Software Engineering Praxis[1]

(Informationsverarbeitung für Aktuare)

Prof. Dr. Franz Schweiggert
und **Dr. Johannes Mayer**

Fakultät Mathematik u. Wirtschaftswissenschaften
Abteilung Angewandte Informationsverarbeitung (*SAI*)

Vorlesungs- und Übungsbegleiter (SS 2006)

Umfang: 4/2 = 6 LVS

Studiengang	LP
WiMa	9
WiWi	12
Informatik	8

Vorbemerkungen

Diese Vorlesung entstand als „Informationsverarbeitung für Aktuare“, um Studenten unserer Fakultät die Möglichkeit zu geben, „zwei Fliegen mit einer Klappe zu schlagen“:

- eine / die Informatikprüfung (samt Schein) im Rahmen des Studiums und gleichzeitig
- eine Prüfung zum Aktuar gemäß den Anforderungen der Deutschen Aktuarvereinigung (DAV) ablegen zu können.

Dazu sind eine Reihe von Vorgaben der DAV einzuhalten!
Gemäß der Devise unserer Abteilung – **Übungen am und mit dem Rechner** – können daher die Übungen inhaltlich nicht vorlesungsbegleitend laufen, sondern bringen teilweise eigenen Stoff, der notwendig ist, um im Team ein kleines Projekt abwickeln zu können.

Dies schlägt sich auch im Vorlesungsbegleiter nieder: Um Stoff für die Übungen zu bekommen, vor allem aber für das zu bearbeitende Projekt, wird der übliche Aufbau einer Software Engineering Vorlesung bewusst durchbrochen:

- **Statt von generellen Konzepten ins Detail zu gehen, werden einige technische Details vorgezogen!**
- **Das Skript soll kein Lehrbuch sein, sondern ist als Vorlesungs*begleiter* gedacht, also als Ergänzung und nicht als Ersatz der Vorlesung. Eine Vorlesung ist ein lebendiger Dialog mit dem Stoff. Deshalb wird es immer wieder zu Abweichungen zwischen Skript und Inhalt der einzelnen Stunden kommen!**

Wenn von Software die Rede ist, so nicht von technischer (hardware- / systemnaher) Software im Sinne von *embedded systems*. Software Engineering **Praxis** meint, dass die Vorlesung mehr am Stand der Technik denn am *state-of-the-art* orientiert ist. Auch können nur ausgewählte Aspekte behandelt werden, gibt es doch zu diesem Thema mehrbändige Lehrbücher mit insgesamt über tausend Seiten!!!

Inhaltsverzeichnis

Teil I

Praxis: Technische Grundlagen

Kapitel 1

Eine Einführung in Java

Im Rahmen der Übungen wird die Programmiersprache Java verwendet, um Sofware Engineering in der Praxis zu üben. Da diese Programmiersprache den meisten Hörern dieser Vorlesung noch nicht bekannt sein dürfte, findet zunächst eine kleine Einführung in diese Programmiersprache statt. Dabei gehen wir davon aus, dass Sie bereits mit einer Programmiersprache wie etwa C, Oberon oder Modula-2 Erfahrungen gesammelt und die Grundkonzepte verstanden haben. Die nun folgende Einführung in Java setzt diese Kenntnisse voraus und vermittelt auch nur so viel von der Sprache Java, wie für die Übungen notwendig ist. Es gäbe sicher noch eine Reihe interessanter „Features“, die den Interessierten aber zum Selbststudium überlassen werden.

1.1 Erste Schritte

1.1.1 Unser erstes Java-Programm

Wie in jeder Programmiersprache beginnen wir auch in Java mit dem obligatorischen Hallo-Welt-Beispiel:

Programm 1.1: Hallo-Welt-Beispiel (*Hello.java*)

```
1 public class Hello {
2     public static void main(String [] args) {
3         System.out.println("Hallo zusammen!");
4     }
5 }
```

Anmerkungen:

- Java ist eine *objektorientierte Programmiersprache*. Deshalb ist alles in sog. *Klassen* – die Module der Objektorientierung – „verpackt“. Selbst für unser erstes kleines Beispiel ist schon eine eigene Klasse nötig.
- Um ein Java-Programm starten zu können, ist analog zu C und C++ eine *main-Methode* – in C und C++ gibt es eine *main-Funktion* – notwendig, die als Parameter ein Array mit den Kommandozeilen-Argumenten erhält.

- Die Datei muss genauso heißen wie die enthaltene *public-Klasse* – in unserem Fall *Hello.java* und *Hello*.
- *System.out* steht für ein sog. *Objekt*, welches die *Standardausgabe* repräsentiert. Von diesem Objekt wird die *Methode* (d. h. Funktion) *println* () aufgerufen, um eine Ausgabe auf die Standardausgabe zu machen.

1.1.2 Vom Programm zur Ausführung

Wir wollen unser erstes Beispielprogramm nun natürlich ausführen. Dazu gehen wir wie folgt vor:

```
chomsky$ ls Hello.*
Hello.java
chomsky$ javac Hello.java
chomsky$ ls Hello.*
Hello.class  Hello.java
chomsky$ java Hello
Hallo zusammen!
chomsky$
```

Anmerkungen:

- Java-Programme müssen *kompiliert* werden. Dies geschieht durch den Aufruf von *javac*. Dieser übersetzt das Java-Programm in *maschinenunabhängigen Bytecode* – deshalb ist Java (im Idealfall ;-) auch *plattformunabhängig*. Für die Klasse *Hello* wird dabei die Datei *Hello.class* erzeugt.
- Der erzeugte *Bytecode* kann, aufgrund seiner *Maschinenunabhängigkeit*, nun natürlich nicht einfach ausgeführt werden. Zur Ausführung wird die *Java Virtual Machine* (oder kurz *JVM*) benötigt. Für die JVM gibt es von unterschiedlichen Herstellern *Implementierungen für alle möglichen Betriebssysteme* wie etwa Solaris, Linux, Windows, etc. Hinter dem Kommando *java* verbirgt sich die JVM. Dabei muss der *Name der Klasse* als Kommandozeilen-Argument übergeben werden, die die *main-Methode* enthält. Eventuell enthält die JVM einen *Just-In-Time-Compiler* (oder kurz *JIT*), der *Java-Bytecode zur Laufzeit in Maschinencode übersetzen* kann, um die mehrfache Ausführung zu beschleunigen – die JVM ist ja „nur“ ein *Interpreter* für Java-Bytecode.

Abbildung 1.1 veranschaulicht den Weg vom Java-Quellcode bis zu seiner Ausführung.

1.1.3 Typische „Anfängerfehler“

Fehler können sich beispielsweise bei der Ausführung eines Java-Programms einschleichen:

```
chomsky$ java Hello.class
Exception in thread "main" java.lang.NoClassDefFoundError: Hello/class
chomsky$ java Helo
Exception in thread "main" java.lang.NoClassDefFoundError: Helo
chomsky$
```

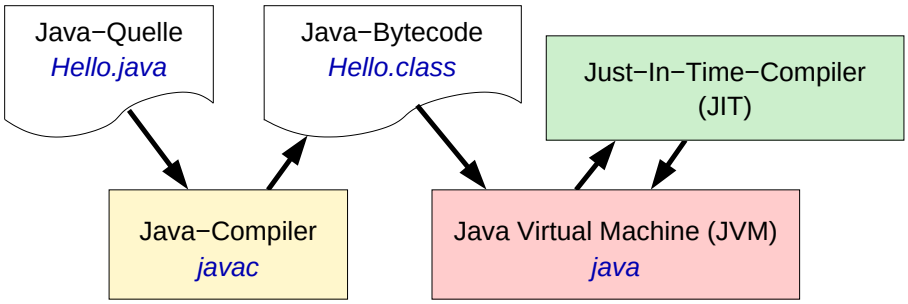


Abbildung 1.1: Kompilation und Ausführung von Java-Programmen

Im ersten Fall wurde nicht der *Klassenname* Hello, sondern der *Name der Klassendatei* Hello.class angegeben. In diesem Fall wird der Dateiname als Klassenname interpretiert. Eine entsprechende Klasse gibt es nicht und so kommt es zum *NoClassDefFoundError*.

Beim Ausführen von Java-Programmen mit java muss man den Klassennamen und nicht den Namen der Klassendatei angeben (also keine Endung '.class').

Im zweiten Fall liegt einfach ein Tippfehler vor. Eine Klasse namens Helo gibt es nicht und so kommt es wie eben zum *NoClassDefFoundError*.

Ein anderer typischer Fehler kann bei der Implementierung passieren:

Programm 1.2: Klasse ohne main-Methode (*NoMain.java*)

```
1 public class NoMain {
2     public int add(int x, int y) {
3         return x + y;
4     }
5 }
```

```
chomsky$ javac NoMain.java
chomsky$ java NoMain
Exception in thread "main" java.lang.NoSuchMethodError: main
chomsky$
```

Will man ein Java-Programm starten, so wird die *main-Methode* der Klasse aufgerufen, die bei java als Kommandozeilen-Parameter angegeben ist. Besitzt diese Klasse keine Methode mit der Signatur

```
public static void main(String [] args)
```

dann führt dies – wie im vorliegenden Fall – zu einem *NoSuchMethodError*, bei dem auch der Name der Methode angegeben ist, die nicht gefunden wurde (in diesem Fall „main“).

1.2 Elemente eines Java-Programms

1.2.1 Grobstruktur

Ein Java-Programm, d. h. eine Datei mit der *Endung „.java“*, besteht aus drei Elementen:

- Zunächst kann eine *Paketangabe* stehen.
  Beispiel: **package** *my.owm.pkg*;
  Wir werden später über Pakete sprechen. Zunächst genügt es, zu wissen, dass Java-Programme, d. h., Java-Klassen, in Paketen organisiert werden können.
- Danach können ein oder mehrere *Import-Anweisungen* folgen.
  Beispiel: **import** *java . util . Vector ;*
  Damit können Klassen aus anderen Paketen zur Verwendung importiert werden. (Bei *java.lang* ist das nicht notwendig. Deshalb auch kein Import bei der Verwendung der Klasse java.lang.System.)
- Draufhin folgen ein oder mehrere *Klassendefinitionen*. Maximal eine dieser Klassen darf *public* sein – was das bedeutet werden wir später erörtern.

Mit den genannten Elementen könnte eine Java-Datei also in etwa wie folgt aussehen:

Programm 1.3: Beispiel einer Java-Datei (*MyClass.java*)

```
1 package my.own.pkg;
2
3 import java . util . Vector ;
4
5 public class MyClass {
6     public static void main( String [] args ) {
7         // ...
8     }
9 }
```

1.2.2 Zeichensatz

Java-Programme basieren auf dem *Unicode-Zeichensatz*. Anders als der 7-Bit-ASCII-Zeichensatz und der 8-Bit-ISO-Latin-1-Zeichensatz, der die Zeichen der wichtigsten westeuropäischen Sprachen enthält, benötigt der Unicode-Zeichensatz 16 Bit und enthält praktisch alle möglichen Schriftzeichen – weltweit! *Unicode-Zeichen* werden normalerweise mittels *UTF-8 byteweise codiert*. Hierbei sind sowohl *ASCII*, als auch *Latin-1 gültige UTF-8 Codierungen*, so dass Java-Programme auch mit einem „normalen“ Editor geschrieben werden können. Unicode-Zeichen können dann mittels `\uxxxx` eingefügt werden, wobei xxxx für den Hexadezimal-Code eines Unicode-Zeichens steht.

1.2.3 Bezeichner

Ein *Java-Bezeichner* muss mit einem Buchstaben, Unterstrich oder Währungssymbol beginnen gefolgt von Buchstaben, Ziffern, Unterstrichen und Währungssymbolen.

Bezeichner mit Währungssymbolen sind aber per Konvention *nur für interne Bezeichner* (die durch Compiler bzw. Präprozessor erzeugt werden) gedacht.

Ein Bezeichner kann in Java Buchstaben und Ziffern anderer Sprachen enthalten! Auch als Währungssymbol gibt es nicht nur „$“! Auch π ist beispielsweise ein Buchstabe im Unicode-Zeichensatz (`\u03c0`). Man kann in Java auch Umlaute für Bezeichner verwenden. Englische Bezeichner sind jedoch eine gute Konvention im Hinblick auf die Lesbarkeit und Wiederverwendbarkeit von Quellcode!

Die folgenden Schlüsselworte sind reserviert und dürfen nicht für eigene Bezeichner verwendet werden:

abstract	**assert**	**boolean**	**break**	**byte**	**case**	**catch**
char	**class**	**const**	**continue**	**default**	**do**	**double**
else	**extends**	**false**	**final**	**finally**	**float**	**for**
goto	**if**	**implements**	**imports**	**instanceof**	**int**	**interface**
long	**native**	**new**	**null**	**package**	**private**	**protected**
public	**return**	**short**	**static**	**strictfp**	**super**	**switch**
synchronized	**this**	**throw**	**throws**	**transient**	**true**	**try**
void	**volatile**	**while**				

Seit Java 1.4 ist **assert** ein reservierter Bezeichner. Die beiden reservierten Bezeichner *const* und *goto* werden gegenwärtig nicht von Java verwendet.

1.2.4 Kommentare

In Java gibt es drei Arten von Kommentaren:

- *Einzeilige Kommentare* beginnen mit `//` und gehen bis zum Ende der Zeile.
  Beispiel: *// mein Kommentar*
- *Mehrzeilige Kommentare* beginnen mit `/*` und gehen bis `*/` – sie können auch nur einzeilig sein.
  Beispiel: */* mein Kommentar */*
  Per Konvention beginnt man jede Zeile in einem solchen Kommentar mit dem Zeichen „*“.
  Beispiel:

```
/* Ein etwas ...
 * ... längerer Kommentar.
 */
```

- Von den mehrzeiligen Kommentaren gibt es auch eine spezielle Version – die sogenannen *Dokumentations-Kommentare* (*doc comments*). Diese beginnen mit `/**` und enden mit `*/`. Auf der Basis dieser Kommentare kann man mit dem Tool *javadoc* automatisch eine Dokumentation generieren – mehr dazu später.

  Beispiel:

```
/**
 * Dies ist nun ein Dokumentations-Kommentar.
 */
```

Kommentare können nicht ineinander geschachtelt werden!

1.3 Datentypen

Java bietet zwei verschiedene Arten von Typen an: *primitive Datentypen* und *Referenztypen*. Auf beide wird im Folgenden näher eingegangen.

1.3.1 Primitive Datentypen

Unter *primitiven Datentypen* sind die grundlegenden Typen zur Repräsentation von *ganzen Zahlen*, *„reellen" Zahlen*, *Booleschen Werten* und *Zeichen* subsumiert.

Java besitzt die folgenden primitiven Datentypen:

Typ	Inhalt	Größe	Standardwert	Wertebereich
byte	ganze Zahl (mit Vorzeichen)	8 Bit	0	-2^7 bis 2^7-1
short	ganze Zahl (mit Vorzeichen)	16 Bit	0	-2^{15} bis $2^{15}-1$
int	ganze Zahl (mit Vorzeichen)	32 Bit	0	-2^{31} bis $2^{31}-1$
long	ganze Zahl (mit Vorzeichen)	64 Bit	0	-2^{63} bis $2^{63}-1$
float	IEEE-754-Gleitkommazahl	32 Bit	0.0	$\pm 1.4 \cdot 10^{-45}$ bis $\pm 3.40\ldots \cdot 10^{38}$
double	IEEE-754-Gleitkommazahl	64 Bit	0.0	$\pm 4.9 \cdot 10^{-324}$ bis $\pm 1.79\ldots \cdot 10^{308}$
boolean	Boolescher Wert	8 Bit	*false*	*true* und *false*
char	Unicode-Zeichen	16 Bit	`\u0000`	`\u0000` bis `\uffff`

Boolesche Werte können in keinen anderen primitiven Datentyp konvertiert werden und umgekehrt. Integer können also nicht als Boolesche Werte verwendet werden – wie in C. Dies bedeutet für C-Programmierer beispielsweise, dass Sie in Zukunft statt **if** (*i*) ... wieder **if** (*i* != 0) ... schreiben müssen.

Alle anderen primitiven Datentypen können *implizit oder explizit ineinander konvertiert* werden (siehe Abbildung 1.2).

Von **byte** und **short** ist jedoch – entgegen der Abbildung – nur eine explizite Konvertierung nach **char** möglich.

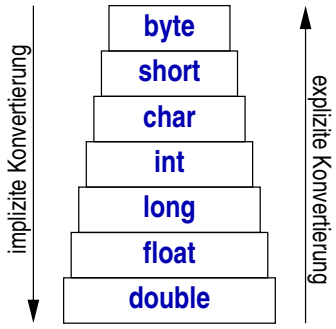


Abbildung 1.2: Implizite und explizite Typkonvertierung

Bei *expliziten Konvertierungen* geht i. A. *Information verloren*. Dies ist bei der impliziten Konvertierung normalerweise nicht der Fall, außer bei der Konvertierung von **int** nach **float** bzw. der Konvertierung von **long** nach **float** und **double**.

Beispiel:

```
int i = 3;
byte b = (byte) i;        // explizite Konvertierung
long l = i;               // implizite Konvertierung
float f = l;              // implizite Konvertierung (evtl. mit Verlust)
double d = 3.1e-10;
boolean b = true;         // Boolescher Wert
char c = '\t';            // Tabulator-Zeichen
```

1.3.2 Referenztypen

Referenzen sind im Prinzip nichts anderes als *typsichere Zeiger*, die nicht dereferenziert werden müssen (und auch nicht dereferenziert werden können). Anders als in C kann man nicht an die Zeigeradresse kommen oder Objekte beliebigen Typs zuweisen.

Referenzen verweisen entweder auf ein Objekt oder haben den Wert **null**, der anzeigt, dass die Referenz auf nichts verweist. (**null** ist der *Standardwert* einer Referenz.)

Referenzen können – je nach Typ – auf Objekte (d. h. Instanzen einer Klasse) oder Arrays[1] verweisen – später mehr dazu.

1.4 Operatoren

Java kennt die folgenden Operatoren:

[1] Auch Arrays sind Objekte in Java.

Priorität	Assoziativität	Operator	Erklärung
15	L	`.`	Elementzugriff bei Objekten
		`[]`	Elementzugriff bei Arrays
		`(Args)`	Methodenaufruf
		`++, --`	Postinkrement/-dekrement
14	R	`++, --`	Präinkrement/-dekrement
		`+, -`	unäres + bzw. -
		`~`	bitweises Komplement
		`!`	logische Negation
13	R	`new`	Objekterzeugung
		`(Typ)`	Typkonvertierung
12	L	`*, /, %`	Multiplikation, Division, „Modulo“
11	L	`+, -`	Addition, Subtraktion
		`+`	Stringkonkatenation
10	L	`<<`	bitweiser Linksshift
		`>>`	bitweiser Rechtsshift (Vorzeichenerw.)
		`>>>`	bitweiser Rechtsshift (Nullerw.)
9	L	`<, <=, >, >=`	Vergleich zweier Zahlen
		`instanceof`	Typvergleich
8	L	`==, !=`	Vergleich (prim. Typen und Referenzen)
7	L	`&`	bitweises und logisches Und
6	L	`^`	bitweises und logisches exklusives Oder (XOR)
5	L	`|`	bitweises und logisches Oder
4	L	`&&`	logisches Und
3	L	`||`	logisches Oder
2	R	`?:`	Auswahloperator
1	R	`=`	Zuweisung
1	R	`*=, /=, %=, ...`	Zuweisung mit Operation

Das folgende kleine Programm illustriert die Verwendung einiger wesentlicher Operatoren.

Programm 1.4: Verwendung einiger Operatoren (*Ops.java*)

```
1  public class Ops {
2      public static void printbin (int wert, int bits) {
3          if (bits > 1)
4              printbin (wert >>> 1, bits - 1);
5          System.out.print (wert & 1);
6      }
7      public static void main(String [] args) {
8          // bitweise Operatoren
9          byte a = 53, b = 47;
10         System.out.print ("a = "); printbin (a, 8); System.out.println ();
11         System.out.print ("b = "); printbin (b, 8); System.out.println ();
12         System.out.print ("~a = "); printbin (~a, 8); System.out.println ();
13         System.out.print ("a & b = "); printbin (a & b, 8); System.out.println ();
14         System.out.print ("a ^ b = "); printbin (a ^ b, 8); System.out.println ();
15         System.out.print ("a | b = "); printbin (a | b, 8); System.out.println ();
16         System.out.print ("a << 1 = "); printbin (a << 1, 8); System.out.println ();
17         System.out.print ("a >> 1 = "); printbin (a >> 1, 8); System.out.println ();
18
19         // Auswahloperator, Stringkonkatenation und Vergleich
20         System.out.println ("Ist a < b? "+(a < b ? "ja" : "nein"));
21
```

```
22          // Prae- und Postinkrement
23          System.out.println("a = " + a);
24          System.out.println("++a = " + (++a));
25          System.out.println("a = " + a);
26          System.out.println("a++ = " + (a++));
27          System.out.println("a = " + a);
28
29          // Zuweisung mit Operator
30          System.out.println("b = " + b);
31          b += 2;                          // aequivalent zu b = b + 2;
32          System.out.println("b = " + b);
33      }
34  }
```

```
chomsky$ java Ops
a =      00110101
b =      00101111
~a =     11001010
a & b =  00100101
a ^ b =  00011010
a | b =  00111111
a << 1 = 01101010
a >> 1 = 00011010
Ist a < b? nein
a =   53
++a = 54
a =   54
a++ = 54
a =   55
b =   47
b =   49
chomsky$
```

1.5 Kontrollstrukturen

In diesem Abschnitt werden kurz die Kontrollstrukturen von Java vorgestellt.

1.5.1 if-Anweisung

```
ifAnweisung ::= "if" "(" bedingung ")" anweisung
                [ "else" anweisung ]
```

Eine *Anweisung* kann ein arithmetischer Ausdruck, ein Funktionsaufruf etc. oder ein Anweisungsblock sein (Folge von Anweisungen in geschweiften Klammern).

Kann der *else-Teil* zu mehreren if-Anweisungen gehören, so gehört er immer zur *unmittelbar vorhergehenden if-Anweisung*. Dies wird auch in folgendem Beispiel deutlich.

Programm 1.5: Beispiel für die if-Anweisung (*If.java*)

```
1  public class If {
2      public static void main(String[] args) {
3          int x = 3;
4
5          if (x > 2)
6              if (x > 5)
7                  System.out.println("x > 2 und x > 5");
8          else // else gehoert zum naechsten if! => Klammerung
9              System.out.println("x <= 2"); // FALSCH!
10
11         // SO?
12         if (x > 2) {
13             if (x > 5)
14                 System.out.println("x > 2 und x > 5");
15         }
16         else
17             System.out.println("x <= 2"); // JETZT KORREKT!
18
19         // ... ODER SO?
20         if (x > 2) {
21             if (x > 5)
22                 System.out.println("x > 2 und x > 5");
23             else // else gehoert zum naechsten if!
24                 System.out.println("x > 2 und x <= 5"); // JETZT KORREKT!
25         }
26     }
27 }
```

1.5.2 switch-Anweisung

```
switchAnweisung ::= "switch" "(" ausdruck ")" "{"
                        ( ( "case" label | "default" ) ":" anweisung+ )+
                    "}"
```

Mit der switch-Anweisung kann man auf einfache Art *Fallunterscheidungen* umsetzen.

Programm 1.6: Beispiel für die switch-Anweisung (*Switch.java*)

```
1  public class Switch {
2      public static void main(String[] args) {
3          char c = 'x';
4          switch (c) {
5              case 'j' : System.out.println("ja");
6                         break; // Ausstieg aus switch (sonst ging's weiter)
7              case 'n' : System.out.println("nein");
8                         break;
9              default :  System.out.println("unentschlossen");
10                        break; // zur Sicherheit (=> Erweiterungen)
11         }
12     }
13 }
```

Fehlt die break-Anweisung am Ende eines Falls, so werden die Anweisungen des nächsten Falls ausgeführt usw.!

1.5.3 while-Anweisung

```
whileAnweisung ::= "while" "(" bedingung ")" anweisung
```

Die Anweisung bzw. der Anweisungsblock wird bei **while** so lange ausgeführt, bis die Bedingung nicht mehr wahr ist. Die Implementierung des Euklid'schen Algorithmus verdeutlicht dies:

Programm 1.7: Beispiel für die while-Anweisung (*While.java*)

```
1  public class While {
2      public static void main(String [] args) {
3          int a = 12, b = 28;
4
5          // Euklid's Algorithmus
6          while (a != b) {
7              if (a > b)
8                  a -= b;
9              else
10                 b -= a;
11         }
12
13         System.out.println ("ggT: "+a);
14     }
15 }
```

1.5.4 do-while-Anweisung

```
doWhileAnweisung ::= "do" anweisung "while" "(" bedingung ")"
```

Im Gegensatz zur while-Schleife wird bei der do-while-Schleife die *Bedingung nicht vor* der Ausführung des Schleifenrumpfes, *sondern danach überprüft*. Somit wird der *Schleifenrumpf* also *mindestens einmal durchlaufen*.

1.5.5 for-Anweisung

```
forAnweisung ::= "for" "(" init ";" bedingung ";" incr ")" anweisung
```

Wie in C ist die Java-for-Schleife allgemeiner als die for-Schleifen bei Pascal & Co. Eine Java for-Schleife kann man wie folgt in eine (nahezu) äquivalente while-Schleife transformieren:

```
for (init; bedingung; incr) anweisung

    =>

        init;
        while (bedingung) {
           anweisung;
           incr
        }
```

Folgendes Programm demonstriert die Verwendung der for-Schleife:

Programm 1.8: Beispiel für die for-Anweisung (*For.java*)

```
1  public class For {
2      public static void main(String[] args) {
3          // alle Zahlen von 0 bis 9 ausgeben
4          for (int i = 0; i < 10; i++)
5              System.out.println("i = "+i);
6
7          // alle geraden Zahlen von 0 bis 9 ausgeben
8          for (int j = 0; j < 10; j += 2)
9              System.out.println("j = "+j);
10     }
11 }
```

Folgendes Programm implementiert dieselbe Funktionalität mit while-Schleifen:

Programm 1.9: Beispiel für die Umsetzung von for- in while-Schleifen (*ForWhile.java*)

```
1  public class ForWhile {
2      public static void main(String[] args) {
3          // alle Zahlen von 0 bis 9 ausgeben
4          int i = 0;
5          while (i < 10) {
6              System.out.println("i = "+i);
7              i++;
8          }
9
10         // alle geraden Zahlen von 0 bis 9 ausgeben
11         int j = 0;
12         while (j < 10) {
13             System.out.println("j = "+j);
14             j += 2;
15         }
16     }
17 }
```

1.5.6 break-Anweisung

Mit der break-Anweisung kann man die direkt umgebende Schleife abbrechen:

Programm 1.10: Beispiel für die break-Anweisung (*Break.java*)

```
1  public class Break {
2      public static void main(String[] args) {
3          for (int x = 1; x < 10; x++) {
4              int y1 = 7*x;
5              int y2 = x*x;
6              if (y1 == y2)
7                  break; // bricht die for-Schleife ab
8              System.out.println("x = "+x);
9          }
10     }
11 }
```

1.5.7 continue-Anweisung

Mit der continue-Anweisung kann man die direkt umgebende Schleife anweisen, die aktuelle Ausführung der Anweisung(en) zu beenden und mit dem nächsten Schleifendurchlauf fortzufahren.

Programm 1.11: Beispiel für die continue-Anweisung (*Continue.java*)

```
1  public class Continue {
2      public static void main(String[] args) {
3          for (int x = 1; x < 10; x++) {
4              int y1 = 7*x;
5              int y2 = x*x;
6              if (y1 == y2)
7                  continue; // beginnt sofort mit dem naechsten Schleifendurchlauf
8              System.out.println("x = "+x);
9          }
10     }
11 }
```

1.5.8 return-Anweisung

Mit der return-Anweisung kann man die Ausführung einer Funktion beenden und einen Wert an den Aufrufer zurückgeben:

Programm 1.12: Beispiel für die return-Anweisung (*Return.java*)

```
1  public class Return {
2      public static int fibonacci(int i) {
3          if (i == 1)
4              return 1;
5          else if (i == 2)
6              return 1;
7          else
8              return fibonacci(i-1) + fibonacci(i-2);
9      }
```

```
10     public static void main(String [] args) {
11         for (int i = 1; i <= 10; i++)
12             System.out.println(" fib ("+i+") = "+fibonacci(i));
13     }
14 }
```

1.6 Pakete

Java-Klassen sind in sogenannten *Paketen* organisiert. Ein Paket enthält Klassen und evtl. auch *Unterpakete*. Paketnamen sind so ähnlich wie Verzeichnisnamen, nur dass die einzelnen Ebenen durch „.“ voneinander getrennt sind.

Beispiele: java.lang, java.io, java.util, java.util.zip

Beginnt eine Java-Quellcode-Datei mit **package** *my.own.pkg*, so gehören alle Klassen in dieser Datei zum Paket *my.own.pkg*. Normalerweise liegt eine solche Datei in einem Unterverzeichnis *my/own/pkg*. Die *Verzeichnisstruktur* entspricht also der *Paketstruktur*.

Fehlt eine Paket-Angabe, so gehören die Klassen zum Standard-Paket (default package), das auch als unbenanntes Paket (alle anderen Pakete sind ja benannt) bezeichnet wird. (Klassen des Standard-Paketes können von Klassen in anderen Paketen nicht importiert und somit nicht verwendet werden!)

Will man eine Klasse ClassXY aus dem Paket my.own.pkg, verwenden, so gibt es zwei Möglichkeiten:

- Man *importiert* die Klasse mittels **import** *my.own.pkg.ClassXY* und kann auf sie dann einfach mittels ClassXY zugreifen.
- Alternativ kann man auf die Klasse *voll qualifiziert* (mit dem Paketnamen) zugreifen: my.own.pkg.ClassXY.

Dies gilt nicht für Klassen aus dem Paket *java.lang*, die standardmäßig bereits importiert sind (z. B. die Klasse System). Bei Klassen im *selben Paket* kann man sich Import und volle Qualifizierung ebenfalls in der Regel sparen.

Folgendes Beispiel zeigt zwei Klassen – die Klasse Main befindet sich im Standard-Paket und die Klasse Test ist im Paket pkg:

Programm 1.13: Klasse Test (*pkg/Test.java*)

```
1 package pkg;
2
3 public class Test {
4     public static void hallo () {
5         System.out.println ("Hallo von pkg.Test!");
6     }
7 }
```

Programm 1.14: Klasse Main (*Main.java*)

```
1 import pkg.Test;
2
3 public class Main {
4     public static void main(String[] args) {
5         Test.hallo(); // ... oder pkg.Test.hallo() ohne import
6         System.out.println("Main");
7     }
8 }
```

```
chomsky$ ls -R
.:
Main.java  pkg

./pkg:
Test.java
chomsky$ javac -classpath . Main.java
chomsky$ ls -R
.:
Main.class  Main.java  pkg

./pkg:
Test.class  Test.java
chomsky$ java -classpath . Main
Hallo von pkg.Test!
Main
chomsky$
```

Anmerkung: Mit der *Option –classpath* kann man das Verzeichnis angeben, in welchem sich die Paket-Verzeichnisse befinden. (Man kann hierbei auch mehrere Verzeichnisse angeben, die unter Unix mit „:“ voneinander getrennt werden.) Ebenso kann man die *Umgebungsvariable CLASSPATH* verwenden.

Will man *mehrere Klassen* aus dem selben Paket my.own.pkg *importieren*, so kann man dies einfach mit **import** *my.own.pkg*.∗ erledigen. Dabei werden jedoch Klassen von *Unterpaketen* nicht importiert!

Gibt es Namenskonflikte – zwei gleichnamige Klassen im selben Paket bzw. importiert –, so kann man auf die betreffenden Klassen nur voll qualifiziert zugreifen.

Damit bei der Benennung von Paketen keine Namenskonflikte auftreten, sollte man die eigene *Domain als Präfix* verwenden. In meinem Fall – ich besitze die Domain johannes-mayer.com – hieße dies: Alle meine Pakete beginnen mit com.johannes-mayer. Ein Paket könnte also beispielsweise com.johannes-mayer.my.own.pkg sein.

1.7 Objektorientierte Konzepte

Im Gegensatz zu C++ ist *Java eine rein objektorientierte Programmiersprache*. Wir haben gesehen, dass wir alle bisherigen Programme in sog. Klassen „verpacken“ mussten. Wir werden in diesem Abschnitt nun die objektorientierten Konzepte in Java kennen lernen.

1.7.1 Objektorientierte versus prozedurale Programmierung

Was sind die wesentlichen Kennzeichen eines *prozeduralen Programms*?

- *Module* enthalten Daten, Typen und Prozeduren bzw. Funktionen
- Prozeduren bzw. Funktionen *verarbeiten Daten*
- *Fokus* liegt auf den *Verarbeitungsschritten* (Prozeduren bzw. Funktionen)
- *Daten* dienen zur *Kommunikation*

Abbildung 1.3 veranschaulicht die Daten in einem prozeduralen Programm und ihre Verwendung.

Abbildung 1.3: Daten in einem prozeduralen Programm

Was kennzeichnet ein *objektorientiertes Programm*?

- *Klassen* sind *Datentypen* und fassen *Daten* und *Operationen* (d. h. Prozeduren und Funktionen – sogenannte *Methoden*) auf diesen Daten zusammen
- *Fokus* liegt auf den *Daten*

Abbildung 1.4 stellt eine Klasse dar, die sowohl Daten als auch Operationen auf diesen Daten kapselt.

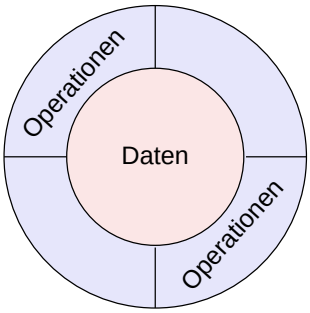


Abbildung 1.4: Klassen – die wesentlichen Elemente objektorientierter Programme

1.7.2 Klassen und Objekte

Eine *Klasse* ist nichts anderes als ein benutzerdefinierter Datentyp. In Programmiersprachen wie Pascal, Oberon, Modula-2 etc. gibt es die *Records* bzw. in C gibt es die *structs*. Diese benutzerdefinierten Datentypen fassen Daten beliebigen Typs zusammen. Klassen gehen hier – entsprechend

dem objektorientierten Konzept – einen Schritt weiter: Sie enthalen zusätzlich zu den Variablen auch Funktionen, sog. *Methoden*. Diese Methoden operieren auf den Daten der Klasse.

Programm 1.15 entählt die erste Version einer Klasse zum Arbeiten mit komplexen Zahlen. Programm 1.16 enthält das zugehörige Hauptprogramm.

Programm 1.15: Komplexe Zahlen – erster Versuch (*v1/Complex.java*)

```
1  public class Complex {
2      // Daten
3      public double re, im;
4
5      // Methoden
6      public void add(Complex c) {
7          re += c.re;
8          im += c.im;
9      }
10 }
```

Programm 1.16: Hauptprogramm zu den komplexen Zahlen – erster Versuch (*v1/Main.java*)

```
1  public class Main {
2      public static void main(String[] args) {
3          Complex c1, c2;
4
5          c1 = new Complex();        // Objekt c1 anlegen
6          c1.re = 1; c1.im = 3;      // und mit Werten initialisieren
7
8          c2 = new Complex();        // Objekt c2 anlegen
9          c2.re = -3; c2.im = 1;     // und auch initialisieren
10
11         // c1 und c2 ausgeben
12         System.out.println("c1 = "+c1.re+" + "+c1.im+" * i");
13         System.out.println("c2 = "+c2.re+" + "+c2.im+" * i");
14
15         c1.add(c2);        // c2 auf c1 addieren (=> nur c1 aendert sich)
16
17         // c1 und c2 ausgeben
18         System.out.println("c1 = "+c1.re+" + "+c1.im+" * i");
19         System.out.println("c2 = "+c2.re+" + "+c2.im+" * i");
20     }
21 }
```

```
chomsky$ java Main
c1 = 1.0 + 3.0 * i
c2 = -3.0 + 1.0 * i
c1 = -2.0 + 4.0 * i
c2 = -3.0 + 1.0 * i
chomsky$
```

Anmerkungen:

- In den Zeilen 5 und 8 von Main.java wird jeweils eine konkrete Ausprägung der Klasse Complex angelegt, ein sog. *Objekt*. Eine Klasse definiert nur bestimmte Elemente – wie ein Record. Diese Elemente gibt's aber noch nicht im Speicher. Ein Objekt hingegen befindet sich irgendwo im Speicher und seine Elemente besitzen konkrete Werte. c1 bzw. c2 ist eine Referenz – im Prinzip also so etwas ähnliches wie ein Zeiger. (In Oberon hätte man mittels `NEW(c1)` Platz beschafft; in C ginge dies in etwa mit `c1 = malloc(sizeof(Complex))`.)
- Die neu angelegten Objekte werden in den Zeilen 6 und 9 von Main.java noch sinnvoll initialisiert.
- Der Aufruf der Methode *add*() in Zeile 15 von Main.java ist auf den ersten Blick ungewohnt. Da steht nicht *add*(*c2*), sondern *c1*.*add*(*c2*). Eine *Objektmethode* – also eine Methode ohne Schlüsselwort **static** – operiert direkt auf den Daten eines Objektes. Deswegen muss man hier – wie beim Zugriff auf ein Datenelement – mit dem .-Operator auf das Element *add*() des Objektes c1 zugreifen. Jetzt ist auch klar, worauf sich *re* und *im* in den Zeilen 7 und 8 von Complex.java beziehen. Beim Aufruf *c1*.*add*(*c2*) sind damit *c1*.*re* und *c1*.*im* gemeint, die dadurch verändert werden.
- Die Methode *main*() von Main.java ist mit dem Schlüsselwort **static** deklariert und daher eine sog. *Klassenmethode*. Eine *Klassenmethode* ist im Gegensatz zu einer Objektmethode mit keinem Objekt assoziiert. Eine Klassenmethode ruft man daher aus derselben Klasse heraus einfach nur mit ihrem Namen auf *main* (...) bzw. ruft sie aus einer anderen Klasse heraus mit der Nennung des Namens der Klasse, in der diese Methode definiert ist auf, also *Main.main* (...) . Das sind also die Funktionen, wie wir sie aus der prozeduralen Programmierung her kennen. Die Klassen sind dann die Module, die diese Funktionen enthalten.
- Natürlich hätten wir die *main-Methode* auch in der Klasse Complex implementieren können und uns damit die Klasse Main gespart. Dies wäre aber vielleicht zu verwirrend. In Zukunft werden wir das aber hin und wieder so machen.

1.7.3 Konstruktoren

Es gibt eine Klassenmethode von Complex im vorigen Beispiel, die automatisch vom Compiler implementiert wird, wenn sie nicht existiert. Das ist der sog. *Konstruktor*, der bei der Erzeugung des Objektes aufgerufen wird. Im vorigen Beispiel wurde mittel *c1* = **new** *Complex*() der parameterlose Konstruktor von Complex aufgerufen, der – weil er nicht implementiert war – in seiner Standard-Implementierung (sog. *Default-Konstruktor*) gar nichts tut. Es wäre jedoch sinnvoll, die komplexe Zahl im Konstruktor zu initialisieren. Dies leistet nun die Erweiterung der Klasse Complex in Programm 1.17. Programm 1.18 ist das zugehörige geändere Hauptprogramm.

Programm 1.17: Komplexe Zahlen – zweiter Versuch (*v2/Complex.java*)

```
1  public class Complex {
2      // Daten
3      public double re, im;
4
5      // Konstruktoren
6      public Complex(double real) {
7          this(real, 0.0); // rufe den anderen Konstruktor auf
8      }
9      public Complex(double real, double imag) {
10         re = real;
11         im = imag;
12     }
```

```
13
14      // Methoden
15      public void add(Complex c) {
16          re += c.re;
17          im += c.im;
18      }
19  }
```

Programm 1.18: Hauptprogramm zu den komplexen Zahlen – zweiter Versuch (*v2/Main.java*)

```
1  public class Main {
2      public static void main(String[] args) {
3          Complex c1 = new Complex(1);              // Objekt c1 anlegen
4          Complex c2 = new Complex(-3, 1);          // Objekt c2 anlegen
5
6          // c1 und c2 ausgeben
7          System.out.println("c1 = "+c1.re+" + "+c1.im+" * i");
8          System.out.println("c2 = "+c2.re+" + "+c2.im+" * i");
9
10         c1.add(c2);       // c2 auf c1 addieren (=> nur c1 aendert sich)
11
12         // c1 und c2 ausgeben
13         System.out.println("c1 = "+c1.re+" + "+c1.im+" * i");
14         System.out.println("c2 = "+c2.re+" + "+c2.im+" * i");
15     }
16 }
```

```
chomsky$ java Main
c1 = 1.0 + 0.0 * i
c2 = -3.0 + 1.0 * i
c1 = -2.0 + 1.0 * i
c2 = -3.0 + 1.0 * i
chomsky$
```

Anmerkungen:

- In den Zeilen 3 und 4 von Main.java wird jeweils ein Objekt der Klasse Complex angelegt. Dazu werden die beiden Konstruktoren der Klasse Complex verwendet. Die *Initialisierung* passiert jetzt also *im Konstruktor* und nicht mehr dort, wo ein Objekt erzeugt wird.
- Der Konstruktor in den Zeilen 9ff. von Complex.java führt die Initialisierung direkt durch, wohingegen der Konstruktor in den Zeilen 6ff. von Complex.java die *Initialisierung an den anderen Konstruktor delegiert* – es wird eine komplexe Zahl erzeugt, deren Imaginärteil 0 ist. Einen (parameterlosen) *Default-Konstruktor* gibt es hier nicht, da explizit Konstruktoren definiert wurden. Einen *parameterlosen Konstruktor* gibt es in diesem Fall nur, wenn er explizit definiert wurde.

Mit einem Konstruktor besorgen wir Speicherplatz für ein Objekt und initialisieren es – wir legen das Objekt an. Die Freigabe des Speichers muss aber nicht „von Hand“ wie etwa in C/C++ mit `free()` bzw. `delete` bzw. in Modula-2 mit `DISPOSE()` erfolgen. Java besitzt (wie etwa auch Oberon) einen Garbage Collector, der alle nicht mehr referenzierten Objekte „einsammelt“.

1.7.4 Referenzen auf Objekte

Im Zusammenhang mit den Datentypen wurden die Referenztypen ja schon einmal angesprochen. Wie gesagt sind Referenztypen nichts anderes als typsichere Zeiger, die nicht explizit dereferenziert werden müssen (und auch nicht können).

Programm 1.19 und Programm 1.20 verdeutlichen dies an der Zuweisung bei Referenzen.

Programm 1.19: Komplexe Zahlen – dritter Versuch (*v3/Complex.java*)

```
1  public class Complex {
2      // Daten
3      public double re, im;
4
5      // Konstruktoren
6      public Complex(double real) {
7          this(real, 0.0); // rufe den anderen Konstruktor auf
8      }
9      public Complex(double real, double imag) {
10         re = real;
11         im = imag;
12     }
13
14     // Methoden
15     public void add(Complex c) {
16         re += c.re;
17         im += c.im;
18     }
19     public String toString() {
20         return re + " + " + im + " * i";
21     }
22 }
```

Programm 1.20: Hauptprogramm zu den komplexen Zahlen – dritter Versuch (*v3/Main.java*)

```
1  public class Main {
2      public static void main(String[] args) {
3          Complex c1 = new Complex(1);          // Objekt c1 anlegen
4          Complex c2 = new Complex(-3, 1);      // Objekt c2 anlegen
5
6          // c1 und c2 ausgeben
7          System.out.println("c1 = "+c1);
8          System.out.println("c2 = "+c2);
9
10         c1 = c2;
11
12         // c1 und c2 ausgeben
13         System.out.println("c1 = "+c1);
14         System.out.println("c2 = "+c2);
15
16         c1.re = 10;
17
18         // c1 und c2 ausgeben
19         System.out.println("c1 = "+c1);
```

```
20          System.out.println("c2 = "+c2);
21      }
22  }
```

```
chomsky$ java Main
c1 = 1.0 + 0.0 * i
c2 = -3.0 + 1.0 * i
c1 = -3.0 + 1.0 * i
c2 = -3.0 + 1.0 * i
c1 = 10.0 + 1.0 * i
c2 = 10.0 + 1.0 * i
chomsky$
```

Was bei der Zuweisung von Referenzen in Main.java genau passiert, das veranschaulicht Abbildung 1.5.

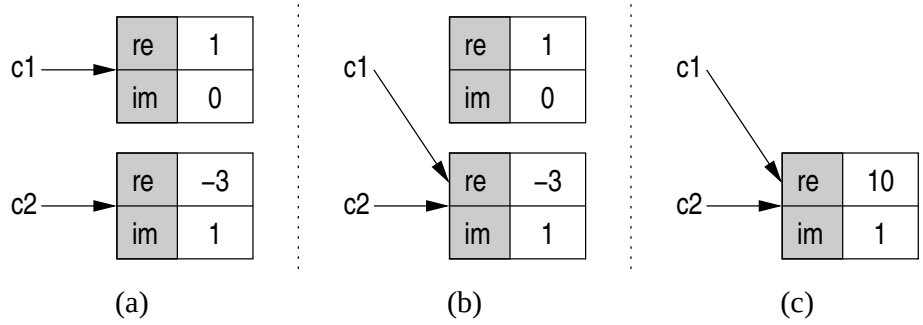


Abbildung 1.5: Zuweisung von Referenzen

Bei der Zuweisung *c1 = c2* (vgl. Abbildung 1.5b) findet keine Kopie des Objektes statt, sondern c1 referenziert danach das selbe Objekt wie c2 (⇝ Analogie zu Zeigern) – es wird also die Referenz (d. h. Adresse) kopiert.

Anmerkungen:

- Nach der Zuweisung gibt es *keine Referenz mehr auf das Objekt, welches c1 vorher referenzierte*. Das Objekt kann also vom *Garbage Collector* „eingesammelt“ werden.
- Damit ist nun klar – weil ja c1 und c2 nach der Zuweisung dasselbe Objekt referenzieren –, dass die Änderung *c1.re = 10* sowohl bei c1 als auch bei c2 Auswirkungen hat.
- Weil wir so viele Ausgaben von komplexen Zahlen machen wollen, ist die Methode *toString()* neu bei Complex hinzu gekommen (Zeilen 19–21 in Complex.java). Diese konkateniert einfach ein paar Strings mit dem Operator + und gibt das Ergebnis zurück. Variablen von einem *primitiven Typ im String-Kontext* (hier: re und im) werden *automatisch in Strings konvertiert* – somit ist es also unproblematisch, dass re und im keine Strings sind. Bei Variablen eines *Referenztyps im String-Kontext* (siehe z. B. Zeilen 7 und 8 in Main.java) wandelt der Java-Compiler dies automatisch in den *Aufruf der toString()-Methode* (aus "c1 = "+*c1* wird also "c1 = "+*c1.toString*()).

1.7.5 Vererbung und Polymorphie

Bei objektorientierter Programmierung ist es aber auch möglich, durch *Vererbung* Klassen zu erweitern. In unserem Beispiel mit den komplexen Zahlen könnten wir uns vorstellen, dass es eine Klasse Real wie in Programm 1.21 gibt, die reelle Zahlen repräsentiert.

Programm 1.21: Reelle Zahlen (*v4/Real.java*)

```
1  public class Real {
2      // Daten
3      public double re;
4
5      // Konstruktoren
6      public Real(double real) {
7          re = real;
8      }
9
10     // Methoden
11     public double real() {
12         return re;
13     }
14     public String toString() {
15         return "" + re; // re im Stringkontext => Umwandlung in String
16     }
17 }
```

Nun können wir die Klasse Complex auf der Basis von Real implementieren. Die Klasse Complex erweitert (⇝ **extends**) also die Klasse Real. Man spricht dann davon, dass Complex von Real *erbt*, Complex *Unterklasse* von Real ist und Real *Oberklasse* von Complex ist. Programm 1.22 zeigt die Implementierung von Complex als Unterklasse von Real.

Programm 1.22: Komplexe Zahlen – vierter Versuch (*v4/Complex.java*)

```
1  public class Complex extends Real {
2      // Daten
3      public double im;
4
5      // Konstruktoren
6      public Complex(double real) {
7          this(real, 0.0); // rufe den anderen Konstruktor auf
8      }
9      public Complex(double real, double imag) {
10         super(real);      // Konstruktor der Oberklasse aufrufen
11         im = imag;        // ... Rest selbst initialisieren
12     }
13
14     // Methoden
15     public double imag() {
16         return im;
17     }
18     public String toString() {
19         return re + " + " + im + " * i";
20     }
21 }
```

Die Klasse Complex enthält alle Variablen und Funktionen von Real und dazu weitere (nämlich *im* und *imag*()). Außerdem kann die Unterklasse Complex auch das Verhalten der geerbten Methoden ändern. Dies passiert hier mit der Methode *toString* (). Man sagt, dass Complex die Methode *toString* () *überschreibt*.

Innerhalb des zweiten Konstruktors der Klasse Complex (Zeile 10 in Complex.java) wird zunächst der *Konstruktor der Oberklasse* Real aufgerufen, um die Oberklasse zuerst zu initialisieren. Erst danach werden die Elemente der Klasse (in diesem Fall die Variable *im*) initialisiert. *Fehlt der Aufruf eines Oberklassenkonstruktors als erste Anweisung* in einem Konstruktor, so wird der parameterlose Konstruktor der Oberklasse aufgerufen (was ggf. zu einem Fehler führt, falls es keinen solchen gibt, wie z. B. bei der Klasse Real).

