängt von abgeleiteten Klassen ab
•G8: Zu viele Informationen
•G9: **Toter Code**
•G10: Vertikale Trennung
•G11: Inkonsistenz
•G12: Müll
•G13: Künstliche Kopplung
•G14: Funktionsneid
•G15: Selektor-Argumente
•G16: Verdeckte Absicht
•G17: Falsche Zuständigkeit
•G18: Fälschlich als statisch deklarierte Methoden
•G19: Aussagekräftige Variablen verwenden
•G20: Funktionsname soll die Aktion ausdrücken
•G21: Den Algorithmus verstehen
•G22: Logische Abhängigkeiten in physische Umwandeln
•G23: Polymorphismus statt If/Else oder Switch/Cas verwenden
•G24: Konventionen beachten
•G25: Magische Zahlen durch bekannte Konstanten ersetzen
•G26: Präzise sein
•G27: Struktur ist wichtiger als Konvention
•G28: Bedingungen einkapseln
•G29: Negative Bedingungen vermeiden
•G30**: Eine Aufgabe pro Funktion!**
•G31: Verborgene zeitliche Kopplungen
•G32: Keine Willkür
•G33: Grenzbedingungen einkapseln
•G44: In Funktionen nur eine Abstraktionsebene tiefer gehen
•G45: Konfigurierbare Daten hoch ansiedeln
•G46: Transitive Navigation vermeiden

Namensgebung
•N1: Deskriptive Namen wählen
•N2: Namen sollten der Abstraktions-ebene entsprechen
•N3: Möglichst die Standardnormen Kultur verwenden
•**N4: Eindeutige Namen**
•N5: Lange Namen für große Geltungs-bereichen
•N6: Codierungen vermeiden
•N7: Namen sollten Nebeneffekte beschreiben

Tests:
T1: Unzureichende Tests
T2: Ein Coverage-Tool verwenden
T3: Triviale Tests nicht überspringen
T4: Ein ignorierter Test zeigt eine Zweideutigkeit auf
T5: Grenzbedingungen testen
T6: Bei Bugs die Nachbarschaft gründlich testen
T7: Das Muster des Scheiterns zur Diagnose nutzen
T8: Hinweise auf Coverage-Patterns
T9: Tests sollten schnell sein

CodeSmell „Redundante Kommentare“
Prinzip der Verbalisierung

Ein Kommentar gilt als redundant, wenn er zu der Selbst-beschreibung eines Objektes keine neuen Informationen hinzufügt.

- Redundante Kommentare sollten vermieden werden
- Kommentare sollten Dinge beschreiben, die der Code nicht selbst ausdrücken kann.

Beispiel1:

```
i++;                    // inkrementiert i um eins
jahr += 1;              // Fahre beim nächsten Jahr fort
                        // Restauriert die Schleifeninvariante.
```

Beispiel2:

```
/**
 * Starts the sale of a given item, defined by its Request
 * @param sellRequest
 * @return
 * @throws ManagedComponentException
*/
private SellResponse beginSellItem(SellRequest sellRequest)
          throws ManagedComponentException {…}
```

CodeSmell „Kommentarbanner“
Prinzip der Verbalisierung

Der Einsatz von **Kommentarbannern** zur Strukturierung sollte generell vermieden werden.

- Verdecken das Wesentliche
- Erzwingt vertikales Scrollen durch Code (➔Lesbarkeit)
- Oft offensichtliche Informationen, nicht wirklich relevant

- Für objektorientierte Sprachen existieren i.d.R. bessere Ansätze
  - ➢ z.B. Ausgliedern von Programm-teilen in separate Klassen

- Ggfs. Bei Skriptsprachen sinnvoll

```
/*****************************************/
/* Längere Ausführung zu diesem Quelltext */
/*****************************************/
```

```
/*****************************************/
/* statische Variablen                   */
/*****************************************/
…
/*****************************************/
/* Instanzvariablen                      */
/*****************************************/
…
/*****************************************/
/* statische Methoden                    */
/*****************************************/
…
/*****************************************/
/* InstanzMethoden                      */
/*****************************************/
…
```

Übersicht der Prinzipien
systematische Programmierung

1) Das Prinzip der Verbalisierung

2) Das Prinzip der problemadäquaten Datentypen

3) Das Prinzip der Verfeinerung

4) Das Prinzip der strukturierten Programmierung

5) Das Prinzip des defensiven Programmierens

6) Das Prinzip der integrierten Dokumentation

Vorgehensweise
Prinzip der Problemadäquaten Datentypen

1) bestehende (Basis-)Datentypen wiederverwenden

- **In Java**: bestehende Klassen, ggfs. primitive Datentypen

2) Wertebereich festlegen und einschränken

- Zulässige Werte sollten aus dem Namen des Datentypen ersichtlich werden
- **In Java:** Aufzählungstypen (enums) anstelle von int, boolean, String, ...