Abbildung 1.6 zeigt ein *UML-Klassendiagramm* für die beiden Klassen Real und Complex. UML-

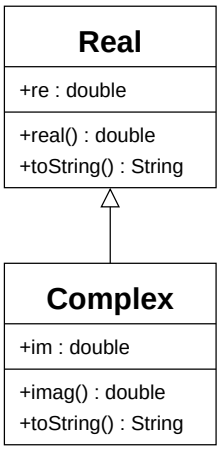


Abbildung 1.6: UML-Klassendiagramm für reelle und komplexe Zahlen

Diagramme werden wir später noch genau kennen lernen. Jede Klasse wird mit einem Kästchen dargestellt, in dem oben fettgedruckt ihr Name steht. Vererbungsbeziehungen werden durch einen Pfeil dargestellt. Außerdem stehen in den Kästchen der Klassen auch die jeweils neu definierten Variablen und Methoden, sowie die überschriebenen Methoden.

Programm 1.23: Hauptprogramm zu den komplexen Zahlen – vierter Versuch (*v4/Main.java*)

```
1  public class Main {
2      public static void main(String [] args) {
3          Real r1 = new Real(3);
4          Complex c1 = new Complex(2, 1);
5
6          // r1 und c1 ausgeben
7          System.out.println("r1 = "+r1); // impliziter Aufruf von toString()!
8          System.out.println("c1 = "+c1); // dito
9
10         Real r2 = c1; // Referenz vom Typ Real kann auch
11                       // ein Objekt einer Unterklasse referenzieren
12
13         // r2 ausgeben
14         System.out.println("r2 = "+r2); // Aufruf von Complex.toString()
15                                         // und nicht Real.toString()!
16
```

```
17          // Zugriff Real- und Imaginaerteil
18          System.out.println("c1.re = "+c1.real());
19          System.out.println("c1.im = "+c1.imag());
20
21          System.out.println("r2.re = "+r2.real());
22          //System.out.println("r2.im = "+r2.imag()); NICHT MOEGLICH!
23
24          Complex c2 = (Complex) r2;
25          System.out.println("c2.re = "+r2.real());
26          System.out.println("c2.im = "+c2.imag());
27      }
28  }
```

```
chomsky$ java Main
r1 = 3.0
c1 = 2.0 + 1.0 * i
r2 = 2.0 + 1.0 * i
c1.re = 2.0
c1.im = 1.0
r2.re = 2.0
c2.re = 2.0
c2.im = 1.0
chomsky$
```

Abbildung 1.7 zeigt die Objekte, die in Main.java erzeugt werden, im Speicher.

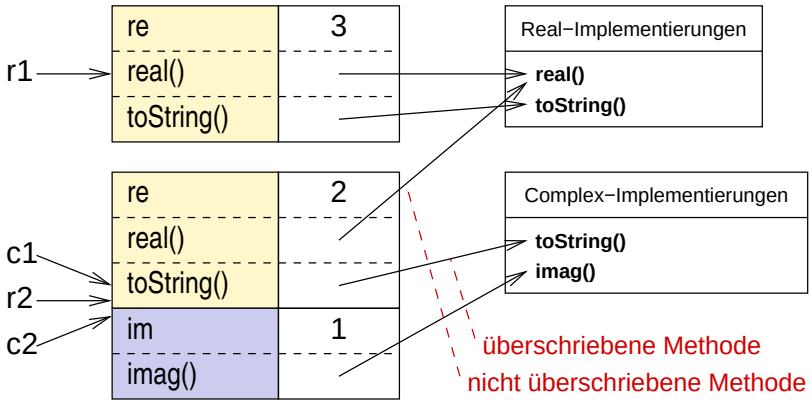


Abbildung 1.7: Realisierung der Vererbung

Anmerkungen:

- Einer Referenz kann man immer auch *Referenzen vom Typ einer Unterklasse zuweisen* (siehe Zeile 10 in Main.java). Hinter der Referenz der Oberklasse kann sich also auch ein Objekt einer Unterklasse verbergen. Dies äußert sich dann in Zeile 14, wo die Methode *toString* () der Klasse Complex aufgerufen wird. Erst zur Laufzeit wird festgestellt, von welcher Klasse die *toString* ()-Methode nun aufgerufen wird (vgl. Abbildung 1.7). Das nennt sich dann *spätes Binden* oder *Late Binding* der Methode *toString* () und man spricht in diesem Zusammenhang von *Polymorphie*.

- Obwohl – aufgrund von Polymorphie – zur Compilezeit nicht klar ist, welche Methode nun aufgerufen wird, so muss es doch eine solche Methode geben. In Zeile 22 darf der Aufruf *r2.imag*() nicht erfolgen, da Real keine solche Methode besitzt! (Die Klasse Real besitzt aber sehr wohl eine Methode *toString*(). Deshalb ist auch der Aufruf dieser Methode in Zeile 14 so erlaubt.)
- Will man z. B. die *imag*()-Methode verwenden und man weiss – wie hier –, dass es sich auch um ein Objekt der Unterklasse Complex handelt, so kann man mit einer *expliziten Typumwandlung* oder genauer mit einer *Typzusicherung* (siehe Zeile 24) eine Referenz vom Typ der Unterklasse Complex erhalten. Damit kann nun die Methode *imag*() aufgerufen werden.

Jede eigene Klasse besitzt eine Oberklasse. Ist keine extends-Klausel angegeben, so wird implizit die Klasse Object aus dem Paket java.lang Oberklasse.

Anmerkungen:

- Man müsste also in Abbildung 1.6 folglich noch die Klasse Object aufnehmen und einen Vererbungspfeil zwischen Object und Real einziehen. (In Klassendiagrammen werden nur Vererbungspfeile zur direkten Oberklasse gezogen – also keine transitive Kante von Complex zu Object.)
- Die Klasse Object enthält eine *toString()-Methode*. Somit hat jede eigene Klasse auch eine *toString*()-Methode. Jedoch ist die *Standard-Implementierung aus Object* im Allgemeinen für eigene Klassen wenig sinnvoll. Es empfiehlt sich also immer (besonders zu Debugging-Zwecken) eine *eigene toString()-Methode* zu implementieren.
- Die Klasse Object erhält außerdem noch einige Methoden, die natürlich auch an alle Klassen vererbt werden. Auf diese gehen wir ein, wenn dies sinnvoll ist.

1.7.6 Kapselung

Einen großen Nachteil hatten die bisherigen Implementierungen der Klasse Complex noch: Obwohl wir Konstruktoren und Methoden zur Initialisierung und Manipulation der Objektdaten hatten, könnte man theoretisch noch die *Elementvariablen re und im setzen*. Das nicht genug: Man sieht also auch von außen, dass Real- und Imaginärteil einfach durch double-Variablen implementiert sind – die *Implementierungsdetails* werden also offengelegt. Was bei der Klasse Complex noch relativ harmlos ist (man würde kaum eine andere Implementierung verwenden und es gibt auch keine Probleme einer direkten Veränderung der Variablen re und im), das sieht bei der Implementierung einer Liste bereits ganz anders aus. Eine Liste könnte man beispielsweise mit einem Array oder auch einer verketteten Liste implementieren. Diese Implementierung sollte natürlich nach außen auf gar keinen Fall sichtbar sein. Wir wollen also die Implementierungsdetails verbergen – dieses Prinzip nennt sich *Information Hiding*.

In Java – wie auch in anderen objektorientierten Programmiersprachen – gibt es das Prinzip der *Kapselung*. Elemente (d. h. Variablen und Methoden) von Klassen können in ihrer *Sichtbarkeit* eingeschränkt werden. Dabei gibt es die folgenden Sichtbarkeitsstufen:

Stufe	Sichbarkeit nur in Methoden
private	der Klasse selbst
package	von Klassen im selben Paket
protected	von Klassen im selben Paket und Unterklassen
public	von allen Klassen

Jetzt ist auch klar, was das Schlüsselwort **public** bei den Elementen einer Klasse für eine Bedeutung hatte: es wurde dadurch für alle Klassen sichtbar. Auch bei *Klassen* steht das Schlüsselwort **public**. Hier gibt es jedoch nur eine *Alternative*: die *Paket-Sichtbarkeit*. Paket-Sichtbarkeit erreicht man immer dadurch, dass man *kein Schlüsselwort* angibt.

Im Folgenden ist das Beispiel zu den komplexen und reellen Zahlen nochmals in leicht veränderter Version dargestellt. Insbesondere die *Sichtbarkeit von re und im* wurde jetzt auf **private** gesetzt. Dies hat zur Folge, dass re nur noch in den Methoden von Real und im nur noch in den Methoden von Complex sichtbar ist. Außerhalb können diese Werte natürlich noch über die **public**-Methoden *real* () und *imag*(), aber nicht mehr direkt zugegriffen werden. Würde man die *Sichtbarkeit von re und im* in **protected** ändern, so wären diese Elemente trotzdem noch *überall* zugreifbar, da die Klassen sich alle im selben Paket befinden. Wären alle Klassen aber in verschiedenen Paketen, dann könnte man re nur noch in Complex und nicht mehr in Main zugreifen.

Programm 1.24: Reelle Zahlen – zweiter Versuch (*v5/Real.java*)

```
1  public class Real {
2      // Daten
3      private double re;
4
5      // Konstruktoren
6      public Real(double real) {
7          re = real;
8      }
9
10     // Methoden
11     public double real() {
12         return re;
13     }
14     public String toString() {
15         return "" + re; // re im Stringkontext => Umwandlung in String
16     }
17 }
```

Programm 1.25: Komplexe Zahlen – fünfter Versuch (*v5/Complex.java*)

```
1  public class Complex extends Real {
2      // Daten
3      private double im;
4
5      // Konstruktoren
6      public Complex(double real) {
7          this(real, 0.0); // rufe den anderen Konstruktor auf
8      }
9      public Complex(double real, double imag) {
10         super(real);      // Konstruktor der Oberklasse aufrufen
11         im = imag;        // ... Rest selbst initialisieren
12     }
```

```
13
14      // Methoden
15      public double imag() {
16          return im;
17      }
18      public String toString () {
19          // re ist private in Real => re nicht sichtbar hier !
20          return real () + " + " + im + " * i";
21      }
22  }
```

Programm 1.26: Hauptprogramm zu den komplexen Zahlen – fünfter Versuch (*v5/Main.java*)

```
1  public class Main {
2      public static void main( String [] args ) {
3          Complex c1 = new Complex(2, 1);
4
5          // c1 ausgeben
6          System.out.println ("c1 = "+c1); // impliziter Aufruf von toString ()!
7
8          // Real- und Imaginaerteil ausgeben
9          System.out.println ("c1.re = "+c1.real ());
10         System.out.println ("c1.im = "+c1.imag ());
11
12         // Real- und Imaginaerteil ausgeben geht so nicht :
13         //System.out.println ("c1.re = "+c1.re );
14         //System.out.println ("c1.im = "+c1.im);
15     }
16 }
```

```
chomsky$ java Main
c1 = 2.0 + 1.0 * i
c1.re = 2.0
c1.im = 1.0
chomsky$
```

Die folgenden Klassen enthalten alle möglichen Zugriffe:

Programm 1.27: Klasse a.A (*pkg/a/A.java*)

```
1  package a;
2
3  public class A {
4      private int i1;
5      int i2;       // Paket- Sichtbarkeit
6      protected int i3;
7      public int i4;
8
9      public void hallo () {
10         System.out.println ("i1 = "+i1);
11         System.out.println ("i2 = "+i2);
12         System.out.println ("i3 = "+i3);
```

```
13          System.out.println("i4 = "+i4);
14      }
15  }
```

Programm 1.28: Klasse a.A1 (*pkg/a/A1.java*)

```
1  package a;
2
3  public class A1 {
4      public void hallo() {
5          A o = new A();
6          //System.out.println("i1 = "+o.i1); NEIN, da private
7          System.out.println("i2 = "+o.i2);
8          System.out.println("i3 = "+o.i3);
9          System.out.println("i4 = "+o.i4);
10     }
11 }
```

Programm 1.29: Klasse b.B (*pkg/b/B.java*)

```
1  package b;
2
3  import a.A;
4
5  public class B extends A {
6      public void hallo() {
7          //System.out.println("i1 = "+i1); NEIN, da private
8          //System.out.println("i2 = "+i2); NEIN, da package
9          System.out.println("i3 = "+i3);
10         System.out.println("i4 = "+i4);
11     }
12 }
```

Programm 1.30: Klasse b.B1 (*pkg/b/B1.java*)

```
1  package b;
2
3  import a.A;
4
5  public class B1 {
6      public void hallo() {
7          A o = new A();
8          //System.out.println("i1 = "+o.i1); NEIN, da private
9          //System.out.println("i2 = "+o.i2); NEIN, da package
10         //System.out.println("i3 = "+o.i3); NEIN, da protected
11         System.out.println("i4 = "+o.i4);
12     }
13 }
```

Programm 1.31: Klasse Main (*pkg/Main.java*)

```
1  import a.*;
2  import b.*;
3
4  public class Main {
```

```
5     public static void main(String[] args) {
6         A o1 = new A(); o1.hallo(); System.out.println("-----");
7         A1 o2 = new A1(); o2.hallo(); System.out.println("-----");
8         B o3 = new B(); o3.hallo(); System.out.println("-----");
9         B1 o4 = new B1(); o4.hallo();
10    }
11 }
```

```
chomsky$ ls -R
.:
Main.java  a  b

./a:
A.java  A1.java

./b:
B.java  B1.java
chomsky$ javac -classpath . Main.java
chomsky$ ls -R
.:
Main.class  Main.java  a  b

./a:
A.class  A.java  A1.class  A1.java

./b:
B.class  B.java  B1.class  B1.java
chomsky$ java -classpath . Main
i1 = 0
i2 = 0
i3 = 0
i4 = 0
-----
i2 = 0
i3 = 0
i4 = 0
-----
i3 = 0
i4 = 0
-----
i4 = 0
chomsky$
```

Anmerkungen:

- Die Klassen A, A1, B und B1 haben einen parameterlosen Default-Konstruktor, da kein Konstruktor implementiert wurde.
- In der Klasse A kann man natürlich in den Objektmethoden (und hallo() ist eine solche) auf die Elemente i1 bis i4 direkt zugreifen. Bei B ist das ebenfalls so, da B ja diese Elemente von A erbt. A1 und B1 erben aber nicht von A. Daher muss man hier erst ein Objekt von A erzeugen und dann kann man über dieses Objekt auf die Elemente zugreifen.

1.7.7 Klassenelemente

Bisher haben wir fast nur mit *Objektvariablen* und *Objektmethoden* (außer der main-Methode) gearbeitet. Diese Elemente einer Klasse sind *mit einem konkreten Objekt assoziiert*. Man braucht also zunächst ein Objekt der Klasse, um auf ein solches Element zugreifen bzw. es aufrufen zu können.

Es gibt aber auch *Klassenvariablen* und *Klassenmethoden*, die direkt *mit der Klasse assoziiert* sind – man braucht hier also kein Objekt. Klassenelemente sind mit dem *Schlüsselwort* **static** gekennzeichnet. Eine bereits bekannte Klassenmethode ist die *main-Methode*. Klassenmethoden sind im Prinzip Funktionen bzw. Prozeduren aus der prozeduralen Welt – die Klasse ist hier einfach das Modul, zu dem die Funktion/Prozedur gehört. Dementsprechend sind Klassenvariablen einfach die modul-globalen Variablen aus der prozeduralen Programmierung. Man verwendet sie, um Dinge zu speichern, die alle Objekte betreffen, z. B. wie die Objektanzahl in folgendem Beispiel:

Programm 1.32: Klassen- versus Objektelemente (*count/Count.java*)

```
1  public class Count {
2      private int i;                          // Objektvariable
3
4      public Count(int i) {
5          this.i = i;       // this ist Referenz auf aktuelles Objekt
6          count++;
7      }
8
9      public void inc() {                     // Objektmethode
10         i++;              // auch count waere zugreifbar
11     }
12
13     public static int count = 0;            // Klassenvariable
14
15     public static int getCount() {          // Klassenmethode
16         return count;    // kein i zugreifbar, da mit keinem Objekt assoziiert
17     }
18
19     public static void main(String[] args) {
20         System.out.println("count = "+count);
21
22         Count c = new Count(7);
23
24         System.out.println("count = "+count);
25
26         new Count(3); new Count(1); new Count(2);
27
28         System.out.println("count = "+count);
29
30         System.out.println("c.i = "+c.i);
31         System.out.println("Count.count = "+Count.count); // ... oder nur count in Count
32     }
33 }
```

```
chomsky$ javac Count.java
chomsky$ java Count
count = 0
count = 1
count = 4
c.i = 7
Count.count = 4
chomsky$
```

Anmerkungen:

- Elemente eines Objektes muss man – außer in den Objektmethoden dieser Klasse – immer qualifiziert zugreifen: „Referenz.Element“
  Beispiel: c . i
- Elemente einer Klasse muss man – außer in den (Objekt- und Klassen-)Methoden dieser Klasse – immer qualifiziert zugreifen: „Klassenname.Element“
  Beispiel: Count . count
- Innerhalb von Objektmethoden kann man **this** als *Referenz auf das aktuelle Objekt* verwenden. Dies kann z. B. nützlich sein, um – wie hier – Namenskonflikte aufzulösen.

1.7.8 Abstrakte Klassen

Manchmal ist es sinnvoll, eine Klasse *abstrakt* zu halten. Man kann z. B. für geometrische Objekte sinnvollerweise eine gemeinsame Oberklasse definieren. Man kann jedoch noch *nicht das komplette Verhalten dieser Klasse festlegen* – z. B. die Flächenberechnung. Und es sollte natürlich auch *kein Objekt einer abstrakten Klasse* geben.

Wenn man eine Klasse mit dem Schlüsselwort **abstract** deklariert (und meistens mindestens eine Methode in einer Klasse als **abstract** deklariert und somit auch *keine Implementierung angibt*), dann handelt es sich um eine *abstrakte Klasse*. Von einer abstrakten Klasse kann es keine Objekte geben und Unterklassen, die nicht mehr abstrakt sein sollen, müssen *mindestens die abstrakten Methoden* implementieren.

Folgendes Programm greift das vorher erwähnte Beispiel mit den geometrischen Figuren auf:

Programm 1.33: Abtrakte Oberklasse für geometrische Figuren (*fig/Figure.java*)

```
1 public abstract class Figure {
2     public abstract double area ();
3 }
```

Programm 1.34: Klasse für Kreise (*fig/Circle.java*)

```
1 public class Circle extends Figure {
2     private double r;
3
4     public Circle (double radius) {
5         r = radius ;
6     }
```

```
7
8       public double area() {
9           return r * r * Math.PI;
10      }
11  }
```

Programm 1.35: Klasse für Rechtecke (*fig/Rectangle.java*)

```
1  public class Rectangle extends Figure {
2      private double a, b;
3
4      public Rectangle(double a, double b) {
5          this.a = a;
6          this.b = b;
7      }
8
9      public double area() {
10         return a * b;
11     }
12 }
```

Programm 1.36: Hauptprogramm für geometrische Figuren (*fig/Main.java*)

```
1  public class Main {
2      public static void main(String[] args) {
3          // Figure f = new Figure();   NICHT erlaubt, da abstrakt !
4          Circle c = new Circle(3);
5          Rectangle r = new Rectangle(2, 4);
6
7          System.out.println("Flaeche von c: "+c.area());
8          System.out.println("Flaeche von r: "+r.area());
9      }
10 }
```

```
chomsky$ java Main
Flaeche von c: 28.274333882308138
Flaeche von r: 8.0
chomsky$
```

Anmerkungen:

- Die beiden Klassen Circle und Rectangle implementieren die abstrakte Methode *area* () und sind somit nicht mehr abstrakt.
- Auch wenn die Klasse Figure in unserem Beispiel keine implementierte Methode und Objektvariable enthielt, wäre dies durchaus möglich. Eine abstrakte Klasse ist also meistens eine Mischung aus implementierten und abstrakten Methoden.

1.7.9 Schnittstellen

Java kennt das *Konzept der Schnittstellen*. Parallel zu den Klassen definieren Schnittstellen ebenfalls *Referenztypen*. Unter einer Schnittstelle versteht man die Menge der *öffentlichen (Objekt-)Methoden*. Schnittstellen sind also so etwas wie abstrakte Klassen, die ausschließlich abstrakte Methoden enthalten – wie z. B. die Klasse Figure in unserem vorigen Beispiel.

Jede Methode in einer Schnittstelle ist automatisch **public** (auch wenn dieses Schlüsselwort fehlt – im Gegensatz zu Klassen wo die Paket-Sichtbarkeit bedeuten würde).

Außerdem gibt es das Konzept der *Vererbung* ebenfalls bei Schnittstellen. Hier gibt es sogar die *Mehrfachvererbung*, die aber unkritisch ist, da ja nur die Vereinigungsmenge der Methodendeklarationen vererbt wird. (Mehrfache Deklarationen derselben Methoden werden einfach zu einer Deklaration – dies ist unproblematisch, da es ja noch keine Implementierung gibt.) Es handelt sich hierbei um eine reine *Typvererbung*, wohingegen bei Klassen auch *Implementierungsvererbung* stattfindet – eine unsaubere Mischform also.

Erweitert eine Schnittstelle eine andere Schnittstelle, so wird hierfür wie bei der Vererbung bei Klassen das Schlüsselwort **extends** verwendet. Eine Klasse kann beliebig viele Schnittstellen implementieren. Dies wird durch das Schlüsselwort **implements** und die Angabe der implementierten Schnittstellen angezeigt.

Das Beispiel der geometrischen Figuren wird in folgendem Programm noch einmal aufgegriffen:

Programm 1.37: Schnittstelle für geometrische Figuren (*fig1/Figure.java*)

```
1 public interface Figure {
2     public double area ();
3 }
```

Programm 1.38: Klasse für Kreise (*fig1/Circle.java*)

```
1  public class Circle implements Figure {
2      private double r;
3
4      public Circle (double radius) {
5          r = radius ;
6      }
7
8      public double area () {
9          return r * r * Math.PI;
10     }
11 }
```

Programm 1.39: Klasse für Rechtecke (*fig1/Rectangle.java*)

```
1  public class Rectangle implements Figure {
2      private double a, b;
3
4      public Rectangle (double a, double b) {
5          this.a = a;
6          this.b = b;
7      }
```

```
8
9       public double area () {
10          return a * b;
11      }
12  }
```

Programm 1.40: Hauptprogramm für geometrische Figuren (*fig1/Main.java*)

```
1  public class Main {
2      public static double totalArea ( Figure [] fs ) {
3          double area = 0.0;
4          for (int i = 0; i < fs.length ; i++)
5              area += fs[i].area ();
6          return area ;
7      }
8
9      public static void main( String [] args ) {
10         Circle c = new Circle (3);
11         Rectangle r = new Rectangle (2, 4);
12
13         System.out.println ("Flaeche von c: "+c.area ());
14         System.out.println ("Flaeche von r: "+r.area ());
15
16         Figure [] fs = new Figure[] {c, r};
17
18         System.out.println ("Gesamtflaeche: "+ totalArea (fs ));
19     }
20 }
```

```
chomsky$ java Main
Flaeche von c: 28.274333882308138
Flaeche von r: 8.0
Gesamtflaeche: 36.27433388230814
chomsky$
```

Anmerkungen:

- Analog zu abstrakten Klassen kann es ebenfalls *keine Instanz* einer Schnittstelle geben.
- Geändert wurde in Figure: **abstract class** in **interface** (außerdem wurde bei der Methode das **abstract** weggelassen – das ist bei Schnittstellen optional)
- Bei Circle und Rectangle wurde nun aus **extends** ein **implements**.
- Die Methode *totalArea* () zeigt ein sehr gutes Beispiel, wie man mit Objekten arbeiten soll. Am besten *verwendet man Objekte über einen Schnittstellentyp* – wie hier Figure. Damit wird die Implementierung der Methode *totalArea* () von konkrekten Klassen – wie etwa Circle und Rectangle – (und somit Implementierungen) der Schnittstelle entkoppelt.

Wie bereits im Zusammenhang mit Referenzen und Klasen erwähnt, kann mit dem Cast-Operator eine Referenz in eine andere Referenz umgewandelt werden, wenn der Ziel-Referenztyp ein Subtyp ist. Wird zur Laufzeit jedoch festgestellt, dass das Objekt nicht von diesem Typ ist, so führt dies zu einer *ClassCastException* – später mehr dazu.

In Java gibt es im Paket java.lang die Schnittstelle *Comparable*, die als einziges die folgende Methode enthält:

```
public int compareTo(Object o)
```

Diese Methode liefert einen Wert < 0, gleich 0 bzw. > 0 je nachdem, ob das aktuelle Objekt kleiner, gleich oder größer als/wie das übergebene Objekt ist. Können beide nicht verglichen werden, so führt dies zu einer *ClassCastException* .

Nun gibt es in der Klasse *java.util.Arrays* eine Methode **public static void** *sort* (*Object* [] *a*), die davon ausgeht, dass alle Objekte die Schnittstelle *Comparable* implementieren. Damit kann diese Methode nun das Array sortieren.

Wir erweitern nun im Folgenden die Klasse Circle und sortieren dann einige Kreise:

Programm 1.41: Klasse für Kreise – nun vergleichbar (*sort/Circle.java*)

```
1  public class Circle implements Figure, Comparable {
2      private double r;
3
4      public Circle (double radius) {
5          r = radius ;
6      }
7
8      public double area () {
9          return r * r * Math.PI;
10     }
11
12     public String toString () {
13         return "Kreis mit Radius "+r;
14     }
15
16     public int compareTo(Object o) {
17         Circle c = (Circle) o; // fuehrt ggf. zu ClassCastException
18         return (r < c.r) ? -1 :
19                            ((r > c.r) ? 1 : 0);
20     }
21 }
```

Programm 1.42: Hauptprogramm zum Sortieren von Kreisen (*sort/Main.java*)

```
1  public class Main {
2      public static void main(String [] args) {
3          Object [] os = new Object[] {new Circle (4), new Circle (2),
4                                       new Circle (1), new Circle (3)};
5
6          System.out.println ("Vor dem Sortieren:");
7          for (int i = 0; i < os.length ; i++)
8              System.out.println (os[i]);
9
10         java.util.Arrays.sort (os);
11
12         System.out.println ();
```

```
13          System.out.println("Nach dem Sortieren:");
14          for (int i = 0; i < os.length; i++)
15              System.out.println(os[i]);
16      }
17  }
```

```
chomsky$ java Main
Vor dem Sortieren:
Kreis mit Radius 4.0
Kreis mit Radius 2.0
Kreis mit Radius 1.0
Kreis mit Radius 3.0

Nach dem Sortieren:
Kreis mit Radius 1.0
Kreis mit Radius 2.0
Kreis mit Radius 3.0
Kreis mit Radius 4.0
chomsky$
```

Man kann also allgemeine Algorithmen – wie z. B. Sortierroutinen auf der Basis von Schnittstellen implementieren.

1.7.10 Typvergleich

Mit dem Operator **instanceof** kann man überprüfen, ob ein Objekt von einem bestimmten Typ ist. Auf der linken Seite steht dabei eine Referenz, die ein Objekt repräsentiert. Auf der rechten Seite steht ein Referenztyp – also eine Klasse oder eine Schnittstelle. Das Ergebnis ist genau dann **true**, wenn die Klasse des Objektes oder eine Oberklasse davon identisch mit der Klasse ist bzw. die Schnittstelle implementiert.

Will man sicher gehen, dass eine Typzusicherung (Cast) zu keiner *ClassCastException* führt, so kann man eine Abfrage mit dem **instanceof**-Operator vorschalten. Dann kann man ausschließen, dass eine *ClassCastException* auftreten kann.

Folgendes Programm überprüft in der Methode *printArea*(), ob das übergebene Objekt ein Circle ist und gibt ggf. dessen Fläche aus:

Programm 1.43: Programm zum Typentest (*sort/TypeTest.java*)

```
1  public class TypeTest {
2      public static void printArea(Object o) {
3          if (o instanceof Circle) {
4              Circle c = (Circle) o; // kann nicht schiefgehen
5              System.out.println("Flaeche: "+c.area());
6          }
7          else
8              System.out.println("o ist kein Circle!");
9      }
10     public static void main(String[] args) {
```

```
11          printArea(new Circle(4));
12          printArea(new Object());  // kein Kreis :-(
13      }
14  }
```

```
chomsky$ java TypeTest
Flaeche: 50.26548245743669
o ist kein Circle!
chomsky$
```

1.8 Arrays

1.8.1 Eindimensionale Arrays

Durch anhängen von `[]` an einen Typ erhält man in Java den zu diesem Typ gehörigen *Array-Typ*. Array-Typen sind auch *Referenztypen* – Arrays sind also Objekte. Die Klassenhierarchie der Array-Typen ist in Abbildung 1.8 dargestellt. In diesem UML-Klassendiagramm ist zu erkennen,

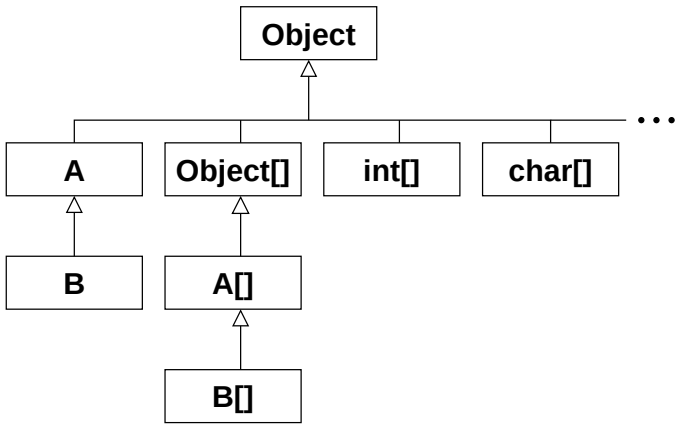


Abbildung 1.8: Klassenhierarchie der Array-Typen

dass die Array-Typen, welche auf primitiven Datentypen basieren, direkte Subtypen von Object sind. Bei Referenztypen gibt es jedoch bei den zugehörigen Arrays dieselbe Klassenhierarchie – insbesondere ist die Klasse Object[] die Oberklasse aller Array-Typen deren Elementtyp ein Arraytyp ist.

Ein Array kann man wie folgt anlegen:

```
int[] a = new int[5];
```

Auf der linken Seite des „=“ steht dabei die Deklaration der Arrayvariablen a. Dieser wird ein neu konstruiertes Array zugewiesen. Dies erfolgt – wie beim Anlegen von Objekten – mit dem Operator **new**. Hinter dem Elementtyp steht bei Arrays dann in Klammern die Größe des Arrays. Alle Elemente des neu angelegten Arrays sind mit dem Standardwert des Elementtyps initialisiert – bei Integern ist das der Wert 0. Man kann jedoch ein Array auch schon beim Anlegen initialisieren. Dies geht wie folgt:

```
int[] a = new int[] {1, 2, 3, 4, 5, 6, 7};
```

Durch das Element *length* eines Arrays kann dessen Länge ermittelt werden:

```
int[] a = new int[5];
// ...
int laenge = a.length;
```

Wie man an der Parameterliste der main-Methode sieht, werden die *Kommandozeilen-Argumente* als Array von Strings übergeben:

Programm 1.44: Arbeiten mit den Kommandozeilen-Argumenten (*Args.java*)

```
1 public class Args {
2     public static void main(String[] args) {
3         for (int i = 0; i < args.length; i++)
4             System.out.println("args["+i+"] = "+args[i]);
5     }
6 }
```

```
chomsky$ java Args
chomsky$ java Args 1
args[0] = 1
chomsky$ java Args 1 "2 3"
args[0] = 1
args[1] = 2 3
chomsky$ java Args -c 1 "2 3"
args[0] = -c
args[1] = 1
args[2] = 2 3
chomsky$
```

Die folgende Klasse, die (math.) Vektoren implementiert, demonstriert die Verwendung von Arrays:

Programm 1.45: Rechnen mit Vektoren (*Vector.java*)

```
1  public class Vector {
2      private int[] v;
3      public Vector(int[] v) {
4          this.v = new int[v.length];
5          for (int i = 0; i < v.length; i++)
6              this.v[i] = v[i];
7      }
8      public void multiplyBy(int c) {
9          for (int i = 0; i < v.length; i++)
10             v[i] *= c;
11     }
12     public String toString() {
13         String out = "(";
14         for (int i = 0; i < v.length; i++)
15             out += (i > 0 ? ", " : "") + v[i];
```

```
16          return out + ")";
17      }
18
19      public static void main(String [] args) {
20          Vector v = new Vector(new int[] {1, 2, 3});
21          System.out.println("v = "+v);
22
23          v.multiplyBy(2);
24          System.out.println("v = "+v);
25      }
26  }
```

```
chomsky$ java Vector
v = (1, 2, 3)
v = (2, 4, 6)
chomsky$
```

Man kann – anstatt [] direkt hinter dem Elementtyp anzugeben – Arrayvariablen auch wie folgt definieren:

```
int as [], x, bs [];
as = new int[] {1, 2};
x = 32;
bs = new int[] {4, 8, 16};
```

Man kann hierbei also die Deklaration von Arrayvariablen und „normalen“ Variablen mischen. Dies ginge, wenn vorne **int** [] steht nicht, dann sind nämlich alles Arrays:

```
int [] as, bs;
as = new int[] {1, 2};
bs = new int[] {4, 8, 16};
```

1.8.2 Mehrdimensionale Arrays

Würde man schreiben **int** [] *as* [], so hat man bereits eine Variable für *zweidimensionale* Arrays definiert. *Mehrdimensionale Arrays* in Java werden einfach sukkzessive durch eindimensionale Arrays aufgebaut. Die Elemente eines zweidimensionalen Arrays sind z. B. einfach Referenzen auf eindimensionale Arrays. So ist also **int** [][] ein Arraytyp dessen Elemente Referenzen auf eindimensionale Integer-Arrays sind. Man legt ein solches Array wie folgt an:

```
int [][] m = new int [10][10];
```

Man kann aber auch das Array selbst aufbauen und hat so die Möglichkeit z. B. eine Dreiecksmatrix zu konstruieren:

```
int [][] m = new int [10][];
for (int i = 0; i < m.length; i++)
    m[i] = new int[i+1];
```

m.length bezeichnet logischerweise die Größe des Arrays mit den Elementen vom Typ **int** [] – also die Größe der ersten Dimension. Mit *m*[*i*]. *length* kann man dann die Größe der zweiten Dimension in Zeile *i* herausfinden.

Natürlich gibt es wie bei eindimensionalen Arrays auch eine Kurzform, um initialisierte mehrdimensionale Arrays anzulegen:

```
int [][] m = new int [][] {{1,2,3},{4,5,6}};
```

Ein zweidimensionales Array – sei es nun eine quadratische oder eine Dreiecksmatrix – kann man mittels length nun wie folgt allgemein durchlaufen:

```
int [][] m = ...;
for (int i = 0; i < m.length ; i++)                    // m.length = Anzahl der Zeilen
    for (int j = 0; j < m[i ]. length ; j++)           // m[i ]. length = Laenge der i-ten Zeile
        // verarbeite Element m[i ][ j ]
```

1.9 Strings

Strings werden in Java durch die *Klasse String* repräsentiert. Für Strings gibt es die einfache Möglichkeit, sie in der Form "dies␣ist␣ein␣String" im Programm zu schreiben. Dabei wird bereits ein *Objekt vom Typ String* erzeugt. Es ist also nicht mehr notwendig und sogar überflüssig zu schreiben:

```
String s = new String("dies␣ist␣ein␣String");
```

Objekte vom Typ String sind *nicht veränderbar*. Durch jede verändernde Operation wird also ein neuer String erzeugt! Dies kann sehr performancekritisch sein. Daher gibt es auch die *Klasse StringBuffer*, die eine *veränderliche Zeichenfolge* repräsentiert. Diese sollte man bei vielen verändernden Operationen verwenden.

Wichtige Methoden der Klasse String sind:

- **int** *length* ()
  liefert die Länge des Strings.
- **char** *charAt*(**int** *index*)
  liefert das Zeichen an der angegebenen Position.
- **int** *compareTo*(*String anotherString*)
  vergleicht den String mit einem anderen. Das Ergebnis ist < 0, gleich 0 bzw. > 0 je nachdem, ob der String kleiner, gleich oder größer – lexikographisch gesehen – als der andere ist.
- **int** *compareToIgnoreCase* (*String str*)
  führt den Vergleich durch wie *compareTo* () und ignoriert dabei Groß- und Kleinschreibung.
- **boolean** *equals* (*Object anObject*)
  liefert genau dann **true**, wenn dieser und der übergebene String identisch sind.

- **boolean** *equalsIgnoreCase* (*String* *anotherString*)
  liefert genau dann **true**, wenn dieser und der übergebene String – unter Nichtbeachtung von Groß- und Kleinschreibung – identisch sind.
- *String* *concat* (*String* *str*)
  liefert einen String, der aus diesem und dem übergebenen String zusammengesetzt wurde.
- **boolean** *startsWith* (*String* *prefix*)
  liefert genau dann **true**, falls der String mit dem übergebenen Präfix beginnt.
- **boolean** *endsWith*(*String* *suffix*)
  liefert genau dann **true**, falls der String mit dem übergebenen Suffix endet.
- **int** *indexOf*(*String* *str*)
  liefert den kleinsten Index, ab dem str in diesem String auftaucht bzw. -1, falls str kein Teilstring ist.
- **int** *indexOf*(*String* *str* , **int** *fromIndex*)
  identisch mit **int** *indexOf*(*String*) – beginnt aber erst bei Index fromIndex mit der Teilstringsuche.
- **int** *lastIndexOf* (*String* *str*)
  liefert den größten Index, ab dem str in diesem String auftaucht bzw. -1, falls str kein Teilstring ist.
- **int** *lastIndexOf* (*String* *str* , **int** *fromIndex*)
  identisch mit **int** *lastIndexOf* (*String*) – beginnt aber erst bei Index fromIndex mit der Teilstringsuche.
- *String* *replace* (**char** *oldChar* , **char** *newChar*)
  ersetzt oldChar durch newChar in der zurückgelieferten Kopie.
- *String* *substring* (**int** *beginIndex* , **int** *endIndex*)
  liefert den Teilstring, der die Buchstaben im angegebenen Indexbereich enthält.
- *String* *toLowerCase* ()
  liefert eine Kopie, bei der alle Groß- in Kleinbuchstaben konvertiert sind.
- *String* *toUpperCase* ()
  liefert eine Kopie, bei der alle Klein- in Großbuchstaben konvertiert sind.
- *String* *trim* ()
  liefert eine Kopie, die am Anfang und Ende keine Whitespaces (d. h. Leerzeichen, Tabulatoren und Zeilenumbrüche) enthält.

Folgendes Beispiel zeigt die Verwendung dieser String-Methoden:

Programm 1.46: Arbeiten mit Strings (*Strings.java*)

```
1 public class Strings {
2     public static void main( String [] args ) {
3         String s = "Software␣Engineering␣Praxis";
4         System.out.println ("length:␣"+s.length ());
5         System.out.println ("compareTo:␣"+s.compareTo("Software" ));
6         System.out.println ("equals:␣"+s.equals ("Software" ));
7         System.out.println ("concat:␣"+s.concat ("␣an␣der␣Uni␣Ulm"));
8         System.out.println ("startsWith:␣"+s.startsWith ("Software" ));
9         System.out.println ("endsWith:␣"+s.endsWith("Praxis" ));
```

```
10          System.out.println("indexOf: "+s.indexOf("Engineering"));
11          System.out.println("replace: "+s.replace('e', 'i'));
12          System.out.println("substring: "+s.substring(3, 17));
13          System.out.println("toLowerCase: "+s.toLowerCase());
14          System.out.println("toUpperCase: "+s.toUpperCase());
15          System.out.println("trim: "+("   blubber         ".trim()));
16      }
17  }
```

```
chomsky$ java Strings
length: 27
compareTo: 19
equals: false
concat: Software Engineering Praxis an der Uni Ulm
startsWith: true
endsWith: true
indexOf: 9
replace: Softwari Enginiiring Praxis
substring: tware Engineer
toLowerCase: software engineering praxis
toUpperCase: SOFTWARE ENGINEERING PRAXIS
trim: blubber
chomsky$
```

1.10 Ausnahmen

1.10.1 Werfen von Ausnahmen

Bisher sind wir in unseren Programmen immer davon ausgegangen, dass alles gut geht. Dies ist jedoch sehr naiv. Zu jedem Programm gehört eine Fehlerbehandlung. Java kennt das Konzept der *Ausnahmen* (*exceptions*). Mittels **throw** kann man eine Ausnahme „werfen“. Dabei gibt man bei **throw** als Parameter eine Referenz vom Typ *Throwable* an:

throw *ausdruck*;

Die Klasse *Throwable* hat die beiden direkten Unterklassen *Exception* („normale“ Ausnahmen) und *Error* („schwere“, nicht behebbare Ausnahmen). Davon gibt es viele weitere Unterklassen. Die Klasse *IllegalArgumentException* wird z. B. verwendet, um zu signalisieren, dass eines der Argumente nicht korrekt war:

```
public int fakultaet (int n) {
    // ueberpruefe Vorbedingung: n >= 0
    if (!(n >= 0))
        // Ausnahme "werfen", wenn Vorbedingung nicht erfuellt ist
        throw new IllegalArgumentException ("n muss nicht-negativ sein");
    if (n == 0)
        return 1;
    else
        return n * fakultaet (n-1);
}
```

So, jetzt wissen wir, wie eine Ausnahme „geworfen“ wird. Aber was passiert dann? Dies zeigt das folgende erweiterte Programm:

Programm 1.47: Ausnahme ohne Behandlung (*Math.java*)

```
1  public class Math {
2      public static int fakultaet (int n) {
3          if (!( n >= 0))
4              throw new IllegalArgumentException ("n␣muss␣nicht−negativ␣sein");
5          if (n == 0)
6              return 1;
7          else
8              return n * fakultaet (n−1);
9      }
10     public static void main( String [] args ) {
11         System.out . println (" fakultaet (3) ␣=␣"+ fakultaet (3));
12         System.out . println (" fakultaet (−1)␣=␣"+ fakultaet (−1));
13     }
14 }
```

```
chomsky$ java Math
fakultaet(3) = 6
Exception in thread "main" java.lang.IllegalArgumentException: \
n muss nicht-negativ sein
        at Math.fakultaet(Math.java:4)
        at Math.main(Math.java:12)
chomsky$
```

Die obige Ausgabe liefert uns die Antwort: Das Programm wird mit der „geworfenen“ Ausnahme beendet, falls die Ausnahme nicht abgefangen wird. Die dabei erzeugte Ausgabe enthält den Typ der Ausnahme, den Parameter (String, der die Ausnahme näher schreibt) und den *Aufrufstack* (*stack trace*). Anhand des Aufrufstack kann man sehen, wo es zu der Ausnahme kam und wie diese Methode aufgerufen wurde. In unserem Fall sieht man, dass die Ausnahme in Zeile 4 von Math.java (genauer: in der Methode Math.fakultaet() auftrat. Diese Methode wurde von Math.main() in der Zeile 12 von Math.java aufgerufen. Diese Angaben können helfen, die Ursache zu finden. In unserem Fall sehen wir, dass bei dem Aufruf in Zeile 12 die Vorbedingung $n \geq 0$ nicht erfüllt ist.

1.10.2 Behandeln von Ausnahmen

Wenn wir nicht wollen, dass eine bestimmte Ausnahme, die wir erwarten, zur Beendigung des Programms führt, so fangen wir diese Ausnahme ab. Dies geht mittels **try**-**catch**- **finally** -Blöcken:

```
try {
    // ...
    // in diesen Anweisungen erwarten
    // wir eine AnException oder eine
    // AnotherException
    // ...
}
```

```
catch (AnException e) {
    // hier reagieren wir auf die
    // Ausnahme e
}
catch (AnotherException e) {
    // hier reagieren wir auf die
    // Ausnahme e
}
finally {
    // diese Anweisungen werden auf jeden
    // Fall ausgefuehrt, wenn hier keine
    // Ausnahme auftritt
}
```

Anmerkungen:

- Auf einen **try**-Block können beliebig viele **catch**-Blöcke folgen (auch keiner).
- Der **finally**-Block ist optional – es gibt also einen oder keinen.
- Nur der erste **catch**-Block, der auf die Ausnahme passt, wird bearbeitet. Dies heißt, dass die Anordnung immer vom Speziellen zum Allgemeinen erfolgen muss. Beispiel: IOException ist eine Unterklasse von Exception. Wenn wir sowohl IOExceptions als auch andere Exceptions abfangen wollten, so müsste zuerst der **catch**-Block für IOException (speziellere Ausnahme) stehen und dann käme erst der **catch**-Block für Exception.
- Fehlt ein **catch**-Block für die geworfene Ausnahme (oder gibt es gar keinen **try**-Block), so wird die Ausnahme an die aufrufende Funktion weitergereicht. Diese kann die Ausnahme durch einen **try**-**catch**-Block abfangen oder sie wird wiederum an die aufrufende Funktion weitergereicht und so weiter. Gibt es bis inkl. der main-Methode keinen passenden **catch**-Block, so wird das Programm wie gehabt abgebrochen.

Folgendes Programm behandelt nun IllegalArgumentExceptions und andere Exceptions:

Programm 1.48: Abfangen von Ausnahmen (*Math1.java*)

```
1  public class Math1 {
2      public static int fakultaet (int n) {
3          if (!(n >= 0))
4              throw new IllegalArgumentException ("n muss nicht-negativ sein");
5          if (n == 0)
6              return 1;
7          else
8              return n * fakultaet (n-1);
9      }
10     public static void main(String [] args) {
11         try {
12             System.out.println (" fakultaet (3) = "+fakultaet (3));
13             System.out.println (" fakultaet (-1) = "+fakultaet (-1));
14         }
15         catch ( IllegalArgumentException e) {
16             System.out.println ("IllegalArgumentException: "+e);
17         }
18         catch (Exception e) {
```

```
19              System.out.println("Exception: "+e);
20          }
21          finally {
22              System.out.println("Diese Ausgabe wird immer gemacht!");
23          }
24      }
25  }
```

```
chomsky$ java Math1
fakultaet(3) = 6
IllegalArgumentException: java.lang.IllegalArgumentException:\
n muss nicht-negativ sein
Diese Ausgabe wird immer gemacht!
chomsky$
```

Nun lassen wir einmal das Abfangen der Ausnahmen weg und verwenden nur noch **finally**:

Programm 1.49: Ausnahmen und finally (*Math2.java*)

```
1  public class Math2 {
2      public static int fakultaet (int n) {
3          if (!(n >= 0))
4              throw new IllegalArgumentException ("n muss nicht-negativ sein");
5          if (n == 0)
6              return 1;
7          else
8              return n * fakultaet (n-1);
9      }
10     public static void do_something () {
11         for (int i = 5; i >= -5; i--)
12             System.out.println (" fakultaet ("+i+") = "+fakultaet (i));
13     }
14     public static void main(String [] args) {
15         try {
16             do_something ();
17         }
18         finally {
19             System.out.println ("Diese Ausgabe wird immer gemacht!");
20         }
21     }
22 }
```

```
chomsky$ java Math2
fakultaet(5) = 120
fakultaet(4) = 24
fakultaet(3) = 6
fakultaet(2) = 2
fakultaet(1) = 1
fakultaet(0) = 1
Diese Ausgabe wird immer gemacht!
Exception in thread "main" java.lang.IllegalArgumentException:\
n muss nicht-negativ sein
        at Math2.fakultaet(Math2.java:4)
        at Math2.do_something(Math2.java:12)
        at Math2.main(Math2.java:16)
chomsky$
```

Man sieht an obiger Ausgabe, dass zuerst noch der **finally**-Block ausgeführt wird, bevor das Programm verlassen wird.

1.10.3 Geprüfte und ungeprüfte Ausnahmen

Wir haben ja schon kennen gelernt, dass Throwable die beiden Unterklassen Error und Exception hat, die etwas über die Schwere der Ausnahme aussagen. Es gibt hier noch eine weitere Unterscheidung: Es gibt nämlich *geprüfte* und *ungeprüfte* Ausnahmen.

- *Ungeprüfte Ausnahmen* sind *Errors* (samt Unterklassen) und *RuntimeExceptions* (samt Unterklassen).
- Dagegen sind *Exception* und alle Unterklassen (außer natürlich RuntimeException) *geprüfte Ausnahmen*.

Wieso diese Unterscheidung? Ungeprüfte Ausnahmen erwartet man nicht. Sie treten aufgrund eines Fehlers auf (z. B. IllegalArgumentException zeigt einen Programmierfehler beim Aufrufer an). Diese Ausnahmen fangen wir *nicht* ab! Wir fangen nur erwartete Ausnahmen ab und das sind die geprüften Ausnahmen, wie z. B. eine FileNotFoundException, die wir erwarten müssen, wenn wir eine Datei öffnen wollen.

Wir fangen nur geprüfte Ausnahmen ab. Ungeprüfte Ausnahmen zeigen immer einen Fehler (im Programm oder der Laufzeitumgebung) an und es macht daher keinen Sinn sie abzufangen. (Wie sollte man auch darauf reagieren?)

Dementsprechend macht es Sinn, dass Java verlangt, dass alle geprüften Ausnahmen entweder mit einem **catch**-Block abgefangen werden oder die Methode entsprechend deklariert wird, dass Sie die entsprechende Ausnahme wirft. Dies gilt auch beim Aufruf einer so deklarierten Methode!

Geprüfte Ausnahmen müssen durch einen **catch**-Block abgefangen werden oder die Methode muss so deklariert sein, dass Sie eine solche Ausnahme werfen kann.

Das folgende Beispiel demonstriert, wie Ausnahmen in einer Methode deklariert werden können. Außerdem werden die beiden Arten, mit einer geprüften Ausnahme umzugehen, vorgeführt:

Programm 1.50: Arbeiten mit geprüften Ausnahmen (*Checked.java*)

```
1  public class Checked {
2      // kein catch-Block, aber throws-Deklaration
3      public static void do1() throws Exception {
4          // ...
5          throw new Exception("Fang!");
6          // ...
7      }
8      // keine throws-Deklaration => catch-Block notwendig
9      public static void do2() {
10         try {
11             do1(); // => macht catch oder throws notwendig
12         }
13         catch (Exception e) {
14             System.out.println("do2: "+e);
15         }
16     }
17     // kein catch => throws-Deklaration notwendig
18     public static void do3() throws Exception {
19         try {
20             do1(); // => macht catch oder throws notwendig
21         }
22         finally {
23             System.out.println("do3!");
24         }
25     }
26     // kein catch => throws
27     public static void main(String[] args) throws Exception {
28         do2();
29         do3(); // => macht catch oder throws notwendig
30     }
31 }
```

```
chomsky$ java Checked
do2: java.lang.Exception: Fang!
do3!
Exception in thread "main" java.lang.Exception: Fang!
        at Checked.do1(Checked.java:5)
        at Checked.do3(Checked.java:20)
        at Checked.main(Checked.java:29)
chomsky$
```

1.11 Parameterübergabe

Java kennt bei der Parameterübergabe nur *Call-By-Value*. Sowohl von den übergebenen primitiven Variablen, als auch von den übergebenen Referenzen wird jeweils eine Kopie angelegt.

Ein Methodenaufruf kann nicht einen Parameter ein anderes Objekt referenzieren lassen. Das referenzierte Objekt kann aber sehr wohl durch den Methodenaufruf verändert werden.

Hier hilft wieder die Vorstellung, dass eine Referenz ja nichts anderes als ein (typsicherer) Zeiger ist (den man nicht dereferenzieren kann und muss). Dann wird klar, dass man bei Call-By-Value zwar nicht die Zeigeradresse verändern kann, aber sehr wohl das Objekt, auf welches der Zeiger zeigt.

Folgendes Beispiel verdeutlicht dies:

Programm 1.51: Parameterübergabe Call-By-Value (*Call.java*)

```
1  public class Call {
2      private int i;
3      public Call(int i) {
4          this.i = i;
5      }
6      public void inc() {
7          i++;
8      }
9      public String toString() {
10         return "Call(i="+i+")";
11     }
12
13     public static void method1(int x) {
14         x = -1;                    // Variable veraendern
15     }
16     public static void method2(Call c) {
17         c = new Call(-1);          // Referenz veraendern
18     }
19     public static void method3(Call c) {
20         c.inc();                   // referenziertes Objekt veraendern
21     }
22
23     public static void main(String[] args) {
24         int x = 2;
25         method1(x);        // kann x nicht veraendern!
26         System.out.println("x = "+x);
27
28         Call c = new Call(2);
29         method2(c);        // kann Referenz c nicht veraendern!
30         System.out.println("c = "+c);
31
32         c = new Call(2);  // eigentlich ueberfluessig ;-)
33         method3(c);        // veraendert referenziertes Objekt
34         System.out.println("c = "+c);
35     }
36 }
```

```
chomsky$ java Call
x = 2
c = Call(i=2)
c = Call(i=3)
chomsky$
```

Methoden können also übergebene Objekte verändern! Hier ist Vorsicht geboten.

1.12 Kopieren von Objekten

Will man verhindern, dass ein Objekt bei der Parameterübergabe verändert wird, so muss man es kopieren und die Kopie übergeben. Aber wie?

Die Klasse *Object* enthält die folgende Methode zum Kopieren:

protected *Object clone* () **throws** *CloneNotSupportedException*

Diese Methode wird auch von jeder Klasse somit geerbt.

Was ist zu tun?

1. Die kopierbare Klasse muss die Schnittstelle *Cloneable* implementieren. Diese Schnittstelle enthält keine Methode. Es handelt sich um eine sogenannte *Markierungs-Schnittstelle* (*marker interface*).
2. Es muss die Methode *clone()* überschrieben und *public* gemacht werden.
3. Bei der Implementierung von *clone()* muss man das *clone()* der Oberklasse aufrufen (und damit die Kopie der Elemente der Oberklasse erzeugen). Dabei muss man die *CloneNotSupportedException* abfangen, die nur auftreten würde, wenn *Cloneable* nicht implementiert wäre. Nun kann also keine solche Ausnahme mehr auftreten und man muss sie also auch nicht mehr deklarieren.

Folgendes Programm demonstriert die Implementierung von clone() an einem einfachen Beispiel:

Programm 1.52: Kopieren von Objekten (*Copy.java*)

```
1 public class Copy implements Cloneable {
2     private int i;
3     public Copy(int i) {
4         this.i = i;
5     }
6     public void inc() {
7         i++;
```

```
8     }
9     public String toString () {
10        return "Copy(i="+i+")";
11    }
12    public Object clone () {
13        try {
14            // Object.clone() erzeugt byteweise Kopie
15            // mit demselben dynamischen Typ (hier Copy):
16            Copy c = (Copy) super.clone ();
17            // ... das reicht hier aus
18            // (andernfalls muesste man hier noch weiter
19            // kopieren – "von Hand").
20            return c;
21        }
22        catch (CloneNotSupportedException e) {
23            // Transformation in eine ungepruefte Ausnahme
24            throw new RuntimeException("Unexpected exception: "+e);
25        }
26    }
27
28    public static void change(Copy c) {
29        c.inc ();
30    }
31
32    public static void main(String [] args) {
33        Copy c = new Copy(1);
34        System.out.println (""+c);
35        change(c);                          // Objekt uebergeben => wird veraendert !
36        System.out.println (""+c);
37        change((Copy) c.clone ());          // Kopie uebergeben => nur Kopie geaendert !
38        System.out.println (""+c);
39    }
40 }
```

```
chomsky$ java Copy
Copy(i=1)
Copy(i=2)
Copy(i=2)
chomsky$
```

Anmerkungen:

- *Object.clone()* erzeugt eine byteweise Kopie mit genau demselben dynamischen Typ wie das kopierte Objekt. Da der statische Typ des Rückgabewertes aber Object ist, muss bei jedem Aufruf von clone() eine Typzusicherung verwendet werden, die aber nicht schiefgehen kann.
- Die *CloneNotSupportedException* fällt etwas aus der Reihe: sie ist nicht erwartet (wenn man die Schnittstelle Cloneable implementiert hat), aber sie ist dennoch geprüft. Das ist eine Unschönheit. Dies kann man beheben, indem man die Ausnahme in eine ungeprüfte Ausnahme transformiert (innerhalb von catch).

Das folgende Programmbeispiel illustriert das Kopieren an einem etwas komplexeren Beispiel:

Programm 1.53: Kopieren von Vektoren (*Vector.java*)

```
1  public class Vector implements Cloneable {
2      private int[] a;
3      public Vector(int[] a) {
4          // Arrays koennen einfach kopiert werden
5          this.a = (int[]) a.clone();
6      }
7      public void multiplyBy(int f) {
8          for (int i = 0; i < a.length; i++)
9              a[i] *= f;
10     }
11     public String toString() {
12         String s = "(";
13         for (int i = 0; i < a.length; i++)
14             s += (i > 0 ? "," : "") + a[i];
15         return s + ")";
16     }
17     public Object clone() {
18         try {
19             // Object.clone() erzeugt byteweise Kopie
20             // mit demselben dynamischen Typ (hier Copy):
21             Vector v = (Vector) super.clone();
22             // ... das reicht hier aber nicht aus,
23             // denn es wurde nur der Referenzwert von a
24             // und nicht das Array kopiert! (=> a verweist
25             // auf dasselbe Array!)
26             v.a = (int[]) a.clone();
27             return v;
28         }
29         catch (CloneNotSupportedException e) {
30             // Transformation in eine ungepruefte Ausnahme
31             throw new RuntimeException("Unexpected exception: "+e);
32         }
33     }
34
35     public static void main(String[] args) {
36         Vector u = new Vector(new int[] {1, 2, 3});
37         System.out.println("u = "+u);
38
39         Vector v = (Vector) u.clone();
40         System.out.println("v = "+v);
41
42         v.multiplyBy(2);
43
44         System.out.println("u = "+u);
45         System.out.println("v = "+v);
46     }
47 }
```

```
chomsky$ java Vector
u = (1,2,3)
v = (1,2,3)
u = (1,2,3)
v = (2,4,6)
chomsky$
```

Anmerkungen:

- *Arrays* besitzen bereits eine öffentliche clone-Methode zum kopieren. Bei Arrays von einem Referenztyp (z. B. *Object* []) werden dabei aber nur die Referenzwerte und nicht die Objekte kopiert!
- In einer eigens implementierten clone-Methode kann man nach dem Aufruf der clone-Methode der Oberklasse noch einzelne Elemente kopieren – wie hier etwa das Array a.

Wieso kopieren wir eigentlich innerhalb der clone-Methode mittels **super**. *clone* () und erzeugen nicht einfach ein Objekt mit Hilfe eines Konstruktoraufrufs, das wäre doch einfacher, oder? Nein, das wäre falsch, denn was ist dann, mit einer Unterklasse? Wenn die dann **super**. *clone* () aufruft, dann ist das erzeugte Objekt nicht vom selben dynamischen Typ, sondern vom Typ der Oberklasse (wegen des Konstruktoraufrufs).

Fazit: In clone() nie mittels Konstruktor „kopieren“, sondern durch den Aufruf von clone() der Oberklasse (also: super.clone()).

1.13 Vergleich von Objekten

Der Vergleich von zwei Zahlen kann ganz einfach mit dem Operator == erfolgen:

```
int a = ..., b = ...;
if (a == b)
    ...
```

Wie kann man jedoch zwei *Objekte miteinander vergleichen*? Was passiert, wenn man hier auch den ==-Operator verwendet? In diesem Fall werden nur die beiden Referenzen miteinander verglichen! D. h. Gleichheit läge dann also nur vor, wenn beide Referenzen *genau dasselbe Objekt* referenzieren. Dies ist jedoch kein sinnvolles Verhalten. Zwei Strings können z. B. verschieden sein, d. h. verschiedene Objekte, aber sie können trotzdem den selben Inhalt haben. In diesem Fall würde man sie natürlich auch als gleich ansehen.

Aus diesem Grund gibt es die Methode *equals()* in der Klasse Object, die somit an alle Klassen vererbt wird. In ihrer Standardimplementierung macht *equals* () auch nichts anderes, als die Referenzen zu vergleichen, aber man kann ja diese Methode überschreiben, um ein sinnvolles Verhalten zu erzielen. So sollte man auch zwei Strings – es handelt sich hierbei um Objekte, auch wenn man das nicht so direkt sieht – den Vergleich mit *equals* () durchführen. Also:

```
String s1 = ..., s2 = ...;
if (s1.equals(s2))        // ... jetzt werden wirklich die Werte der Strings verglichen
    ...
```

Die Methode *equals* () besitzt die folgende Signatur:

```
public boolean equals(Object o)
```

Innerhalb einer eigenen Implementierung muss man zuerst mit **instanceof** sehen, ob o den richtigen Typ hat und dann ggf. eine Typzusicherung machen. Erst danach kann man die Werte vergleichen. Dies wird nun am Beispiel der komplexen Zahlen verdeutlicht:

Programm 1.54: Vergleich komplexer Zahlen (*Complex.java*)

```
1  public class Complex {
2      private int re, im;
3      public Complex(int re, int im) {
4          this.re = re;
5          this.im = im;
6      }
7      public Complex(int re) {
8          this(re, 0);
9      }
10     public String toString() {
11         return re+" + "+im+" i";
12     }
13     public boolean equals(Object o) {
14         if (o instanceof Complex) {
15             Complex c = (Complex) o;
16             return re == c.re && im == c.im;
17         }
18         else
19             return false;
20     }
21
22     public static void main(String[] args) {
23         Complex c1 = new Complex(3, 0);
24         Complex c2 = new Complex(3, 0);
25
26         System.out.println("c1 = "+c1);
27         System.out.println("c2 = "+c2);
28
29         System.out.println("c1 == c2:      "+(c1 == c2));
30         System.out.println("c1.equals(c2): "+c1.equals(c2));
31     }
32 }
```

```
chomsky$ java Complex
c1 = 3 + 0 i
c2 = 3 + 0 i
c1 == c2:      false
c1.equals(c2): true
chomsky$
```

Eine Implementierung von equals() muss sicherstellen, dass es sich beim Vergleich um eine Äquivalenzrelation (reflexiv, transitiv und symmetrisch) handelt, der null-Wert mit keinem Objekt identisch ist und wiederholte Aufrufe – solange das Objekt nicht verändert wurde – zum selben Ergebnis kommen.

Die Klasse Object enthält noch die folgende Methode

```
public int hashCode()
```

die einen Hashwert liefert.

Sind zwei Objekte gleich bzgl. equals(), so müssen sie auch denselben Hashwert haben.

Implementieren wir also die Methode *equals*(), so müssen wir in den meisten Fällen auch die Methode *hashCode*() implementieren, damit obige Bedingung erfüllt ist. Für Complex würde eine passende Implementierung beispielsweise wie folgt aussehen:

```
public int hashCode() {
    int result = 17;
    result = result * 37 + re;
    result = result * 37 + im;
    return result;
}
```

Es ist wichtig, hierbei *Primzahlen* zu verwenden, um eine *gute Streuung* zu erzielen. Für Objekte die nicht gleich sind, darf der Hashwert zwar identisch sein, doch dies ist nicht günstig. Daher versucht man eine möglichst gute Streuung zu erreichen.

1.14 Container-Datenstrukturen

Im Paket *java.util* sind eine ganze Reihe von *Container-Klassen* und *Container-Schnittstellen* enthalten. Ein paar werden wir im Folgenden kennen lernen.

1.14.1 Listen und dynamische Arrays

Die Schnittstelle *List* wird von allen Listen implementiert. Bei einer Liste kann man via Index auf die einzelnen Elemente zugreifen, Elemente anhängen und entfernen. Natürlich sind unterschiedliche Implementierungen möglich, wie z. B. als doppelt verkettete Liste (Klasse *LinkedList*) oder auf Basis eines Arrays (Klasse *Vector*). Die Klasse Vector, die wir uns nun genauer ansehen werden, ist ein dynamisch wachsendes Array, dessen Elemente Referenzen mit statischen Typ Object sind.

Die wichtigsten Methoden der Klasse *Vector* sind:

- **int** *size* ()
  liefert die Anzahl der Elemente im Vector.
- **boolean** *isEmpty*()
  liefert genau dann **true**, wenn der Vector leer ist.
- **boolean** *add*(*Object o*)
  hängt das Objekt o an und liefert **true**.
- **void** *add*(**int** *index*, *Object element*)
  fügt element an der angegebenen Position ein.
- **void** *clear* ()
  leert den Vector.
- *Object get*(**int** *index*)
  liefert das Element an der angegebenen Stelle.
- *Object set*(**int** *index*, *Object element*)
  setzt das Objekt an der angegebenen Stelle neu.
- *Object firstElement* ()
  liefert das erste Element.
- *Object lastElement* ()
  liefert das letzte Element.
- *Object remove*(**int** *index*)
  entfernt das Objekt an der angegebenen Stelle und liefert es. (Damit rücken dann die „hinteren“ Elemente auf.)
- *Iterator iterator* ()
  liefert einen Iterator. Mit diesem können die Elemente durchlaufen werden.

Außer den Methoden *firstElement* () und *lastElement* () gehören alle aufgezählten Methoden bereits zur Schnittstelle *List*. Die verändernden Methoden (*add*(), *set* (), *clear* () und *remove*()) sind dort jedoch nur *optional*, d. h. es kann sein, dass sie bei einer beliebigen Liste nicht implementiert sind und so zu einer UnsupportedOperationException führen. (Vector implementiert diese Methoden jedoch.)

Folgendes Programm demonstriert die Verwendung der Klasse Vector:

Programm 1.55: Verwendung der Klasse Vector (*Obst.java*)

```
1  import java.util.Vector;
2  import java.util.Iterator;
3
4  public class Obst {
5      public static void main(String[] args) {
6          Vector v = new Vector();
7          v.add("Apfel");
8          v.add("Birne");
9          v.add("Banane");
10         v.add("Ananas");
11         System.out.println("v = "+v);
12         System.out.println("Groesse: "+v.size());
13         System.out.println("Erstes: "+v.firstElement());
```

```
14          System.out.println("Letztes: "+v.lastElement());
15          System.out.println(" Stelle 2: "+v.get(2));
16          v.set(2, "Pflaume");
17          System.out.println("v = "+v);
18          v.remove(1);
19          System.out.println("v = "+v);
20          for (Iterator i = v.iterator(); i.hasNext();) {
21              System.out.println("> "+i.next());
22          }
23      }
24  }
```

```
chomsky$ java Obst
v = [Apfel, Birne, Banane, Ananas]
Groesse: 4
Erstes: Apfel
Letztes: Ananas
Stelle 2: Banane
v = [Apfel, Birne, Pflaume, Ananas]
v = [Apfel, Pflaume, Ananas]
> Apfel
> Pflaume
> Ananas
chomsky$
```

Der *Iterator*, den die Methode iterator() liefert, ermöglicht das durchlaufen einer beliebigen Datenstruktur. Die *Schnittstelle Iterator* hat dazu die folgenden Operationen:

- **boolean** *hasNext*()
  liefert genau dann **true**, wenn es noch ein Element gibt.
- *Object next*()
  liefert das nächste Element (sofern vorhanden). Gibt es kein Element mehr, so kommt es zur ungeprüften Ausnahme NoSuchElementException (die man aber durch vorherigen Test mittels hasNext() verhindern kann).
- **void** *remove*()
  entfernt das aktuelle Element. Diese Operation muss nicht unterstützt werden (was zu einer UnsupportedOperationException führt). Wird remove() vor dem ersten Aufruf von next() oder zweimal für ein Element aufgerufen, so führt dies zu einer IllegalStateException.

Wird etwas an der Datenstruktur geändert, von der man noch einen Iterator hat, so wird dieser Iterator ungültig.

1.14.2 Assoziative Arrays

Die Schnittstelle *Map* ist für assoziative Arrays gedacht (also Arrays, die nicht nur mit Integern indiziert werden können, sondern z. B. auch mit Strings). Die beiden Klassen *TreeMap* und *HashMap*

implementieren diese Schnittstelle auf Basis eines balancierten Baumes bzw. einer Hashtabelle. Entsprechend müssen die Schlüsselelemente für TreeMap die Schnittstelle *Comparable* (konsistent zu equals()) implementieren. Für eine HashMap muss sichergestellt sein, dass die Methode *hashCode()* der Schlüsselemente korrekt implementiert ist (konsistent mit equals()). Aufgrund der unterschiedlichen Art der Speicherung und des Zugriffs, unterscheiden sich natürlich die beiden Implementierungen auch hinsichtlich der Zugriffszeiten.

Die wichtigsten Methoden der Schnittstelle *Map* sind:

- **int** *size* ()
  liefert die Anzahl der Schlüsselwerte in der Map.
- **boolean** *isEmpty*()
  liefert genau dann **true**, wenn die Map leer ist.
- **void** *clear* ()
  leert die Map.
- **boolean** *containsKey* (*Object key*)
  liefert genau dann **true**, wenn die Map den Schlüssel enthält.
- *Object get* (*Object key*)
  liefert den Wert zum angegebenen Schlüssel.
- *Object put* (*Object key* , *Object value*)
  macht einen neuen Eintrag in die Map bzw. ändern den Wert zum angegebenen Schlüssel.
- *Object remove* (*Object key*)
  entfernt einen ggf. vorhandenen Eintrag zum angegebenen Schlüssel.
- *Set keySet* ()
  liefert eine Menge mit den Schlüsselwerten. Von dieser Menge kann man die Methode *iterator* () verwenden, um einen Iterator über alle Schlüsselwerte zu erhalten.

Wie im vorigen Abschnitt sind auch bei dieser Schnittstelle die verändernden Methoden optional.

Das folgende Programm demonstriert die Verwendung der Klasse HashMap:

Programm 1.56: Verwendung einer HashMap (*Einkaufsliste.java*)

```
1  import java.util.HashMap;
2  import java.util.Iterator;
3
4  public class Einkaufsliste {
5      public static void main(String[] args) {
6          HashMap m = new HashMap();
7          m.put("Butter", "3 Stücke");
8          m.put("Sahne", "4 Becher");
9          m.put("Milch", "3 Tüten");
10         m.put("Zucker", "1 Packung");
11         System.out.println("m = "+m);
12         System.out.println("Groesse: "+m.size());
13         System.out.println("Leer? "+m.isEmpty());
14         System.out.println("Eintrag für Butter? "+m.containsKey("Butter"));
15         System.out.println("Eintrag für Käse? "+m.containsKey("Käse"));
16         System.out.println("Wieviel Butter? "+m.get("Butter"));
```

```
17          m.remove("Butter"); // ... bereits erledigt
18          System.out.println("m = "+m); // = neuer Einkaufszettel
19          for (Iterator i = m.keySet().iterator(); i.hasNext();) {
20              Object key = i.next();
21              System.out.println("> "+key+" => "+m.get(key));
22          }
23      }
24  }
```

```
chomsky$ java Einkaufsliste
m = {Zucker=1 Packung, Sahne=4 Becher, Butter=3 Stücke, Milch=3 Tüten}
Groesse: 4
Leer? false
Eintrag für Butter? true
Eintrag für Käse? false
Wieviel Butter? 3 Stücke
m = {Zucker=1 Packung, Sahne=4 Becher, Milch=3 Tüten}
> Zucker => 1 Packung
> Sahne => 4 Becher
> Milch => 3 Tüten
chomsky$
```

Bei Verwendung einer HashMap müssen die Schlüsselwerte eine (mit equals() konsistente) hashCode()-Methode besitzen! Bei Verwendung einer TreeMap ist die Implementierung der Schnittstelle Comparable (oder ein Comparator) erforderlich (ebenfalls konsistent mit equals()).
Beachte: Die Schlüsselwerte müssen vergleichbar sein! (Sie dürfen also nicht von verschiedenen Klassen sein.)

1.15 Wrapper-Klassen

Was wäre, wenn wir beim Beispiel mit der Einkaufsliste als Werte direkt die Menge als Integer hätten eintragen wollen? Das wäre nicht gegangen, denn Integer ist ein primitiver Datentyp und als Wert benötigen wir einen Referenztyp. Dies ist der Grund, warum Java auch sogenannte *Wrapper-Klassen* enthält. Diese „verpacken“ den primitiven Datentyp in Form einer Klasse. Für den primitiven Datentyp *int* gibt es z. B. die Wrapper-Klasse *Integer*. Mit dieser Klasse können wir nun das Einkaufslisten-Beispiel entsprechend ändern:

Programm 1.57: Verwendung von Wrapper-Klassen (*Einkaufsliste.java*)

```
1 import java.util.HashMap;
2 import java.util.Iterator;
3
4 public class Einkaufsliste {
5     public static void main(String[] args) {
6         HashMap m = new HashMap();
7         m.put("Butter", new Integer(3));
8         m.put("Sahne", new Integer(4));
9         m.put("Milch", new Integer(3));
```

```
10          m.put("Zucker", new Integer (1));
11          System.out.println ("m = "+m);
12          System.out.println ("Groesse: "+m.size ());
13          System.out.println ("Leer? "+m.isEmpty ());
14          System.out.println ("Eintrag für Butter? "+m.containsKey("Butter" ));
15          System.out.println ("Eintrag für Käse? "+m.containsKey("Käse" ));
16          System.out.println ("Wieviel Butter? "+m.get("Butter" ));
17          m.remove("Butter"); // ... bereits erledigt
18          System.out.println ("m = "+m); // = neuer Einkaufszettel
19          for ( Iterator i = m.keySet (). iterator (); i.hasNext ();) {
20              Object key = i.next ();
21              System.out.println (">  "+key+" => "+m.get(key));
22          }
23      }
24  }
```

```
chomsky$ java Einkaufsliste
m = {Zucker=1, Sahne=4, Butter=3, Milch=3}
Groesse: 4
Leer? false
Eintrag für Butter? true
Eintrag für Käse? false
Wieviel Butter? 3
m = {Zucker=1, Sahne=4, Milch=3}
> Zucker => 1
> Sahne => 4
> Milch => 3
chomsky$
```

Die Wrapper-Klasse für Integer enthält die folgenden wesentlichen Methoden:

- **int** *intValue* ()
  liefert den Integer-Wert des Objektes.
- **static** *String* *toBinaryString* (**int** *i*)
  liefert die Binärdarstellung von *i*.
- **static** *String* *toOctalString* (**int** *i*)
  liefert die Oktaldarstellung von *i*.
- **static** *String* *toString* (**int** *i*)
  liefert die Dezimaldarstellung von *i*.
- **static** *String* *toHexString* (**int** *i*)
  liefert die Hexadezimaldarstellung von *i*.
- **static** *String* *toString* (**int** *i*, **int** *radix*)
  liefert die Darstellung von *i* zur Basis *radix*.
- **static** *Integer* *valueOf* (*String* *s*)
  parst *s* und liefert die enthaltene Integer, die als vorzeichenbehaftete Dezimalzahl interpretiert wird. Dabei kann eine *NumberFormatException* auftreten.

- **static** *Integer valueOf*(*String s*, **int** *radix*)
  parst *s* und liefert die enthaltene Integer, die als vorzeichenbehaftete Zahl zur Basis *radix* interpretiert wird. Dabei kann eine *NumberFormatException* auftreten.

Außerdem gibt es natürlich eine sinnvolle equals()- und hashCode()-Methode und die Schnittstelle Comparable wird auch implementiert. Wrapper-Klassen enthalten hauptsächlich die Klassenmethoden zum Parsen und zur Ausgabe von Daten des betreffenden Typs.

Folgendes Programmbeispiel illustriert die Verwendung der Methoden von Integer:

Programm 1.58: Verwendung der Wrapper-Klasse Integer (*Wrapper.java*)

```
1  public class Wrapper {
2      public static void main(String [] args) {
3          int i = 3;
4          Integer wrapi = new Integer(i);
5          System.out.println("i = "+i);
6          System.out.println("wrapi = "+wrapi);
7          int j = wrapi.intValue();
8          System.out.println("j = "+j);
9          Integer i1 = Integer.valueOf("16"); // Dezimalzahl
10         Integer i2 = Integer.valueOf("16", 16); // Hexadezimalzahl
11         System.out.println("i1 = "+i1);
12         System.out.println("i2 = "+i2);
13         j = i2.intValue();
14         System.out.println("j = "+Integer.toBinaryString(j));
15         System.out.println("j = "+Integer.toOctalString(j));
16         System.out.println("j = "+Integer.toString(j)); // Dezimal
17         System.out.println("j = "+Integer.toHexString(j));
18         System.out.println("j = "+Integer.toString(j, 8)); // zur Basis 8
19     }
20 }
```

```
chomsky$ java Wrapper
i = 3
wrapi = 3
j = 3
i1 = 16
i2 = 22
j = 10110
j = 26
j = 22
j = 16
j = 26
chomsky$
```

Außerdem gibt es im Paket *java.lang* ähnliche Wrapper-Klassen für **boolean**, **byte**, **char**, **float**, **double**, **long** und **short**: *Boolean*, *Byte*, *Character*, *Float*, *Double*, *Long* und *Short*.

1.16 Ein-/Ausgabe

Schon von Anfang an haben wir immer die Ausgaben mit *System.out.println* () gemacht. Was steckt aber dahinter? *out* ist einfach eine Klassenvariable der Klasse *System*. Und der Typ dieser Klassenvariable ist *PrintStream*. Analog dazu ist *System.in* – die Standardeingabe – vom Typ *InputStream*.

Nun wollen wir uns im Detail mit er Ein- und Ausgabe beschäftigen. Dabei spielen zwangsläufig *Streams* eine wichtige Rolle. Unter einem Stream versteht man eine geordnete Folge von Bytes oder Zeichen mit einer Quelle oder einem Ziel. Dabei muss man zwischen zwei Arten von Streams unterscheiden:

- **Byte-Streams:** *InputStream* und *OutputStream* sind die abstrakten Oberklassen dieser Sorte von Streams.
- **Zeichen-Streams:** Diese haben die beiden abstrakten Oberklassen *Reader* und *Writer*.

Im Folgenden behandeln wir nur die öfters benötigte zeichenweise Ein- und Ausgabe.

1.16.1 Zeichenweise Eingabe mit Reader & Co.

Im Wesentlichen stellt die abstrakte Klasse *Reader* Methoden zum Lesen eines oder mehrerer Zeichen zur Verfügung. Ein Reader ist eine *Abstraktion*, hinter der sich eine beliebige Zeichenquelle (Standardeingabe, Datei, etc.) verbergen kann.

Eine interessante Unterklasse ist *BufferedReader*, der (auf der Basis eines anderen Readers) eine Methode *readLine()* zum Lesen einer Zeile zur Verfügung stellt:

String readLine () **throws** *IOException*

Der Rückgabewert **null** signalisiert das Stream-Ende. Außerdem gibt es noch die Klasse *InputStreamReader*, mit der man aus einem *InputStream* einen Reader machen kann. Damit können wir nun komfortabel von der Standardeingabe lesen:

Programm 1.59: Lesen von der Standardeingabe (*Echo.java*)

```
1  import java.io.*;
2
3  public class Echo {
4      public static void main(String[] args) throws IOException {
5          InputStreamReader r = new InputStreamReader(System.in);
6          BufferedReader in = new BufferedReader(r);
7
8          String line;
9          while ((line = in.readLine()) != null) {
10             System.out.println(">␣"+line);
11         }
12     }
13 }
```

```
chomsky$ java Echo
Test
> Test
1, 2 und 3
> 1, 2 und 3
chomsky$
```

Außerdem gibt es noch die Klasse *FileReader* zum Lesen aus einer Datei. Das geht dann analog:

Programm 1.60: Lesen aus einer Datei (*Cat.java*)

```
1  import java.io.*;
2
3  public class Cat {
4      public static void main(String[] args) throws IOException {
5          if (args.length != 1) {
6              // Ausgabe auf die Fehlerausgabe
7              System.err.println("usage: java "+Cat.class.getName()+" <filename>");
8              System.exit(1);
9          }
10         // Lesen aus der Datei mit Name args[0]
11         FileReader r = new FileReader(args[0]);
12         BufferedReader in = new BufferedReader(r);
13
14         String line;
15         while ((line = in.readLine()) != null) {
16             System.out.println("> "+line);
17         }
18     }
19 }
```

```
chomsky$ java Cat
usage: java Cat <filename>
chomsky$ java Cat Echo.java
> import java.io.*;
>
> public class Echo {
>     public static void main(String[] args) throws IOException {
>         InputStreamReader r = new InputStreamReader(System.in);
>         BufferedReader in = new BufferedReader(r);
>
>         String line;
>         while ((line = in.readLine()) != null) {
>             System.out.println("> "+line);
>         }
>     }
> }
chomsky$
```

Weitere nützliche Klassen zur Eingabe sind *StringReader* – ein weiterer Reader –, mit dem man aus einem String einlesen kann (genauso wie von der Standardeingabe oder aus einer Datei).

Außerdem ist die Klasse *StreamTokenizer* nützlich zum Zerlegen einer Eingabe in Tokens – sie ist aber nicht einfach zum handhaben.

1.16.2 Zeichenweise Ausgabe mit Writer & Co.

Die Reader entsprechende *Abstraktion* für zeichenweise Ausgabe ist die abstrakte Klasse *Writer*. Mit dieser Abstraktion kann man einzelne Zeichen, Folgen von Zeichen und Strings schreiben. Hier spielt es ebenfalls keine Rolle, ob die Ausgabe nun auf der Konsole, in einer Datei, in einem String oder wo auch immer „landet“.

Analog gibt es auch einen *BufferedWriter*, der die Ausgabe durch Pufferung effizienter macht. (Hier können mehrere write()-Aufrufe zu einem „gebündelt“ werden.)

Ein besonderer Writer ist der *PrintWriter*, der *print()-Methoden* und *println()-Methoden* für alle möglichen Datentypen enthält. Die *Standardausgabe System.out* ist beispielsweise von diesem Typ.

Ein weiterer Writer ist *FileWriter*, mit dem man in eine Datei schreiben kann:

Programm 1.61: Schreiben in eine Datei (*Log.java*)

```
1  import java.io.*;
2
3  public class Log {
4      public static void main(String[] args) throws IOException {
5          if (args.length != 1) {
6              // Ausgabe auf die Fehlerausgabe
7              System.err.println("usage: java "+Log.class.getName()+" <filename>");
8              System.exit(1);
9          }
10         // Datei zum schreiben öffnen
11         FileWriter w = new FileWriter(args[0]);
12         BufferedWriter bw = new BufferedWriter(w);
13         PrintWriter out = new PrintWriter(bw);
14
15         out.println("dies");
16         out.println("... ist ein Log!");
17         out.println("Ciao!");
18         out.close(); // schliessen nicht vergessen !
19     }
20 }
```

```
chomsky$ java Log
usage: java Log <filename>
chomsky$ java Log xy
chomsky$ cat xy
dies
... ist ein Log!
Ciao!
chomsky$
```

1.17 Javadoc-Kommentare

Wie anfangs schon einmal erwähnt, kennt Java einen besonderen Kommentartyp /** ... */ – die sogenannten *Javadoc-Kommentare*. Wozu Javadoc-Kommentare? Es gibt das Tool *javadoc*, mit dem automatisch eine Dokumentation (in HTML) erzeugt werden kann. Dabei werden sowohl die Schnittstelle (also die Methoden und Variablen) einbezogen, als auch die Javadoc-Kommentare bei den Klassen, Methoden und Objekt- oder Klassenvariablen. Die Kommentare werden dabei als HTML-Text interpretiert. Einige Tags, die mit „@“ beginnen, werden dabei gesondert behandelt.

Eine Klasse kann man wie folgt dokumentieren:

```
/**
* Dies ist die Dokumentation von MyClass
* evtl. mit einem kleinen Beispiel:
* <pre>
* MyClass c = new MyClass();
* // ...
* </pre>
*
* @author Johannes Mayer
* @version 1.0, 2004-10-09
* @see java.lang.Integer
*/
public class MyClass {
// ...
```

Der Javadoc-Kommentar enthält die Angabe eines Autors mit dem @author-Tag, einer Version mit dem @version-Tag und einen Verweis auf die Klasse Integer mit dem @see-Tag. Die Syntax des @see-Tags bedarf ein bisschen der Erläuterung:

```
SEE_TAG ::= "@see" (PACKAGE_NAME | CLASS ["#" ELEMENT] | "#" ELEMENT)
CLASS   ::= [PACKAGE_NAME "."] CLASS_NAME
ELEMENT ::= VARIABLE_NAME | METHOD
METHOD  ::= METHOD_NAME ["(" PARAMETER_LIST ")"]
PARAMETER_LIST ::= TYPE {"," TYPE}
```

Man kann mit @see also ein Paket, eine Klasse, ein Element der aktuellen Klasse (`"#" ELEMENT`) oder ein Element einer anderen Klasse (`CLASS "#" ELEMENT`) referenzieren. Ein Element kann entweder eine Variable oder eine Methode sein. Das @see-Tag kann auch bei Methoden und Variablen im Javadoc-Kommentar verwendet werden.

Eine Variable kann man wie folgt kommentieren:

```
/**
* Der Name des Punktes.
*/
private String name;

/**
* Die beiden Koordinaten des Punktes.
*
* @see AndereKlasse# variable
*/
private int x, y;
```

Dabei kann – wie im zweiten Beispiel – ein Kommentar auch für mehrere Variablen verwendet werden.

Der Kommentar einer Methode sieht wie folgt aus:

```
/**
* Der Kommentar zur Methode toString ().
*
* @param radix   die Basis für die Konvertierung .
* @return        die Zahl in der Darstellung zur Basis radix
*                in einen String konvertiert .
* @throws        MyException
*                wenn die Basis negativ ist .
* @see #toString ()
*/
public String toString (int radix) throws MyException {
// ...
```

Mit @param kann man jeden Parameter beschreiben, d. h. ein @param-Tag für jeden Parameter. @return enthält die Beschreibung des Rückgabewertes. Und mit evtl. mehreren @throws-Tags kann man die deklarierten Ausnahmen erklären und angeben, unter welchen Bedingungen sie geworfen werden. Wie bereits besprochen, kann natürlich auch hier das @see-Tag verwendet werden, um eine Referenz herzustellen – in diesem Fall zur parameterlosen toString()-Methode.

Folgendes Programmbeispiel enthält nun Kommentare für die Klasse, die Variablen und die Methoden:

Programm 1.62: Verwendung von Javadoc-Kommentaren (*Complex.java*)

```
1  /**
2  * Eine Klasse zur Repräsentation von komplexen Zahlen.
3  *
4  * @author Johannes Mayer
5  * @version 1.0, 2004-10-09
6  */
7  public class Complex {
8      /**
9      * Der Realteil der komplexen Zahl.
10     *
11     * @see #im
12     */
13     private double re;
14     /**
15     * Der Imaginärteil der komplexen Zahl.
16     *
17     * @see #re
18     */
19     private double im;
20     /**
21     * Erzeugt die komplexe Zahl 0.
22     */
23     public Complex() {
24         this (0.0, 0.0);
25     }
26     /**
```

```
27      * Erzeugt eine komplexe Zahl mit Realteil <i>re</i>
28      * und Imaginärteil 0.
29      *
30      * @param re der Realteil der komplexen Zahl.
31      */
32     public Complex(double re) {
33         this(re, 0.0);
34     }
35     /**
36      * Erzeugt eine komplexe Zahl mit Realteil <i>re</i>
37      * und Imaginärteil <i>im</i>.
38      *
39      * @param re der Realteil der komplexen Zahl.
40      * @param im der Imaginärteil der komplexen Zahl.
41      */
42     public Complex(double re, double im) {
43         this.re = re; this.im = im;
44     }
45     /**
46      * Liefert den Realteil der komplexen Zahl.
47      *
48      * @return der Realteil.
49      */
50     public double real() {
51         return re;
52     }
53     // ...
54 }
```

Mit dem Kommando *javadoc* kann man nun daraus eine HTML-Dokumentation erzeugen:

```
chomsky$ javadoc -author -private -version -d doc Complex.java
Loading source file Complex.java...
Constructing Javadoc information...
Standard Doclet version 1.4.2_02
Generating doc/constant-values.html...
Building tree for all the packages and classes...
Building index for all the packages and classes...
Generating doc/overview-tree.html...
Generating doc/index-all.html...
Generating doc/deprecated-list.html...
Building index for all classes...
Generating doc/allclasses-frame.html...
Generating doc/allclasses-noframe.html...
Generating doc/index.html...
Generating doc/packages.html...
Generating doc/Complex.html...
Generating doc/package-list...
Generating doc/help-doc.html...
Generating doc/stylesheet.css...
chomsky$
```

Mit den Optionen `-author`, `-private` und `-version` erreicht man, dass Autor, Version und auch die privaten Elemente (Variablen und Methoden) in die Dokumentation eingebunden werden. Mit der Option `-d` kann man ein Zielverzeichnis für die Dokumentationsdateien angeben (in diesem Fall „doc“). Eine wichtige – hier nicht verwendete – Option ist `-classpath` mit der üblichen Bedeutung. Nach den Optionen folgen dann die Namen der Quelldateien und Pakete, für die die Dokumentation erzeugt werden soll. Weitere Einzelheiten sind (wie immer) auf der Manpage zu javadoc zu finden.

Die erzeugte HTML-Dokumentation ist in Abbildung 1.9 dargestellt.

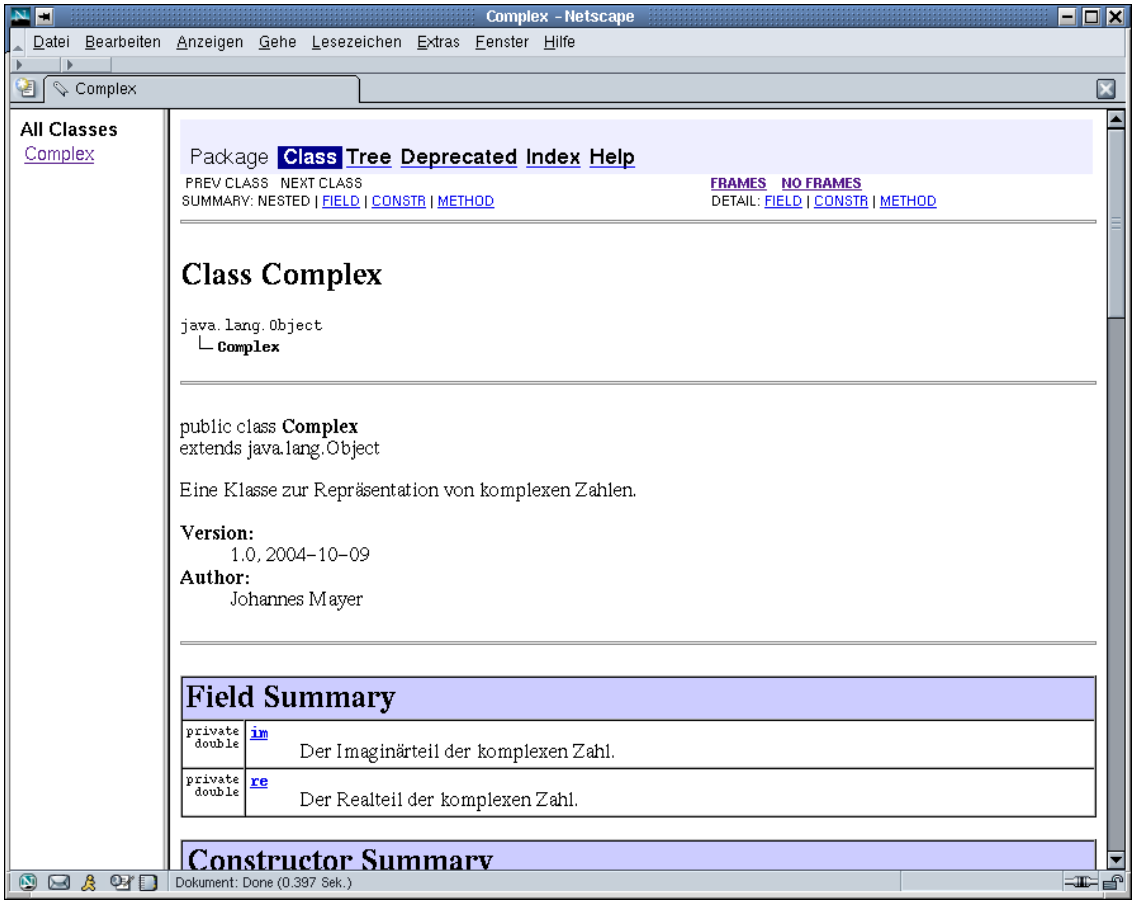


Abbildung 1.9: Darstellung der Javadoc-Dokumentation der Klasse Complex

1.18 JARs – Java-Archive

Genauso wie man eine Menge Dateien in ein ZIP-Archiv oder ein TAR-Archiv einpacken kann, so gibt es auch die Möglichkeit, Klassen in ein *Java-Archiv* (das im Prinzip ein ZIP-Archiv ist) zu verpacken. Java-Archive haben die *Endung „.jar“*. Mit dem Tool *jar* kann man diese Archive erstellen, ändern und auspacken. Das Gute ist nun, dass man nicht nur Verzeichnisse, sondern auch *Java-Archive* in den *Klassenpfad* aufnehmen kann. Somit muss man ein solches Archiv nicht „von Hand“ entpacken.

Gegeben sind die folgenden Klassen:

```
chomsky$ ls -R bin/ src/
bin/:

src/:
Main.java  a  b

src/a:
A.java

src/b:
B.java
chomsky$ javac -classpath src/ -d bin/ src/Main.java
chomsky$ ls -R bin/ src/
bin/:
Main.class  a  b

bin/a:
A.class

bin/b:
B.class

src/:
Main.java  a  b

src/a:
A.java

src/b:
B.java
chomsky$
```

Diese werden nun mittels *jar* in ein Java-Archiv „verpackt“:

```
chomsky$ cd bin
chomsky$ jar cvf ../myclasses.jar *
added manifest
adding: Main.class(in = 308) (out= 234)(deflated 24%)
adding: a/(in = 0) (out= 0)(stored 0%)
adding: a/A.class(in = 376) (out= 265)(deflated 29%)
adding: b/(in = 0) (out= 0)(stored 0%)
adding: b/B.class(in = 376) (out= 265)(deflated 29%)
chomsky$
```

Dann kann man die main-Methode der Klasse Main (im Java-Archiv) wie folgt ausführen:

```
chomsky$ java -classpath myclasses.jar Main
A!
B!
chomsky$
```

Mit den folgenden Kommandos kann man den Inhalt eines Java-Archivs ansehen und die *Manifest-Datei* extrahieren:

```
chomsky$ jar tvf myclasses.jar
     0 Sat Oct 09 16:37:12 CEST 2004 META-INF/
    71 Sat Oct 09 16:37:12 CEST 2004 META-INF/MANIFEST.MF
   308 Sat Oct 09 16:35:10 CEST 2004 Main.class
     0 Sat Oct 09 16:35:10 CEST 2004 a/
   376 Sat Oct 09 16:35:10 CEST 2004 a/A.class
     0 Sat Oct 09 16:35:10 CEST 2004 b/
   376 Sat Oct 09 16:35:10 CEST 2004 b/B.class
chomsky$ jar xf myclasses.jar META-INF/MANIFEST.MF
chomsky$ cat META-INF/MANIFEST.MF
Manifest-Version: 1.0
Created-By: 1.4.2_02 (Sun Microsystems Inc.)

chomsky$
```

Das Manifest wurde nun wie folgt verändert und zum Java-Archiv hinzugefügt. Dann kann man die main-Methode der Klasse Main einfach aufrufen:

```
chomsky$ cat META-INF/MANIFEST.MF
Manifest-Version: 1.0
Created-By: Johannes Mayer
Main-Class: Main
chomsky$ jar ufm myclasses.jar META-INF/MANIFEST.MF
Oct 9, 2004 4:51:36 PM java.util.jar.Attributes read
WARNING: Duplicate name in Manifest: Manifest-Version
Oct 9, 2004 4:51:36 PM java.util.jar.Attributes read
WARNING: Duplicate name in Manifest: Created-By
chomsky$ java -jar myclasses.jar
A!
B!
chomsky$
```

Anmerkungen:

- Das *Manifest-Attribut Main-Class* ermöglich die Angabe einer Klasse im Archiv, von der die main-Methode aufgerufen werden soll, falls das Kommando java mit der Option `-jar` verwendet wird.
- Es gäbe auch noch das *Manifest-Attribut Class-Path* zur Angabe eines Klassenpfades.

- Wie oben dargestellt, kann das Manifest ersetzt werden. Dabei kommt es zu Warnungen, wenn Attribute ersetzt werden.
- Mit der Option `-jar` und ohne Angabe einer Klasse kann so – unter Verwendung des Manifest-Attributes Main-Class – ein Java-Archiv (eigentlich genauer: die Hauptklasse des Archives) auf einfache Weise gestartet werden.

Wie immer gilt auch hier: Weitere Details zum Kommando jar finden Sie in der Manpage.

1.19 Literatur

[Arnold00] Arnold, K.; Gosling, J.; Holmes, D.: *The Java Programming Language.* Dritte Ausgabe, Addison-Wesley, 2000.

[Bloch01] Bloch, J.: *Effective Java: Programming Language Guide.* Addison-Wesley, 2001.

[Flanagan02] Flanagan, D.: *Java in a Nutshell.* Vierte Ausgabe, O'Reilly, 2002.

[Flanagan04] Flanagan, D.: *Java Examples in a Nutshell.* Dritte Ausgabe, O'Reilly, 2004.

[Gosling00] Gosling, J.; Joy, B.; Steele, G.; Bracha, G.: *The Java Language Specification.* Zweite Ausgabe, Addison-Wesley, 2000.

Kapitel 2

UML-Einführung

2.1 UML-Überblick

Glücklicherweise gibt es nun auch für Analyse und Entwurf in der objektorientierten Softwareentwicklung einen Standard, nämlich die *Unified Modeling Language (UML)* Mittlerweile liegt sie bereits in der Version 2 vor. Von den 15 Diagrammarten, die die UML in der aktuellen Version enthält, werden wir aber nur Ausschnitte der wichtigsten Arten für Design und Architektur kennen lernen. Die im Folgenden näher betrachteten Diagramme sind:

- Klassendiagramm
- Paketdiagramm
- Objektdiagramm
- Sequenzendiagramm

Man kann diese Diagramme nun noch nach verschiedenen Kriterien unterteilen. Eine Möglichkeit ist die Einteilung in Diagramme, die die Struktur beschreiben, und solche, die das Verhalten beschreiben:

Strukturdiagramme	Verhaltensdiagramme
Klassendiagramm Paketdiagramm Objektdiagramm	Sequenzdiagramm

Eine andere Möglichkeit besteht darin, sie danach zu unterteilen, ob sie eine statische Sicht des Systems oder eine Laufzeitsicht beschreiben:

statische Sicht	Laufzeitsicht
Klassendiagramm Paketdiagramm	Objektdiagramm Sequenzendiagramm

Zur UML gibt es ganze Bücher (siehe Literaturangaben) allein zu diesem Thema. Von daher ist es klar, dass dieses Kapitel nur einen kleinen Überblick und Einblick geben kann.

2.2 Use Case Diagramme

Beispiel[1]

Im Beispiel sei das Aussendienstsystem einer Bausparkasse neu zu entwickeln oder ein bestehendes zu erweitern oder zu verbessern. Dazu wird nach erfolgter Aufnahme der Anforderungen ein Use-Case-Modell entwickelt.

. . . Beratung zum Thema *Hausbau finanzieren* erwartet. Bezogen auf die von der Bausparkasse angebotenen Bausparverträge kann die Bausparkasse diese Dienstleistung in der Regel erbringen, wenngleich die gesamte Finanzierung in der Regel in Zusammenarbeit mit einer Bank erfolgt. Dieser Zusammenhang wäre in einer **Verfeinerung** des Use-Cases *Hausbau finanzieren* thematisierbar. Für diese Beratung wird in der Regel eine Vorhersage über den Zuteilungszeitpunkt benötigt. Deshalb benutzt der Use-Case *Hausbau finanzieren* den Use-Case *Ansparsimulation* und für die Darlehensphase den Use-Case *Darlehenssimulation*.