Wenn 1) nicht möglich:

3) Benutzerdefinierten Datentyp verwenden

- **In Java:** Neue Klasse erzeugen

Beispiel Typkonstruktor „Verbund“
Prinzip der Problemadäquaten Datentypen

Die komplexe Zahl als problemadäquater Datentyp:

```
public class ComplexNumber {

  private double real;
  private double imaginary;

  ComplexNumber (double real, double imaginary) { ... }

  public void setReal(double real) { ... }

  public double getReal() { ... }

  ...
}
```

Beispiele
Prinzip der Problemadäquaten Datentypen

Beispiele problemadäquater Datentypen:

```
enum Familienstand (LEDIG, VERHEIRATET, VERWITWET, GESCHIEDEN)
enum Ampelfarbe (ROT, GELB, GRÜN)
enum Bauteil (R, L, C, U, I)
enum Geschlecht (MÄNNLICH, WEIBLICH, X)
```

- `enum` ist existierender Basisdatentyp
- Die möglichen Werte
  - sind problemspezifisch
  - schränken den Datentypen ein
  - sind aus dem Namen des Datentypen ableitbar

Beispiel „Aufzählungstypen“
Prinzip der Problemadäquaten Datentypen

Modellieren eines Ventilzustands mit drei Zuständen: *zu*, *auf*, *halbauf*

Ohne ENUM:

```
class Ventilzustand {

  static final int AUF = 0;
  static final int HALBAUF = 1;
  static final int ZU = 2;
  int zustand;

  void dreheAuf () {

    switch (zustand) {
      case(AUF)     : zustand = AUF;     break;
      case(HALBAUF) : zustand = AUF;     break;
      case(ZU)      : zustand = HALBAUF; break;
      default       :
          throw new IllegalStateException ();
    }
  }

  ...
}
```

Mit ENUM

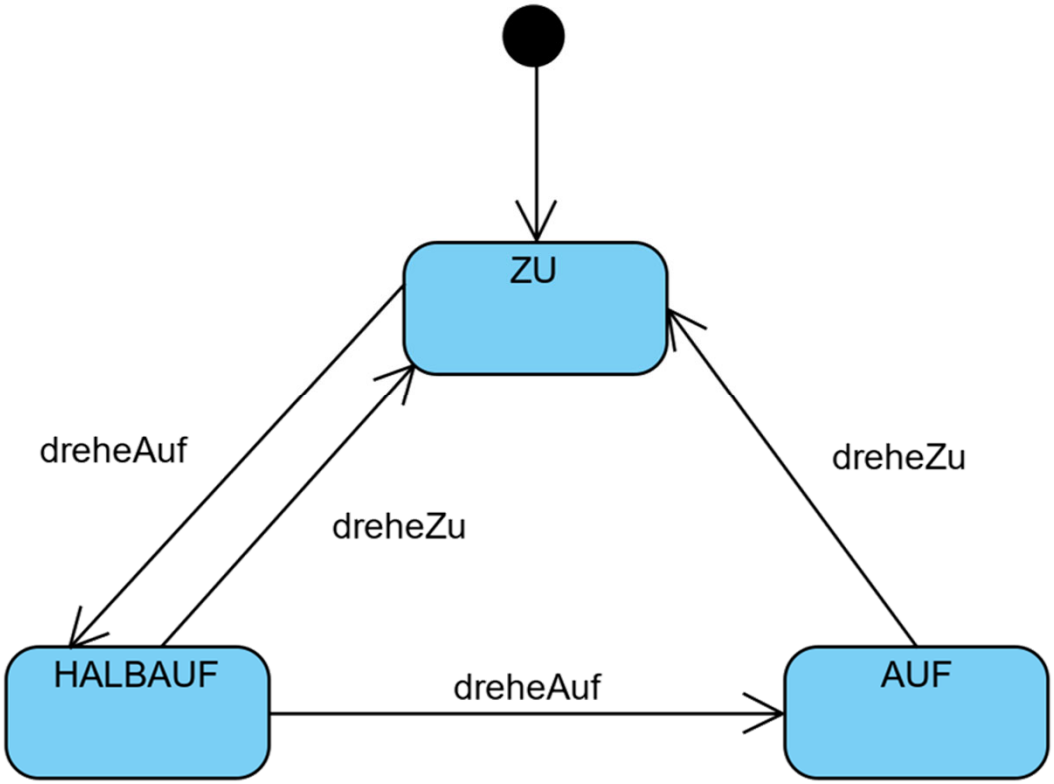
```
enum Ventilzustand {

  AUF {
    Ventilzustand dreheAuf() {
      return AUF;
    }
    Ventilzustand dreheZu() {
      return ZU;
    }
  },
  HALBAUF {...},
  ZU {...}

  abstract Ventilzustand dreheAuf();
  abstract Ventilzustand drehezu();
}
```

Beispiel „Ventilzustand“ im Automatendiagramm
Prinzip der Problemadäquaten Datentypen

Modellieren eines Ventilzustands mit drei Zuständen: *zu*, *auf*, *halbauf*



Mit ENUM

```
enum Ventilzustand {

  AUF {
    Ventilzustand dreheAuf() {
      return AUF;
    }
    Ventilzustand dreheZu() {
      return ZU;
    }
  },
  HALBAUF {...},
  ZU {...}

  abstract Ventilzustand dreheAuf();
  abstract Ventilzustand drehezu();
}
```

Zusammenfassung
Prinzip der Problemadäquaten Datentypen

- Wiederverwendung Standard-Klassen
- Wertebereiche weder über- noch unterspezifiziert,
  da 1:1 Abbildung auf fachliche Problemstellung
- Macht statische Typprüfung durch Compiler möglich
- Erhöht Verständlichkeit
- Erhöht Lesbarkeit
- Erhöht Wartbarkeit