Außerdem bietet die Bausparkasse (hier vertreten durch den **Akteur** *Berater*) den Abschluss einer oder mehrerer Bausparverträge an (*Abschlussantrag stellen*) und bietet dann letztlich, nachdem diese zuteilungsreif sind, dem Bausparer einen Darlehensvertrag an (*Darlehensvertrag beantragen*) Hier will der Bausparer wissen, wie seine Rückzahlungsmodalitäten aussehen, weshalb auch der Use-Case *Darelehensvertrag beantragen* den Use-Case *Darlehenssimulation* benutzt.

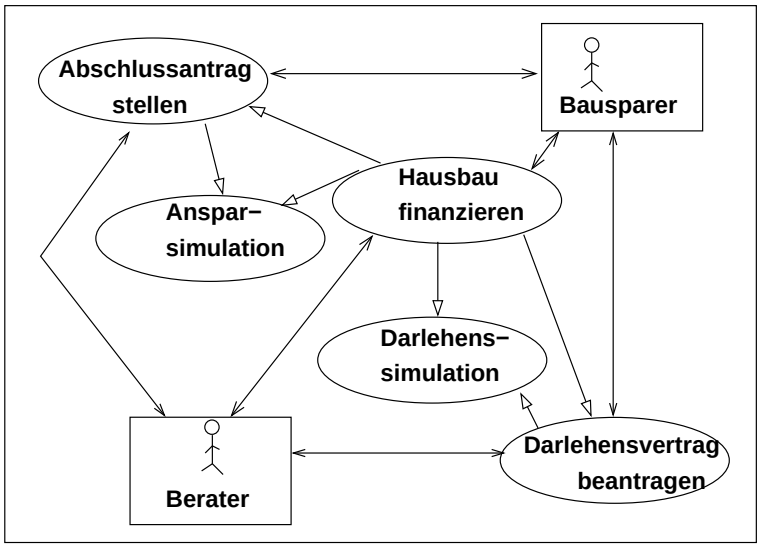


Abbildung 2.1: Use-Case-Diagramm zum System Bausparkasse

[1] aus R. Burkhardt: *UML - Unified Modeling Language.* Addison-Wesley Longman Verlag GmbH, Bonn, 1997

2.3 Klassendiagramme

Klassendiagramme bestehen im Wesentlichen aus Klassen und deren Beziehungen zueinander.

2.3.1 Klassen

Grundsätzlich wird eine Klasse mit fettgedruckten Namen in einem Kästchen dargestellt (siehe Abbildung 2.2). Für einen Überblick über die Klassenstruktur können die Attribute (d. h. Varia-

(a) (b)

Abbildung 2.2: Zwei unterschiedlich detaillierte Darstellungen einer Klasse

blen der Klasse) und Operationen (d. h. Methoden der Klasse) auch weggelassen werden.

2.3.2 Attribute und Operationen

Werden *Attribute* angegeben, so gilt für sie folgende Syntax:

```
[Sichtbarkeit] ["/"] Name [: Typ] [Multiplizität] [= Standardwert]
```

Für die *Sichtbarkeit* kann man aus folgenden Möglichkeiten wählen:

- `-`: *private*
- #: *protected* (anders als in Java: nicht auch package!)
- +: *public*
- ~: *package*

Mit / zeigt man an, dass es sich um ein *abgeleitetes Attribut* handelt – es kann also aus anderen Attributen etc. berechnet werden. Beispiel: In einer Datumsklasse mit Tag, Monat und Jahr wäre beispielsweise der Wochentag ein abgeleitetes Attribut, da er sich aus dem Datum berechnen lässt.

Über die *Multiplizität* kann man erreichen, dass Attribut mehrere Werte eines bestimmten Typs haben kann. Diese gibt man in eckigen Klammern an – in der Form „Untergrenze `..` Obergrenze“. Ist die Obergrenze unendlich, so notiert man dies durch `*`. Sind Ober- und Untergrenze identisch, so kann man abkürzend nur eine dieser beiden Zahlen nennen. Beispiele für Multiplizitäten sind also: `[0..1]` (ein optionales Attribut), `[1..*]`, `[3]` (abkürzend für `[3..3]`).

Hat eine Person mindestens einen, aber potenziell mehrere Vornamen, so kann man dies mit der Multiplizität `[1..*]` notieren.

Mit dem *Standardwert* kann man eine Initialisierung für dieses Attribut (passend zu seinem Typ) angeben.

Außerdem gäbe es noch *Eigenschaften*, die man an eine Attributdefinition anhängen kann – wie z. B. `{readOnly}` für Konstanten.

Für *Operationen* gilt die folgende Syntax:

```
[Sichtbarkeit] Name(Parameterliste) [: <Typ des Rückgabewertes>]
```

Die *Parameterliste* ist eine Folge von Parametern, die voneinander durch Semikolon getrennt sind. Ein für *Parameter* gilt die folgende Syntax:

```
[Richtung] Name [: Typ] [Multiplizität] [= Standardwert]
```

Die *Richtung* er Parameterübergabe kann die folgenden Werte annehmen:

- *in*: Eingabeparameter
- *out*: Ausgabeparameter
- *inout*: Ein- und Ausgabeparameter
- *return*: Parameter für den Rückgabewert

Eigenschaften kann man auch an das Ende einer Parameter- und Operationsdefinition anfügen.

Klassenattribute und *Klassenoperationen* werden *unterstrichen* dargestellt.

Ein Klassendiagramm für eine Personen-Klasse könnte etwa wie in Abbildung 2.3 aussehen. Eine

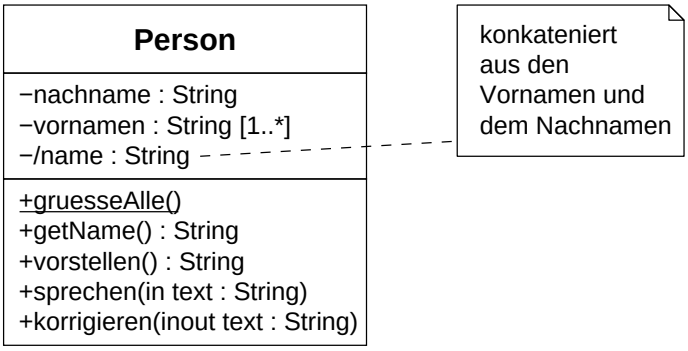


Abbildung 2.3: Klassendiagramm für eine Personen-Klasse mit Attributen und Operationen

Personen hat einen Nachnamen (vereinfachend!) und einen oder mehrere Vornamen, sowie einen Namen, den man durch Konkatenation der Vornamen und des Nachnamens erhalten kann. All diese Attribute sind privat. Eine Person bietet Operationen, um den Namen zu erfragen, eine Vorstellung dieser Person zu arrangieren, sie einen bestimmten Text sprechen zu lassen und einen Text zu korrigieren. AuSSerdem gibt es für alle Personen eine Klassenoperation, um alle Personen zu grüßen.

Kommentare werden in einem Kästchen mit „eingeknickter Ecke“ dargestellt und durch eine gestrichelten Linie mit dem entsprechenden Element (Klasse, Attribut, Operation, ...) verbunden (siehe Abbildung 2.3).

2.3.3 Vererbung

Vererbung bei Klassen wir in einem Klassendiagramm durch einen Pfeil mit Dreiecksspitze von der Unter- zur Oberklasse dargestellt (siehe Abbildung 2.4).

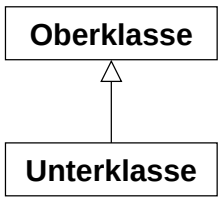


Abbildung 2.4: Vererbung bei Klassen im Klassendiagramm

2.3.4 Abstrakte Klassen

Es gibt zwei Möglichkeiten, eine *abstrakte Klasse* darzustellen. Entweder, indem man den Name der Klasse kursiv setzt, oder durch die *Eigenschaft* `{abstract}` unter dem Klassennamen (siehe Abbildung 2.5). *Abstrakte Operationen*, d. h. Operationen, die in einer abstrakten Klasse nur

(a) (b)

Abbildung 2.5: Zwei Darstellungsmöglichkeiten für abstrakte Klassen

deklariert, aber nicht definiert sind, werden ebenfalls kursiv dargestellt (siehe Abbildung 2.5).

2.3.5 Schnittstellen

Zur Darstellung von *Schnittstellen* wird der *Stereotyp „interface“* verwendet. Schnittstellenoperationen müssen aber – im Gegensatz zu abstrakten Methoden – nicht kursiv gesetzt werden, da es ja in diesem Fall keine Implementierung geben kann (siehe Abbildung 2.6).

Zur Darstellung der *Implementierung einer Schnittstelle* gibt es mittlerweile zwei Notationen (siehe Abbildung 2.7). Analog dazu gibt es auch für die Verwendung von Klassen über Schnittstellen zwei mögliche Darstellungen (siehe Abbildung 2.8).

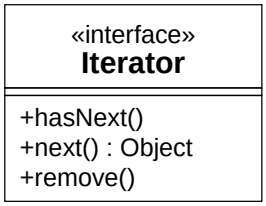


Abbildung 2.6: Klassendiagramm für eine Schnittstelle

Abbildung 2.7: Klassendiagramm-Varianten zur Implementierung einer Schnittstelle

Abbildung 2.8: Klassendiagramm-Varianten zur Verwendung einer Klasse über eine Schnittstelle

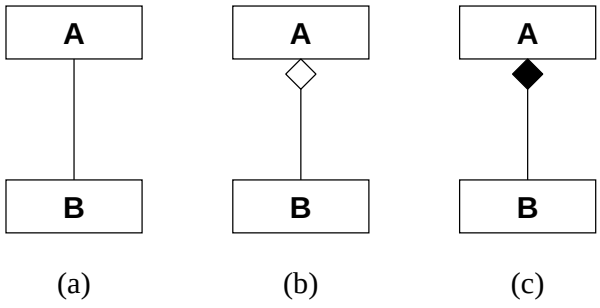


Abbildung 2.9: Assoziation, Aggregation und Komposition

2.3.6 Assoziationen

Beziehungen zwischen Ausprägungen einer Klasse können mit Hilfe von *Assoziationen* (siehe Abbildung 2.9a) annotiert werden.

Es gibt die folgenden Sonderformen:[2]

- *Aggregation*: Beschreibt eine Zuordnungs-Beziehung (siehe Abbildung 2.9b und Programm 2.1). Beispiel: Ein Mitarbeiter ist in einer Abteilung enthalten. (Er kann evtl. auch in mehreren Abteilungen sein.) Die Abteilung besteht aber weiterhin, wenn dieser Mitarbeiter kündigt.
- *Komposition*: Beschreibt eine Besteht-aus-Beziehung (siehe Abbildung 2.9c und Programm 2.2). Beispiel: Ein Student hat eine Matrikelnummer. Ohne diese kann man sich einen Studenten kaum ;-) vorstellen. Man kann sich auch nicht vorstellen, dass mehrere Studenten dieselbe Matrikelnummer haben.

Beispiele zu Aggregation und Komposition:

Programm 2.1: Beispiel zu Aggregation (*Abteilung.java*)

```
1  public class Abteilung {
2      private Angestellter chef;                          // Aggregation
3      private List mitarbeiter = new Vector();            // Aggregation
4      // ...
5      public Abteilung( Angestellter chef ) {
6          this.chef = chef;                               // ... deswegen
7          // ...
8      }
9      public void neuerMitarbeiter ( Angestellter mitarb ) {
10         mitarbeiter.add( mitarb );                      // ... deswegen
11     }
12     public void kündigen( Angestellter mitarb ) {
13         mitarbeiter.remove( mitarb );
14     }
15     // ...
16 }
```

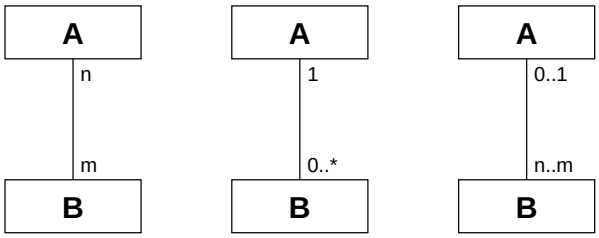
Programm 2.2: Beispiel zu Komposition (*Stack.java*)

```
1  public Stack {
2      private Vector v;                    // Komposition
3      public Stack () {
4          v = new Vector();                // ... deshalb
5      }
6      public boolean isEmpty() {
7          return v.isEmpty();
8      }
9      public void push( Object o ) {
10         v.add( o );
11     }
```

[2] Gamma et al. haben die Begriffe Aggregation und Komposition genau anders herum definiert. Wir halten uns aber an den UML-Standard.

```
12      public Object pop() {
13          return v.remove(v.size()-1);
14      }
15  }
```

Assoziationen, Aggregationen und Kompositionen können mit *Multiplizitäten* an beiden Enden annotiert sein (siehe Abbildung 2.10). In der Abbildung bedeutet dies bei (a), dass es zu jedem

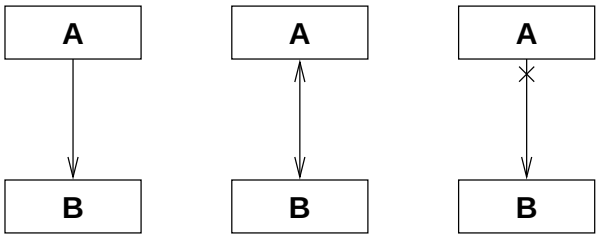


(a) (b) (c)

Abbildung 2.10: Assoziationen mit Multiplizitäten

A-Objekt genau m B-Objekte gibt und zu jedem B-Objekt genau n A-Objekte, bei (b), dass es zu jedem A-Objekt beliebig viele B-Objekte gibt und zu jedem B-Objekt genau ein A-Objekt, und bei (c), dass es zu jedem A-Objekt mindestens n und höchstens m B-Objekte und zu jedem B-Objekt maximal ein A-Objekt gibt. Im Zusammenhang mit *Kompositionen und Multiplizitäten* gibt es eine wichtige Einschränkung: Die Multiplizität bei A in Abbildung 2.9c muss immer 1 sein, da ein B-Objekt immer zu genau einem A-Objekt gehört.

Außerdem kann man durch einen Pfeil am Assoziationsende die *Navigierbarkeit* bzw. durch ein Kreuzchen die *Nicht-Navigierbarkeit* anzeigen. Eine Pfeilspitze bei B, d. h. ein Pfeil von A nach B, bedeutet, dass jedes A-Objekt zu „seinen“ B-Objekten navigieren kann. Ein Kreuzchen am Pfeilende bei B bedeutet, dass man eben gerade von den A-Objekten nicht zu den B-Objekten gelangen kann. Fehlen Kreuzchen und Pfeil, so ist die Navigierbarkeit unspezifiziert. Abbildung 2.11 enhält ein paar Beispiele. In (a) kann man von den A-Objekten zu den B-Objekten gelangen. Über



(a) (b) (c)

Abbildung 2.11: Navigierbare Assoziationen

die andere Richtung ist nichts ausgesagt. Bei (b) kann man sowohl von den A-Objekten zu den B-Objekten, als auch umgekehrt gelangen. In (c) schließlich kann man nur von den A-Objekten zu den B-Objekten gelangen, aber nicht umgekehrt. Bei *Aggregationen und Kompositionen* kann natürlich an der Stelle der Raute kein Kreuzchen oder Pfeil stehen. Dies wäre auch nicht sinnvoll bei einer solchen Beziehung.

Eine Assoziation kann man außerdem noch mit einem *Namen* versehen und den beiden Partnern der Assoziation kann man *Rollen* in dieser Assoziation zuteilen (siehe Abbildung 2.12). Die As-

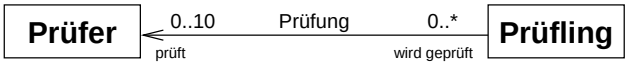


Abbildung 2.12: Assoziationen mit Namen und Rollen

soziation in der Abbildung heißt „Prüfung“. Der Prüfer nimmt dabei die Rolle „prüft“ und der Prüfling nimmt die Rolle „wird geprüft“ ein.

Außerdem kann man einer Assoziation auch noch eine *Assoziationsklasse* zuordnen (siehe Abbildung 2.13). Auf diese Weise wird jeder Ausprägung der Assoziation ein Objekt der Assoziations-

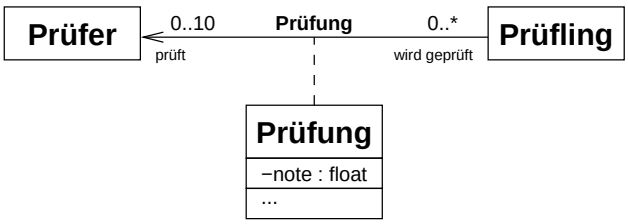


Abbildung 2.13: Assoziationen mit Assoziationsklasse

klasse zugeordnet. Auf diese Weise kann man im Beispiel die Note zu einer Prüfung speichern, was weder beim Prüfer noch beim Prüfling ginge.

2.3.7 Abhängigkeitsbeziehung

In Abbildung 2.7 haben wir bereits eine *Abhängigkeitsbeziehung* zwischen einer Klasse und einer Schnittstelle gesehen.

2.4 Paketdiagramme

Ein *Paket* wird in einem Paketdiagramm wie ein Registerblatt dargestellt (siehe Abbildung 2.14). Ein Paket kann *Elemente* wie Unterpakete und Klassen enthalten. Diese können mit einem „-“ für Paketsichtbarkeit bzw. „+“ für globale Sichtbarkeit versehen sein (siehe Abbildung 2.15). Ist ein Paket leer, so muss der Name nicht auf der Lasche stehen, sondern er kann auch im Innern stehen. Als Beziehungen zwischen Paketen sind für uns die Abhängigkeiten mit den Stereotypen *access und import* von Bedeutung. Bei „import“ findet ein öffentlicher Import des anderen Paketes statt (d. h. die Elemente des importierten Paketes sind danach auch von außen im importierenden Paket sichtbar). Bei „access“ findet im Gegensatz dazu ein privater Import statt, bei dem die importierten Pakete nicht von außen sichtbar sind (siehe Abbildung 2.16). Paketdiagramme können beispielsweise sehr gut verwendet werden, um die *Schichten einer Architektur* zu modellieren (siehe Abbildung 2.17). Ein solches Diagramm kann dann nach und nach verfeinert werden um weitere Elemente (Pakete und Klassen) in den Schichten und so nimmt der Detaillierungsgrad immer mehr zu.

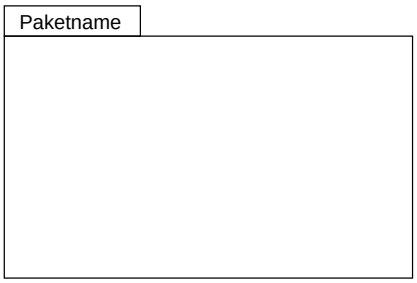


Abbildung 2.14: Darstellung eines Paketes

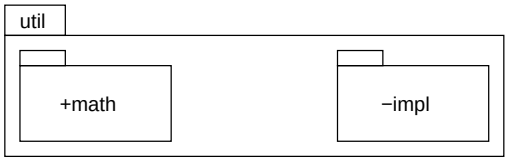


Abbildung 2.15: Paket mit Unterpaketen unterschiedlicher Sichtbarkeit

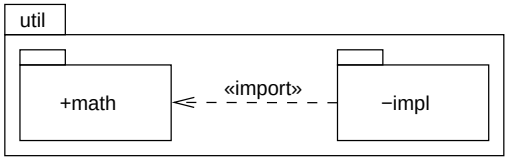


Abbildung 2.16: Paket mit Unterpaketen und Importbeziehungen

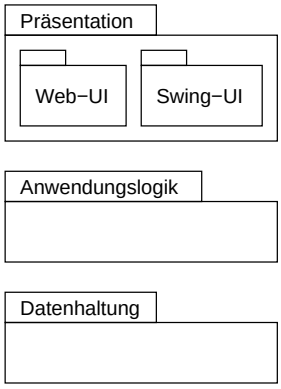


Abbildung 2.17: Schichten einer Architektur als Paketdiagramm

2.5 Objektdiagramme

Durch ein *Objektdiagramm* kann man einen *Systemzustand* bestehend aus *Objekten* (Instanzen von Klassen), *Verknüpfungen* (Ausprägungen von Assoziationen) und *Attributwerten* darstellen. In Abbildung 2.19 ist ein Objektdiagramm zum Klassendiagramm in Abbildung 2.18 dargestellt. Die im Objektdiagramm dargestellte Situation beschreibt den Prüfling Hans Muster, der mo-

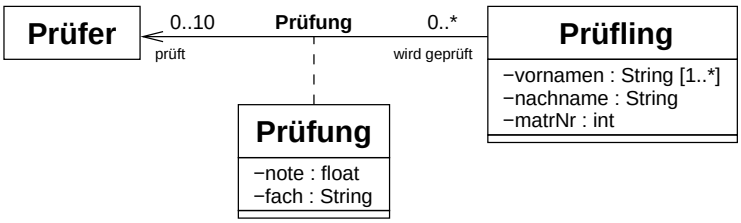


Abbildung 2.18: Klassendiagramm zu Prüfungen

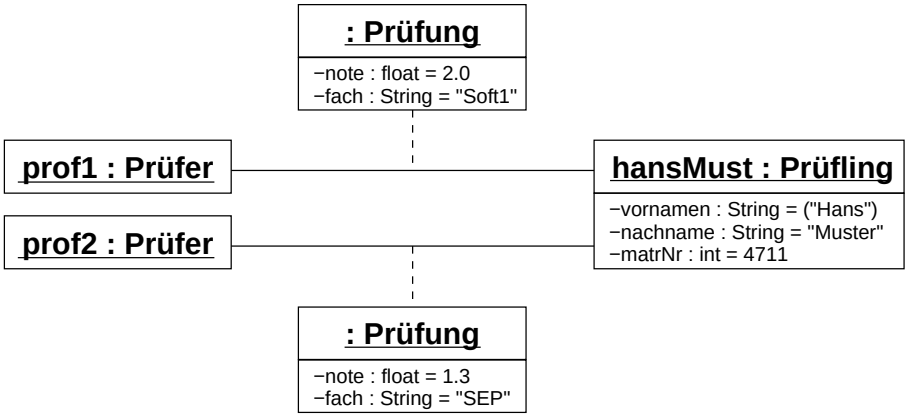


Abbildung 2.19: Objektdiagramm zu Prüfungen

mentan zwei Prüfungen bei zwei Prüfern abgelegt hat. Es gibt also Objekte für den Prüfling und die Prüfer. Diese werden analog zu Klassen in Kästchen dargestellt, wobei jetzt der Klassenname unterstrichen wird, und der Name des Objektes (optional) sowie ein Doppelpunkt vorangestellt werden. Da es sich um Objekte handelt, haben die Attribute ganz konkrete Werte. Außerdem stellt es auch kein Problem dar, dass Prüfer zweimal auftaucht – es handelt sich um zwei verschiedene Prüfer-Objekte. Aus der Assoziation zwischen Prüfer und Prüfling wurden jetzt die beiden Verknüpfungen, die durch Linien gezeichnet sind. An diesen Verknüpfungen „hängen“ die Verknüpfungsobjekte, die diese näher beschreiben (durch die Note und das geprüfte Fach).

2.6 Sequenzendiagramme

Die bisher betrachteten Diagramme haben sich allein auf die Struktur konzentriert. Das *Sequenzendiagramm* hingegen beschreibt den zeitlichen Ablauf von *Nachrichten* (siehe Abbildung 2.20). Eine Nachricht kann folgendes sein:

- Aufruf einer Operation
- Rücksprung aus einer Operation
- Ereignis
- Signal (z. B. Zeitgeber)

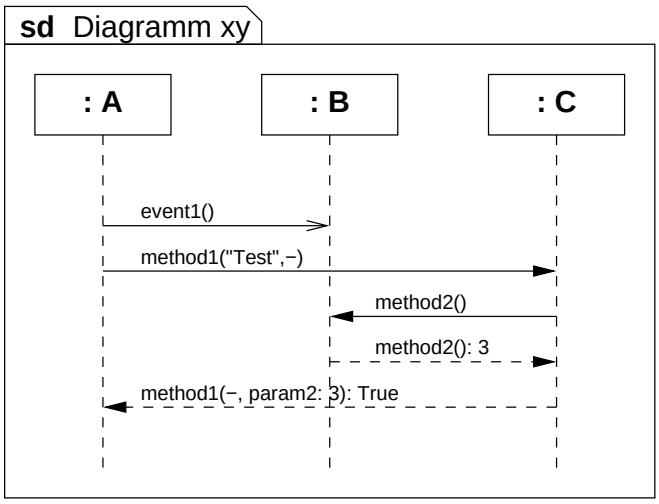


Abbildung 2.20: Sequenzendiagramm

Der *zeitliche Verlauf* ist im Sequenzendiagramm vertikal dargestellt, d. h. die vertikale Achse ist die Zeitachse. Je weiter unten etwas steht, desto später kommt es.

Die *Lebenslinie* wird in einem Sequenzendiagramm durch das Kästchen für das Objekt und die anschließende gestrichelte Linie dargestellt.

Es gibt zwei Formen von Nachrichten:

- *Synchrone Nachrichten* werden durch einen Pfeil mit ausgefüllter Spitze dargestellt. Der Sender wartet so lange, bis er die *Antwortnachricht* (gestrichelter Pfeil mit ausgefüllter Spitze) erhält.
- *Asynchrone Nachrichten* werden durch einen Pfeil mit offener Spitze dargestellt. In diesem Fall wartet der Sender nicht.

Die Pfeile der Nachrichten werden annotiert mit folgender Syntax:

```
Nachricht(Parameterwert {, Parameterwert})
```

Als Parameter werden dabei die Werte der Parameter beim Aufruf angegeben.

Für die Antwortnachricht gilt folgende Syntax:

```
Nachricht(Name ":" Parameterwert {, Name ":" Parameterwert}): Rückgabewert
```

Hier wird immer noch der Name des (in)out-Parameters zum Wert mit angegeben.

Interessiert ein Parameterwert nicht, oder kann er nicht angegeben werden, so kann er schlicht durch „–“ ersetzt werden.

2.7 Literatur

[Jeckle04] Jeckle, M., Rupp, C., Hahn, J., Zengler, B., Queins, S.: *UML 2 glasklar.* Carl Hanser Verlag, 2004.

[Oesterreich04] Oesterreich, B.: *Objektorientierte Softwareentwicklung: Analyse und Design mit der UML 2.0.* 6. Auflage, Oldenbourg Verlag, 2004.

Kapitel 3

Relationale Datenbanken, SQL, MySQL und JDBC

3.1 Einführung

Datenbank-Management-System (**DBMS**): Software zur Verwaltung / zum Management großer Datenmengen in einem Daten-Pool (siehe Abb. 3.1)

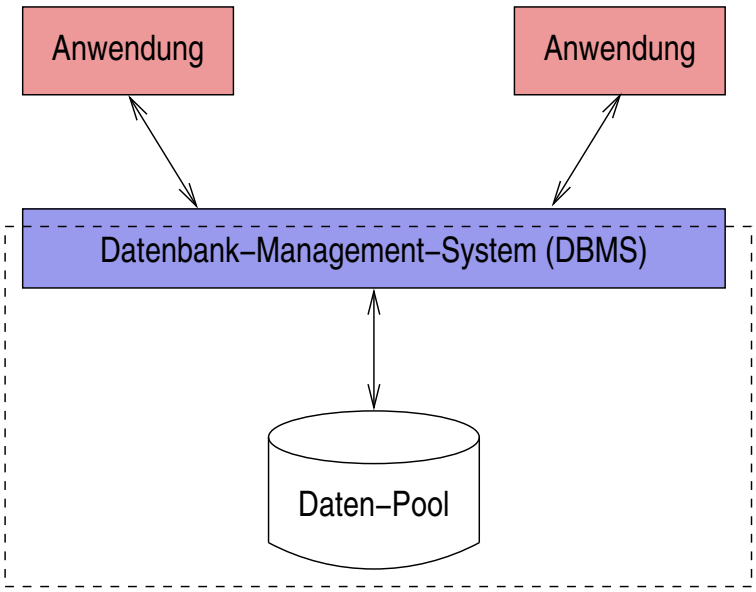


Abbildung 3.1: Datenbank-Management-System (DBMS)

Entwicklung:

- *Ursprünglich:* Anwendungs-Programme, bei denen der *Datenpool eine Menge von Dateien* darstellte und die Programme entsprechend ihrer Funktionalität direkt auf diese Dateien zugriffen

- Dazu mussten die Programme exakt die *Struktur der Daten in den Dateien kennen* (⇝ *Daten-Programm-Abhängigkeit*):
  - Änderungen in den Datenstrukturen ziehen Änderungen in den Programmen nach sich und umgekehrt
  - aus Effizienzgründen wurden dazu Dateien in entsprechender Struktur aufbereitet, was zu einer entsprechenden *Redundanz* in den Daten führt (ein- und dasselbe Datum war mehrfach gespeichert, was u. A. bei Änderungen zu *Inkonsistenzen* führen kann).

Sicht auf Daten in Form von Tabellen (siehe Abb. 3.2):

Artikel	Artikel-#	Bezeichnung	Gewicht	Preis
	13	Schraube M1	11	33
	27	Schraube M3	15	64
	2	Nagel N12	8	11

Kunde	Kunden-#	Name	Ort	Bonität
	K12	Fa. Meyer	Illertissen	gut
	K03	Fa. Hug	Ulm	mittel
	K46	Fa. Huber	Breitenthal	mies

Abbildung 3.2: Tabellen-Sicht

- *Darstellung einzelner Datensätze* (d. h. die Daten zu / *Attribute* von einem abgrenzbaren, realen Objekt) als Zeile in der Tabelle; Bsp.: Daten, die einen Kunden oder Artikel beschreiben; als *Entität* („Wesenseinheit“) oder *Entity* bezeichnet; die Menge aller möglichen konkreten Entitäten gleichen Typs werden als *Entity-Typ* bezeichnet
- *Darstellung der Beziehungen* zwischen den Datensätzen verschiedener Entitäten (Verbindung eines Kunden zu einem konkreten Artikel) – wie wird ausgedrückt, dass ein bestimmter Kunde einen bestimmten Artikel in einer bestimmten Menge zu bestimmten Konditionen bestellt hat? (⇝ Tabelleneintrag mit den Fremdschlüsseln, die Kunde und Artikel identifizieren)

Sichten in DBMS:

- *interne Sicht* (*internes Modell*, beschrieben im internen Schema): tatsächliche Speicherung auf physikalischem Medium
- *logische Sicht* (*logisches Modell*, beschrieben im logischen Schema): die vom jeweiligen DBMS zur Verfügung gestellte Sichtweise (Datenbeschreibungssprache – *data definition language, DDL*)

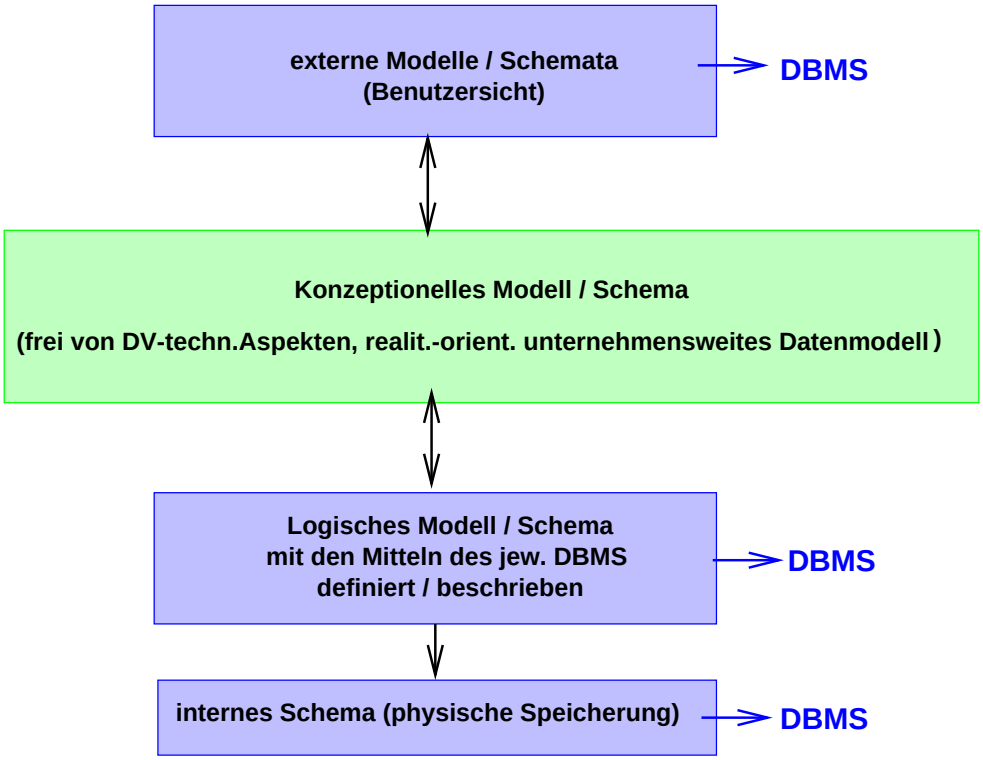


Abbildung 3.3: DBMS – Sichten

- *konzeptionelle Sicht* (*konzeptionelles Modell*, beschrieben im konzeptionellen Schema): DBMS-unabhängig, übergreifend, deshalb häufig auch als „unternehmensweites Datenmodell“ bezeichnet
- *externe Sicht* (*externes Modell*, beschrieben im externen Schema): abgeleitete Sicht für spezielle Nutzer-Bedürfnisse (z. B. andere Zusammenstellung, abgeleitete Daten)

3.2 DBMS – Modelle

Unterscheidung nach logischem Modell:

- *Hierarchische Datenbanken* (z. B. IMS/VS): Hierarchische Anordnung der Entity-Typen entlang von „enthält-Beziehungen“
- *Netzwerk-Datenbanken* (z. B. CODASYL): definierte Einstiegspunkte ($\sim$ Entity-Typen) und „Verdrahtung“ der Beziehungen zwischen Entity-Typen zur Navigation (weniger verbreitet)
- *Relationale Datenbanken*: Daten in Form vom Relationen ($\rightsquigarrow$ Tabellen-Sicht) – diese werden im folgenden wegen ihrer aktuellen Bedeutung näher behandelt
- *Objekt-relationale bzw. objekt-orientierte Datenbanken*: aktuelle Entwicklungen, noch wenig verbreitet

3.3 Einige Kennzeichen relationaler DBMSe

- *Entity-Typen* (und ggf. auch *Beziehungen*) werden in Form von *Tabellen* (Relationen) beschrieben
- Alle Tabellen sind gleichberechtigt
- *Spalten* (= *Attribute*) werden wie Tabellen nur über ihre *Namen* angesprochen
- Die Zeilen einer Tabelle bilden eine *Menge* ($\rightsquigarrow$ Relation)
- Zu jeder Tabelle gibt es (mind.) einen *Schlüssel*, d. h. eine minimale Spaltenkombination, die eine Zeile eindeutig identifiziert. Eine Tabelle kann mehrere Schlüssel haben.
  Unterscheidung: *Schlüssel-* / *Nicht-Schlüsselattribute*
- Die *Reihenfolge* der Datensätze spielt keine Rolle (z. B. der „5-te“ Eintrag)
- *Beziehungen* zwischen Entity-Typen können *über Wertgleichheit in Spalten* (Beziehungsattribute) beschrieben werden
  $\rightsquigarrow$ *Fremdschlüssel* (siehe Abb. 3.4)

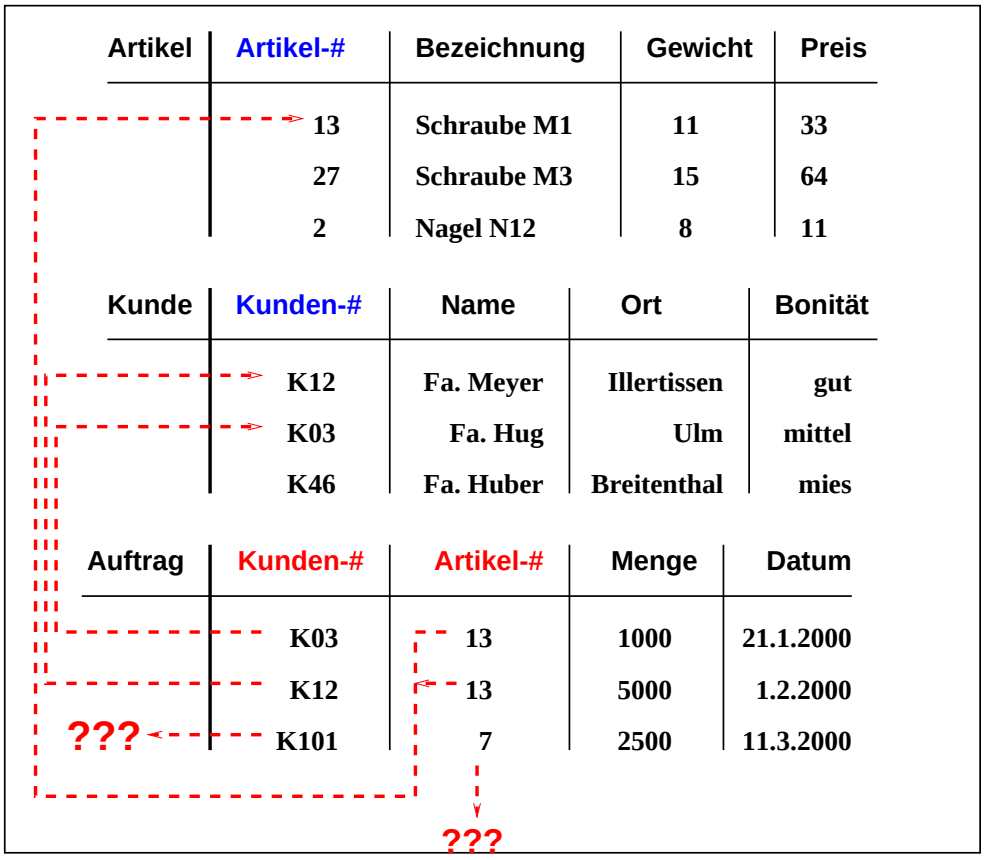


Abbildung 3.4: Von Tabelle zu Tabelle

3.4 Einige generelle Anforderungen an ein DBMS

- **Unterstützung von NULL-Werten**:

Insb. bei Attributen, die als Wertebereich ganze Zahlen haben, muss zwischen dem *Zahlenwert 0* und der Tatsache, dass *nichts* eingegeben wurde, unterschieden werden können. Manche Datenbanken stellen dieses *Nichts* oder *unbekannt* intern durch die Zahl *0* (*default*-Wert) dar.

Problem am Beispiel: Für das Attribut Alkoholkonsum (g/Tag) ist die Unterscheidung zwischen 0 g/Tag und unbekannt u. U. essentiell wichtig!

- **Transaktionskonzept**:

  Datenbankoperationen bestehen oft aus mehreren elementaren Operationen, die nacheinander (oder auch parallel) ausgeführt werden. Aus Sicht der Anwendung müssen diese aber als *atomare (= unteilbare) Einheit* stattfinden, d. h. alle diese elementaren Operationen müssen entweder vollständig oder dürfen überhaupt nicht ausgeführt werden. Anwendungsoperationen müssen von einem konsistenten wieder in einen konsistenten Zustand überführen.

  Problem am Beispiel: Bei einer Buchung sind zwei Konten (also zwei Tabellen) betroffen; elementar besteht eine Buchung aus zwei Operationen – würde nur die erste korrekt ausgeführt, die zweite aus welchem Grund auch immer nicht, so wäre das Kontensystem inkonsistent!

- **Synchronisation**:

  Im Multi-User-Betrieb kann es vorkommen, dass mehrere Benutzer gleichzeitig lesend / schreibend auf denselben Datensatz zugreifen. Bei schreibenden Operationen muss der Zugriff exklusiv sein, das heißt, die Operationen müssen synchronisiert werden.

  Problem am Beispiel: Ein Benutzer will einen sehr langen Datensatz lesen, die Datenbank liefert zunächst die erste Hälfte, danach die zweite; dazwischen, also nach dem Lesen der ersten Hälfte, verändert ein anderer Benutzer den gesamten Datensatz; in Folge erhält der Lesende die erste Hälfte des alten und die zweite Hälfte des neuen Datensatzes ($\rightsquigarrow$ Inkonistenz)!

- **Recovery**:

  Beim Betrieb eines DBMS kann es durchaus zu Störungen („Systemabstürze“) kommen, die einen geregelten Neustart (oder Wiederanlauf) und damit verbunden ein Wiederherstellen eines konsistenten Systemzustandes verlangen. ($\rightsquigarrow$ korrekter Umgang mit noch nicht abgeschlossenen Transaktionen)

- **Vergabe von Zugriffsrechten**:

  Nicht jeder Benutzer soll alles „dürfen“, d. h. bezogen auf einzelne Benutzer werden Rechte beim Zugriff auf Tabellen oder einzelne Datensätze oder einzelne Felder eingeschränkt (nur lesen, kein ändern, . . .)!

- **Zugriffsbeschleunigung**:

  Die Suche nach einem durch einen Schlüssel ausgewählten Datensatz kann linear erfolgen, d. h. die Datensätze werden sukzessive nach dem definierten Datensatz durchsucht (sehr langsam).

  Statt dessen kann über diesen Schlüssel eine andere Zugriffsorganisation gelegt werden:

  - *Hash-Organisation*: aus dem angegebenen Schlüssel wird (direkt) die Speicherposition berechnet ($\rightsquigarrow$ sehr schneller Zugriff, Problem von Kollisionen)
  - *Index* über diesen Schlüssel (anschaulich wie in einem Lexikon); Methode: B-Bäume etc. ($\rightsquigarrow$ erhebliche Beschleunigung beim Zugriff)

Integrität bedeutet im Zusammenhang mit DBMS die *Konsistenz (Widerspruchsfreiheit)* aller Daten in sich und zur realen Welt.

- **Objekt-Integrität**: Der Schlüssel muss stets eindeutig und vollständig definiert sein!
- **Beziehungsintegrität** (siehe Abb. 3.4)
  - In *Auftrag* können in *Kunden-#* und *Artikel-#* nur Werte eingetragen werden, die in den entsprechenden Spalten in *Kunde* bzw. *Artikel* bereits definiert sind.
  - In *Kunde* kann ein Datensatz nur dann gelöscht werden, wenn es zu dieser *Kunden-#* keinen Datensatz in *Auftrag* mehr gibt!

3.5 Normalformen

Ziel bei der Entwicklung von DBMS: u. A. Reduktion der *Redundanz* in den Daten

⇝ *Normalformen*

1. Normalform (**1NF**):
Alle Attribute sind elementar (besitzen also keine relevante Unterstruktur).

Beispiele für die Verletzung der 1NF:

- Adresse: Straße, Postleitzahl, Ort, ...
- Hobbys: einzelne Personen können eine unterschiedliche Zahl an Hobbys haben

2. Normalform (**2NF**):
Erste Normalform und zusätzlich: Alle Nichtschlüsselattribute hängen vom ganzen Schlüssel ab und nicht nur von Teilen davon.

Beispiel für die Verletzung der 2NF: In Abbildung 3.5 ist die Aufnahme der Bonität in die Tabelle Auftrag dargestellt. Dies verletzt die 2. Normalform, da Bonität nicht vom ganzen Schlüssel (Kunden-# und Artikel-#), sondern nur von der Kunden-# abhängt.

3. Normalform (**3NF**):
Erste Normalform und zusätzlich: Alle Nichtschlüsselattribute hängen unmittelbar (direkt) vom Schlüssel ab und nicht nur mittelbar (indirekt).

Beispiel für die Verletzung der 2NF: In Abbildung 3.6 führt die Aufnahme des Wochentages zur Verletzung der 3. Normalform, da sich der Wochentag aus dem Datum berechnen lässt.

Daraus lässt sich weiter folgende Regel ableiten:
Nur die Attribute in den (Basis-)Tabellen aufnehmen, die sich nicht aus anderen Attributen ableiten lassen!

Beachte: Es ist immer ein Kompromiss zu finden zwischen klaren Strukturen und Performance. Man kann und muss häufig gegen das Prinzip von klaren Strukturen verstoßen – dies sollte aber bewusst geschehen!

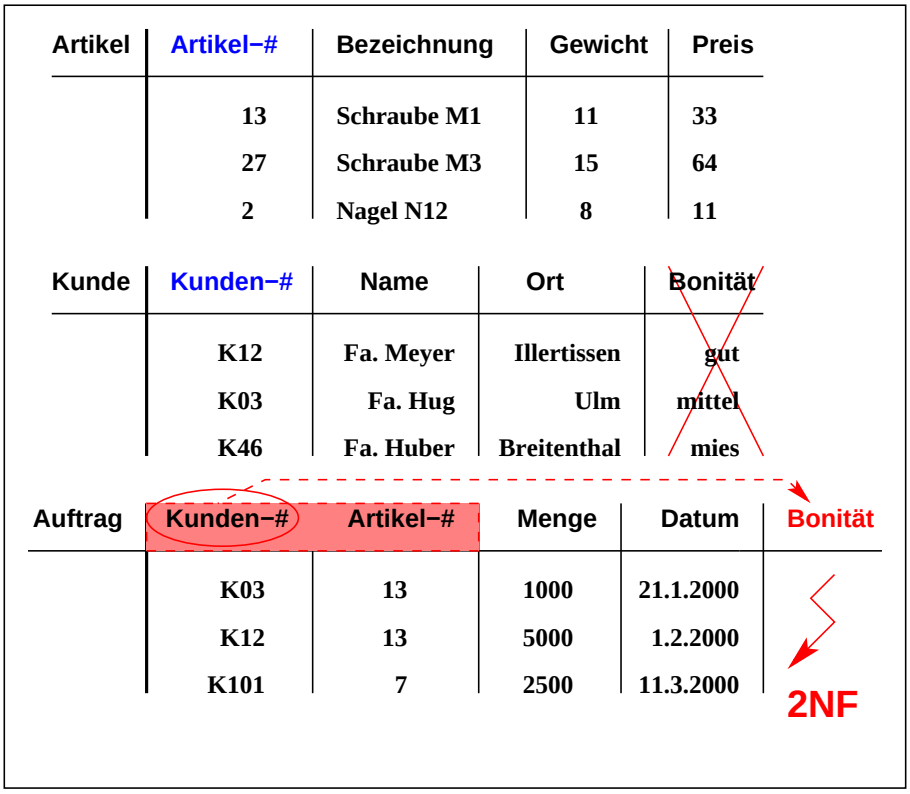


Abbildung 3.5: Verletzung der 2NF

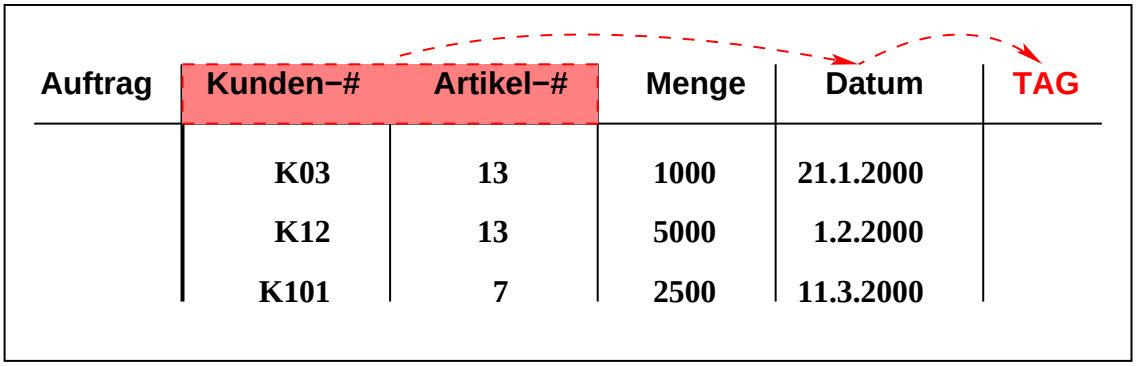


Abbildung 3.6: Verletzung der 3NF

3.6 SQL – Structured Query Language

- im Bereich *relationaler DBMSe* Standardsprache zur Datendefinition wie auch zur Datenmanipulation (und auch weitere Operationen)
- im Folgenden in Grundzügen beschrieben
- anhand der freien Datenbank *MySQL* (auch für Windows)

(Der folgende Text ist sehr eng an das Vorlesungsmanuskript *Unix-basierte Implementierung kleiner Datenbanken II* von Dr. A. Borchert, Abt. Angewandte Informationsverarbeitung der Universität Ulm, angelehnt.)

3.6.1 Etwas Historie

- Nach der *Einführung des relationalen Datenbankmodells von Codd (1969 und 1970)*, gab es zunächst nur wenige experimentelle Implementierungen, die insbesondere keine umfassende Abfragesprache offerierten.
- Von großer Bedeutung für alle nachfolgenden relationalen Datenbanken und insbesondere SQL war das *Forschungssystem System/R*, das in der Zeit von 1974 bis 1979 beim IBM San Jose Research Laboratory entwickelt worden ist.
- SQL wurde später von kommerziellen Anbietern relationaler Datenbanken von System/R übernommen und danach auch zu einem *formalen Standard*:

1986	SQL	ISO/TC 97/SC 21/WG 3 N 117 und ANSI X3.135-1986
1992	SQL-2	ISO/IEC 9075:1992 und ANSI X3.135-1992
1999	SQL-3	ISO/IEC 9075:1999

  Momentan aktuell: *SQL-2*, d. h. *SQL-92* wird von den meisten DBMSen mehr oder weniger vollständig unterstützt.
- Normalerweise wird SQL entweder interaktiv verwendet (über einen sogenannten Monitor, eine Art Shell für eine Datenbank) oder innerhalb von anderen Programmiersprachen (*embedded SQL*).

3.6.2 SQL vs. MySQL

- *MySQL* ist ein Produkt der schwedischen Firma MySQL AB, die von Anfang an ihre Quellen unter eine Open-Source-Lizenz (GPL) gestellt hat.
- MySQL orientiert sich an dem Standard von *SQL-2*, bietet jedoch keine vollständige Implementierung.
- Verschachtelte *select*-Anweisungen werden erst ab Version 4.1 unterstützt. Virtuelle Tabellen (Views) und Trigger gibt es erst ab Version 5 (noch experimentell). Transaktionen gibt es für einige Tabellenrepräsentierungen ab der Version 4. Fremdschlüssel gibt es seit Version 3.23.44 für eine der Tabellenrepräsentierungen.
- Bei einigen nicht unterstützten Konstrukten wird zwar die Syntax der entsprechenden Anweisungen akzeptiert, sie aber im weiteren ignoriert.
- Ursprünglich hat MySQL nicht das *Rechtevergabesystem* aus SQL-2 unterstützt. Stattdessen wurde die Rechte mit Tabellen repräsentiert, die normal mit SQL manipuliert werden können. Seit Version 4 gibt es auch *GRANT*. *GRANT* und normale Datenbank-Operationen auf den zugehörigen Tabellen können parallel verwendet werden.

3.6.3 SQL Übersicht (Auswahl)

Tabellenorientierte Anweisungen	
alter table	Änderungen der Definition einer Tabelle
create table	Erzeugen einer Tabelle
show columns	Anzeige der Definition einer Tabelle
drop table	Entfernen einer Tabelle mitsamt Inhalt
Tupelorientierte Anweisungen	
delete	Löschen von Tupeln
insert	Neueinfügen von Tupeln
select	Auswahl und Anzeige von Tupeln
update	Ändern von Tupeln
replace	Ersetzung von Tupeln
Sichten	
create view	Definition einer virtuellen Tabelle
drop view	Entfernen einer virtuellen Tabelle
Zugriffsbeschleunigung	
create index	Anlegen weiterer Indizes
drop index	Indizes entfernen
Vergabe von Zugriffsrechten	
grant	Zugriffsrechte vergeben
revoke	Zugriffsrechte zurücknehmen
Synchronisation	
lock tables	Eine Menge von Tabellen exklusiv reservieren
unlock tables	Freigabe gesperrter Tabellen
Transaktionen	
start transaction	Beginn einer Transaktion
commit transaction	Abschluß einer Transaktion
rollback transaction	Abbruch der Transaktion
savepoint	Definition von Zwischenpunkten, zu denen eine Rückkehr mit *rollback* möglich ist

3.6.4 Datentypen bei SQL (Auswahl)

In diesem Teil unterscheiden sich auf dem Markt befindliche DBMSe i.a. in einem mehr- oder weniger reichhaltigen Angebot. Die folgende Tabelle gibt einige grundlegende Datentypen:

integer	ganze Zahlen (32 Bit)
float	Gleitkommazahlen
char(n)	Zeichenkette mit der festen Länge $n \leq 255$
varchar(n)	Zeichenkette mit der maximalen Länge $n \leq 255$
date	Datum in dem Format *'YYYY-MM-DD'*
time	Uhrzeit im Format *'HH:MM:SS'*
datetime	Zeitstempel im Format *'YYYY-MM-DD HH:MM:SS'*
text	Text mit der maximalen Länge von 65535 Zeichen
blob	Binäre Zeichenfolge mit der maximalen Länge von 65535 Zeichen
enum('value', ...)	Zeichenkette, die nur einen der angegebenen Werte (einschließlich *null*) annehmen kann.
set('value', ...)	Menge aus den angegebenen Werten.

3.6.5 Repräsentierung von Daten

- Relationale Datenbanken bevorzugen aus Performance-Gründen *Datensätze fester Länge*. Deswegen ergibt sich mit Ausnahme von *text* und *blob* immer eine festgelegte Größe eines entsprechenden Datenfeldes.
- Typischerweise werden *bei text und blob* die eigentlichen *Daten separat abgelegt*. Bei dem entsprechenden Datenfeld verbleibt dann nur noch ein Zeiger.
- Aus diesem Grund unterliegen auch *text* und *blob* der Beschränkung, dass sie *nicht indizierbar sind und insbesondere nicht Teil eines Primärschlüssels* sein können.
- Nach außen hin ist normalerweise immer nur eine textuelle Repräsentierung der einzelnen Datentypen sichtbar.
- Sofern nicht explizit ausgeschlossen, können alle Datenfelder – ausgenommen natürlich Schlüsselfelder – den Wert *NULL* besitzen.

3.6.6 Syntax von SQL bzw. MySQL

- Alle *Schlüsselwörter* von SQL werden unabhängig von der Verwendung von *Klein- oder Großbuchstaben* erkannt, d. h. sowohl *CREATE TABLE* als auch *create table* sind zulässig.
- Bei (selbst definierten) *Namen für Tabellen und Spalten* ist *Groß- / Kleinschreibung* jedoch signifikant.
- MySQL unterstützt mehrzeilige Kommentare analog zu C (/* . . . */).
- *Zeichenketten* können entweder in *einfache Apostrophen* oder *Doppel-Apostrophen* eingeschlossen werden.

3.6.7 Beispiel-Tabellen

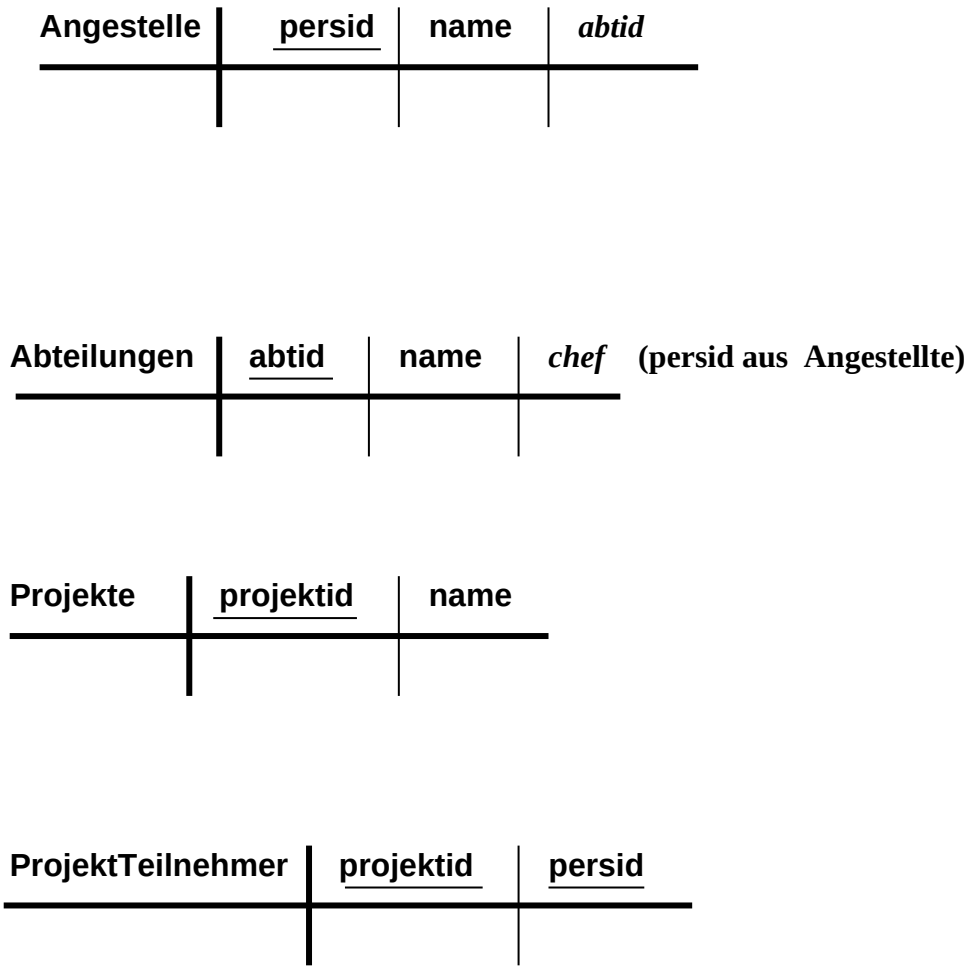


Abbildung 3.7: Beispiel-Tabellen

3.6.8 CREATE TABLE - Anlegen von Tabellen

Diese und die folgenden Syntax-Angaben wurden der MySQL-Dokumentation entnommen.

```
CREATE TABLE table_name ( create_definition,... )

create_definition:
  column_name type [NOT NULL | NULL]
                   [DEFAULT default_value]
                   [AUTO_INCREMENT]
                   [ PRIMARY KEY ]
                   [reference_definition]
  or    PRIMARY KEY ( index_column_name,... )
  or    KEY [index_name] KEY( index_column_name,...)
```

```
  or    INDEX [index_name] ( index_column_name,...)
  or    UNIQUE [index_name] ( index_column_name,...)
  or    FOREIGN KEY index_name ( index_column_name,...)
           [reference_definition]
  or    CHECK (expr)

index_column_name:
        column_name [ (length) ]

reference_definition:
        REFERENCES table_name [( index_column_name,...)]
                  [ MATCH FULL | MATCH PARTIAL]
                  [ ON DELETE reference_option]
                  [ ON UPDATE reference_option]

reference_option:
        RESTRICT | CASCADE | SET NULL | NO ACTION |
        SET DEFAULT
```

Programm 3.1: Anlegen der Beispieltabellen (*create-tables*)

```
1  CREATE TABLE Angestellte (
2      persid   VARCHAR(32) NOT NULL PRIMARY KEY,
3               INDEX (persid),
4      name VARCHAR(255) NOT NULL,
5      abtid    VARCHAR(32) NOT NULL REFERENCES Abteilungen
6  );
7
8  CREATE TABLE Abteilungen (
9      abtid    VARCHAR(32) NOT NULL PRIMARY KEY, INDEX (abtid),
10     name VARCHAR(255) NOT NULL,
11     chef     VARCHAR(32) REFERENCES Angestellte
12 );
13
14 CREATE TABLE Projekte (
15     projektid    VARCHAR(32) NOT NULL PRIMARY KEY,
16        INDEX (projektid ),
17     name VARCHAR(255) NOT NULL
18 );
19
20 CREATE TABLE ProjektTeilnehmer (
21     projektid    VARCHAR(32) NOT NULL REFERENCES Projekte,
22        INDEX (projektid ),
23     persid   VARCHAR(32) NOT NULL REFERENCES Angestellte,
24        INDEX (persid),
25     PRIMARY KEY (projektid, persid)
26 );
```

Erläuterungen zu Programm 3.1:

- Wenn der *Primärschlüssel* nur aus *einem einzigen Attribut* besteht, kann *PRIMARY KEY* direkt bei der Spezifikation des Attributs angegeben werden. Bei *nicht-elementaren Primärschlüsseln* ist eine *extra Definition* notwendig, die die beteiligten Attribute aufzählt (siehe *ProjektTeilnehmer*).

- Bei *Primärschlüsseln und Fremdschlüsseln* zu einer Beziehung mit Komplexitätsgrad 1 ist darauf zu achten, dass *NOT NULL* angegeben wird.
- MySQL (und auch andere dem SQL-2 Standard entsprechende Datenbanken) achten darauf, dass die *Primärschlüssel auch wirklich eindeutig* bleiben. Wenn diese Eigenschaft auch für andere Attribute gewünscht wird, dann kann dies mit *UNIQUE* angegeben werden.
- Die Angaben von Beziehungen (*REFERENCES*), die zur Erhaltung und Überprüfung der Konsistenz dienen, werden von MySQL ignoriert. Dennoch sind sie alleine schon aus Gründen der Dokumentation sinnvoll.
- Indizes werden von MySQL unterstützt, wenn sie bei *CREATE TABLE* gewünscht werden oder sie später bei *ALTER TABLE* hinzugefügt (oder auch entfernt werden). Im Gegensatz dazu wird die Anweisung *CREATE INDEX* ignoriert.

3.6.9 INSERT – Einfügen von Datensätzen

```
   INSERT INTO table [ (column_name,...) ]
   VALUES (expression,...)
or
   INSERT INTO table [ (column_name,...) ] SELECT ...
```

- Mit der *ersten Variante* wird *ein Datensatz eingefügt*, während mit der zweiten viele Datensätze auf einmal generiert werden können.
- Normalerweise werden *vollständige Tupel* angegeben, aber es ist auch möglich, eine Teilmenge zu spezifizieren – der Rest wird dann (soweit zulässig!) auf *NULL* gesetzt.
- Die *Reihenfolge der Werte* entspricht genau der Reihenfolge der Attributdefinitionen der *CREATE TABLE-Anweisung*.

Programm 3.2: Insert – Datensatz einfügen (*insert-tupel*)

```
1 INSERT INTO Angestellte VALUES
2    (1, 'Franz Schweiggert', 'SAI');
```

3.6.10 SELECT – Selektieren von Datensätzen

```
SELECT [STRAIGHT_JOIN] [DISTINCT | ALL] select_expression,...
   [INTO OUTFILE 'file_name' ...]
   [ FROM table_references [WHERE where_definition]
      [GROUP BY column,...]
      [HAVING where_definition]
      [ ORDER BY column [ASC | DESC] ,..]
      [LIMIT [offset,] rows] [PROCEDURE procedure_name]
   ]
```

- *select_expression* besteht im einfachen Falle aus einer durch Kommata getrennten *Aufzählung von Feldnamen* oder aus *, falls alle Attribute gewünscht werden.
- Werden bei *FROM mehrere Tabellen* angegeben, so müssen *Feldnamen mit dem Namen der Tabelle qualifiziert* werden, um Eindeutigkeit herzustellen.
- Statt Feldnamen sind auch Konstanten oder auch kompliziertere Ausdrücke einschließlich eine Reihe interessanter Funktionen möglich.
- *where_definition* ist ein beliebiger Ausdruck mit einem *Booleschen Wert*. Alle Datensätze bzw. alle Kombinationen von Datensätzen (wenn mehrere Tabellen angegeben sind), für die diese Bedingung zutrifft, werden selektiert und in der von *select_expression* spezifizierten Form zurückgeliefert.

Beispiele:

```
mysql> SELECT * FROM Angestellte;
+--------+------------------+-------+
| persid | name             | abtid |
+--------+------------------+-------+
| 1      | Franz Schweiggert | SAI   |
| 2      | Andreas Borchert  | SAI   |
| 3      | Johannes Mayer    | SAI   |
+--------+------------------+-------+
3 rows in set (0.01 sec)
```

Anmerkung: * ist eine Kurzform für alle Feldnamen der Tabelle *Angestellte* (in der selben Reihenfolge wie bei CREATE TABLE angegeben).

```
mysql> SELECT name, abtid FROM Angestellte;
+------------------+-------+
| name             | abtid |
+------------------+-------+
| Franz Schweiggert | SAI   |
| Andreas Borchert  | SAI   |
| Johannes Mayer    | SAI   |
+------------------+-------+
3 rows in set (0.01 sec)
```

Anmerkung: Natürlich können auch weniger Felder gewünscht werden und die Reihenfolge beliebig bestimmt werden.

```
mysql> SELECT * FROM Angestellte WHERE persid >= 2;
+--------+-----------------+-------+
| persid | name            | abtid |
+--------+-----------------+-------+
| 2      | Andreas Borchert | SAI   |
| 3      | Johannes Mayer   | SAI   |
+--------+-----------------+-------+
2 rows in set (0.02 sec)
```

Anmerkung: Bei *where* können beliebige Bedingungen zur Selektion angegeben werden.

```
mysql> SELECT Angestellte.name, Angestellte.abtid
    ->    FROM Angestellte, Abteilungen
    ->    WHERE Angestellte.persid = Abteilungen.chef;
+------------------+----------+
| name             | abtid    |
+------------------+----------+
| Franz Schweiggert | SAI      |
| Frank Stehling    | WiWi     |
| Volker Schmidt    | Stoch    |
+------------------+----------+
3 rows in set (0.02 sec)
```

Anmerkung: Bei einer *Verknüpfung von zwei (oder mehr Tabellen)* wird das *kartesische Produkt* gebildet und dann entsprechend der *WHERE*-Bedingung selektiert. Dank der Optimierungstechniken relationaler Datenbanken ist dies weniger teuer als es sich anhört. Trotzdem ist es natürlich wichtig, dass für Fremdschlüsselzugriffe entsprechende Indizes vorhanden sind.

```
mysql> SELECT Angestellte.name, Projekte.name
    ->    FROM Angestellte, Projekte, ProjektTeilnehmer
    ->    WHERE
    ->    Angestellte.persid = ProjektTeilnehmer.persid AND
    ->    ProjektTeilnehmer.projektid = Projekte.projektid;
+------------------+-------------------------------------+
| name             | name                                |
+------------------+-------------------------------------+
| Franz Schweiggert | Implementierung kleiner Datenbanken |
| Andreas Borchert  | Implementierung kleiner Datenbanken |
| Andreas Borchert  | Ulmer Oberon-System                 |
| Franz Schweiggert | GeoStoch                            |
| Johannes Mayer    | GeoStoch                            |
+------------------+-------------------------------------+
5 rows in set (0.04 sec)
```

Anmerkungen:

- Diese Abfrage liefert alle Projekte mitsamt ihren Projektteilnehmern.
- Wegen der zwischenliegenden Tabelle *ProjektTeilnehmer*, ist dafür eine zweifache *Tabellenverknüpfung (Join)* notwendig.

Beispiel für eine geschachtelte Select-Anweisung (Sub-Query): (erst ab Version 4.1 in MySQL!)

```
SELECT Angestellte.name
FROM Angestellte
WHERE Angestellte.persid IN
      (SELECT Abteilungen.chef FROM Abteilungen);

+-------------------+
| name              |
+-------------------+
| Franz Schweiggert |
| Frank Stehling    |
| Volker Schmidt    |
+-------------------+
```

Erklärung: Selektiere die Namen der Angestellten, deren *persid* in der Menge (durch die innere *select*-Anweisung bestimmt) der Werte von *Abteilungen.chef* vorkommt.

Tupelvariable:

⇝ Referenzieren verschiedener Tupel, d. h. Datensätze, in einer Tabelle

Fragestellung: Wie heisst der Leiter der Abteilung, in der Herr Borchert (persid = 2) arbeitet?

```
mysql> SELECT ang1.name
    -> FROM Angestellte ang1, Angestellte ang2, Abteilungen
    -> WHERE ang2.persid = 2
    ->    AND
    ->    ang2.abtid = Abteilungen.abtid
    ->    AND
    ->    Abteilungen.chef = ang1.persid;
+-------------------+
| name              |
+-------------------+
| Franz Schweiggert |
+-------------------+
1 row in set (0.2 sec)
```

Gruppierung:

Aufgabe: Finde die Personal-Nummern (persid) aller Mitarbeiter, die an mehr als einem Projekt mitarbeiten!

```
mysql> SELECT pt.persid
    -> FROM ProjektTeilnehmer pt
    -> GROUP BY pt.persid
    -> HAVING COUNT(pt.projektid) > 1;
+--------+
| persid |
+--------+
| 1      |
| 2      |
+--------+
2 rows in set (0.02 sec)
```

Anmerkungen:

- Diese *group*-Klausel ermöglicht die *Zerlegung einer Tabelle* entsprechend den Werten eines Attributes: *die Tupel jeder entstehenden „Gruppe“ besitzen für dieses Attribut denselben Wert!*
- Dies kann sehr vielfältig verwendet werden, da es so möglich ist, mit Teilmengen (den Gruppen) als Einheiten zu arbeiten. Diese können nun als Ganzes angesprochen werden, z. B. so, dass die Gruppen gelöscht werden, die bestimmte Bedingung nicht erfüllen.
- Typischerweise bauen diese Bedingungen – formuliert in der *having*-Klausel – auf Standardfunktionen auf: Anzahl, Summe, Durchschnitt, ...

Standardfunktionen:

```
mysql> SELECT COUNT(*) FROM Angestellte
    ->         WHERE Angestellte.abtid = 'SAI';
+----------+
| COUNT(*) |
+----------+
|        4 |
+----------+
1 row in set (0.01 sec)
```

```
mysql> SELECT MIN(Angestellte.persid) FROM Angestellte
    ->           WHERE Angestellte.abtid = 'SAI';
+-------------------------+
| MIN(Angestellte.persid) |
+-------------------------+
| 1                       |
+-------------------------+
1 row in set (0.01 sec)
```

3.6.11 UPDATE – Aktualisieren von Datensätzen

```
UPDATE table
   SET column=expression,...
   [WHERE where_definition]
```

- Mit *UPDATE* ist es möglich, *einzelne Attribute* der mit *WHERE* selektierten Tupel zu *modifizieren*.
- *Wenn WHERE weggelassen* wird, bezieht sich die Operation auf *alle Tupel* der angegebenen Tabelle.
- Auf der *rechten Seite einer Zuweisung* bei *SET* kann der *Name eines Attributs* verwendet werden, der dann für den jeweils alten Wert steht.
- *Primärschlüssel* sollten nicht auf diese Weise aktualisiert werden, obwohl MySQL (und wohl auch andere relationale Datenbanken) dies akzeptieren. Besser wäre in so einem Fall die *REPLACE-Anweisung*.

Beispiel:

```
mysql> UPDATE Angestellte
    ->    SET abtid = 'WiWi'
    ->    WHERE persid = '3';
Query OK, 1 row affected (0.10 sec)
```

3.6.12 REPLACE – Das Ersetzen von Datensätzen

```
        REPLACE INTO table [ (column_name,...) ]
           VALUES (expression,...)
or      REPLACE INTO table [ (column_name,...) ]
           SELECT ...
```

REPLACE arbeitet *analog zu INSERT*, abgesehen davon, dass zuvor Datensätze mit den gleichen primären Schlüsseln (oder mit *UNIQUE* gekennzeichneten Feldern!) gelöscht werden.

Beispiel:

```
mysql> REPLACE INTO Angestellte VALUES
    ->     (2, 'Kurt Nachfolger', 'SAI');
Query OK, 2 rows affected (0.10 sec)
```

3.6.13 DELETE – Das Löschen von Datensätzen

```
DELETE [LOW_PRIORITY] FROM tbl_name
   [WHERE where_definition] [LIMIT rows]
```

Analog zu UPDATE bezieht sich eine DELETE auf alle Datensätze, wenn die WHERE-Klausel fehlt.

```
mysql> DELETE FROM Angestellte
    ->    WHERE persid <= 2;
Query OK, 2 rows affected (0.02 sec)
```

3.6.14 Import und Export von Daten

Man kann auch Daten in eine Textdatei exportieren, indem man bei SELECT ein *OUTFILE* angibt:

```
1 SELECT * FROM Angestellte
2    INTO OUTFILE "ang.dat" FIELDS TERMINATED BY ":";
```

Dabei wird der Inhalt der Tabelle Angestellte in eine „flache Textdatei“ „ang.dat“ exportiert, wobei die Datensätze durch Zeilenumbruch und die einzelnen Spalten durch „:“ (default: Tabulator) voneinander getrennt sind. (NULL-Werte werden als `\N` dargestellt.)

Die erzeugte Datei liegt im Verzeichnis der Datenbank; in unserem Beispiel also *mysql/mydb/*.

Diese Datei kann man auch wieder in die Tabelle „laden“:

```
1 LOAD DATA LOCAL INFILE "ang.dat"
2     INTO TABLE Angestellte FIELDS TERMINATED BY ":";
```

Das „local“ zeigt an, dass die Datei auf dem Client liegt.

3.6.15 Metainformationen

Es gibt ein paar Besondere SQL-Anweisungen, um Metainformationen zu „beschaffen“:

- *SHOW TABLES*
  Liefert die Namen aller vorhandenen Tabellen.
- *DESCRIBE <table>* bzw. *SHOW COLUMNS FROM <table>*
  Liefert die Beschreibung der Spalten der angegebenen Tabelle.
- *SHOW CREATE TABLE <table>*
  Liefert die CREATE TABLE-Anweisung für die angegebene Tabelle.
- *SHOW INDEXES FROM <table>*
  Liefert die Indizes, die für die angegebene Tabelle existieren.

3.7 Die JDBC-Schnittstelle

Java's Verbindung zur Welt der Datenbanksysteme ist die *JDBC-Schnittstelle*. JDBC steht dabei für *Java Database Connectivity*. Bisher haben wir ja SQL-Kommandos direkt an der „Kommandozeile“ von mysql abgesetzt. Mit Hilfe von JDBC ist es möglich, SQL-Anweisungen in Java-Programme einzubetten – sogenanntes *Embedded SQL*.

3.7.1 JDBC und MySQL

Wie die Bezeichnung „Schnittstelle“ schon andeuten soll, ist JDBC unabhängig von konkreten Datenbanken gehalten. Um nun konkret mit einer Datenbank zu arbeiten wird ein sogenannter *Treiber* benötigt. Dieser wird dann von JDBC intern aufgerufen, um die tatsächliche Kommunikation mit der Datenbank durchzuführen.

Momentan haben wir für MySQL den Treiber in *mysql-connector-java-3.0.15-ga-bin.jar*. Dieses Java-Archiv muss natürlich irgendwie im Klassenpfad vorkommen, sonst kann der Treiber nicht gefunden werden.

Programm 3.3: Laden des MySQL-Treibers (*LoadDriver.java*)

```
1  public class LoadDriver {
2      public static void main(String[] args) {
3          try {
4              // MySQL-Treiber über Java-Reflection laden
5              Class.forName("com.mysql.jdbc.Driver").newInstance();
6          }
7          catch (Exception e) {
8              // evtl. Exception transformieren
9              throw new RuntimeException("MySQL driver not found: "+e);
10         }
11     }
12 }
```

Anmerkungen:

- Die Treiber-Klasse wird über *Java-Reflection* versucht zu laden. Ein Fehler hierbei, z. B. Klassenname falsch geschrieben, fällt daher erst zur Laufzeit auf, da der Klassenname ja auch als String übergeben ist.
- Ist das Laden des Treibers erfolgreich, so *registriert* sich der Treiber beim *DriverManager*, der nun mit MySQL-URLS umzugehen weiss.

```
theseus$ javac LoadDriver.java
theseus$ java LoadDriver
Exception in thread "main" java.lang.RuntimeException: \
MySQL driver not found: java.lang.ClassNotFoundException: \
com.mysql.jdbc.Driver
        at LoadDriver.main(LoadDriver.java:9)
theseus$ . /home/sep2004/software/installed/setvars
theseus$ java LoadDriver
theseus$
```

Den MySQL-Treiber (mysql-connector-java-3.0.15-ga-bin.jar) also nicht vergessen, in den Klassenpfad aufzunehmen. Dies geht am einfachsten durch den Aufruf `. /home/sep2004/software/installed/setvars`. Der anfängliche Punkt gefolgt von einem Leerzeichen bedeutet, dass das Skript in der aktuellen Shell ausgeführt wird und somit die Variablen (⇝ CLASSPATH, PATH etc.) gesetzt werden.

Nachdem der Treiber geladen ist, können wir eine Verbindung zur MySQL-Datenbank aufbauen. Dazu müssen wir zunächst die Datei *setenv* im Verzeichnis der Datenbank ansehen:

```
theseus$ ls mysql/
ib_arch_log_0000000000  ibdata1                mysql
ib_logfile0             logfile                setenv
ib_logfile1             mydb
theseus$ cat mysql/setenv
MYSQL_UNIX_PORT=/home/thales/jmayer/mysql/socket
MYSQL_TCP_PORT=14317
theseus$
```

Der Eintrag *MYSQL_TCP_PORT* sagt uns, an welchem *TCP-Port* der MySQL-Server „lauscht“. Außerdem benötigen wir noch den *Benutzernamen* (sep) und das *Passwort*, welches wir beim Einrichten der Datenbank gesetzt haben (hier verwenden wir im Folgenden immer „test“ als Passwort). Eine letzte Info, die wir noch benötigen, ist der *Name der Datenbank*. Dieser lässt sich aber leicht finden, wenn man ins Datenbankverzeichnis sieht (hier haben wir „mydb“ verwendet – darum auch oben ein Verzeichniseintrag „mydb“).

Von Ihrem Browser kennen Sie sicher URLs der Form `http://www....`. Auch zur *Beschreibung der Datenbankverbindung* werden *URLs* verwendet. In unserem Beispiel sieht die URL wie folgt aus:

jdbc:mysql://theseus:14317/mydb?user=sep&password=test

Erläuterungen:

- Vor dem ersten Doppelpunkt steht das Protokoll. Es handelt sich also um eine JDBC-Verbindung.
- Dann folgt mittels „mysql“ die Auswahl des MySQL-Treibers, der natürlich zuvor – wie beschrieben – geladen werden muss.
- Durch „theseus:14317“ wird schließlich angegeben, zu welchem Rechner und TCP-Port der Treiber eine Verbindung aufbauen soll.
- Schließlich ist noch der Name der Datenbank (vor dem Fragezeichen) und der Benutzername (user) und das Passwort (password) wichtig, um die Verbindung zur Datenbank aufzubauen.

Über die angegebene URL kann man nun – via *DriverManager* – eine Verbindung herstellen und erhält dabei ein Objekt, das diese Verbindung bezeichnet.

Programm 3.4: Verbindung zu einer MySQL-Datenbank herstellen (*MySqlConnection.java*)

```
1  import java.sql.*;
2
3  public class MySqlConnection {
4      public static void main(String[] args) {
5          try {
6              // MySQL-Treiber über Java-Reflection laden
7              Class.forName("com.mysql.jdbc.Driver").newInstance();
8          }
9          catch (Exception e) {
10             // evtl. Exception transformieren
11             throw new RuntimeException("MySQL driver not found: "+e);
12         }
13
14         Connection con = null;
15
16         try {
17             String url = "jdbc:mysql://theseus:14317/mydb?user=sep&password=test";
18             // Verbindung herstellen
19             con = DriverManager.getConnection(url);
20
21             // ... mit der MySQL-Verbindung arbeiten ...
```

```
22          }
23          catch (SQLException e) {
24              System.err.println("Exception: "+e);
25          }
26          finally {
27              // in jedem Fall die Verbindung sauber beenden
28              if (con != null) {
29                  try { con.close(); }
30                  catch (Exception e) { }
31              }
32          }
33      }
34  }
```

```
theseus$ java MySqlConnection
Exception in thread "main" java.lang.RuntimeException: \
MySQL driver not found: java.lang.ClassNotFoundException: \
com.mysql.jdbc.Driver
        at MySqlConnection.main(MySqlConnection.java:11)
theseus$ . /home/sep2004/software/installed/setvars
theseus$ java MySqlConnection
Exception: java.sql.SQLException: Unable to connect to any hosts \
due to exception: java.net.ConnectException: Connection refused
...
theseus$ ./startdb
041115 16:30:13  InnoDB: Started
./usr/local/mysql/bin.orig/mysqld: ready for connections
theseus$ java MySqlConnection
theseus$
```

Damit eine Verbindung zur Datenbank möglich ist, muss der MySQL-Server natürlich gestartet sein.
Vergessen Sie nicht, eine Datenbankverbindung mittels finally-Klausel immer (!) sauber zu beenden.

Anmerkung: Alternativ kann man *Benutzername und Passwort* auch als zweiten und dritten Parameter bei *getConnection()* mit übergeben – anstatt diese in die URL zu codieren:
`con = DriverManager.getConnection(url, "sep", "test");`

3.7.2 Datenbankanweisungen in Java einbetten

Nachdem nun eine Verbindung zur Datenbank steht, können wir mit dem eigentlichen Arbeiten beginnen. JDBC unterscheidet zwischen *zwei verschiedenen Arten von Anweisungen*: solche, die Änderungen irgendwelcher Art durchführen (sog. Updates) und Anfragen (sog. Queries).

Zunächst muss man mit der Methode *createStatement()* über die Verbindung ein *Statement-Objekt* erzeugen. Mit diesem kann man dann über die Methode *executeUpdate()* eine ändernde Anweisung ausführen. Diese liefert als Rückgabewert eine Integer, die die *Anzahl der betroffenen Datensätze* angibt.

Eine INSERT-Anweisung gehört natürlich zu dieser Kategorie. Die vorigen Beispiele müssen nun um folgenden Code ergänzt werden, um eine INSERT-Anweisung auszuführen:

Programm 3.5: Ändernde SQL-Anweisung mit JDBC ausführen (*Update.java*)

```
17          String url = "jdbc:mysql://theseus:14317/mydb";
18          // Verbindung herstellen
19          con = DriverManager.getConnection(url, "sep", "test");
20
21          Statement stmt = null;
22
23          try {
24              // Objekt zum Ausführen von SQL-Anweisungen erzeugen
25              stmt = con.createStatement();
26
27              // SQL-Anweisung vorbereiten
28              String name = "Armin Rutzen";
29              String sql = "INSERT INTO Angestellte VALUES "
30                           +"(4, '"+name+"', 'SAI')";
31
32              // SQL-Anweisung ausführen
33              int count = stmt.executeUpdate(sql);
34
35              System.out.println(count+" Datensätze betroffen");
36          }
37          finally {
38              // Statement wieder sauber schlieSSen
39              if (stmt != null) {
40                  try { stmt.close(); }
41                  catch (SQLException e) { }
42              }
43          }
```

```
theseus$ java Update
1 Datensätze betroffen
theseus$ ./workondb
mysql> SELECT * FROM Angestellte;
+--------+--------------+-------+
| persid | name         | abtid |
+--------+--------------+-------+
| 4      | Armin Rutzen | SAI   |
+--------+--------------+-------+
1 row in set (0.00 sec)
```

Statement-Objekte muss man genauso wie Verbindungs-Objekte wieder schließen. Normalerweise passiert dies beim Schließen der Datenbankverbindung implizit, man kann sich bei einigen Datenbanken (z. B. IBM DB2) aber nicht darauf verlassen.

Ruft man die Methode *executeQuery()* bei einem *Statement-Objekt* auf, so kann man eine Anfrage ausführen. Eine Anfrage liefert bei Erfolg ein *ResultSet-Objekt*. Dieses ist eine *Iterator*, mit dem man die komplette Ergebnismenge durchlaufen kann. Am Anfang steht der Iterator *vor dem ersten Datensatz*. Nach einem *next()-Aufruf*, der als Rückgabewert die Antwort auf die Frage „Gibt es noch weitere Datensätze?“ liefert, kann man mit den Methoden *get...()* entweder indiziert durch den *Spaltennamen* oder durch den *Spaltenindex* (1, 2, ...) auf die Attribute des Datensatzes zugreifen. Dabei gibt es praktisch für jeden SQL-Datentyp eine entsprechende get-Methode – außerdem funktioniert *getString()* und *getObject()* bei jedem Typ.

Programm 3.6: Anfrage mit JDBC ausführen (*Query.java*)

```
17          String url = "jdbc:mysql://theseus:14317/mydb";
18          // Verbindung herstellen
19          con = DriverManager.getConnection(url, "sep", "test");
20
21          Statement stmt = null;
22
23          try {
24              // Objekt zum Ausführen von SQL-Anweisungen erzeugen
25              stmt = con.createStatement();
26
27              // SQL-Anweisung vorbereiten
28              String sql = "SELECT * FROM Angestellte";
29
30              // SQL-Anweisung ausführen
31              ResultSet rs = stmt.executeQuery(sql);
32
33              // ResultSet iterieren
34              while (rs.next()) {
35                  int persid = rs.getInt("persid");          // bei Index 1
36                  String name = rs.getString("name");        // bei Index 2
37                  String abtid = rs.getString("abtid");      // bei Index 3
38
39                  System.out.println("| "+persid+" | "+name+" | "+abtid+" |");
40              }
41          }
42          finally {
43              // Statement wieder sauber schlieSSen
44              if (stmt != null) {
45                  try { stmt.close(); }
46                  catch (SQLException e) { }
47              }
48          }
```

Zum Zugriff auf die Elemente kann man statt den Spaltennamen auch Indizes verwenden:

Programm 3.7: Anfrage mit JDBC ausführen (*Query1.java*)

```
33              // ResultSet iterieren
34              while (rs.next()) {
35                  int persid = rs.getInt(1);          // persid
36                  String name = rs.getString(2);      // name
37                  String abtid = rs.getString(3);     // abtid
38
39                  System.out.println("| "+persid+" | "+name+" | "+abtid+" |");
40              }
```

```
theseus$ java Query
| 1 | Franz Schweiggert | SAI |
| 2 | Andreas Borchert | SAI |
| 3 | Johannes Mayer | SAI |
theseus$ java Query1
| 1 | Franz Schweiggert | SAI |
| 2 | Andreas Borchert | SAI |
| 3 | Johannes Mayer | SAI |
theseus$ ./workondb
mysql> SELECT * FROM Angestellte;
+--------+------------------+-------+
| persid | name             | abtid |
+--------+------------------+-------+
| 1      | Franz Schweiggert | SAI   |
| 2      | Andreas Borchert  | SAI   |
| 3      | Johannes Mayer    | SAI   |
+--------+------------------+-------+
3 rows in set (0.00 sec)
```

Nach Aufruf einer get-Methode kann man durch Aufruf der Methode *wasNull()* des ResultSets ermitteln, ob es sich bei dem eben gelesenen Wert um einen NULL-Wert handelt. Bei einem Integer-Wert kann man dies nämlich nicht mehr feststellen:

Programm 3.8: Umgang mit NULL-Werten (*Query2.java*)

```
33              // ResultSet iterieren
34              while (rs.next()) {
35                  int id = rs.getInt("id");
36
37                  int count = rs.getInt("count");
38                  // war letzter mit get ...() gelesener
39                  // Wert eigentlich NULL?
40                  boolean count_null = rs.wasNull();
41
42                  System.out.print("id="+id+" count="+count);
43                  System.out.println(" (count=NULL? "+count_null+")");
44              }
```

```
theseus$ java Query2
id=1 count=0 (count=NULL? false)
id=2 count=0 (count=NULL? true)
theseus$ ./workondb
mysql> SELECT * FROM Dummy;
+----+-------+
| id | count |
+----+-------+
|  1 |     0 |
|  2 |  NULL |
+----+-------+
2 rows in set (0.00 sec)
```

3.7.3 Informationen über den ResultSet

Von einem *ResultSet* kann man mit der Methode

ResultSetMetaData getMetaData()

Meta-Daten erhalten. Diese beinhalten im Wesentlichen die Anzahl und Namen der Spalten. Die Klasse *ResultSetMetaData* besitzt hierzu die folgenden beiden Methoden:

- **int** *getColumnCount*()
  liefert die Anzahl der Spalten.
- *String getColumnName*(**int** *i*)
  liefert den Namen der *i*-ten Spalte.

3.7.4 Vorbereitete Abfragen/SQL-Anweisungen

Es gibt zwei gravierende Nachteile der eben vorgestellten Form, SQL-Anweisungen in Java einzubetten:

- Bei *mehreren gleichen Abfragen*, bei denen sich nur die Werte unterscheiden, dauert die Verarbeitung bisher unnötig lange.
  Beispiel: Sie möchten wie in Programm 3.5 eine Reihe von Datensätzen in die Tabelle Angestellte einfügen. Dann ist die SQL-Anweisung immer dieselbe. Nur die Werte ändern sich.
- Wenn Sie *Werte wie bisher in die SQL-Anweisung einbetten* (siehe z. B. Zeile 30 in Programm 3.5), dann ist das problematisch. Stellen Sie sich vor, dass im Beispiel name ein einfaches Hochkomma enthält. Dann ist an dieser Stelle der Namens-String zu Ende!

Um diese Probleme zu umgehen, kennt JDBC vorbereitete Anweisungen – *PreparedStatement*. Diese kann man mit der Methode *prepareStatement()* der Verbindung erzeugen. Der dabei übergebene String repräsentiert die SQL-Anweisung. Diese darf anstatt von Werten auch „?“ enthalten. Diese müssen dann mittels *set...()-Methoden* noch vor der Ausführung der Anweisung eingesetzt werden, wobei als erster Prameter von set der Index des Fragezeichens (1, 2, ...) und als zweiter Parameter der einzusetzende Wert erwartet wird. Sind alle Werte eingesetzt, kann man mittels *executeUpdate()* bzw. *executeQuery()* das *PreparedStatement* ausführen. Dies geschieht in der Regel mehrmals, wobei jeweils vorher die Werte geändert werden.

Programm 3.9: Vorbereitete ändernde SQL-Anweisung (*Prepared.java*)

```
17  String url = "jdbc:mysql://theseus:14317/mydb";
18  // Verbindung herstellen
19  con = DriverManager.getConnection(url, "sep", "test");
20
21  PreparedStatement stmt = null;
22
23  try {
24      // SQL-Anweisung vorbereiten
25      String sql = "INSERT INTO Angestellte VALUES (?,?,?)";
26
```

```
27              // Objekt zum Ausführen von SQL-Anweisungen erzeugen
28              stmt = con.prepareStatement(sql);
29
30              // Fragezeichen mit Werten belegen
31              stmt.setInt(1, 5);
32              stmt.setString(2, "Ulrike Krause");
33              stmt.setString(3, "SAI");
34
35              // SQL-Anweisung ausführen => liefert Anzahl betroffender DS
36              int count = stmt.executeUpdate();
37
38              System.out.println(count+" betroffene Datensätze");
39          }
40          finally {
41              // Statement wieder sauber schließen
42              if (stmt != null) {
43                  try { stmt.close(); }
44                  catch (SQLException e) { }
45              }
46          }
```

Programm 3.10: Vorbereitete SQL-Abfrage (*Prepared1.java*)

```
22          String url = "jdbc:mysql://theseus:14317/mydb";
23          // Verbindung herstellen
24          con = DriverManager.getConnection(url, "sep", "test");
25
26          PreparedStatement stmt = null;
27
28          try {
29              // SQL-Anweisung vorbereiten
30              String sql = "SELECT * FROM Angestellte WHERE persid = ?";
31
32              // Objekt zum Ausführen von SQL-Anweisungen erzeugen
33              stmt = con.prepareStatement(sql);
34
35              // Fragezeichen mit Wert belegen
36              stmt.setString(1, args[0]);
37
38              // SQL-Anweisung ausführen => liefert ResultSet
39              ResultSet rs = stmt.executeQuery();
40
41              // ResultSet iterieren
42              while (rs.next()) {
43                  int persid = rs.getInt("persid");
44                  String name = rs.getString("name");
45                  String abtid = rs.getString("abtid");
46
47                  System.out.println("| "+persid+" | "+name+" | "+abtid+" |");
48              }
49          }
50          finally {
51              // Statement wieder sauber schließen
52              if (stmt != null) {
```

```
53                  try { stmt.close(); }
54                  catch (SQLException e) { }
55              }
56          }
```

```
theseus$ ./workondb
mysql> select * from Angestellte;
+--------+------------------+-------+
| persid | name             | abtid |
+--------+------------------+-------+
| 1      | Franz Schweiggert | SAI   |
| 2      | Andreas Borchert  | SAI   |
| 3      | Johannes Mayer    | SAI   |
+--------+------------------+-------+
theseus$ java Prepared
1 betroffene Datensätze
theseus$ ./workondb
mysql> select * from Angestellte;
+--------+------------------+-------+
| persid | name             | abtid |
+--------+------------------+-------+
| 1      | Franz Schweiggert | SAI   |
| 2      | Andreas Borchert  | SAI   |
| 3      | Johannes Mayer    | SAI   |
| 5      | Ulrike Krause     | SAI   |
+--------+------------------+-------+
4 rows in set (0.00 sec)
theseus$ java Prepared1
java Prepared1 <persid>
theseus$ java Prepared1 1
| 1 | Franz Schweiggert | SAI |
theseus$ java Prepared1 3
| 3 | Johannes Mayer | SAI |
theseus$
```

3.8 Literatur

[Balzert96] Balzert, H.: *Lehrbuch der Software-Technik.* Spektrum Akademischer Verlag, Heidelberg, Berlin, Oxford, 1996.

[Reese00] Reese, G.: *Database Programming with JDBC and Java*. Zweite Ausgabe, O'Reilly, 2000.

[Schlageter83] Schlageter, G.; Stucky, W.: *Datenbanksysteme: Konzepte und Modelle*. Teubner-Verlag, 1983.

[Vossen88] Vossen, G.; Witt, K.-U.: *Das SQL/DS-Handbuch.* Addison-Wesley, 1988.

[Zehnder89] Zehnder, C.A.: *Informationssysteme und Datenbanken.* Teubner-Verlag, 1989.

Kapitel 4

Web-Anwendungen mit Java Servlets

In diesem Kapitel geht es nun konkret um die Implementierung von Web-Anwendungen mittels *Java Servlets*. Das sind einfach besonders strukturierte Java-Programme. Genauer gesagt, ist ein Servlet für das WWW (World Wide Web) eine Unterklasse von *HttpServlet*. Hierbei spielt natürlich das *HTTP* (Hyper Text Transfer Protocol) eine wichtige Rolle.

4.1 Ein kurzer Blick auf das HTTP

Um Servlets zu verstehen, sollte man ein bisschen über das HTTP Bescheid wissen. Daher zunächst ein kleiner Einblick in dieses Protokoll. Abbildung 4.1 zeigt eine kleine Kommunikation zwischen Browser und Web-Server. Der Browser fordert die Seite hinter der URL „/hello“ an

Abbildung 4.1: Kommunikation zwischen Browser und Web-Server

und der Server meldet, dass die Anfrage bearbeitet wurde und liefert den angeforderten Inhalt.

Die genaue Anfrage:

```
GET /hello HTTP/1.0
```

Der Server erkennt anhand einer *Leerzeile* das *Ende der Anfrage*.

Die genaue Antwort des Web-Servers:

```
HTTP/1.1 200 OK
Content-Type: text/plain
Content-Length: 28
Date: Mon, 22 Nov 2004 15:01:14 GMT
Server: Apache-Coyote/1.1
Connection: close

Hello World!
```

In der ersten Zeile der Antwort ist die *Protokoll-Version* (HTTP/1.1) und der *Status-Code* (200) enthalten. Letzterer signalisiert, das alles gut ging. Diese erste Zeile ist die *Statuszeile*. Diese ist gefolgt von einigen *Kopfzeilen*, die z. B. den Dateityp angeben (Content-Type: text/plain). Nach einer *Leerzeile* folgen dann die *angeforderten Daten* (d. h. das Dokument, Bild, etc.).

Bei diesem Beispiel wurde die *GET-Methode* von HTTP verwendet. Gebräuchlich ist außerdem noch die *POST-Methode* z. B. bei Formularen. Die Felder eines Formulars werden als Parameter der Anfrage übergeben. Wohingegen die GET-Methode eine Limitierung der Parametergröße besitzt, ist dies bei POST praktisch nicht der Fall. Daher ist auch die POST-Methode der Standard für Formulare.

4.2 Unser erstes Servlet

Wie immer beginnen wir auch hier mit einem Hallo-Welt-Beispiel:

Programm 4.1: Unser erstes Servlet (*HelloWorld.java*)

```
1  import java.io.*;
2  import javax.servlet.*;
3  import javax.servlet.http.*;
4
5  public class HelloWorld extends HttpServlet {
6      public void doGet(HttpServletRequest req, HttpServletResponse res)
7          throws IOException, ServletException {
8          res.setContentType("text/plain");
9          PrintWriter out = res.getWriter();
10         out.println("Hello World!");
11     }
12 }
```

Anmerkungen:

- Dieses Servlet überschreibt für die GET-Methode die Servlet-Methode *doGet()* – die POST-Methode wird von diesem Servlet also nicht unterstützt.

- doGet() erhält zwei Parameter. Der erste Parameter (*HttpServletRequest*) repräsentiert die *Anfrage*, der zweite (*HttpServletResponse*) die *Antwort*.

Um unser Servlet zu komplilieren, muss man das Java-Archiv *servlet.jar*, das sich beispielsweise im Unterverzeichnis *common/lib/* der Tomcat-Installation befindet, in den *Klassenpfad* mit aufnehmen. (Dies geht einfacher mit dem Skript *setvars*.)

Was ist nun zu tun, um das Servlet auszuführen? Dazu muss die erzeugte Klassendatei in das Verzeichnis *WEB-INF/classes/* unserer Web-Anwendung kopiert werden. Für die ROOT-Anwendung ist dies das Verzeichnis *webapps/ROOT/WEB-INF/classes/* in Ihrer lokalen (!) Tomcat-Installation. Danach ist das Servlet via *http://theseus:4711/servlet/HelloWorld* erreichbar (wobei wir hier davon ausgehen, das Ihr Tomcat auf Port 4711 auf der Theseus „lauscht“ und das Servlet „HelloWorld“ heißt).

4.3 Innenleben eines HttpServlets

Die Oberklasse *HttpServlet* für alle Servlets für HTTP vereinfacht das Arbeiten sehr. Normalerweise muss nämlich jedes (allgemeine) Servlet eine *service()-Methode* implementieren. Dies übernimmt jedoch die *abstrakte Klasse HttpServlet* für uns. Wir müssen nur noch mindestens eine der Methoden *doGet()* bzw. *doPost()* etc. implementieren (siehe Abbildung 4.2). An diese Metho-

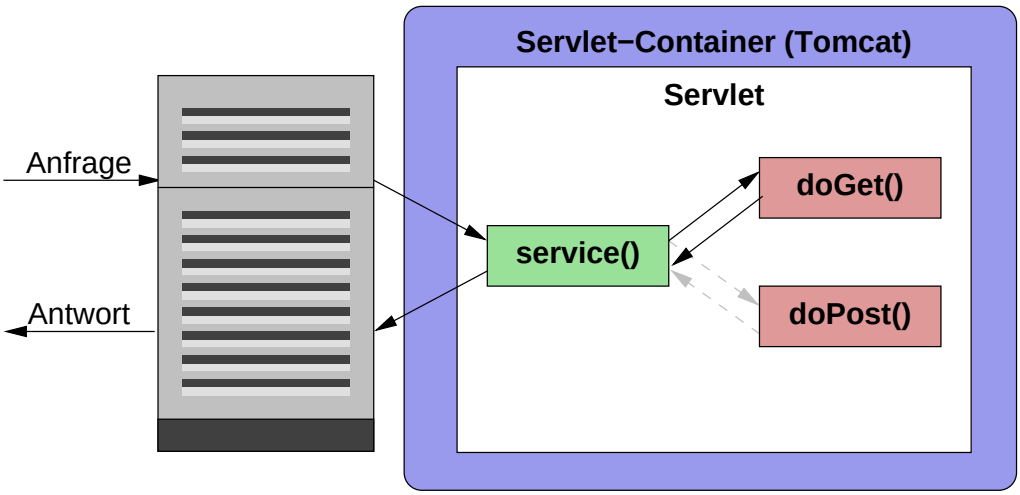


Abbildung 4.2: Innenleben eines HttpServlets

den delegiert service() die HTTP-Anfragen der entsprechenden HTTP-Methode. Ist eine *do...()-Methode* nicht implementiert (genauer: nicht überschrieben), so bedeutet dies, dass eine entsprechende Anfrage nicht unterstützt wird. Unser (erstes) HelloWorld-Servlet unterstützt beispielsweise die POST-Methode nicht. Daher erhalten wir auf die Anfrage *POST /servlet/HelloWorld HTTP/1.0* auch folgende Antwort:

```
HTTP/1.1 400 HTTP method POST is not supported by this URL
Content-Type: text/html;charset=ISO-8859-1
Content-Language: de-DE
Date: Mon, 22 Nov 2004 16:04:14 GMT
Server: Apache-Coyote/1.1
Connection: close

<html><head><title>Apache Tomcat/4.1.30 - Error
report</title><STYLE><!--H1{font-family : sans-serif,Arial,Tahoma;color
: white;background-color : #0086b2;} H3{font-family :
sans-serif,Arial,Tahoma;color : white;background-color : #0086b2;}
BODY{font-family : sans-serif,Arial,Tahoma;color : black;background-color
: white;} B{color : white;background-color : #0086b2;} HR{color :
#0086b2;} --></STYLE> </head><body><h1>HTTP Status 400 - HTTP method POST
is not supported by this URL</h1><HR size="1" noshade><p><b>type</b>
Status report</p><p><b>message</b> <u>HTTP method POST is not
supported by this URL</u></p><p><b>description</b> <u>The request
sent by the client was syntactically incorrect (HTTP method POST is
not supported by this URL).</u></p><HR size="1" noshade><h3>Apache
Tomcat/4.1.30</h3></body></html>
```

Bei dieser Antwort ist der *Status-Code* nun nicht mehr 200, sondern 400, was einen Fehler signalisiert. Klar, die POST-Methode wird nicht unterstützt. Daher muss eine POST-Anfrage auch schief gehen.

Wollte man auch die POST-Methode unterstützen und auf diese Anfragen gleich wie auf GET-Anfragen reagieren, so könnte man doPost() wie folgt implementieren:

Programm 4.2: Implementierung der POST-Methode (*HelloWorld1.java*)

```
12    public void doPost( HttpServletRequest req, HttpServletResponse res )
13        throws IOException, ServletException {
14        doGet( req, res );
15    }
```

4.4 Innenleben eines Servlet-Containers

Ein Servlet-Container (beispielsweise Tomcat) enthält von jedem Servlet nur *eine Instanz* (d h. Objekt), an das er die Anfragen (an dieses Servlet) delegiert (siehe Abbildung 4.3). Programm 4.3 veranschaulicht das sehr schön:

Programm 4.3: Zählendes Servlet (*Counter.java*)

```
1  import java.io.*;
2  import javax.servlet.*;
3  import javax.servlet.http.*;
4
5  public class Counter extends HttpServlet {
6      private int count = 0;
7      public void doGet( HttpServletRequest req, HttpServletResponse res )
```

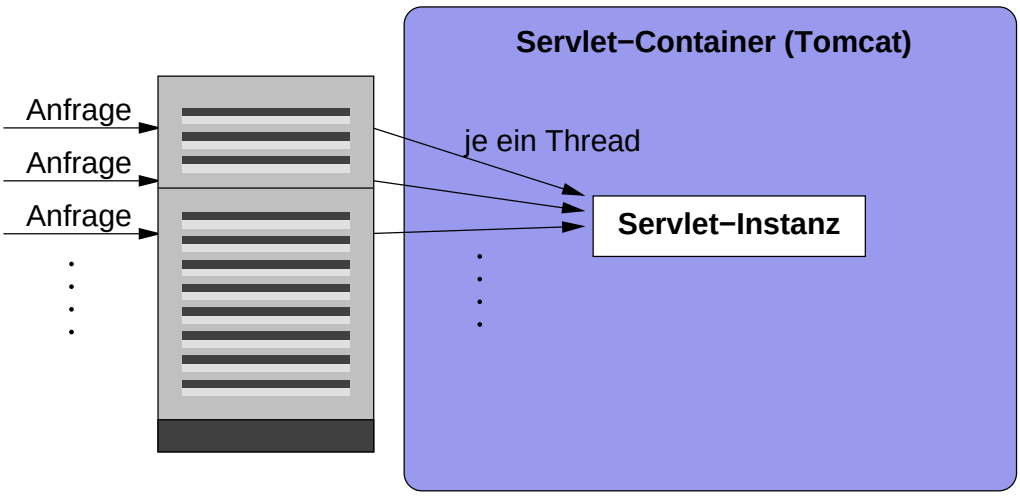


Abbildung 4.3: Innenleben eines Servlet-Containers

```
8           throws IOException,  ServletException  {
9           res.setContentType("text/plain");
10          PrintWriter  out = res.getWriter();
11          count++;
12          out.println("count = "+count);
13      }
14  }
```

Bei der Ausführung des Counter-Servlets ist das Ergebnis beim ersten Aufruf „count = 1“, beim zweiten Aufruf „count = 2“, usw. Man sieht also, dass alle Aufrufe an dasselbe Servlet-Objekt gehen, da die *Änderung der Objektvariable count* beim folgenden Aufruf „sichtbar“ ist.

Bei *parallelen Anfragen* werden eben *mehrere Threads* (das sind leichtgewichtige Prozesse, die sich den Adressraum – also Daten – teilen) erzeugt. Diese Threads rufen dann jeweils die service()-Methode auf *demselben Servlet-Objekt* auf. Somit könnte es zu folgender Situation kommen:

1. Thread 1: count++ inkrementiert count von 0 nach 1
2. Thread 2: count++ inkrementiert count von 1 nach 2
3. Thread 2: gibt „count = 2“ aus
4. Thread 1: gibt „count = 2“ aus

In unserem Fall wäre das nicht tragisch. Sollten jedoch die erzeugten Integer-Werte eindeutig sein, so haben wir ein Problem. Man muss also sehen, wie man die parallelen Threads beim *Zugriff auf Objektvariablen* synchronisieren kann. Dies geht, indem man den zu schützenden Bereich (Lesen und Inkrementieren von count), der *atomar* durchgeführt werden soll, in einen *synchronized-Block* einschließt. In einem solchen Block kann zu jedem Zeitpunkt nur maximal ein Thread sich befinden. Dann gibt es also keine Konflikte.

Programm 4.4: Thread-sicheres zählendes Servlet (*SafeCounter.java*)

```
1  import java.io.*;
2  import javax.servlet.*;
3  import javax.servlet.http.*;
4
5  public class SafeCounter extends HttpServlet {
6      private int count = 0;
7      public void doGet( HttpServletRequest req, HttpServletResponse res )
8          throws IOException, ServletException {
9          res.setContentType("text/plain");
10         PrintWriter out = res.getWriter();
11         int local_count;
12         synchronized(this) {
13             count++;
14             local_count = count;
15         }
16         out.println("count = "+local_count);
17     }
18 }
```

Beim Zugriff auf Objekt- oder Klassenvariablen eines Servlets muss man auf Synchronisation achten.
Außerdem sollte man dafür Sorge tragen, dass synchronisierte Bereiche nur so klein wie möglich sind.
Dass ein Servlet-Container von jedem Servlet nur ein Objekt enthält, ist nur die halbe Wahrheit. In Wirklichkeit gibt es für jeden registrierten Namen eine Instanz.

Was nun ein „registrierter Name“ eines Servlets ist, das lernen wir im folgenden Abschnitt kennen.

4.5 Web-Anwendungen

Eine Servlet-Web-Anwendung, wie z. B. die ROOT-Anwendung, besitzt die folgende Verzeichnisstruktur:

- *webapps/<webapp>/* enthält alle Dateien auf dem Server, z. B. die Startseite index.html bzw. index.jsp
- *webapps/<webapp>/WEB-INF/* enthält den sog. *Deployment-Deskriptor web.xml*, der Infos zur Web-Anwendung, Servlet-Namen und Servlet-URL-Mappings enthält (siehe Programm 4.5)

Programm 4.5: Deployment-Deskriptor (*web.xml*)

```
1  <?xml version="1.0" encoding="ISO-8859-1"?>
2
3  <!DOCTYPE web-app
4      PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
5      "http://java.sun.com/dtd/web-app_2_3.dtd">
6
7  <web-app>
```

```
8   <display-name>Welcome to Tomcat</display-name>
9   <description>
10      Welcome to Tomcat
11  </description>
12 </web-app>
```

- *webapps/<webapp>/WEB-INF/classes/* enthält die Servlet-Klassen
- *webapps/<webapp>/WEB-INF/libs/* enthält Java-Archive mit Servlet-Klassen

In einem Deployment-Deskriptor kann man aber auch einen Namen für ein Servlet definieren:

Programm 4.6: Deployment-Deskriptor mit Servlet-Eintrag (*v1/web.xml*)

```
1  <?xml version="1.0" encoding="ISO-8859-1"?>
2
3  <!DOCTYPE web-app
4      PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
5      "http://java.sun.com/dtd/web-app_2_3.dtd">
6
7  <web-app>
8    <display-name>Welcome to Tomcat</display-name>
9    <description>
10     Welcome to Tomcat
11   </description>
12
13   <servlet>
14     <servlet-name>instances</servlet-name>
15     <servlet-class> Instances </servlet-class>
16   </servlet>
17   <servlet-mapping>
18     <servlet-name>instances</servlet-name>
19     <url-pattern>/instances</url-pattern>
20   </servlet-mapping>
21 </web-app>
```

Dieser Deskriptor definiert den Namen *instances* für die Servlet-Klasse *Instances*. Außerdem definiert er, dass dieses Servlet über die URL */instances* erreichbar ist. Somit existieren nun die beiden URLs */servlet/Instances* (vom InvokerServlet) und */instances* für dieses Servlet. Der Quelltext des Servlets *Instances* folgt nun:

Programm 4.7: Zählen der Servlet-Instanzen (*Instances.java*)

```
1  import java.io.*;
2  import javax.servlet.*;
3  import javax.servlet.http.*;
4  import java.util.*;
5
6  public class Instances extends HttpServlet {
7      private static Hashtable instances = new Hashtable();
8      public void doGet( HttpServletRequest req, HttpServletResponse res )
9          throws IOException, ServletException {
10         res.setContentType("text/plain");
11         PrintWriter out = res.getWriter();
```

```
12            instances.put(this, this);
13            out.println("Es gibt momentan "+instances.size()+" Instanzen.");
14        }
15    }
```

Da man dieses Servlet nun über zwei Namen erreichen kann (den Klassennamen „Instances“, der vom Invoker-Servlet als Name eingetragen wird, und den Namen „instances“), gibt es zwei Servlet-Instanzen, sobald ein Aufruf über jede der beiden URLs erfolgt ist.

Will man nun noch sicher stellen, dass put() und size() *atomar* erfolgen, so muss man wieder *synchronisieren* (aber jetzt nicht mehr über this, sondern nun über *instances* – eine Referenz – selbst, da mehrere Servlet-Objekte jetzt alle auf *instances* Zugriff haben, weil es eine Klassenvariable ist):

Programm 4.8: Zählen der Servlet-Instanzen mit Synchronisation (*Instances1.java*)

```
1  import java.io.*;
2  import javax.servlet.*;
3  import javax.servlet.http.*;
4  import java.util.*;
5
6  public class Instances1 extends HttpServlet {
7      private static Hashtable instances = new Hashtable();
8      public void doGet(HttpServletRequest req, HttpServletResponse res)
9          throws IOException, ServletException {
10         res.setContentType("text/plain");
11         PrintWriter out = res.getWriter();
12         int local_size;
13         synchronized(instances) {
14             instances.put(this, this);
15             local_size = instances.size();
16         }
17         out.println("Es gibt momentan "+local_size+" Instanzen.");
18     }
19 }
```

Zur Synchronisation kann man bei Referenzvariablen die Variable selbst verwenden. Andernfalls kann man ein Objektreferenz nur zu Synchronisationszwecken anlegen oder bei Objektvariablen this und bei Klassenvariablen <Klassenname>.class verwenden.

Anmerkung: Im Zusammenhang mit Servlets ist es wichtig, dass die verwendeten Klassen (wie z. B. Hashtable) *thread-safe* sind, das bedeutet, dass man sie auch in „parallelem Betrieb“ problemlos verwenden kann.

4.6 Lebenszyklus eines Servlets

Wird ein Servlet erzeugt, so wird die Methode *init()* aufgerufen, bevor das Servlet die erste Anfrage bearbeiten soll. Analog dazu wird am Lebensende des Servlets die Methode *destroy()* aufgerufen. Die Signaturen der beiden Methoden sehen wie folgt aus:

```
public void init() throws ServletException
public void destroy()
```

In einem eigenen Servlet können diese Methoden nun überschrieben werden, um Aktion zu Beginn und am Ende des „Lebenes eines Servlets“ auszuführen.

4.7 Initialisierungs-Parameter

Besonders im Zusammenhang mit *Initialisierungs-Parametern* kann die Methode *init()* sinnvoll Verwendung finden. Im Deployment-Deskriptor kann man Parameter für die Initialisierung des Servlets angeben (hier *init_count* als Startwert des Zählers):

Programm 4.9: Deployment-Deskriptor mit Servlet-Initialisierungs-Parametern (*v2/web.xml*)

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<!DOCTYPE web-app
    PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
    "http://java.sun.com/dtd/web-app_2_3.dtd">

<web-app>
  <display-name>Welcome to Tomcat</display-name>
  <description>
     Welcome to Tomcat
  </description>

  <servlet>
    <servlet-name>init</servlet-name>
    <servlet-class>InitCounter</servlet-class>
    <init-param>
      <param-name>init_count</param-name>
      <param-value>100</param-value>
    </init-param>
  </servlet>
  <servlet-mapping>
    <servlet-name>init</servlet-name>
    <url-pattern>/init</url-pattern>
  </servlet-mapping>
</web-app>
```

Mit der Methode *getInitParameter()* kann man dann (am besten in *init()*) den Startwert für den Zähler lesen und setzen:

Programm 4.10: Zähler mit Startwert (*InitCounter.java*)

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class InitCounter extends HttpServlet {
    private int count;
    public void init()
        throws ServletException {
        String init_count = getInitParameter("init_count");
        try {
            count = Integer.parseInt(init_count);
        }
        catch (NumberFormatException e) {
            count = 0;
        }
```

```
16      }
17      public void doGet( HttpServletRequest req , HttpServletResponse res )
18          throws IOException , ServletException {
19          res . setContentType ( " text / plain " );
20          PrintWriter out = res . getWriter ();
21          int local_count ;
22          synchronized (this) {
23              count++;
24              local_count = count ;
25          }
26          out . println ( "count = "+local_count );
27      }
28  }
```

Will man alle Initialisierungs-Parameter herausfinden, so geht dies mit der Methode *public java.util.Enumeration getInitParameterNames()*, wobei diese einen Iterator vom Typ *Enumeration* liefert. Diese könnte man wie folgt durchlaufen (analog zu *Iterator*):

```
Enumeration e = getInitParameterNames ();
while (e .hasMoreElements ()) {
    String name = ( String ) e . nextElement ();
    String value = getInitParameter (name);
    out . println (name + " = " + value );
}
```

Initialisierungs-Parameter kann man z. B. für die Datenbank-Verbindung verwenden. Dann muss man nicht URL, Benutzername und Passwort „hart“ ins Servlet kodieren.

4.8 Eine kleine Einführung in HTML

HTML steht für *Hyper Text Markup Language*. Das ist *die* Sprache, in der momentan Dokumente für das *WWW* (*World Wide Web*) erstellt werden. Eine ausführliche Dokumentation zu HTML ist unter de.selfhtml.org zu finden.

Ein HTML-Dokument besteht aus einer Menge von sogenannten *Tags*. Das sind Schlüsselwörter in spitzen Klammern (z. B. <**html**>). Es gibt Tags, die *alleine* stehen (z. B. <**br** /> für einen Zeilenumbruch), und solche die *paarweise* auftreten (z. B. <**html**> und </**html**>). Paarweise Tags haben einen Inhalt – eben alles was zwischen öffnendem und schließendem Tag steht. Tags können auch *Parameter* erhalten (z. B. <**body bgcolor**="#00ff00">). Kommentare haben die Form *<!−−Kommentar −−>*.

Programm 4.11: Hallo Welt in HTML (*hello.html*)

```
1 <html>
2   <head>
3     < title >Hello World!</ title >
4   </head>
5   <body>
6   Hello World!
7   </body>
8 </html>
```

Anmerkungen: Eine HTML-Seite ist immer in genau ein *html-Tag* eingebettet. Darin steht zunächst der Header (*head-Tag*) und dann der Body (*body-Tag*) mit dem tatsächlichen Seiteninhalt. Der Titel im Header wird nur im Fenstertitel des Browsers angezeigt.

Im Body kann nun u. A. Folgendes stehen:

- Eine *Überschrift*, d. h. Text geklammert in eines der Tag-Paare *h1, h2, h3 oder h4* (z. B. <**h1**>*Überschrift*</**h1**>)
- *Fettgedruckter Text*, d. h. in ein *b-Tag-Paar* eingeschlossener Text (z. B. <**b**>*fetter Text*</**b**>)
- *Kursiver Text* mit einem *i-Tag-Paar* (z. B. <**i**>*kursiver Text*</**i**>)
- Beginn eines *neuen Absatzes*: p-Tag (<**p** />)
- *Zeilenumbruch*: br-Tag (<**br** />)
- *Link*: a-Tag-Paar (z. B. <**a href**="http://www.uni-ulm.de">*Uni Ulm*</**a**>)
- *Bild*: img-Tag (z. B. <**img src**="/rot.gif" />)
- *Tabelle* mit table-,tr- und td-Tag-Paaren; Beispiel:

```
<table>
  <tr> <!-- 1. Zeile -->
    <td> <!-- 1. Zelle -->
      Inhalt
    </td>
    <td> <!-- 2. Zelle -->
      Inhalt
    </td>
  </tr>
  <tr> <!-- 2. Zeile -->
    <td> <!-- 1. Zelle -->
      Inhalt
    </td>
    <td> <!-- 2. Zelle -->
      Inhalt
    </td>
  </tr>
</table>
```

- *Formular* mit form-Tag-Paar und input- sowie select-Tags; Beispiel:

```
<form action="/servlet/Blubber" method=post>
  <!-- type=text => Eingabefeld -->
  Zahl: <input type=text name="zahl" size=5 maxlength=5 /><br />
  <!-- Combobox -->
  Farbe: <select name="tipp" size="1">
           <option value="0">rot</option>
           <option value="1">grün</option>
         </select><br />
  <!-- type=submit => Button zum Abschicken -->
  <input type=submit value="Tippen" />
</form>
```

Beim form-Tag muss man die *Aktion* (⇝ Servlet) und die Methode (GET oder POST; siehe HTTP) angeben. In einem Formular sollte mindestens ein *Button zum Absenden* sein (type=submit).

4.9 Ein Servlet mit HTML-Ausgabe

Bisher hatten unsere Servlets immer einen einfachen Text (*Content-Type: text/plain*) als Ausgabe. Nun können wir natürlich auch ein Servlet angeben, das HTML-Ausgabe (*Content-Type: text/html*) erzeugt. Dann wird aber die Ausgabe natürlich länglicher. Das liegt einfach an HTML.

Programm 4.12: Hallo Welt-Beispiel mit HTML-Ausgabe (*HelloWorld2.java*)

```
1  import java.io.*;
2  import javax.servlet.*;
3  import javax.servlet.http.*;
4
5  public class HelloWorld2 extends HttpServlet {
6      public void doGet(HttpServletRequest req, HttpServletResponse res)
7          throws IOException, ServletException {
8          res.setContentType("text/html");
9          PrintWriter out = res.getWriter();
10         out.println("<html>");
11         out.println("  <head>");
12         out.println("    <title>Hello World!</title>");
13         out.println("  </head>");
14         out.println("  <body>");
15         out.println("    <h1>Hello World!</h1>");
16         out.println("  </body>");
17         out.println("</html>");
18     }
19 }
```

4.10 Parameter der Anfrage

Über eine *URL* mit einem „Anhang“ der Form *?param1=wert&param2=wert* kann man eine GET-Anfrage parametrisieren. Außerdem werden beim Übermitteln eines *Formulars* die einzelnen Elemente als Parameter übermittelt.

In einem Servlet kann man auf Parameter mit den folgenden Methoden von *HttpServletRequest* zugreifen:

- *Enumeration getParameterNames()*
  liefert eine *Enumeration* aller Parameter-Namen.
- *String getParameter(String name)*
  liefert den ersten Wert eines Parameters. (Ein Parameter kann mehrere Werte haben!)
- *String[] getParameterValues(String name)*
  liefert alle Werte eines Parameters.

Folgendes Servlet arbeitet mit Parametern:

Programm 4.13: Zugriff auf Parameter der Anfrage (*Parameters.java*)

```
1  import java.io.*;
2  import javax.servlet.*;
3  import javax.servlet.http.*;
4  import java.util.Enumeration;
5
6  public class Parameters extends HttpServlet {
7      public void doGet(HttpServletRequest req, HttpServletResponse res)
8          throws IOException, ServletException {
9          if (req.getParameter("vorname") == null) {
10             res.setContentType("text/html");
11             PrintWriter out = res.getWriter();
12             out.println("<html>");
13             out.println("  <head>");
14             out.println("    <title>Parameters</title>");
15             out.println("  </head>");
16             out.println("  <body>");
17             out.println("    <form action=\"/servlet/Parameters\" method=post>");
18             out.println("      Vorname: <input type=text name=\"vorname\" size=20 />");
19             out.println("      <br />");
20             out.println("      <input type=hidden name=\"nachname\" value=\"Muster\" />");
21             out.println("      <input type=submit value=\"Absenden\">");
22             out.println("    </form>");
23             out.println("  </body>");
24             out.println("</html>");
25         }
26         else {
27             res.setContentType("text/plain");
28             PrintWriter out = res.getWriter();
29
30             // getParameter() liefert den ersten Wert des Parameters
31             // (oder null, wenn es keinen solchen Parameter gibt)
32             out.println("Hallo "+req.getParameter("vorname")+"!");
33
34             out.println();
35             out.println("Hier kommen alle Parameter:");
36
37             // wir iterieren nun über alle Parameter-Namen ...
38             Enumeration e = req.getParameterNames();
39             while (e.hasMoreElements()) {
40                 String name = (String) e.nextElement();
41                 out.print(name+" = ");
42                 // alle Werte des Parameters name ausgeben
43                 // geht mit der Methode getParameterValues()
44                 String[] values = req.getParameterValues(name);
45                 for (int i = 0; i < values.length; i++)
46                     out.print((i > 0 ? ", " : "") + values[i]);
47                 out.println();
48             }
49         }
50     }
51     public void doPost(HttpServletRequest req, HttpServletResponse res)
52         throws IOException, ServletException {
```

```
53          doGet(req, res);
54      }
55  }
```

Anmerkung: Um *Anführungszeichen* in einen String einzufügen, müssen Sie *mit einem Backslash quotiert* werden. Dies macht es etwas unleserlich.

Zunächst gibt das Servlet ein Eingabeformular aus mit einem Eingabefeld für den Vornamen und einen Button „Absenden“. Gibt man als Vornamen „Johannes“ ein und betätigt dann den Button, so erscheint folgende Ausgabe:

```
Hallo Johannes!

Hier kommen alle Parameter:
nachname = Muster
vorname = Johannes
```

Das eben vorgestellte Servlet kennt *zwei Zustände*: *Parameter vorname vorhanden* bzw. *Parameter vorname nicht vorhanden*. Von diesem Zustand ist die erzeugte Ausgabe abhängig. Gibt es noch keinen Parameter, so wird ein Formular generiert. Andernfalls wird die Begrüßung und die Ausgabe aller Parameter generiert.

Mit Hilfe von Parametern lässt sich auch leicht ein Servlet zum *Zahlen-Raten* implementieren:

Programm 4.14: Zahlen-Raten mit einem Servlet (*Raten.java*)

```
1  import java.io.*;
2  import javax.servlet.*;
3  import javax.servlet.http.*;
4  import java.util.Random;
5
6  public class Raten extends HttpServlet {
7      public void doGet(HttpServletRequest req, HttpServletResponse res)
8          throws IOException, ServletException {
9          res.setContentType("text/html");
10         PrintWriter out = res.getWriter();
11
12         out.println("<html>");
13         out.println("  <head>");
14         out.println("    <title>Raten</title>");
15         out.println("  </head>");
16         out.println("  <body>");
17
18         final int MAX = 128;
19         int geheim = 0;
20         boolean neu_waehlen = true;
21
22         // evtl. Tipp auswerten
23         if (req.getParameter("geheim") != null) {
24             try {
25                 geheim = Integer.parseInt(req.getParameter("geheim"));
26                 int tipp = Integer.parseInt(req.getParameter("tipp"));
```

```
27                  if (geheim < tipp) {
28                      out.println("Die␣geheime␣Zahl␣ist␣<i>kleiner</i>!");
29                      neu_waehlen = false;
30                  }
31                  else if (geheim > tipp) {
32                      out.println("Die␣geheime␣Zahl␣ist␣<i>grösSer</i>!");
33                      neu_waehlen = false;
34                  }
35                  else
36                      out.println("<b>Toll,␣Sie␣haben␣die␣geheime␣Zahl␣erraten!</b>");
37              }
38              catch (NumberFormatException e) {
39                  // Log-Meldung schreiben
40                  log(""+e);
41                  out.println("Sie␣müssen␣schon␣eine␣Zahl␣eingeben!");
42              }
43          }
44
45          // evtl. eine neue geheime Zahl wählen
46          if (neu_waehlen) {
47              Random r = new Random();
48              geheim = r.nextInt(MAX) + 1;
49              out.println("<p␣/>");
50              out.println("Ich␣habe␣eine␣Zahl␣zwischen␣1␣und␣"+MAX+"␣gewählt!");
51          }
52
53          out.println("␣␣␣␣<p␣/>");
54
55          out.println("␣␣␣␣<form␣action=\"/servlet/Raten\"␣method=post>");
56          out.println("␣␣␣␣␣␣Ihr␣Tipp:␣<input␣type=text␣name=\"tipp\"␣size=5␣/>");
57          out.println("␣␣␣␣␣␣<br␣/>");
58          out.println("␣␣␣␣␣␣<input␣type=hidden␣name=\"geheim\"␣value=\""+geheim+"\"␣/>");
59          out.println("␣␣␣␣␣␣<input␣type=submit␣value=\"Tipp␣absenden\">");
60          out.println("␣␣␣␣</form>");
61          out.println("␣␣</body>");
62          out.println("</html>");
63      }
64      public void doPost(HttpServletRequest req, HttpServletResponse res)
65          throws IOException, ServletException {
66          doGet(req, res);
67      }
68  }
```

Anmerkungen:

- Mit *log()* kann man eine Log-Meldung in die Log-Datei *localhost_log.<datum>.txt* im Unterverzeichnis *logs* der lokalen Tomcat-Installation schreiben. Dies ist besonders zu *Debugging-Zwecken* sinnvoll.
- Die geheime Zahl verstecken wir als *Hidden-Field* im Formular. Das hat natürlich den Nachteil, dass der HTML-Quelltext sofort die geheime Zahl verrät.

4.11 Sitzungen

Das Problem bei unserer bisherigen Implementierung von Zahlen-Raten war die *geheime Zahl im HTML-Quelltext*. Dies ließe sich umgehen, wenn wir auf dem Server etwas speichern könnten, was beim nächsten Aufruf wieder zur Verfügung steht. Genau das geschieht schon durch die *Sitzungsverwaltung* (*session management*) des Servlet-Containers. Obwohl HTTP keine Sitzungen, d. h. mehrere zusammenhängende Anfragen unterstützt, wird die vom Servlet-Container angeboten. Dieser assoziiert jede Sitzung mit einem eigenen Sitzungsobjekt vom Typ *HttpSession*, das aufgrund einer *Sitzungs-ID* (*session id*) zugeordnet wird (siehe Abbildung 4.4).

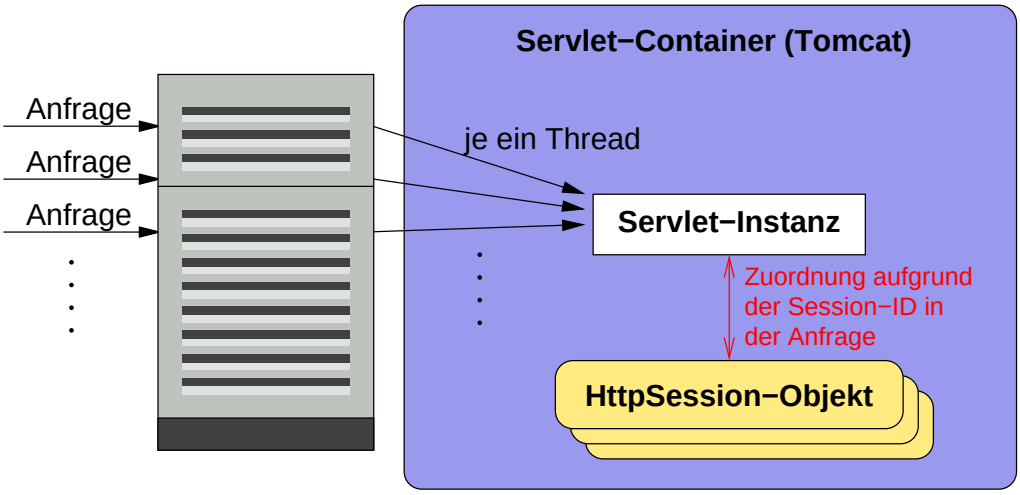


Abbildung 4.4: Zuordnung eines Sitzungs-Objektes zu einer Servlet-Sitzung

Folgende Methoden sind zunächst von Bedeutung:

- *HttpSession HttpServletRequest . getSession* ()
  liefert das assoziierte Sitzungsobjekt (vom Typ *HttpSession*) bzw. erzeugt ein neues und liefert dieses.
- *Object HttpSession . getAttribute* (*String name*)
  liefert ein Attribut der Sitzung bzw. **null**, wenn es kein solches gibt.
- **void** *HttpSession . setAttribute* (*String name*, *Object value*)
  setzt ein Sitzungsattribut.
- *Enumeration HttpSession . getAttributeNames* ()
  liefert einen Iterator für die Sitzungs-Attribute.
- **void** *HttpSession . removeAttribute* (*String name*)
  entfernt ein Sitzungsattribut.

Praktisch ist, dass an ein Sitzungsobjekt *beliebige Objekte als Attribute* angehängt werden können.

Das folgende Servlet verwendet nun ein Sitzungsobjekt:

Programm 4.15: Servlet unter Verwendung von Sitzungen (*Session.java*)

```
1  import java.io.*;
2  import javax.servlet.*;
3  import javax.servlet.http.*;
4  import java.util.Enumeration;
5
6  public class Session extends HttpServlet {
7      public void doGet( HttpServletRequest req, HttpServletResponse res )
8          throws IOException, ServletException {
9          res.setContentType("text/html");
10         PrintWriter out = res.getWriter();
11
12         // Sitzungsobjekt besorgen (bzw. neu anlegen)
13         HttpSession sess = req.getSession();
14
15         // Attribut count der Sitzung lesen
16         Integer count = (Integer) sess.getAttribute("count");
17
18         // Attribut count erhöhen oder mit 1 initialisieren
19         if (count == null)
20             count = new Integer(1);
21         else
22             count = new Integer(count.intValue() + 1);
23
24         // Sitzungsattribut count setzen
25         sess.setAttribute("count", count);
26
27         out.println("<html>");
28         out.println("  <body>");
29         out.println("    count = "+count+"<br/>");
30         // selbst referenzierende URL erzeugen
31         String url = res.encodeURL(req.getRequestURI());
32         out.println("    <a href=\""+url+"\">Ruf mich auf!</a>");
33         out.println("    <p/>");
34         out.println("    <b>Alle Attribute:</b><br/>");
35
36         // alle Attribute ausgeben
37         Enumeration e = sess.getAttributeNames();
38         while (e.hasMoreElements()) {
39             String name = (String) e.nextElement();
40             out.println(name+" = "+sess.getAttribute(name));
41         }
42
43         out.println("  </body>");
44         out.println("</html>");
45     }
46 }
```

Anmerkungen:

- Jetzt haben wir keine *lokale Variable* count mehr, sondern ein *Sitzungs-Attribut* count. Da ein Servlet-Objekt ja für alle Sitzungen verwendet wird, macht das schon einen Unterschied.

- Bisher haben wir bei der *Aktion eines Formulars* einfach direkt die URL angegeben. Dies funktioniert nun so nicht mehr. Wir müssen nicht nur die *URL* verwenden, sondern auch die *Sitzungs-ID*. Dies geht, indem wir mit *res.encodeURL(req.getRequestURI())* die URL der Anfrage beschaffen und in sie die Sitzungs-ID „hineinkodieren“.

Mit Hilfe eines Sitzungsobjektes lässt sich nun das Zahlen-Raten so implementieren, dass die geheime Zahl auch wirklich geheim bleibt:

Programm 4.16: Zahlen-Raten unter Verwendung einer Sitzung (*Raten1.java*)

```
1  import java.io.*;
2  import javax.servlet.*;
3  import javax.servlet.http.*;
4  import java.util.Random;
5
6  public class Raten1 extends HttpServlet {
7      public void doGet(HttpServletRequest req, HttpServletResponse res)
8          throws IOException, ServletException {
9          res.setContentType("text/html");
10         PrintWriter out = res.getWriter();
11
12         HttpSession sess = req.getSession();
13
14         out.println("<html>");
15         out.println("  <head>");
16         out.println("    <title>Raten</title>");
17         out.println("  </head>");
18         out.println("  <body>");
19
20         final int MAX = 128;
21         Integer geheim = (Integer) sess.getAttribute("geheim");
22
23         // evtl. Tipp auswerten
24         if (geheim != null) {
25             try {
26                 int tipp = Integer.parseInt(req.getParameter("tipp"));
27                 int anzTipps = ((Integer) sess.getAttribute("anzahl_tipps")).intValue()+1;
28                 if (geheim.intValue() < tipp)
29                     out.println("Die geheime Zahl ist <i>kleiner</i>!");
30                 else if (geheim.intValue() > tipp)
31                     out.println("Die geheime Zahl ist <i>grösSer</i>!");
32                 else {
33                     out.println("<b>Toll, Sie haben die geheime Zahl erraten!</b>");
34                     geheim = null; // => neu wählen
35                 }
36                 out.println("<p />");
37                 out.println("Das war Ihr "+anzTipps+"-ter Tipp.");
38                 sess.setAttribute("anzahl_tipps", new Integer(anzTipps));
39             }
40             catch (NumberFormatException e) {
41                 // Log-Meldung schreiben
42                 log(""+e);
43                 out.println("Sie müssen schon eine Zahl eingeben!");
44             }
```

```
45          }
46
47          // evtl. eine neue geheime Zahl wählen
48          if (geheim == null) {
49              Random r = new Random();
50              geheim = new Integer(r.nextInt(MAX) + 1);
51              sess.setAttribute("geheim", geheim);
52              sess.setAttribute("anzahl_tipps", new Integer(0));
53              out.println("<p />");
54              out.println("Ich habe eine Zahl zwischen 1 und "+MAX+" gewählt!");
55          }
56
57          out.println("    <p />");
58
59          String url = res.encodeURL(req.getRequestURI());
60          out.println("    <form action=\""+url+"\" method=post>");
61          out.println("      Ihr Tipp: <input type=text name=\"tipp\" size=5 />");
62          out.println("      <br />");
63          out.println("      <input type=submit value=\"Tipp absenden\">");
64          out.println("    </form>");
65          out.println("  </body>");
66          out.println("</html>");
67      }
68      public void doPost(HttpServletRequest req, HttpServletResponse res)
69          throws IOException, ServletException {
70          doGet(req, res);
71      }
72  }
```

Wichtig im Zusammenhang mit einer Sitzung sind außerdem auch noch folgende Methoden:

- **boolean** *HttpSession.isNew*()
  liefert genau dann **true**, wenn die Sitzung soeben neu erzeugt wurde.
- **void** *HttpSession.invalidate*()
  macht die Sitzung ungültig. Danach kann man mit *getSession*() ein neues Sitzungsobjekt anlegen. (Damit lässt sich „Ausloggen“ realisieren.)
- **long** *HttpSession.getCreationTime*()
  liefert die Erstellungszeit dieser Sitzungs in Millisekunden seit Mitternacht 1.1.1970 GMT.
- **long** *HttpSession.getLastAccessedTime*()
  liefert die Zeit des letzten Zugriffs dieser Sitzungs in Millisekunden seit Mitternacht 1.1.1970 GMT.

Eine Sitzung lebt nicht ewig. Nach einer gewissen Zeit wird sie automatisch „aufgeräumt“. Folgende Methoden können dazu verwendet werden, diese Zeit zu bestimmen bzw. zu verändern:

- **int** *HttpSession.getMaxInactiveInterval*()
  liefert die maximal mögliche inaktive Zeit dieser Sitzung in *Sekunden*.
- **void** *HttpSession.setMaxInactiveInterval*(**int** *secs*)
  setzt die maximal mögliche inaktive Zeit dieser Sitzung in *Sekunden*.

Am besten macht man jedoch eine geeignete Voreinstellung für den Timeout im *Deployment-Deskriptor* der Web-Anwendung:

Programm 4.17: Session-Timeout im Deployment-Deskriptor (*v3/web.xml*)

```
1  <?xml version="1.0"  encoding="ISO-8859-1"?>
2
3  <!DOCTYPE web-app
4      PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
5      " http://java.sun.com/dtd/web-app_2_3.dtd">
6
7  <web-app>
8    <display-name>Welcome to Tomcat</display-name>
9    <description>
10       Welcome to Tomcat
11   </description>
12
13   <session-config>
14     <session-timeout>
15       60 <!-- in Minuten -->
16     </session-timeout>
17   </session-config>
18 </web-app>
```

4.12 Literatur

[Hunter01] Hunter, J.; Crawford, W.: *Java Servlet Programming*. Zweite Ausgabe, O'Reilly, 2001.

Kapitel 5

Konfigurationsmanagement Praxis mit Ant und CVS

Das Konfigurationsmanagement ist eine Tätigkeit im Software Engineering, die sich durch alle Phasen eines Projektes hindurch zieht. Darunter fällt die Versionsverwaltung aller Dokumente. Man will jederzeit auf alle Versionen eines Dokumentes Zugriff haben – sei es nun ein „echtes“ Dokument oder ein Quelltext oder ein Testfall. Außerdem ist das Buildmanagement, d. h. die Erzeugung des Programms aus den Quelltexten, von enormer Bedeutung. Zu diesen beiden „Säulen“ des Konfigurationsmanagements werden wir im Folgenden die beiden Tools CVS und Ant kennen lernen.

5.1 Build-Management mit Ant

5.1.1 Ein erstes Beispiel

Beginnen wir mit einem Beispiel. Am Anfang sieht die Verzeichnisstruktur wie folgt aus:

```
thales$ ls -R
.:
build.xml  src

./src:
Main.java  a  b

./src/a:
A.java

./src/b:
B.java
thales$
```

Im Unterverzeichnis *src* befinden sich die Java-Quelldateien. Im aktuellen Verzeichnis befindet sich die *Ant-Build-Datei build.xml*.

Programm 5.1: Unsere erste Ant-Build-Datei (*ex1/build.xml*)

```
1  <?xml version="1.0"?>
2
3  <project name="einfache Build-Datei" default="compile" basedir=".">
4
5      <target name="prepare">
6          <mkdir dir="build"/>
7          <mkdir dir="build/classes"/>
8          <mkdir dir="build/lib"/>
9      </target>
10
11     <target name="compile" depends="prepare" description="Kompilation der Quellen">
12         <javac srcdir="src" destdir="build/classes"/>
13     </target>
14
15     <target name="jar" depends="compile" description="Erzeugung eines Java-Archivs">
16         <jar jarfile="build/lib/my.jar" basedir="build/classes"/>
17     </target>
18
19     <target name="clean" description="Alle erzeugten Dateien entfernen">
20         <delete dir="build"/>
21     </target>
22
23 </project>
```

Anmerkungen:

- Das Format der Build-Datei ist *XML* (siehe Webseiten mit HTML). Das zeigt die erste Zeile der Build-Datei an. Zu jedem *öffnenden Tag* (z. B. <project>), das etwas enthalten kann, gibt es ein zugehöriges *schließendes Tag* (z. B. </project>). Soll ein Tag nichts enthalten, so kann es auch gleich *in sich abgeschlossen* sein (z. B. <javac ... / >). Außerdem kann ein Tag – zusätzlich zu den Elementen zwischen öffnendem und schließendem Tag – auch noch *Parameter* enthalten (z. B. <javac srcdir="src" ... / >).
- Eine Ant-Build-Datei besteht nun aus einem *project-Tag*, das den *Namen des Projektes*, das *Default-Target* und das *Basis-Verzeichnis* („.“ bedeutet gleiches Verzeichnis wie die Build-Datei) als Parameter erhält.
- In das project-Tag können nun beliebig viele *target-Tags* eingebettet sein. Diese beschreiben einzelne Ziele. Ziele hängen im Allgemeinenen voneinander ab. Dies kann man mittels *depends* notieren. Bei den Zielen unterscheidet man solche mit Beschreibung (sog. *Hauptziele* bzw. *main targets*) und solche ohne Beschreibung (sog. *Unterziele* bzw. *subtargets*). Diese verhalten sich gleich. Letztere kann man dazu verwenden, die Hauptziele zu strukturieren, ohne dass diese Aufgaben „von außen“ erforderlich wären.
- Ein target-Tag kann nun beliebig viele *task-Tags* enthalten, um das Ziel zu erreichen. Ant kennt eine große Zahl vordefinierter Aufgaben (z. B. javac, jar, mkdir, delete, etc.). Außerdem lässt sich Ant auch um Aufgaben erweitern.
- Was nun die *Portabilität* von Ant-Dateien angeht, so ist es sehr gut darum bestellt. Da die Aufgaben *keine Kommandozeilen* (wie etwa bei make), sondern Ant-Tasks sind und, da der *Verzeichnistrenner* „/“ unter Windows automatisch nach „\“ konvertiert wird, sieht eine Ant-Build-Datei unter Unix und Windows identisch aus.

Wie gehabt (bei MySQL und Tomcat) müssen wir zum Arbeiten zunächst das *Skript setvars* aufrufen, um den Pfad zu setzen. Danach können wir einfach *ant* auf der Kommandozeile eingeben.

```
thales$ . /home/sep2004/software/installed/setvars
thales$ ant
Buildfile: build.xml

prepare:
    [mkdir] Created dir: /home/thales/jmayer/progs/ex1/build
    [mkdir] Created dir: /home/thales/jmayer/progs/ex1/build/classes
    [mkdir] Created dir: /home/thales/jmayer/progs/ex1/build/lib

compile:
    [javac] Compiling 3 source files to /home/thales/jmayer/progs/ex1/build/classes

BUILD SUCCESSFUL
Total time: 4 seconds
thales$ ant jar
Buildfile: build.xml

prepare:

compile:

jar:
      [jar] Building jar: /home/thales/jmayer/progs/ex1/build/lib/my.jar

BUILD SUCCESSFUL
Total time: 2 seconds
thales$
```

Anmerkungen:

- Ist beim Aufruf von Ant keine Build-Datei angegeben, so wird nach einer Datei *build.xml* gesucht. (Mit `-buildfile mybuild.xml` könnte man das ändern.)
- Ist auch kein Target angegeben beim Ant-Aufruf, so wird das *Default-Target* (siehe *Parameter „default“* des project-Tags) genommen. In obigem Beispiel ist dies beim ersten Aufruf der Fall. Das Default-Target ist hier compile.

```
thales$ ant -projecthelp
Buildfile: build.xml

Main targets:

 clean    Alle erzeugten Dateien entfernen
 compile  Kompilation der Quellen
 jar      Erzeugung eines Java-Archivs
Default target: compile
thales$
```

Anmerkung: Mit der Option `-projecthelp` kann man Ant die Hauptziele (inkl. Beschreibung) einer Build-Datei entlocken. Außerdem erfährt man auch, welches das Default-Ziel ist.

Nach der Kompilation und der Erzeugung des Java-Archivs gibt es die folgenden Dateien und Verzeichnisse.

```
thales$ ls -R
.:
build  build.xml  src

./build:
classes  lib

./build/classes:
Main.class  a  b

./build/classes/a:
A.class

./build/classes/b:
B.class

./build/lib:
my.jar

./src:
Main.java  a  b

./src/a:
A.java

./src/b:
B.java
thales$
```

Mittels *ant clean* könnte man alle erzeugten Dateien auf einmal löschen.

5.1.2 Eigenschaften

Was bisher sehr unschön war in dem Beispiel ist, dass die *Verzeichnisse „hart" reinkodiert* waren. Das ist im Prinzip genauso wie, wenn bei einem Programm irgendwelche Konstanten ohne Namen auftauchen. Das soll vermieden werden! Und es geht auch, und zwar mit *Eigenschaften* (engl. *properties*). Damit kann man quasi Variablen definieren, die man später referenzieren darf.

Programm 5.2: Verbesserte Build-Datei (*ex2/build.xml*)

```
1 <?xml version="1.0"?>
2
3 <project name="einfache Build-Datei" default="compile" basedir=".">
4
5     <!-- Quellverzeichnis -->
```

```
6     <property name="src.dir" value="src"/>
7
8     <!-- Zielverzeichnisse -->
9     <property name="build.dir" value="build"/>
10    <property name="classes.dir" value="${build.dir}/classes"/>
11    <property name="lib.dir" value="${build.dir}/lib"/>
12
13    <!-- Name des Java-Archivs -->
14    <property name="archive.name" value="my.jar"/>
15
16    <target name="prepare">
17        <mkdir dir="${build.dir}"/>
18        <mkdir dir="${classes.dir}"/>
19        <mkdir dir="${lib.dir}"/>
20    </target>
21
22    <target name="compile" depends="prepare" description="Kompilation der Quellen">
23        <javac srcdir="${src.dir}" destdir="${classes.dir}"/>
24    </target>
25
26    <target name="jar" depends="compile" description="Erzeugung eines Java-Archivs">
27        <jar jarfile ="${lib.dir}/${archive.name}" basedir="${classes.dir}"/>
28    </target>
29
30    <target name="clean" description="Alle erzeugten Dateien entfernen">
31        <delete dir="${build.dir}"/>
32    </target>
33
34 </project>
```

Sobald eine Eigenschaft erst einmal definiert ist, kann ihr werden innerhalb von Parameter-Werten mittels *${Name der Eigenschaft}* eingefügt werden.

5.1.3 Klassenpfade und Datentypen

Muss man in den Klassenpfad eine Reihe von Bibliotheken einbinden, so könnte man das natürlich wie folgt machen:

```
<property name="classpath" value="${libs.dir}/a.jar:${libs.dir}/b.jar:${libs.dir}/c.jar"/>
```

Dabei muss man aber einen ziemlich länglichen String erzeugen und bei Änderungen (neues Java-Archiv etc.) muss jedes Mal dieser String verändert werden.

Man kann aber auch für Pfade *filesets* (d. h. Mengen von Dateien) verwenden.

```
<path id="classpath">
    <fileset dir="${libs.dir}">
        <include name="a.jar"/>
        <include name="b.jar"/>
        <include name="c.jar"/>
    </fileset>
</path>
```

Für filesets kann man (im Gegensatz zu Eigenschaften) auch *Wildcards* verwenden:

```
<path id="classpath">
    < fileset  dir="${libs.dir}">
        <include name="*.jar"/>
    </ fileset >
</path>
```

Will man auch Java-Archive aus den Unterverzeichnissen hinzufügen, so schreibe man einfach ***/*.jar*.

Einen so definierten Pfad kann man nun wie folgt verwenden:

```
<javac  ... >
    <classpath  refid="classpath"/>
</javac>
```

Anmerkung: Auch Pfade sind portabel. Während unter Unix/Linux der Trenner „:“ verwendet wird, findet unter Windows „;“ Verwendung.

5.1.4 Weitere nützliche Ant-Tasks

Erzeugung von Dokumentation Mit der *javadoc-Task* kann man Javadoc erzeugen:

```
<javadoc packagenames="geostoch.*"
         sourcepath="${src.dir}"
         destdir="${doc.dir}"
         author="true"
         version="true"
         use="true">
    <classpath  refid="classpath"/>
</javadoc>
```

Kopieren von Dateien Mit der *copy-Task* kann man die Dateien aus einem oder mehreren filesets an ein Ziel kopieren:

```
<copy todir="${dist.dir}/lib">
    <!-- eine Datei -->
    < fileset  dir="${lib.dir}" includes="${archive.name}"/>
    <!-- evtl. mehrere Dateien -->
    < fileset  dir="${libs.dir}" includes="*.jar"/>
</copy>

<copy todir="${dist.dir}/doc">
    <!-- ein Verzeichnis -->
    < fileset  dir="${doc.dir}"/>
</copy>
```

5.1.5 Ein abschließendes Beispiel

Folgende Build-Datei enthält nun die besprochenen Details:

Programm 5.3: Beispiel für eine Build-Datei (*ex3/build.xml*)

```
1  <?xml version="1.0"?>
2
3  <project name="einfache Build-Datei" default="compile" basedir=".">
4
5      <!-- Quellverzeichnis -->
6      <property name="src.dir" value="src"/>
7
8      <!-- Zielverzeichnisse -->
9      <property name="build.dir" value="build"/>
10     <property name="classes.dir" value="${build.dir}/classes"/>
11     <property name="lib.dir" value="${build.dir}/lib"/>
12     <property name="doc.dir" value="${build.dir}/doc"/>
13     <property name="dist.dir" value="dist"/>
14
15     <!-- Verzeichnis mit den verwendeten Java-Archiven -->
16     <property name="libs.dir" value="libs"/>
17
18     <!-- Name des Java-Archivs -->
19     <property name="archive.name" value="my.jar"/>
20
21     <path id="classpath">
22         <fileset dir="${libs.dir}">
23             <include name="**/*.jar"/>
24         </fileset>
25     </path>
26
27     <target name="prepare">
28         <mkdir dir="${build.dir}"/>
29         <mkdir dir="${classes.dir}"/>
30         <mkdir dir="${lib.dir}"/>
31         <mkdir dir="${doc.dir}"/>
32     </target>
33
34     <target name="compile" depends="prepare" description="Kompilation der Quellen">
35         <javac srcdir="${src.dir}" destdir="${classes.dir}">
36             <classpath refid="classpath"/>
37         </javac>
38     </target>
39
40     <target name="jar" depends="compile" description="Erzeugung eines Java-Archivs">
41         <jar jarfile="${lib.dir}/${archive.name}" basedir="${classes.dir}"/>
42     </target>
43
44     <target name="doc" depends="prepare" description="Erzeugung der Javadoc">
45         <javadoc packagenames="*.*"
46                  sourcepath="${src.dir}"
47                  destdir="${doc.dir}"
48                  author="true"
```

```
49                  version="true"
50                  use="true">
51              <classpath refid="classpath"/>
52          </javadoc>
53      </target>
54
55      <target name="install" depends="compile,jar" description="Anwendung installieren">
56          <copy todir="${dist.dir}">
57              <fileset dir="${build.dir}"/>
58          </copy>
59      </target>
60
61      <target name="clean" description="Alle erzeugten Dateien entfernen">
62          <delete dir="${build.dir}"/>
63      </target>
64
65  </project>
```

Für alle make-Benutzer: Ant unterscheidet sich gravierend von make. Zum einen verwendet es eigene Tasks statt Kommandos und zum anderen haben die Targets bei Ant nichts (direkt) mit Dateien zu tun.

5.2 Versionsmanagement mit CVS

Im Folgenden werden wir nun kennen lernen wie man verschiedene Versionen von Dateien mit *CVS* (*Concurrent Versions System*) verwalten kann.

5.2.1 Anlegen eines Repositorys

```
chomsky$ export CVSROOT=/home/jmayer/cvs
chomsky$ cvs init
chomsky$ ls -R /home/jmayer/cvs
/home/jmayer/cvs:
CVSROOT

/home/jmayer/cvs/CVSROOT:
Emptydir        commitinfo,v  cvswrappers,v  modules    rcsinfo    val-tags
checkoutlist    config        history        modules,v  rcsinfo,v  verifymsg
checkoutlist,v  config,v      loginfo        notify     taginfo    verifymsg,v
commitinfo      cvswrappers   loginfo,v      notify,v   taginfo,v

/home/jmayer/cvs/CVSROOT/Emptydir:
chomsky$
```

Anmerkungen:

- Zunächst muss man die *Umgebungsvariable CVSROOT* auf den Katalog des CVS-Repositorys setzen. Dieser Katalog existiert am Anfang noch nicht.
- Durch das Kommando *cvs init* wird das Repository (samt angegebenem Katalog) angelegt.
- Die Dateien im Repository enthalten momentan nur Verwaltungsinformationen.

5.2.2 Importieren von Modulen

```
chomsky$ ls -R
.:
pkg

./pkg:
Test.java
chomsky$ cd pkg/
chomsky$ cvs import -m "Aller Anfang ..." pkg avendor arelease
N pkg/Test.java

No conflicts created by this import

chomsky$
```

Anmerkungen:

- Durch das Kommando *cvs import* kann man ein *Modul* importieren. CVS versteht unter einem *Modul* einfach ein Verzeichnis samt Inhalt (reguläre Dateien und Unterverzeichnisse).
- Beim Import muss man eine Log-Meldung (hinter `-m`) und ein Vendor-, sowie ein Release-Tag angeben.
- CVS arbeitet normalerweise *rekursiv*, d. h. es werden auch Unterverzeichnisse mit einbezogen.

5.2.3 Auschecken einer Arbeitskopie

```
chomsky$ pwd
/home/jmayer/user1
chomsky$ ls
chomsky$ cvs checkout pkg
cvs checkout: Updating pkg
U pkg/Test.java
chomsky$ ls -R
.:
pkg

./pkg:
CVS  Test.java

./pkg/CVS:
Entries  Repository  Root
chomsky$
```

Anmerkungen:

- Wir setzen nun beim Arbeiten mit CVS – auch hier – immer voraus, dass die *Umgebungsvariable CVSROOT* passend gesetzt ist.
- Mit *cvs checkout* kann man sich nun eine Arbeitskopie des Moduls pkg besorgen. Das können auch mehrere Benutzer gleichzeitig tun.

5.2.4 Änderungen permanent machen

```
chomsky$ cat pkg/Test.java
public class Test {
    public static void main(String[] args) {
    }
}
chomsky$ # Änderung durchführen
chomsky$ cat pkg/Test.java
public class Test {
    public static void main(String[] args) {
        System.out.println("Test");
    }
}
chomsky$ cvs commit -m "mit Ausgabe" pkg/Test.java
Checking in pkg/Test.java;
/home/jmayer/cvs/pkg/Test.java,v  <--  Test.java
new revision: 1.2; previous revision: 1.1
done
chomsky$
```

Anmerkungen:

- Will man nach einer Änderung diese ins Repository übernehmen, so muss man das Kommando *cvs commit* verwenden.
- Hierbei muss man eine *Log-Meldung* (Option `-m`) angeben.
- Lässt man den Namen der Datei weg, so werden alle Dateien und Unterverzeichnisse „eingecheckt“. ($\rightsquigarrow$ CVS ist *rekursiv*)

5.2.5 Gleichzeitiges Arbeiten

Bisher hat immer Benutzer 1 alleine gearbeitet. Nun könnte auch ein Benutzer 2 eine weitere Arbeitskopie „auschecken“ und danach Änderungen am Repository vornehmen:

```
chomsky$ pwd
/home/jmayer/user2/pkg
chomsky$ cat Test.java
public class Test {
    public static void main(String[] args) {
        System.out.println("Test ...");
    }
}
chomsky$ cvs commit -m "Ausgabe geändert" Test.java
Checking in Test.java;
/home/jmayer/cvs/pkg/Test.java,v  <--  Test.java
new revision: 1.3; previous revision: 1.2
done
chomsky$
```

Nun möchte Benutzer 1 auch wieder auf den aktuellen Stand kommen:

```
chomsky$ pwd
/home/jmayer/user1/pkg
chomsky$ cvs status Test.java
===================================================================
File: Test.java         Status: Needs Patch

    Working revision:    1.2     Wed Dec  1 16:35:07 2004
    Repository revision: 1.3     /home/jmayer/cvs/pkg/Test.java,v
    Sticky Tag:          (none)
    Sticky Date:         (none)
    Sticky Options:      (none)

chomsky$ cvs update
cvs update: Updating .
U Test.java
chomsky$ cat Test.java
public class Test {
    public static void main(String[] args) {
        System.out.println("Test ...");
    }
}
chomsky$
```

Anmerkungen:

- Mit *cvs status* kann man sich den Zustand einer Datei anzeigen lassen. Dabei sehen wir, dass wir (Benutzer 1) mit einer veralteten Revision (1.2) arbeiten ($\rightsquigarrow$ „Needs Patch“).
- Mit *cvs update* kann Benutzer 1 nun seine Kopie von Test.java wieder auf den aktuellen Stand bringen. Da keine Datei angegeben ist, wird hier ein Update auf alle Dateien (auch in Unterverzeichnissen durchgrführt)! ($\rightsquigarrow$ CVS ist *rekursiv*)

5.2.6 Zusammenführen gleichzeitiger Änderungen

Momentan sind Benutzer 1 und 2 auf dem aktuellen Stand. Benutzer 1 macht nun Änderungen an der Datei Test.java und übernimmt diese ins Repository:

```
chomsky$ cat Test.java
public class Test {
    public static void main(String[] args) {
        System.out.println("1");
        System.out.println("Test ...");
    }
}
chomsky$ cvs commit -m "weitere Ausgabe" Test.java
Checking in Test.java;
/home/jmayer/cvs/pkg/Test.java,v  <--  Test.java
new revision: 1.4; previous revision: 1.3
done
chomsky$
```

Ebenso fügt Benutzer 2 eine Zeile in Test.java ein und will diese ins Repository übernehmen (nachdem Benutzer 1 dies bereits getan hat):

```
chomsky$ cat Test.java
public class Test {
    public static void main(String[] args) {
        System.out.println("Test ...");
        System.out.println("2");
    }
}
chomsky$ cvs commit -m "weitere Ausgabe" Test.java
cvs commit: Up-to-date check failed for ‘Test.java’
cvs [commit aborted]: correct above errors first!
chomsky$ cvs update Test.java
RCS file: /home/jmayer/cvs/pkg/Test.java,v
retrieving revision 1.3
retrieving revision 1.4
Merging differences between 1.3 and 1.4 into Test.java
M Test.java
chomsky$ cat Test.java
public class Test {
    public static void main(String[] args) {
        System.out.println("1");
        System.out.println("Test ...");
        System.out.println("2");
    }
}
chomsky$ cvs commit -m "weitere Ausgabe" Test.java
Checking in Test.java;
/home/jmayer/cvs/pkg/Test.java,v  <--  Test.java
new revision: 1.5; previous revision: 1.4
done
chomsky$
```

Anmerkungen:

- Das Einchecken klappt nun nicht sofort, da sich inzwischen ja das Repository geändert hat („Up-to-date check failed“).
- Zuerst muss jetzt also die Version mit dem Repository abgeglichen werden, was mittels *cvs update* passiert. Dies klappt hier sogar automatisch („merging“). Das Ergebnis ist, dass unsere lokale Kopie die Ausgabezeile von Benutzer 1 zusätzlich enthält. Diese konnte *konfliktfrei* eingefügt werden, da Benutzer 2 an einem anderen Bereich der Datei gearbeitet hat.
- Nach dem Update klappt das Einchecken der Datei dann problemlos.

5.2.7 Zusammenführen von Änderungen mit Konflikten

Nun führt Benutzer 1 wieder ein *cvs update* aus, um auf dem aktuellen Stand zu sein – genauso wie Benutzer 2. Danach fügt er ans Ende der main-Methode eine Ausgabe an und checkt die lokale Kopie ins Repository ein:

```
chomsky$ pwd
/home/jmayer/user1/pkg
chomsky$ cvs update Test.java
U Test.java
chomsky$ # Test.java ändern
chomsky$ cat Test.java
public class Test {
    public static void main(String[] args) {
        System.out.println("1");
        System.out.println("Test ...");
        System.out.println("2");
        System.out.println("3");
    }
}
chomsky$ cvs commit -m "noch eine weitere Ausgabe" Test.java
Checking in Test.java;
/home/jmayer/cvs/pkg/Test.java,v  <--  Test.java
new revision: 1.6; previous revision: 1.5
done
chomsky$
```

Benutzer 2 führt jetzt auf seiner lokalen Kopie auch eine Änderung durch: Er fügt ebenfalls etwas ans Ende der main-Methode an. Da ein *cvs commit* wie eben nicht funktioniert, muss zunächst ein *cvs update* erfolgen, was hier aber zu *„conflicts“* führt. Was ist der Grund dafür? Beide Änderungen (von Benutzer 1 und 2) beziehen sich auf denselben Bereich im Quelltext!

```
omsky$ pwd
/home/jmayer/user2/pkg
chomsky$ # Test.java ändern
chomsky$ cat Test.java
public class Test {
    public static void main(String[] args) {
        System.out.println("1");
        System.out.println("Test ...");
        System.out.println("2");
        System.out.println("4");
    }
}
chomsky$ cvs commit -m "noch eine weitere Ausgabe" Test.java
cvs commit: Up-to-date check failed for ‘Test.java’
cvs [commit aborted]: correct above errors first!
chomsky$ cvs update Test.java
RCS file: /home/jmayer/cvs/pkg/Test.java,v
retrieving revision 1.5
retrieving revision 1.6
Merging differences between 1.5 and 1.6 into Test.java
rcsmerge: warning: conflicts during merge
cvs update: conflicts found in Test.java
C Test.java
chomsky$
```

Nach dem Update ist der Bereich im Quelltext gekennzeichnet, der nicht automatisch zusammengeführt werden konnte:

```
chomsky$ cat Test.java
public class Test {
    public static void main(String[] args) {
        System.out.println("1");
        System.out.println("Test ...");
        System.out.println("2");
<<<<<<< Test.java
        System.out.println("4");
=======
        System.out.println("3");
>>>>>>> 1.6
    }
}
chomsky$
```

Nach dem Benutzer 2 nun von Hand eingegriffen hat und den Konflikt beseitigt hat, kann die Datei ganz normal „eingecheckt“ werden:

```
chomsky$ # Test.java ändern
chomsky$ cat Test.java
public class Test {
    public static void main(String[] args) {
        System.out.println("1");
        System.out.println("Test ...");
        System.out.println("2");
        System.out.println("3");
        System.out.println("4");
    }
}
chomsky$ cvs commit -m "noch eine weitere Ausgabe" Test.java
Checking in Test.java;
/home/jmayer/cvs/pkg/Test.java,v  <--  Test.java
new revision: 1.7; previous revision: 1.6
done
chomsky$
```

5.2.8 Dateien hinzufügen

Benutzer 1 legt nun die Datei *Main.java* neu an und fügt sie mittels *cvs add* dem Repository hinzu. Dies passiert bei regulären Dateien – im Gegensatz zu Verzeichnissen – jedoch nicht sofort, sondern erst beim nächsten *cvs commit*:

```
chomsky$ pwd
/home/jmayer/user1/pkg
chomsky$ cat Main.java
public class Main {
    public static void main(String[] args) {
        System.out.println("Hello world!");
    }
}
chomsky$ cvs add Main.java
cvs add: scheduling file ‘Main.java’ for addition
cvs add: use ‘cvs commit’ to add this file permanently
chomsky$ cvs commit -m "Main.java neu" Main.java
RCS file: /home/jmayer/cvs/pkg/Main.java,v
done
Checking in Main.java;
/home/jmayer/cvs/pkg/Main.java,v  <--  Main.java
initial revision: 1.1
done
chomsky$
```

Durch ein *cvs update* kann nun auch Benutzer 2 seine lokale Kopie wieder auf den aktuellsten Stand bringen:

```
chomsky$ pwd
/home/jmayer/user2/pkg
chomsky$ cvs update
cvs update: Updating .
U Main.java
chomsky$ ls
CVS  Main.java  Test.java
chomsky$
```

5.2.9 Revisionen und Releases

Wie Sie eben beim „Einchecken“ gesehen haben, ging die Revision von Test.java von 1.1 nach 1.2. CVS verwaltet für jede Datei alle Revisionen. Bei CVS ist es aber so, dass verschiedene Dateien im Repository unterschiedliche Revisionsnummern haben können. Will man daher einen bestimmten Stand beschreiben, z. B. Release 1, so kann man alle die Dateien im Repository (unabhängig von Ihrer Revision) mit dem Tag „release1“ versehen. Dann kann man Dateien unterschiedlicher Revision über das vergebene Tag trotzdem als zusammengehörig identifizieren. Ein Tag kann man mittels *cvs tag* bzw. *cvs rtag* vergeben. Ersteres ist besser, da es die „ausgecheckten“ Revisionen „taggt“ und somit resistent gegenüber gleichzeitigen Änderungen ist.

Benutzer 2 zeigt nun zuerst die ausgecheckten Revisionen an (mit *cvs status*) und vergibt dann das Tag *release1* an diesen Stand:

```
chomsky$ pwd
/home/jmayer/user2/pkg
chomsky$ cvs status
cvs status: Examining .
===================================================================
File: Main.java         Status: Up-to-date

    Working revision:    1.1     Wed Dec  1 23:06:49 2004
    Repository revision: 1.1     /home/jmayer/cvs/pkg/Main.java,v
    Sticky Tag:          (none)
    Sticky Date:         (none)
    Sticky Options:      (none)

===================================================================
File: Test.java         Status: Up-to-date

    Working revision:    1.7     Wed Dec  1 22:53:10 2004
    Repository revision: 1.7     /home/jmayer/cvs/pkg/Test.java,v
    Sticky Tag:          (none)
    Sticky Date:         (none)
    Sticky Options:      (none)

chomsky$ cvs tag release1
cvs tag: Tagging .
T Main.java
T Test.java
chomsky$
```

5.2.10 Änderungshistorie

Mit dem Kommando *cvs annotate* kann man anzeigen lassen, wer wann welche Zeile beigesteuert hat:

```
chomsky$ cvs annotate Test.java

Annotations for Test.java
***************
1.1          (jmayer   01-Dec-04): public class Test {
1.1          (jmayer   01-Dec-04):     public static void main(String[] args) {
1.4          (jmayer   01-Dec-04):         System.out.println("1");
1.3          (jmayer   01-Dec-04):         System.out.println("Test ...");
1.5          (jmayer   01-Dec-04):         System.out.println("2");
1.6          (jmayer   01-Dec-04):         System.out.println("3");
1.7          (jmayer   01-Dec-04):         System.out.println("4");
1.1          (jmayer   01-Dec-04):     }
1.1          (jmayer   01-Dec-04): }
chomsky$
```

5.2.11 Vergleich von Revisionen

Wie sich mit *cvs status* zeigt, arbeitet Benutzer 1 noch mit Revision 1.6. Mit *cvs diff* kann er nun vergleichen, was sich gegenüber seiner Version in Revision 1.6 bzw. Release „release1“ geändert hat:

```
chomsky$ cvs status Test.java
===================================================================
File: Test.java         Status: Needs Patch

   Working revision:    1.6     Wed Dec  1 22:50:33 2004
   Repository revision: 1.7     /home/jmayer/cvs/pkg/Test.java,v
   Sticky Tag:          (none)
   Sticky Date:         (none)
   Sticky Options:      (none)

chomsky$ cvs diff -r 1.7 Test.java
Index: Test.java
===================================================================
RCS file: /home/jmayer/cvs/pkg/Test.java,v
retrieving revision 1.7
retrieving revision 1.6
diff -r1.7 -r1.6
7d6
<         System.out.println("4");
chomsky$ cvs diff -r release1 Test.java
Index: Test.java
===================================================================
RCS file: /home/jmayer/cvs/pkg/Test.java,v
retrieving revision 1.7
retrieving revision 1.6
diff -r1.7 -r1.6
7d6
<         System.out.println("4");
chomsky$
```

Anmerkung: Bei Verwendung von „release1“ wird von Test.java die Revision 1.7 gewählt, da diese mit dem angegebenen Tag versehen wurde.

5.2.12 Dateien aus dem Repository entfernen

Mit *cvs remove* gefolgt von *cvs commit* kann man eine Datei aus dem Repository löschen, wenn man diese zuvor im Dateisystem (in der Arbeitskopie) gelöscht hat:

```
chomsky$ pwd
/home/jmayer/user1/pkg
chomsky$ rm Main.java
chomsky$ cvs remove Main.java
cvs remove: scheduling ‘Main.java’ for removal
cvs remove: use ‘cvs commit’ to remove this file permanently
chomsky$ cvs commit -m "Main.java entfernt" Main.java
Removing Main.java;
/home/jmayer/cvs/pkg/Main.java,v  <--  Main.java
new revision: delete; previous revision: 1.1
done
chomsky$
```

5.2.13 Zurück zu alter Revision

Zunächst sehen wir mal, mit welcher Revision wir eigentlich arbeiten:

```
chomsky$ cvs status
cvs status: Examining .
===================================================================
File: Test.java         Status: Up-to-date

   Working revision:    1.7     Wed Dec  1 23:33:18 2004
   Repository revision: 1.7     /home/jmayer/cvs/pkg/Test.java,v
   Sticky Tag:          (none)
   Sticky Date:         (none)
   Sticky Options:      (none)

chomsky$ cat Test.java
public class Test {
    public static void main(String[] args) {
        System.out.println("1");
        System.out.println("Test ...");
        System.out.println("2");
        System.out.println("3");
        System.out.println("4");
    }
}
chomsky$
```

Mit *cvs update* können wir nun die Unterschiede zwischen der aktuellen Revision und einer vorigen Revision rückgängig machen. Dazu geben wir mit *–j 1.7* zunächst die aktuelle Revision und dann mit *–j 1.5* die gewünschte Revision, zu der wir zurürckkehren wollen, an:

```
chomsky$ cvs update -j 1.7 -j 1.5
cvs update: Updating .
RCS file: /home/jmayer/cvs/pkg/Test.java,v
retrieving revision 1.7
retrieving revision 1.5
Merging differences between 1.7 and 1.5 into Test.java
chomsky$ cat Test.java
public class Test {
    public static void main(String[] args) {
        System.out.println("1");
        System.out.println("Test ...");
        System.out.println("2");
    }
}
chomsky$
```

Diesen Stand könnten wir jetzt als aktuelle Revision mittels *cvs commit* ins Repository aufnehmen.

5.2.14 Zusammenfassung

CVS besitzt also die folgenden Kommandos:

Kommando	Erklärung
cvs import	Importieren eines neuen Moduls
cvs checkout	„Auschecken“ einer lokalen Arbeitskopie
cvs commit	„Einchecken“ einer lokalen Arbeitskopie
cvs update	Aktualisieren der lokalen Arbeitskopie
cvs add	Dateien und Verzeichnisse zu Modulen hinzufügen
cvs tag	Release-Tag auf die (ausgecheckten) Revisionen vergeben
cvs annotate	Änderungshistorie anzeigen
cvs status	Revision etc. anzeigen
cvs diff	Revisionen miteinander vergleichen
cvs remove	Dateien aus dem Repository entfernen

5.3 Literatur

[Budszuhn04] Budszuhn, F.: *CVS.* Galileo Computing, 2004.

[Tilly02] Tilly, J.; Burke, E. M.: *Ant: The Definitive Guide.* O’Reilly, 2002.

Teil II

Software Engineering

Kapitel 6

Einleitung

6.1 Hardware – Software

Rasante Entwicklung der Hardware-Technik / Elektronik (Prozessoren und Chips):
immer schneller, kleiner, zuverlässiger, preiswerter

Funktionalität von technischen Geräten wie betrieblichen Informationssystemen wird immer mehr bzw. primär durch Software bestimmt!!!

Software:

> Bestehende Software-Produkte werden mit immer mehr Funktionalität „angereichert“, es wird selten etwas „weggenommen“, was sowohl die Kompliziertheit wie auch die Komplexität in erheblichem Maße steigert.

N. Wirth: **fat software** ([?]:

- *Die Software-Industrie steht vor dem Dilemma, entweder ihre bisherigen Investitionen in der Form von Programmen oder Systemen beizubehalten oder von den modernen Paradigmen und Methoden der Software-Entwicklung zu profitieren. Wie in solchen Fällen üblich, sucht sie einen Kompromiß und hofft, die Lösung im Hinzufügen von neuen Features in alten Systemen zu finden. Dies führt jedoch unweigerlich zu immer umfangreicheren Konstruktionen, deren Inneres komplex und undurchsichtig wird und die viel beschworenen Prinzipien der systematischen Struktur, Klarheit und Zuverlässigkeit Lügen straft. Solche Systeme sind nur noch dank der stetig wachsenden Leistung moderner Hardware implementierbar, wobei letztere oft sehr ineffektiv eingesetzt wird.*
- *Erste Ansätze bei der Software-Konstruktion führen meist zu komplexen Gebilden. Erst wiederholte Verbesserungen oder gar Neuanfänge unter Berücksichtigung der erhaltenen Erkenntnisse erbringen die gewünschte Qualität. Die erfolgreichsten Schritte sind diejenigen, die eine Vereinfachung herbeiführen oder gar Teile eines Programms als überflüssig erscheinen lassen. Entwicklungen dieser Art trifft man jedoch in der Praxis selten an, weil sie zeitaufwendiges Überdenken erfordern, das meistens kaum belohnt wird. Statt dessen werden Unzulänglichkeiten durch rasch gefundene Zusätze überdeckt, deren Summe schließlich zur bekannten fat software führt.*

6.2 Begriffe

Software:

- *(engl., eigtl. „weiche Ware"), Abk. SW, Sammelbezeichnung für Programme, die für den Betrieb von Rechensystemen zur Verfügung stehen, einschl. der zugehörigen Dokumentation* (Brockhaus-Enzyklopädie).

- *Computer programs, procedures, rules, and possibly associated documentation and data pertaining to the operation of a computer system (IEEE Standard Glossary of Software Engineering Terminology)*

- *Software* besteht aus Anweisungen und Daten an / für die „Maschine" (konkrete / virtuelle Maschine) **und** Anweisungen / Informationen an „Menschen", die einen **effizienten / effektiven Umgang** mit dieser Software ermöglichen. Dieser „Umgang" kann sein:

Entwicklung	Weiter-Entwicklung	Pflege	Prüfung
Benutzung	Operating	Portierung	...

Software-Entwicklung:

Vorrangige Berücksichtigung der mit der zu erstellenden Software verbundenen Prozesse

- bei Anwendungssoftware die Anwendungsprozesse,

- bei technischer Software die technischen Prozesse,

- aber auch die zukünftig damit verbundenen Prozesse wie z. B. die Pflege (Aufrechterhaltung der Gebrauchstauglichkeit)

„Probleme" in der Entwicklung von Informationssystemen / Anwendungssoftware ([?], IBM, Schweiz – frei übersetzt):

> *Wir können drei verschiedene Kern-Systeme in jeder Art von Projekt differenzieren, die aber in gewisser Beziehung zueinander stehen:*
>
> - *Es wird ein System entwickelt, das kurz als Produkt oder Anwendung bezeichnet werden kann. Dieses System wird vom Kunden angenommen und soll bei ihm die Ergebnisse liefern, für die es konzipiert wurde.*

- *Es gibt ein System, mit dem o.g. System entwickelt wird. Dieses kann als Management-System bezeichnet werden und beschreibt das Projekt-Management, den Test, die Qualitätssicherung. Dieses „stirbt“ spätestens bei Projektende.*

- *Es gibt ein „Umgebungs-System“ – das Produkt läuft schließlich in einem Anwendungsumfeld, mit dem auch o.g. Management-System leben muss. Das Produkt braucht Ressourcen vom Kunden und muss letztlich so entworfen und entwickelt werden, das das „Umgebungs-System“ damit zurechtkommt. Dies geht weit über funktionale Anforderungen hinaus, es sind Aspekte wie Industrie-Standards, Kultur oder Verhaltensweisen zu berücksichtigen.*

Analysiert man Qualitätsprobleme bei Software, so kann man in etwa von folgender Verteilung ausgehen:

System	Problem-Anteil
Produkt	10%
Management	70%
Anwendungsumgebung	20%

Besonderheiten von Software im Vergleich zu anderen technischen Produkten:

- immaterielles Produkt
- kein Verschleiß, aber „Alterung“ (Anforderungen ändern sich)
- im Prinzip Unikat
- schwer zu „vermessen“
- im Prinzip keine physikalischen Grenzen (menschlicher Geist!)
- leicht(!) und schnell(!) änderbar (i.P. Textverarbeitung)

Veränderungen in der Software-Entwicklung in den letzten 10-20 Jahren:

- Immense Bedeutung für Industrie und Gesellschaft
- Wachsende Komplexität von Software
- Nachfragestau und Engpassfaktor
- Mehr und mehr ‘Standard’software
- Problem der Altlasten (Jahr2000-Problem, sog. Aufwärtskompatibilität)
- Zunehmend ‘Außer-Haus’-Entwicklung
- Zunehmende Qualitätsanforderungen (Rechtssprechung, Markt-Konkurrenz, . . .)
- . . .

Software-Komplexität:

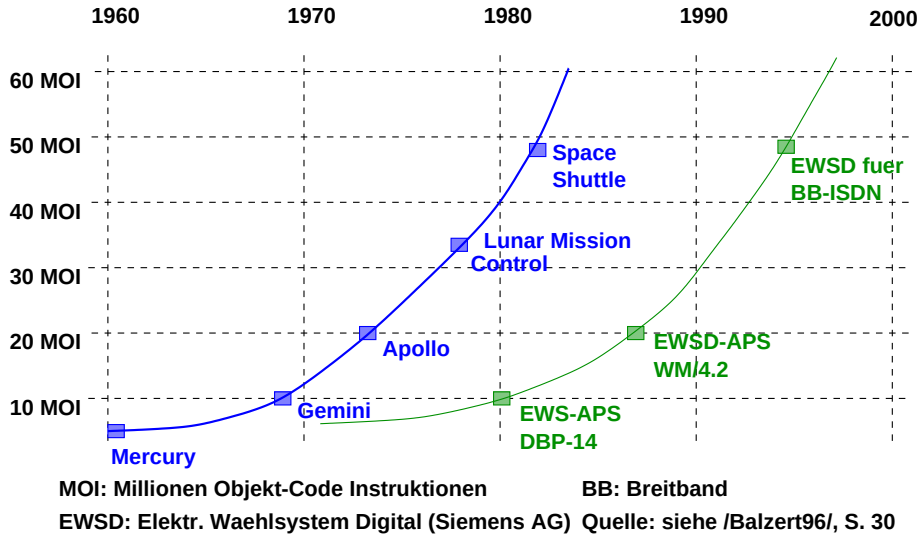


Abbildung 6.1: Software-Komplexität

Bedeutung in der Technik:

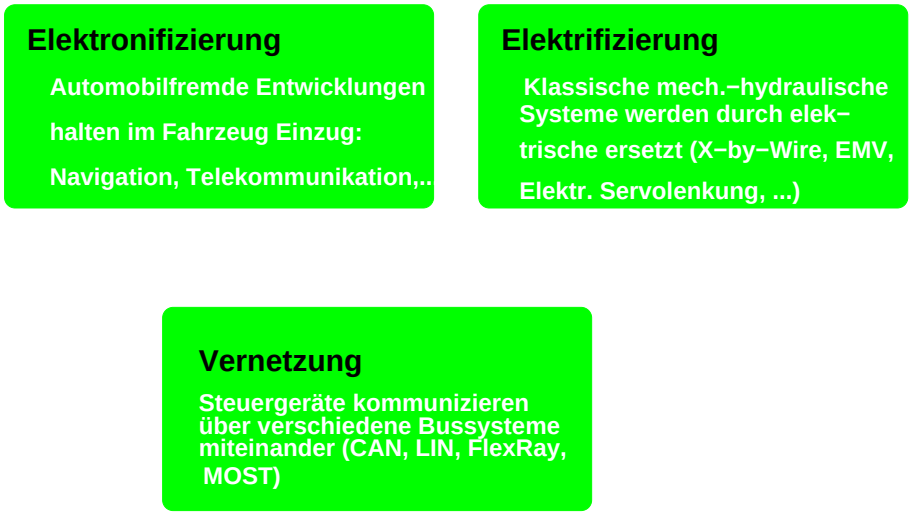


Abbildung 6.2: Innovation durch Software

Grundlegende Problematik:

- Während der Entwicklung ändern sich
  - die Produktanforderungen
  - die HW- und System-SW-Komponenten
  - die Produktionsmittel
- Hohe Portabilitätsanforderungen sind einzuhalten, da
  - Software-Produkte eine meist wesentlich längere Lebensdauer besitzen als die zugrundliegende Hardware und
  - oft auf mehreren Plattformen eingesetzt werden.
- Entwickelt man heute Anwendungssoftware, dann bedeutet dies, dass
  - während der Lebenszeit der Anwendungssoftware mindestens einmal die zugrunde liegende Systemsoftware und mindestens zweimal die Hardware ausgetauscht wird,
  - die Zielsysteme, für die die Software entwickelt wird, zur Entwicklungszeit oft noch nicht vorhanden sind,
  - weltweit vertriebene Software-Produkte leicht länderspezifische Varianten erlauben müssen (Oberflächen, gesetzl. Randbedingungen, ...)

Software Engineering:

> *das ingenieurmäßige Entwerfen, Herstellen und Implementieren von Software sowie die ingenieurwissenschaftliche Disziplin, die sich mit Methoden und Verfahren zur Lösung der damit verbundenen Problemstellungen befasst*
> (Brockhaus-Enzyklopädie)

Software Engineering umfasst unter anderem

- Einsatz bewährter Methoden des Projektmanagements (wie Netzplantechnik, Qualitäts-, Risiko-, Ressourcenmanagement)
- Methodischer Entwurf unter besonderer Berücksichtigung von Arbeitsteilung, Prüfbarkeit, Integrierbarkeit, Wiederverwendbarkeit, Pflegbarkeit, u.dgl.m.
- **Frühzeitige** Überlegungen zur Validation (z. B. durch Prototyping) und zur Verifikation (z. B. parallel zur Konstruktion laufende Testplanungen)
- **Systematische** Fehlersuche und Fehleranalyse
- Prinzipien der Entwicklung gelten auch für die Pflege

Projekt – Charakteristika:

- einmaliges Vorhaben (hebt sich von der Tagesroutine ab)
- zeitlich begrenzt durch Start- und Endtermine
- (mehr oder weniger) klare Ziele
- komplexes Vorhaben mit verschiedenen Techniken und Methoden
- **Zusammenarbeit von Menschen aus unterschiedlichen Fachgebieten mit unterschiedlichen Kenntnissen und unterschiedlichen Sprech- / Denkgewohnheiten und Realitätsvorstellungen** („Modellen“ der Realität)
- neue, unbekannte Probleme (für die Beteiligten)
- limitiertes Budget
- Risiken (Termine – Kosten – neuartige, unausgereifte Techniken – Verfügbarkeit von Spezialisten – Mitarbeit des Auftraggebers – u.v.a.m)
- besonderer Druck für die Beteiligten

6.3 Systeme und Modelle

- **Systemtheorie** ↪ stellt Definitionen für eine abstrahierende Sichtweise auf beliebige Sachverhalte zur Verfügung

- **System** ↪ Anordnung von Gebilden (für die jeweilige Sicht abgrenzbaren Elemente), die aufeinander durch Relationen einwirken und die durch eine Hüllfläche, die Systemgrenze, von ihrer Umgebung abgegrenzt sind (offene vs. geschlossene Systeme)

- **Systemkonzepte**
  ↪ funktionale, strukturelle, hierarchische Betrachtung (ergänzende Sichtweisen)

  – Beim **funktionalen** Konzept steht die Wandlung von Eingabeoperanden zu Ausgabeoperanden, durch die die Funktion des Systems beschrieben wird, im Betrachtungsmittelpunkt

  – Das **strukturelle** Konzept stellt allgemein „Elemente“, ihr „Verhalte“n und die sie verknüpfenden „Relationen“ in den Vordergrund. Ergänzend wird das System gegenüber der Umgebung abgegrenzt.

  – Das **hierarchische** Konzept beschreibt ein System durch sukzessives Zerlegen in Teilsysteme

Unternehmen / Projekt als System:

- **künstlich** (von Menschenhand geschaffen)
- **dynamisch** (wechselndes Verhalten der einzelnen Elemente zueinander)
- **real** (weitgehend gegenständlich, beobachtbar)
- **offen** (Beziehungen zur Umwelt)
- **probabilistisch** (Vorhersagen über zukünftiges Verhalten können nur mit einer gewissen Wahrscheinlichkeit, die jedoch mit dem Verständnis des Systems zunimmt, gemacht werden)
- **komplex** (hohe Zahl von Zuständen, in denen sich das System befinden kann)

Modell:

- Ein Modell ist ein System, das ein anderes (reales) System abbildet.
- Eine **Modellbildung** erfordert geeignete Modellierungsmethoden.
- Eine **Methode** ist ein Vorgehensprinzip zur Lösung von Aufgaben.
- Eine Modellierungsmethode umfasst Konstrukte und eine Vorgehensweise, die beschreibt, wie diese Konstrukte wirkungsvoll bei der Modellbildung anzuwenden sind.
- Konstrukte umfassen die Elemente einer „Beschreibungssprache“ und die Regeln für die Verknüpfung der Elemente.
- **Modell-Merkmale:** das Abbildungsmerkmal, das Verkürzungs-/Abstraktions-Merkmal, das pragmatische Merkmal
  - Das **Abbildungsmerkmal** kennzeichnet das Modell als Abbild des realen Systems. Man unterscheidet zwischen isomorphen (strukturgleichen) und homomorphen (strukturähnlichen) Modellen.
  - Das **Verkürzungsmerkmal** ergibt sich daraus, dass das Modell nur die Elemente und Relationen des realen Systems abbildet, die dem Modellbildner (subjektiv) als relevant erscheinen.
  - Das **pragmatische Merkmal** (Pragmatik: Ziel- / Zweckorientierung) bezeichnet Ziel und Zweck der Modellbildung.
    Man kann unterscheiden in Beschreibungs- / Bezugs-, Erklärungs- und Entscheidungsmodelle.

Zweck von **Beschreibungsmodellen** ist die Informationsgewinnung über die Beschaffenheit eines Systems in einem als statisch angenommenen Zustand durch eine Beschreibung des Systems und seiner Entscheidungssituationen. Es kann als Bezug / Ausgangspunkt für weitere Modelle sein (↪ Software-Entwicklung).

Erklärungsmodelle sollen reale Erscheinungen eines Systems erklären und prognostizieren, also Aussagen über künftige Systemzustände ermöglichen.

Entscheidungsmodelle sollen bestimmte Handlungsmassnahmen aus vorgegebenen Zielsetzungen, Randbedingungen und Entscheidungsvariablen ableiten.

Kapitel 7

Anwendungssysteme / Anwendungsarchitekturen

7.1 Einleitung

Globale Unternehmensziele (z. B.):

- Verbesserung der Rentabilität des eingesetzten Kapitals,
- Erhöhung der Marktanteile,
- Erschliessung neuer Absatzmärkte durch neue Produkte,

Abgeleitete Ziele (z. B.):

- Verbesserung der Kundenzufriedenheit oder
- Senkung der Kosten

Unterziele (z. B.):

- Verbesserung der Termintreue oder des Kundenservice,
- schnellere Reaktion auf Kundenanfragen,
- Produkte, die sich am Bedürfnis der Kunden stärker orientieren,
- Reduktion der Kosten (des Produkts wie auch dessen Nutzung)

Folge: Veränderungen in den bisherigen Unternehmensstrukturen, sowohl auf Seiten der Ablauforganisation (Veränderung der Geschäftsprozesse), der Aufbauorganisation („Träger“ der Geschäftsprozesse) wie auch in der Unternehmenskultur (Entscheidungs-, Berichtswege, Motivation, . . .).

Kurz: **Verbesserung der Qualität der Geschäftsprozesse**

> **Qualität** ist die Gesamtheit von Eigenschaften und Merkmalen eines Produkts oder einer Tätigkeit, die sich auf deren Eignung zur Erfüllung gegebener Erfordernisse beziehen

„Erfordernisse“ abgeleitet aus den Unternehmenszielen ↪ Strategien: mit welchen grundlegenden Vorgehensweisen, welchen Maßnahmen und mit welchen Hilfsmitteln können diese „Erfordernisse“ erfüllt werden

a) Das Bestehende wird vollkommen durch etwas Neues ersetzt („Neubau einer Produktionsstätte“) mit neu geschaffenen Aufbau- und Ablaufstrukturen.

b) Das Bestehende wird vollständig „umgekrempelt“.

c) Das Bestehende wird schrittweise verändert und auf die neuen Anforderungen hin „entwickelt“.

Notwendig: Modell der betrieblichen Abläufe, der notwendigen Funktionen und der zugrundeliegenden Datenstrukturen, mit dem die Zielerreichung und die Zielverfolgung (Strategie) sichergestellt werden kann (**Soll-Modell**).

Dieses Modell kann nun realisiert werden

(1) durch vollständige Neu-Entwicklung von Software (**Individual-Software**),

(2) durch Einsatz (neu zu beschaffender) **Standard-Software** oder

(3) durch schrittweise Weiterentwicklung bestehender Software.

(2) und (3) ↪ Ist-Modell der realisierten / realisierbaren betrieblichen Abläufe, der gegebenen Funktionalität und der zugrundliegenden Datenstrukturen (z. B. über Referenzmodelle)

Architektur:

- ein Modell, bestehend aus betrieblichen Abläufen (Geschäftsprozessen) zur Leistungserbringung, Funktionalitäten und zugrundliegenden Datenstrukturen
- zusätzlich: Vorstellungen, wie die Erstellung solcher Modelle erfolgen kann
- Systemtheorie: funktionales, strukturelles und auch hierarchisches System-Konzept

7.2 ARIS-Architektur

Geschäftsprozessorientierte Software-Entwicklung:

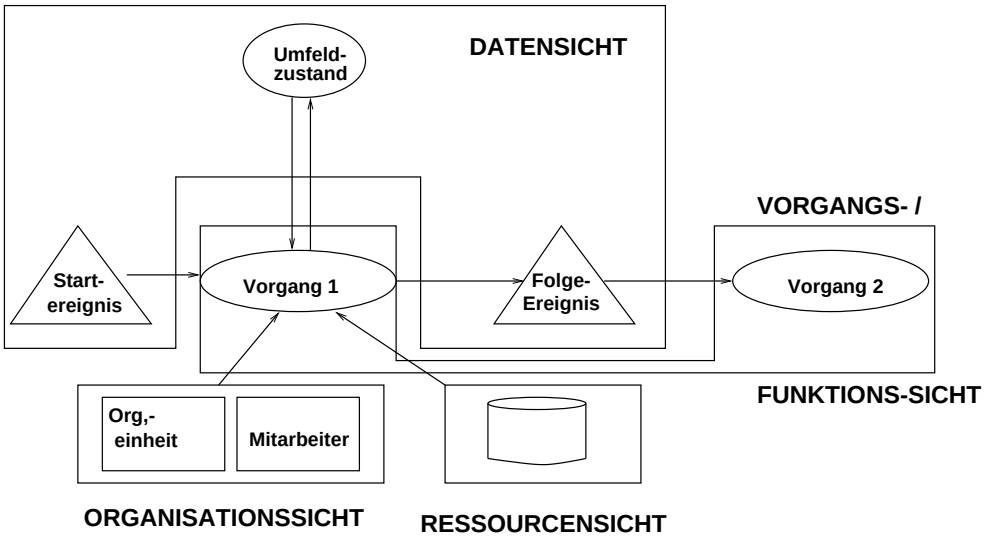


Abbildung 7.1: ARIS

zusätzlich: Steuerungssicht

betriebswirtschaftliche Anwendungskonzept → Fachkonzept → DV-Konzept → Implementierung → Betrieb/Wartung

7.3 Beispiel: Versicherungs-Anwendungs-Architektur

siehe dazu insbesondere **www.gdv-online.de/vaa/**

7.3.1 Fachliche Anwendungsarchitektur

Orientiert an fachlichen Aufgaben:

- Vertriebswege: Außendienstmitarbeiter, Makler, Banken, Internet
  - CRM
    - Kundenübersicht
    - Kontaktmanagement
  - Akquisition
    - Vertriebsunterstützung
    - Angebot / Antrag
    - Kampagnenmanagement
- Fachabteilungen, Sachbearbeitung, Mathematik
  - Partner
    - Partnerverwaltung
  - Vertrieb / Provision
    - Vermittler
    - Provision
  - Risiko
    - Risikoprüfung
    - Rück- / Mitversicherung
  - Produkt
    - Produktentwicklung (Einzel / Kollektiv)
    - Produktdefinition (Einzel / Kollektiv)
    - Mathematik
    - betriebliche Altersversorgung (bAV) – Versorgungsordnungen
    - Pensionskassen / Pensionsfonds
  - Vertrag
    - Bestand
    - Schaden / Leistung
  - Rechnungswesen
    - Inkasso / Exkasso
    - Finanzbuchhaltung

- Interne Funktionen: Betriebsorganisation (BO), Informationstechnik (IT), *Human Ressources* (HR)
  - Basisdienste
    - Benutzerverwaltung
    - Workflowmanagement
  - Dokumente
    - Dokumentenmanagement
  - Kapitalanlage
    - Kapitalanlage
    - Fonds

- Management: Vorstand, Controlling, Marketing
  - Unternehmenssteuerung / *Business Information*
    - Analysen / Reports
    - Asset-Liability-Management (ALM) / Risikomanagement
    - Rechnungslegung

IT-Unterstützung – nicht isoliert, sondern im Systemverbund:

- CRM-Systeme (spartenunabhängig)
- Vertriebsunterstützungssystem (spartenunabhängig)
- Partnersystem (spartenunabhängig)
- Produktsystem / Produktserver (spartenunabhängig)
- Provisionssystem (spartenunabhängig)
- In/Exkasso-System (spartenunabhängig)
- Finanzbuchhaltungssystem (spartenunabhängig)
- Data Warehouse (spartenunabhängig)
- Bestandsführungssystem (spartenspezifisch)
- Risikoprüfungssystem (spartenspezifisch)

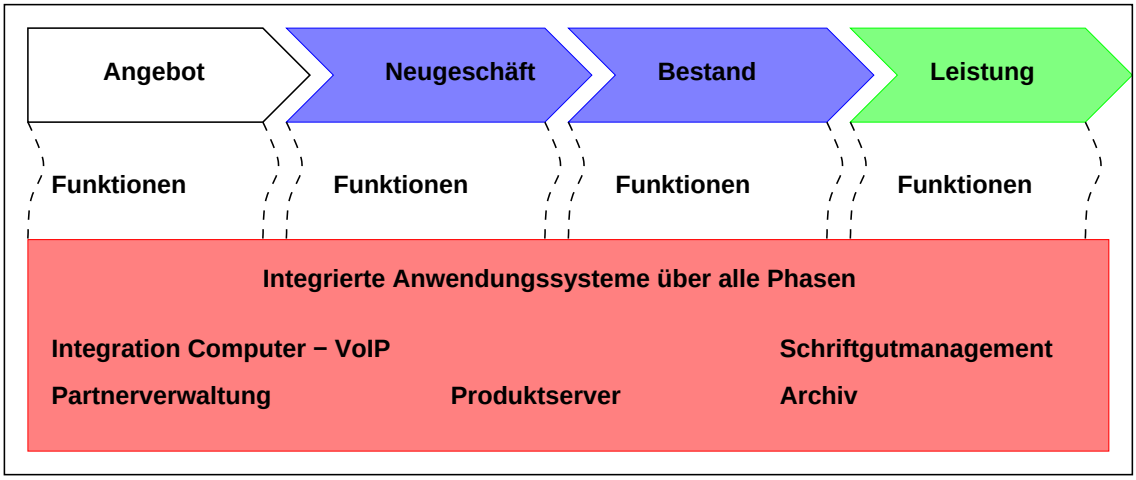


Abbildung 7.2: Integrierte Systeme – Beispiel: Bestandsführung

Beispiele für die Funktionen

- Angebot: Darstellung der Angebote, effektive Umsetzung der vertrieblichen Kernprozesse, integrierte Versicherungsmathematik, ...
- Neugeschäft: Effiziente Erfassung, Plausibilitäts- / Vollständigkeitsprüfungen, Archivierung, Prozessverfolgung (Terminüberwachung, Wiedervorlagen), integrierte Provisionierung, ...
- Bestand: maschinelle Abwicklung alles planmäßigen Geschäftsvorfälle, maschinelle Unterstützung aller außerplanmäßigen Geschäftsvorfälle inkl. Historie, maschinelle Ermittlung der technischen Bilanzwerte, integriertes Inkasso, ...
- Leistung: maschinelle Abwicklung alles planmäßigen Leistungsanforderungen, maschinelle Unterstützung aller außerplanmäßigen Leistungsanforderungen, integriertes Exkasso, ...

7.3.2 Ziele der VAA

- Schaffung der Voraussetzungen für einen Komponentenmarkt, der Anwendungssoftware für **VU** (VU: Versicherungsunternehmen) anbietet
- Förderung der Entwicklung flexibler, offener, herstellerunabhängiger Informationssysteme für die VU, und damit auch langfristige Kostenreduzierung und Senkung von Entwicklungsrisiken für das einzelne Unternehmen
- Geschäftsprozessorientierung und Ausnutzung der damit verbundenen Chancen
- Schnelle und preiswerte Anpassung von Software an die sich unter dem Marktdruck immer schneller wandelnden Geschäftsprozesse
- Performante Verarbeitung auch für große VU

VAA bietet generellen Rahmen, innerhalb dessen

- ein übergreifendes Konstruktionsprinzip herrscht,
- durch Normung von Schnittstellen die Integration von Anwendungen unterstützt wird,
- für generelle Aufgaben Standardbausteine bereitstehen,
- die Endbenutzer im Verhalten „ähnliche“ Systeme erhalten,
- die Anwendungen für den Entwickler transparenter sind und
- der Entwicklungsprozess optimiert wird.

7.3.3 Geschäftsprozessorientierung

Unter einem **Geschäftsprozess** (Synonyme: Vorgangstyp, Workflow) versteht man einen abgrenzbaren, meist arbeitsteiligen Prozess zur Erstellung/Verwertung betrieblicher Leistungen (z. B.: Neugeschäft „Antrag bearbeiten"), bei dem im Mittelpunkt der dynamische Ablauf steht; dieser Prozess wird initiiert durch Auslöser (zeitliches, externes oder internes Ereignis) und besteht aus elementaren Arbeitsschritten (sequentiell, selektiv, iterativ, parallel). Geschäftsprozesse werden auf mehreren Detaillierungsebenen (Teilprozess, Elementarprozess) verfeinert.

Ein **Arbeitsschritt** (Synonym: Vorgangsschritt, Aktivität, Arbeitsgang, Elementarprozess) ist eine Tätigkeit, die aus fachlicher Sicht nicht mehr sinnvoll zu unterteilen ist und als manuelle, DV-gestützte oder DV-Funktion umgesetzt wird.

Geschäftsprozessorientierung bedeutet in vielen Fällen eine grundlegende Organisationsveränderung und eine intensive Nutzung der Informations- und Kommunikations-Technik (Schlagworte: Geschäftsprozess-Optimierung, *Workflow-Management*, *Process Re-Engineering*, ...).

Process Re-Engineering ist das fundamentale Überdenken und der radikale Re-Design der Geschäftsprozesse, um dramatische Verbesserungen bzgl. kritischer Größen wie Kosten, Qualität, Service und Zeit zu erreichen.

Noch vorzufindende Ausgangsituation in VU:

(ebenso in vielen anderen Branchen):

- Spartenspezifische Aufbau- / Ablauforganisation
- betriebliche Abläufe geprägt von starker Arbeitsteiligkeit (Taylorismus)
- Konsequenzen:
  - lange Liege- wie Durchlaufzeiten wegen ineffizienter und ineffektiver Arbeitsteilung
  - geringe Produktivität durch lange Bearbeitungszeiten, auch bei erhöhtem IT-Einsatz
  - fehlende ganzheitliche Unterstützung betrieblicher Abläufe
  - häufige Medienbrüche und Neuerfassungen wegen fehlender Integration von Daten und Texten

Ergo: Optimierung der Geschäftsprozesse („Business Process Reengineering")

Ziele des *Process Re-Engineering* können sein:

- schnelle Reaktion auf Marktveränderungen
- Vereinfachung und Beschleunigung der Prozesse
- Vermeidung von Doppelarbeiten
- Verringerung der Kosten
- Verbesserung des Kundenservice (kurze Reaktionszeiten auf Anfragen)
- Flexibilität, Transparenz und Wiederverwendbarkeit der Prozesse (z. B. um Arbeitsschritte der Angebots-/Antragsbearbeitung in den Aussendienst zu verlagern)
- Schaffung integrierter Informationssysteme zur Produktentwicklung, Vertragsbearbeitung und Vertriebsunterstützung

Geschäftsprozessunterstützung durch Workflow-Managementsysteme (nach [Heilmann1994])

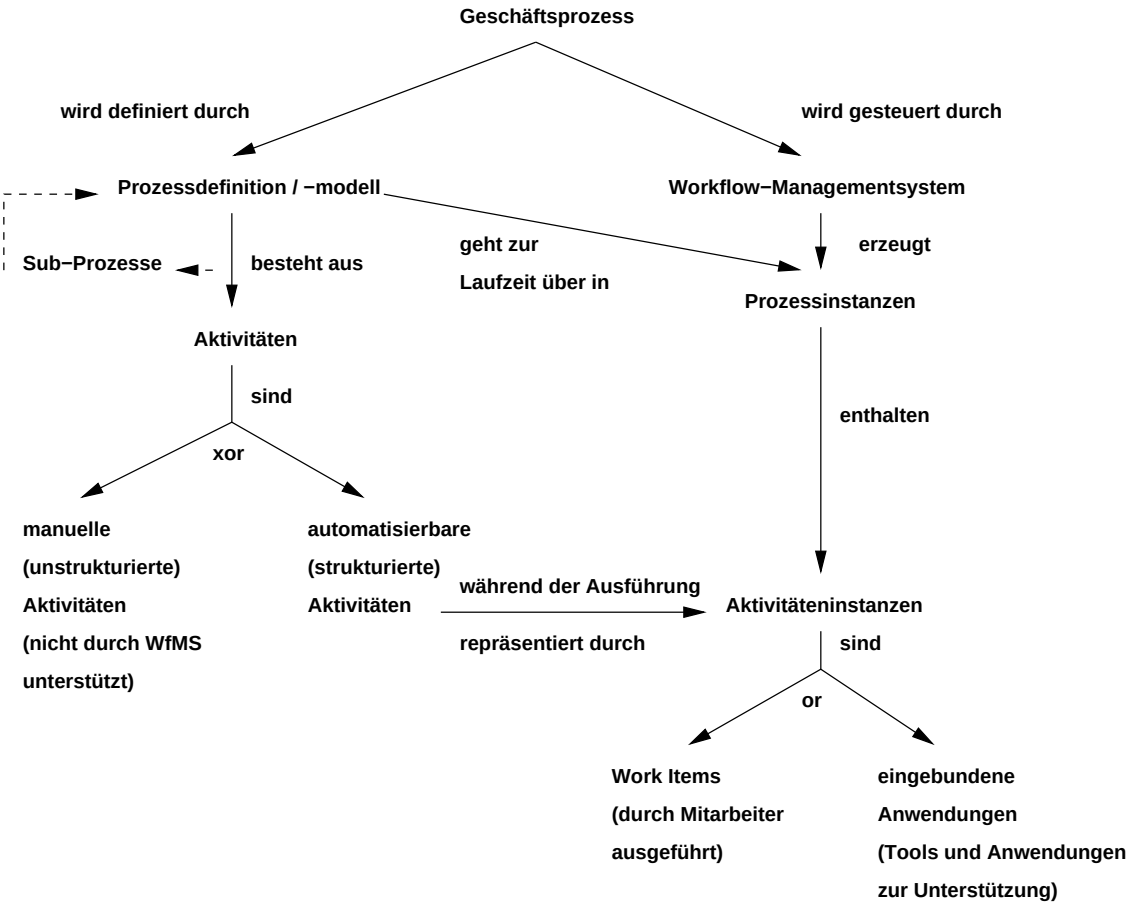


Abbildung 7.3: Geschäftsprozesse und WfMS

Anders dargestellt:

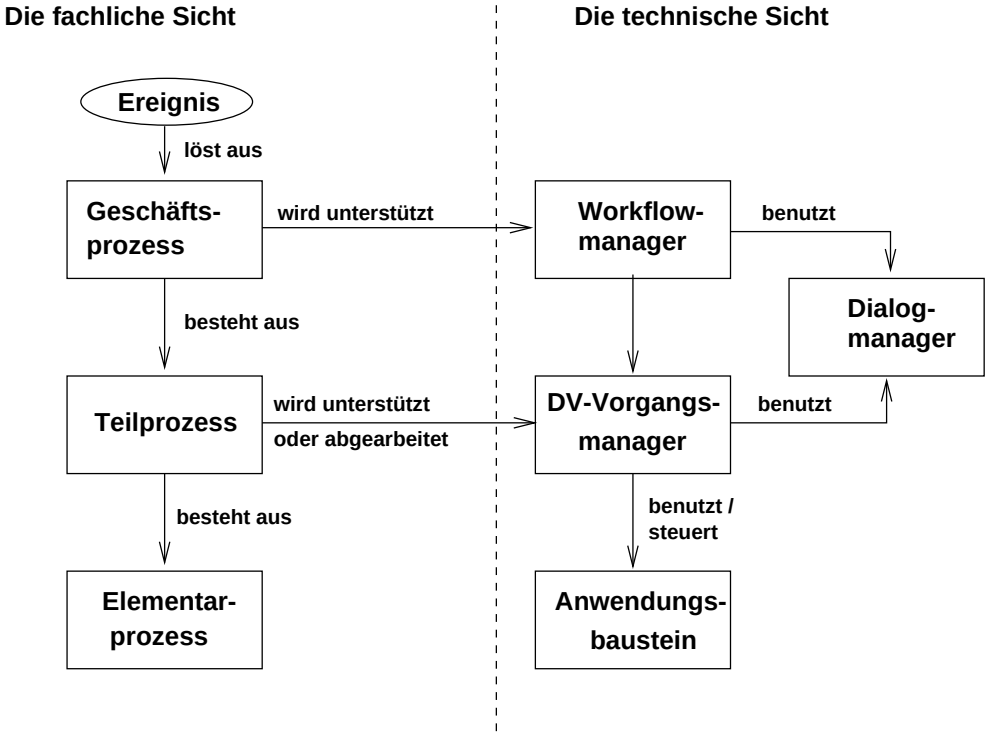


Abbildung 7.4: Geschäftsprozess und WfMS – andere Darstellung

Ablauf:

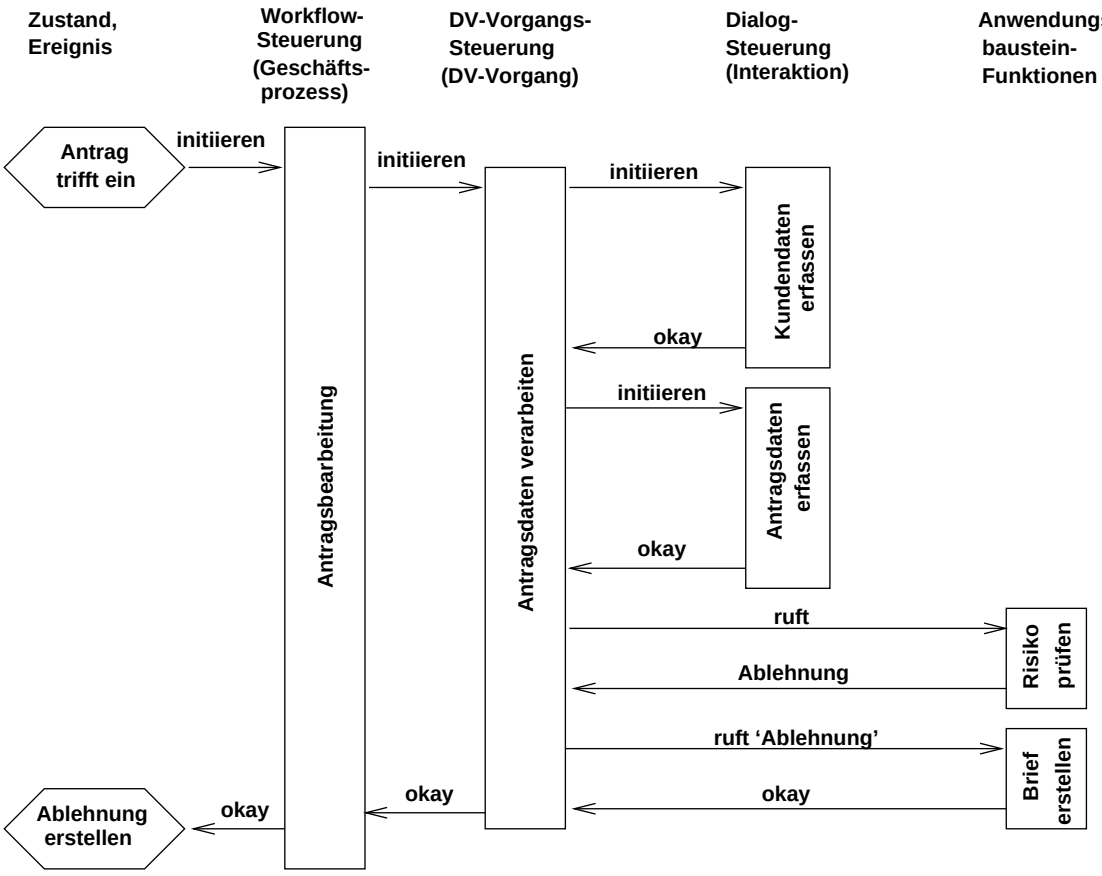


Abbildung 7.5: Geschäftsprozess und WfMS – Ablauf

7.3.4 Die VAA-Architektur

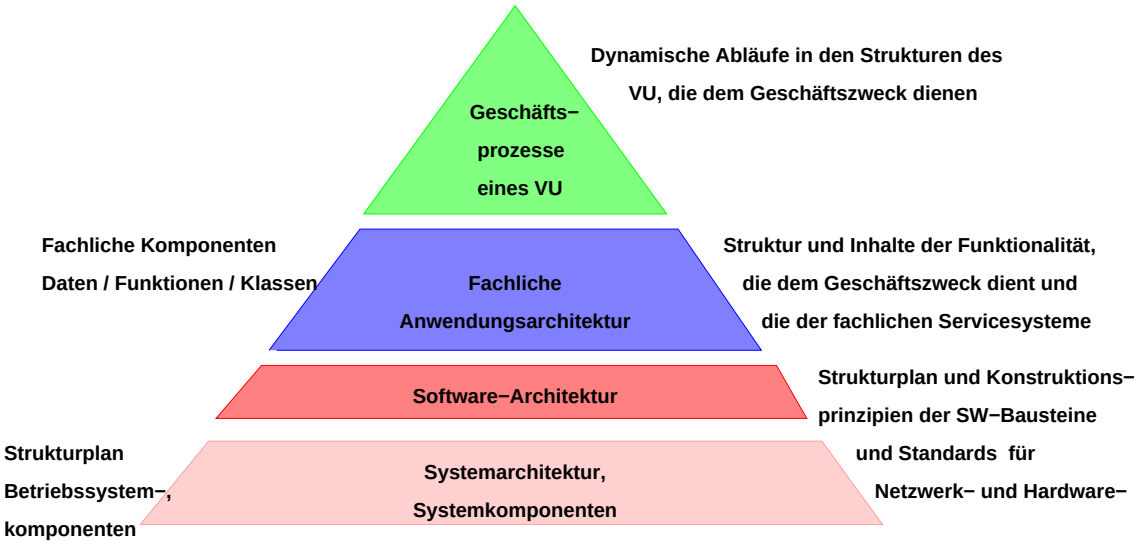


Abbildung 7.6: Quelle: VAA

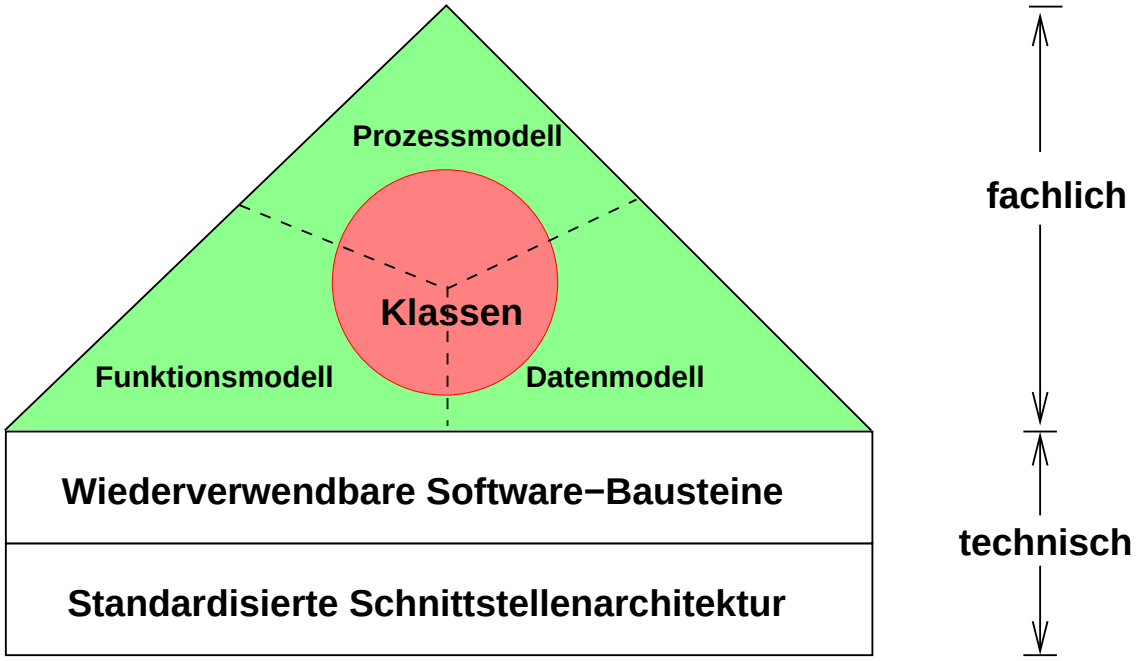


Abbildung 7.7: Quelle: VAA

Die fachlichen Modelle:

- Datenmodell: Entitäten („Wesenseinheiten“, Objekte der Realität), ihre Attribute und die Beziehungen zwischen den Entitäten
- Funktionenmodell: Art der Funktionen und deren betriebswirtschaftliche Rollen, ihre interne statische Struktur und interner Ablauf
- Prozessmodell: geregelte Folge von Teilprozessen zur Erledigung der Aufgaben eines VU

Das technische Rahmenwerk:

- beschreibt die Anwendungsplattform mit generellen, parametrisierbaren und dadurch wiederverwendbaren Bausteinen
- beschreibt die standardisierten Schnittstellen zwischen Softwareteilen, die den Kontrollfluss, die Parameter- und Datenübergabe, die Modulsynchronisation, usw. festlegen

VAA – Entwicklung und Status

- 1993: 3 Arbeitskreise des Ausschusses Betriebswirtschaft des GDV
- 1995: Entscheidung, einzelne Komponenten auszuarbeiten
- 1996/1997: Veröffentlichung der ersten prozeduralen Version
- 1999: VAA Edition 1999
  - prozedural (pVAA), Version 2.1
  - objektorientiert (oVAA) Version 1.0
- 2000: VAA Edition 2000 (Abschlussversion)
  - prozedural (pVAA), Version 2.1
  - objektorientiert (oVAA) Version 2.0

NB: Ergebnisse stehen vollständig im Internet zur Verfügung!

7.3.5 VAA – Dokumentenstruktur

- Management Summary
- Anforderungen und Prinzipien
- Glossar
- VAA prozedural (pVAA)
  - Prozeduraler Rahmen
    - Technische Beschreibung
      - Datenmanager
      - Datenmanager Historienführung und Anhang
      - Dialogmanager
      - Parametersystem
      - Workflow- / Vorgangsmanager
    - Fachliche Beschreibung
      - Inkasso / Kontokorrent
      - Partner
      - Partner / Anhang
      - Produkt
      - Provision
      - Schaden / Leistung
      - Vertrag
  - VAA objektorientiert (oVAA)
    - Das Objektorientierte Technische Referenzmodell
    - Das Objektorientierte Fachliche Referenzmodell
      - Fachliche Modelle in MID-Innovator-Format
      - Fachliche Modelle in HTML-Format
      - Fachliches Referenzmodell aus Edition 1999
      - Generischer Produktansatz
    - Das VAA-Komponentenmodell

7.3.6 Geschäftsprozesse eines VU

Übersicht:

- Produkt entwickeln
- Produkt vertreiben
- Antrag bearbeiten
- Vertrag bearbeiten/Bestände verwalten
  Vertrag planmäßig entwickeln (Sollstellung, Dynamikerhöhung, ...)
  Vertrag außerplanm. techn. ändern (Summe, Laufzeit, Rück-/Teilrückkauf, ...)
  Vertrag juristisch ändern (Zahlungsverkehr, Bezugsrecht, Wechsel des VN)
  Auskunft bearbeiten (Wertstand, Bezugsrecht, ...)
- Schaden/Leistung bearbeiten
- Be- und Abrechnung durchführen
- Abrechnung mit Dritten durchführen

7.3.7 Produktsystem

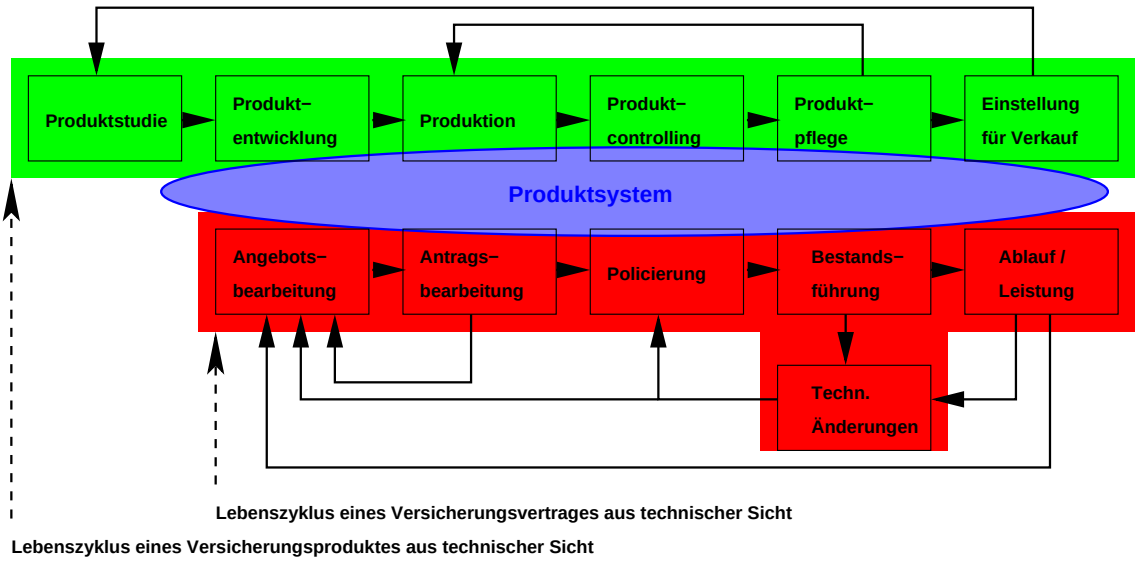


Abbildung 7.8: Versicherungstechn. Lebenszyklen

Produktsystem

besteht aus:

- Produktdefinitionssystem
- Produktdatenbank
- versicherungsmathematischem Rechenkern

und

- verwaltet das gesamte Produktwissen
- stellt es den verschiedenen Anwendungssystemen zur Verfügung

Produktentwicklungssystem (*Actuarial Workbench*

- unterstützt Entwicklung wie auch Analyse von Versicherungsprodukten
- analysiert Produktvorschläge unter verschiedenen Sichtweisen

Einsatzgebiete einer Produktdatenbank:

- Provisionsberechnung
- Inkasso
- Produktbeschreibung
- Kundeninformationen
- Tarifklassifikation
- Optionen / Rechte
- Plausibilitätsregeln
- Risikoprüfung
- Ablaufsteuerung
- Policentextbausteine
- Rechnungslegung
- Produktstrukturdefinition
- Kapitalanlagen
- ...

Vetriebsunterstützungssystem – Teil eines Systemverbunds

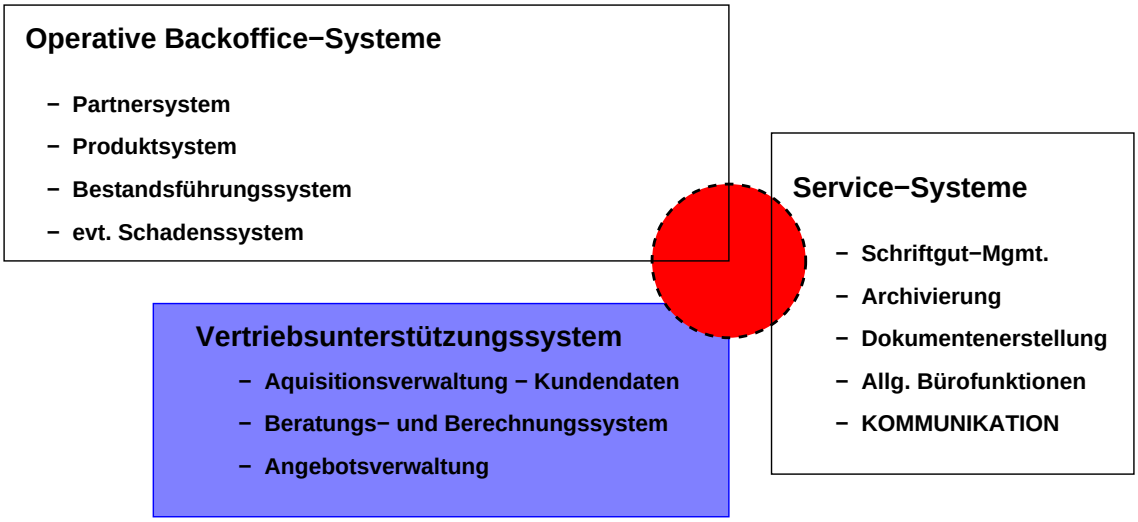


Abbildung 7.9: Vertriebsunterstützung im Systemverbund

7.3.8 Geschäftsprozessmodellierung – Vorgehen

- Bedarfsanalyse erstellen (Vorstudie)
  Geschäftsprozesse identifizieren
  Geschäftsprozesse auswählen (ABC-Analyse – s.u.)
  Geschäftsprozesse grob (abstrakt, nicht ungenau) modellieren
  Maßnahmen ableiten

- Ist-Prozesse modellieren
  Ist-Prozesse erheben und strukturieren
  Ist-Prozesse analysieren
  Schwachstellen herausarbeiten

- Soll-Prozesse modellieren (iterativer Vorgang)
  Soll-Prozesse konstruieren
  Soll-Prozesse simulieren und analysieren
  Soll-Prozesse bewerten

- Soll-Prozesse operativ umsetzen
  Geschäftsvorfälle steuern
  Geschäftsvorfälle protokollieren

- Geschäftsprozesse kontrollieren
  protokollierte Ist-Prozesse analysieren (z. B. hinsichtlich Liege-, Transport- und Bearbeitungszeiten)
  ausgewählte Ist-Prozesse optimieren und modellieren (optimierte Soll-Prozesse)

ABC-Analyse – Beispiel

	Prozessdaten			
Geschäfts-prozess	Anzahl /Periode	Prozess-zeit	Gesamt-zeit	Prozent. zeitanteil
GP 1	140	100	14.000	1.7%
GP 2	180	150	27.000	3.4%
GP 3	260	140	36.400	4.6%
GP 4	12.000	8	96.000	11.9%
GP 5	96.000	5	480.000	59.8%
GP 6	60.000	2	120.000	14.9%
GP 7	30.000	1	30.000	3.7%
			803.400	100%

Prozessrangfolge		
Geschäfts-prozess	Zeit-anteil	Kumulierter Zeitanteil
GP 5	59.8%	59.8%
GP 6	14.9%	74.7%
GP 4	11.9%	86.6%
GP 3	4.6%	91.2%
GP 7	3.7%	94.9%
GP 2	3.5%	98.3%
GP 1	1.7%	100.0%

Der Workflow-Management-Zyklus nach [Heilmann1996]:

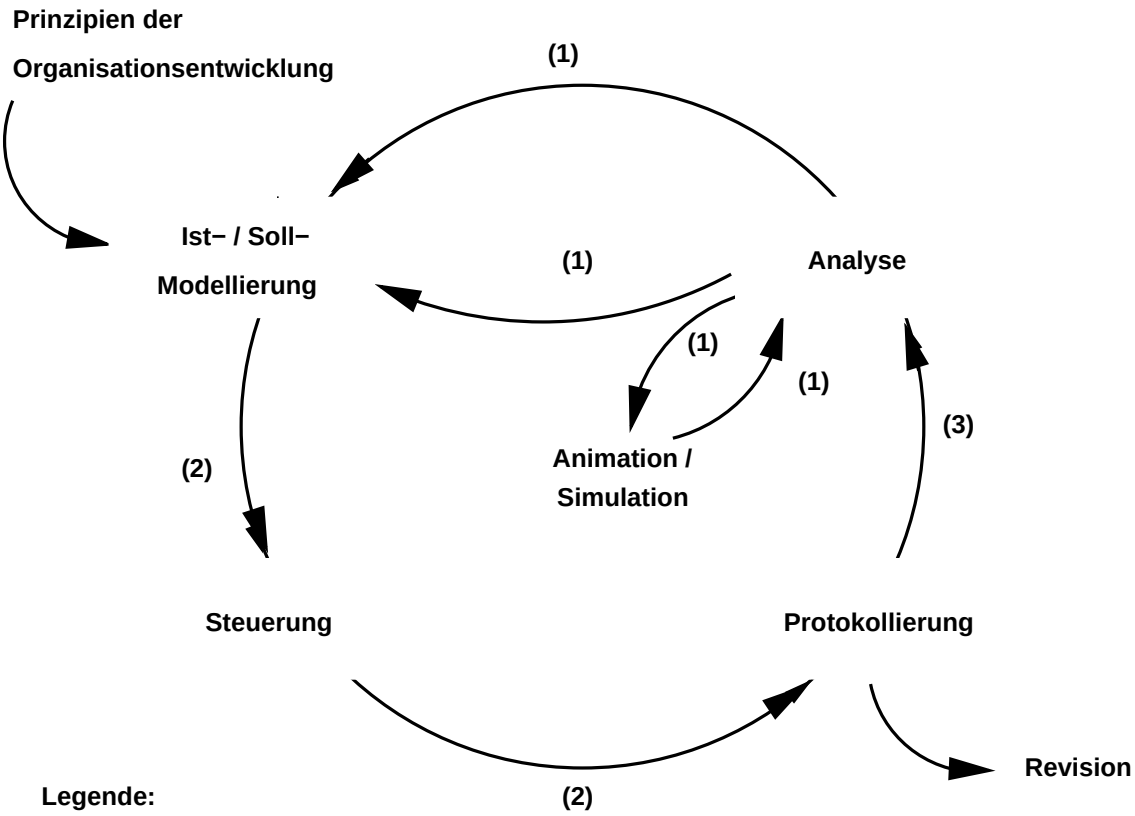


Abbildung 7.10: Workflow-Management-Zyklus

7.3.9 Funktionenmodell

Betriebswirtschaftliche Funktionen eines VU		
Unternehmen führen/leiten - F1 -	**Produktionsfaktoren beschaffen** - F2 -	**Leistungen erstellen (Produktion)** - F3 -
• Ziele setzen • Maßnahmen planen • Maßnahmen entscheiden • Durchführung steuern • Ergebnisse kontrollieren	• Personal/Arbeitsleistungen beschaffen • Dienst- und Werklstng.en beschaffen • Betriebs-/Hilfsmittel einkaufen • Eigen-/Fremdkapital beschaffen • Rückversicherungsschutz einkaufen • Informationen beschaffen	• Organisation/Verfahren regeln • Schäden/Leistungen bearbeiten • Rückversicherung/ fremde Anteile berechnen • Provision berechnen • Verträge bearbeiten – V. annehmen – V. pflegen – V. beenden
Produkte absetzen - F4 -	**Finanzierung sichern** - F5 -	**Unternehmen verwalten** - F6 -
• Markt/Bedarf erforschen • Produkt entwickeln, gestalten, testen, pflegen • Produktabsatz vorbereiten • Produktabsatz organisieren • Produkt verkaufen	• Liquidität/Solvabilität sichern • Forderungen einziehen • Verbindlichkeiten auszahlen • Geld/Mittel anlegen	• Personal verwalten • Hilfsdienste/Nebenbetriebe verwalten • Unternehmensergebnis dokumentieren

Hierarchische Gliederung

Die Funktionen werden sukzessive in Teil-Funktionen zerlegt!

Beispiel für ein Funktionsdiagramm:

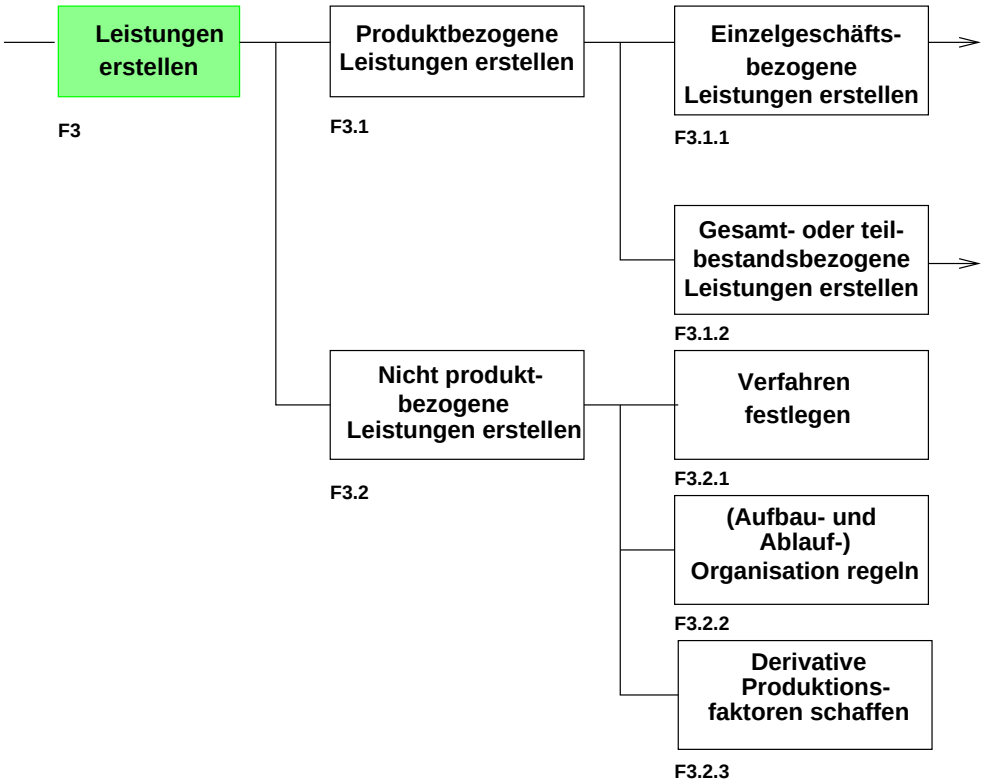


Abbildung 7.11: Funktionsdiagramm

Beschreibung:

F3.1 Produktbezogene Leistungen erstellen

Definitionen und Erläuterungen

Produktbezogene [...] Leistungen beziehen sich auf den gesamten Versicherungsbestand oder auf abgegrenzte Teile davon (z. B. auf einzelne Versicherungszweige, Vertragstypen, Risikotypen, Kundengruppen, Regionen) oder auf einzelne Versicherungsgeschäfte. (siehe Farny, a.a.O., S.503) Die Erstellung dieser produktbezogenen Leistungen ist Gegenstand der in diesem Abschnitt zusammengefaßten Funktionen.

F.3.1.1 Einzelgeschäftsbezogene Leistungen erstellen

Definitionen und Erläuterungen

Die Erstellung einzelgeschäftsbezogener Leistungen fasst die betrieblichen Funktionen zusammen, die sich auf den Risiko-, Dienstleistungs- und ggf. Spar-/Entsparteil der einzelnen Versicherungsgeschäfts beziehen (vgl. Farny, a.a.O., S. 504). Dies schließt die Bearbeitung mehrerer Verträge in einem Geschäftsvorfall und mehrerer Risiken in einem Vertrag ein. Hier finden sich die Aufgaben, aus denen jeweils die zu den folgenden Kategorien zusammengefassten Geschäftsprozesse

- Erst- (Antrags-),
- Folge- (Bestands-),
- Schaden- bzw. Leistungsfall-,
- Schluß- sowie
- Rückversicherungs- und Beteiligungsbearbeitung

gebildet werden. Die Funktionen ihrerseits lassen sich unter „Vertragsgrundlagen bestimmen“, „Berechnungen durchführen“ und „Ergebnisse dokumentieren“ subsumieren.

7.3.10 Das Prozess-Modell

- Ausgangspunkt für Geschäftsprozessmodellierung: Funktionenmodell
- Geschäftsprozessmodellierung zum Zweck der Standardisierung:
  hohes Abstraktionsniveau (Individualität der VU, Wettbewerbssituation der VU)

Prozesse und Funktionen:

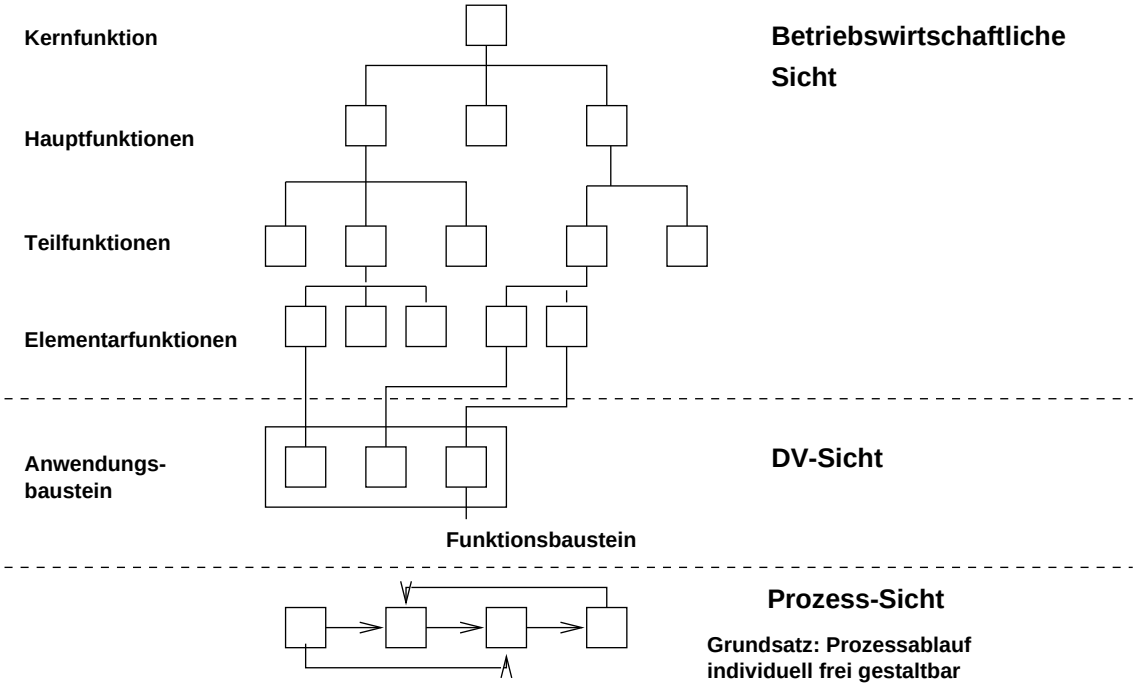


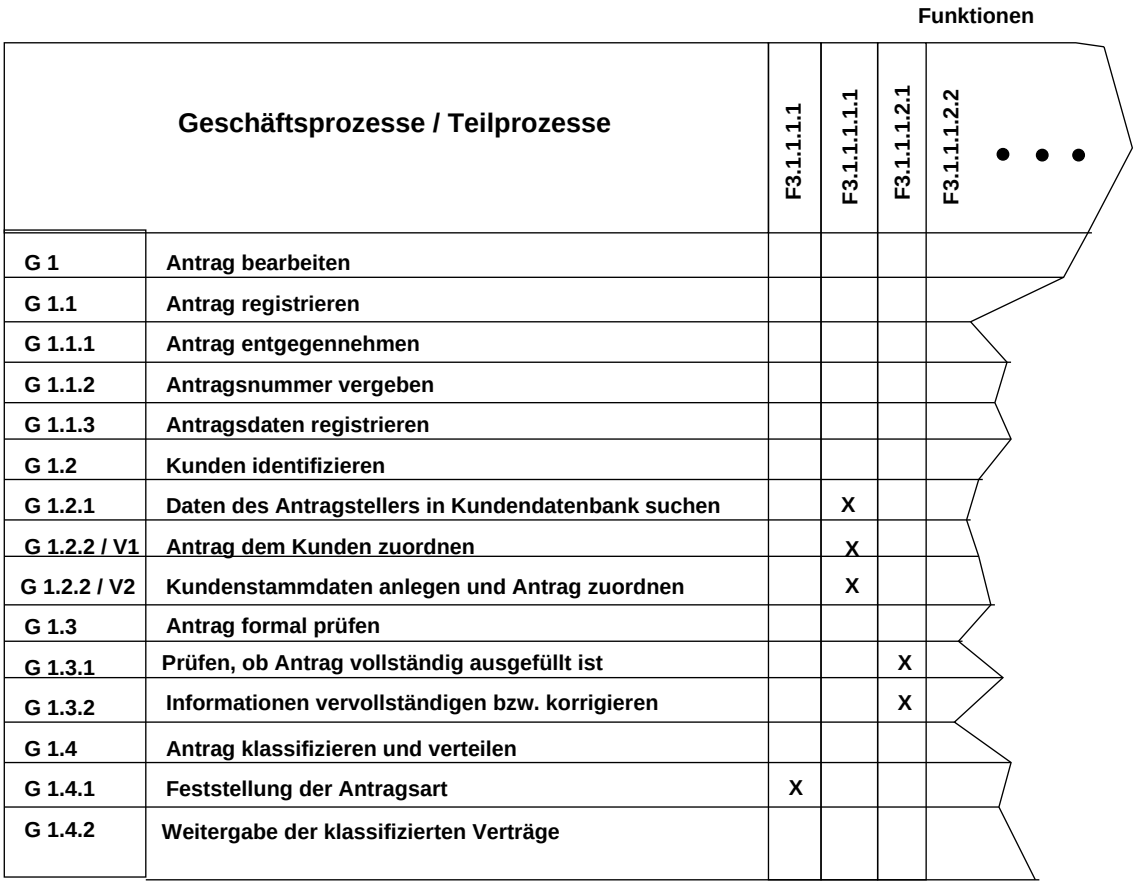
Abbildung 7.12: Prozess-Modell

Beispiel:

G1	**ANTRAG BEARBEITEN**
Kategorie:	Geschäftsprozess
Auslöser:	Antrag auf Versicherungsschutz (einseitige Willenserklärung)
Beschreibung:	Der Prozess beschreibt den Ablauf vom Eingang des Antrags auf Versicherungsschutz bis zum geprüften, vervollständigten und entschiedenen Antrag.
Teilprozesse:	
G1.1	Antrag registrieren
G1.2	Kunden identifizieren
G1.3	Antrag formal prüfen sowie Antrag vervollständigen bzw. korrigieren (formale Aspekte)
G1.4	Antrag inhaltlich prüfen sowie Antrag vervollständigen bzw. korrigieren (inhaltliche Aspekte)
Ergebnis:	geprüfter und vervollständigter bzw. korrigierter und entschiedener Antrag.
Folgeprozesse:	Es sind je nach Ergebnis der Antragsbearbeitung drei Folgeprozesse denkbar: • ein beidseitig unterschriebener Antrag (Folgeprozess: Policierung) • ein vom VU nicht unterschriebener Antrag (Folgeprozess: Ablehnung mit Gegenangebot) • ein vom VU nicht unterschriebener Antrag (Folgeprozess: Ablehnung ohne Gegenangebot)

G 1.1	**ANTRAG REGISTRIEREN**	
Kategorie:	Teilprozess	
Auslöser:	Antrag auf Versicherungsschutz (einseitige Willenserklärung)	
Beschreibung	Der Teilprozess „Antrag registrieren“ beinhaltet die Teilprozesse von der Entgegennahme des Antrags bis zu seiner Registrierung	
Teilprozesse:		Funktion:
G1.1.1	Antrag entgegennehmen (z. B. bei schriftlichem Antrag: Post annehmen, Eingangspoststempel setzen,...)	F4.4.2.4
G1.1.2	Antragsnummer vergeben	F6.1.3
G1.1.3	Antragsdaten registrieren (Grunddaten des Antrags mit Antragsnummer registrieren (elektronisch oder manuell))	F6.x
Ergebnis:	registrierter Antrag	
Folgeprozess:	Kunden identifizieren	

Zusammenführung der Teilprozesse zum Funktionenmodell:



Geschäftsprozesse / Teilprozesse		F3.1.1.1.1	F3.1.1.1.1.1	F3.1.1.1.2.1	F3.1.1.1.2.2 • • •
					Funktionen
G 1	Antrag bearbeiten				
G 1.1	Antrag registrieren				
G 1.1.1	Antrag entgegennehmen				
G 1.1.2	Antragsnummer vergeben				
G 1.1.3	Antragsdaten registrieren				
G 1.2	Kunden identifizieren				
G 1.2.1	Daten des Antragstellers in Kundendatenbank suchen		X		
G 1.2.2 / V1	Antrag dem Kunden zuordnen		X		
G 1.2.2 / V2	Kundenstammdaten anlegen und Antrag zuordnen		X		
G 1.3	Antrag formal prüfen				
G 1.3.1	Prüfen, ob Antrag vollständig ausgefüllt ist			X	
G 1.3.2	Informationen vervollständigen bzw. korrigieren			X	
G 1.4	Antrag klassifizieren und verteilen				
G 1.4.1	Feststellung der Antragsart	X			
G 1.4.2	Weitergabe der klassifizierten Verträge				

Abbildung 7.13: Prozesse / Funktionen

Dies wird im Zusammenhang mit allgemeinen Modellierungssprachen detaillierter behandelt.

7.3.11 Das Datenmodell

Darauf wird in einem späteren Kapitel allgemein eingegangen!

7.3.12 Das Komponentenmodell

Darauf wird im Kontext mit UML und OO-Software-Entwicklung eingegangen!

7.4 Insurance Application Architecture (IAA) von IBM

Umfasst drei Dimensionen:

- *Layer*
  - *Business Model* (fachliche Schicht)
  - *System Model* (technische Schicht)
- *Detail*
  - Allgemeine Elemente
  - Unternehmensspezifische Elemente
- *Scope*
  - Datenmodell, Informationen
  - Funktionsmodell, Tätigkeiten
  - Prozessmodell

Entwicklung und Status

- Start: 1990
- Übergreifende Initiative mit internationaler Beteilung von VU unter Federführung von IBM
- 1993: Edition 3 – prozedural mit Daten-, Funktions- und Prozessmodell
- 1999: Edition 4.1 – mit unternehmensweitem *Business Object Model* (BOM)
- aktuell: Weiterentwicklung in Richtung “komponenten-basiert” und XML
- Weltweite Standardarchitektur für VU-spezifische IBM Software und Dienste
- Umfassendes und spartenneutrales Modell, hoher Abstraktionsgrad

Kapitel 8

Weitere Methoden und Techniken

8.1 Daten-Modellierung – klassisch

8.1.1 Grundlegendes

Objekte der realen Welt werden bezüglich einer gegebenen Zielsetzung durch ihre relevanten, statischen Eigenschaften beschrieben:

- Worüber werden welche Informationen benötigt?
- Was sind die Objekte der Betrachtung (Wesenseinheiten oder Entitäten)?
- Welche Eigenschaften von diesen sind von Bedeutung?
- Wie stehen diese Objekte untereinander in Beziehung oder welche Relationen gibt es zwischen diesen Entitäten?

Objekte der „realen“ Welt (z.B. eines Unternehmens)

- **Objekte (Objektklassen, Objekttypen)/'Entities' (Entity-Typen)**
  → Informationen über Objekte der „realen Welt“
  Worüber werden Informationen (Daten) benötigt?
  z.B. Kunden, Artikel, Mitarbeiter, Projekte, Aufträge, ...

  Konkrete Objekte (Kunden 'Maier', 'Huber', ...) werden zu einer Klasse zusammengefaßt, wenn sie „gleichartig“ sind.

- **Attribute/Eigenschaften/Merkmale**
  Welche Informationen werden über diese Objekte benötigt?

  → Attribute dieser Objekte
  Kunden (Entity-Typ 'Kunde'): Name, Anschrift, Tel., Bonität, ...

  elementare/strukturierte Attribute

  - elementare Attribute besitzen keine relevante (Semantik) Unterstruktur
  - strukturierte Attribute sind zusammengesetzt aus elementaren/ strukturierten Attributen (Bsp.: Anschrift = (Straße, PLZ, Ort))

- **Schlüssel:**
  Wie werden die einzelnen Objekte eine Objektklasse identifiziert (unterschieden)?

  Eine minimale Attributkombination, die das einzelne Objekt aus der Objektklasse eindeutig identifiziert, ist ein Schlüssel dieser Klasse.
  Minimal heißt: Läßt man ein Attribut aus der Kombination weg, so ist der Rest nicht mehr identifizierend.

- **Wertebereiche, Integrität**
  Welche Ausprägungen der elementaren Attribute sind (prinzipiell) möglich?
  → Wertebereiche (PLZ: 5-stellig, Ziffern); semantisch sind u.U. nicht alle sinnvoll.

  Welche Ausprägungen der Kombinationen elementarer Attribute zu einem nichtelementaren Attribut oder Objekt sind möglich/zulässig?

- **Beziehungen (Beziehungsklassen) - (Objekt)-Relationen** (*Relationships*)

  → Welche Zusammenhänge/Abhängigkeiten bestehen zwischen den Objekten?
  Beispiel
  Artikel(Artikel#, Bezeichnung, Gewicht, Preis)
  Kunde(Kunden#, Anschrift, Bonität)

  Dadurch, dass ein Kunde einen Artikel bestellt, entsteht eine Beziehung zwischen Kunde und Artikel: → Auftrag.
  Diese Beziehung hat selbst wieder Attribute: Menge, Datum.

8.1.2 Entity-Relationship-(ER)-Diagramme

Entity(-Typen) werden dargestellt als *Rechtecke*, Beziehungen als *Rauten* mit *Kanten* zu den an der Beziehung beteiligten Rechtecke:

- *Attribute*: das Objekt identifizierend/beschreibend, direkt und unmittelbar
- alle Objekte einer Objektklasse werden „gleichartig" beschrieben

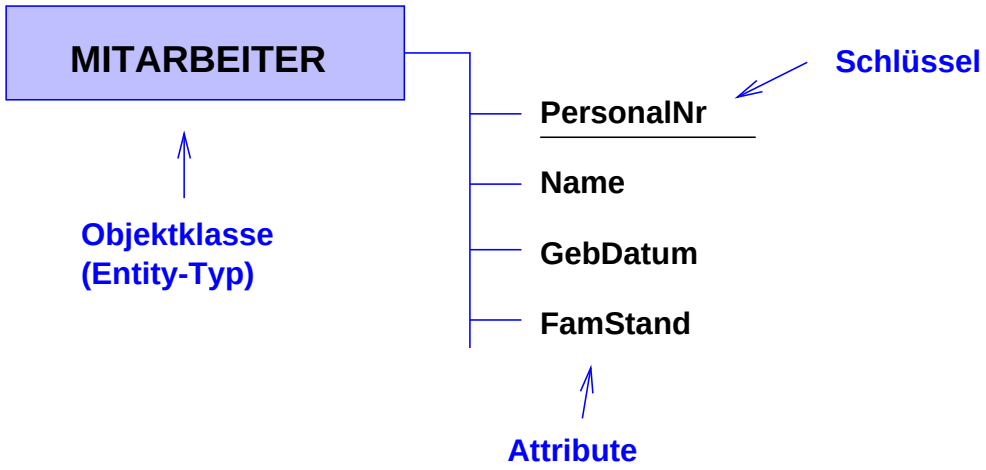


Abbildung 8.1: Darstellung von Objekttypen, Attributen und Schlüsseln

Beziehungen zwischen den Objektklassen

meist als Verb ('arbeitet mit', 'nimmt teil', 'wird geleitet von') — suggeriert oft eine „Lese-Richtung", manchmal auch als Substantiv (eine mündliche 'Prüfung' stellt eine Beziehung zwischen dem Kandidaten, dem Professor und dem Beisitzer dar!).

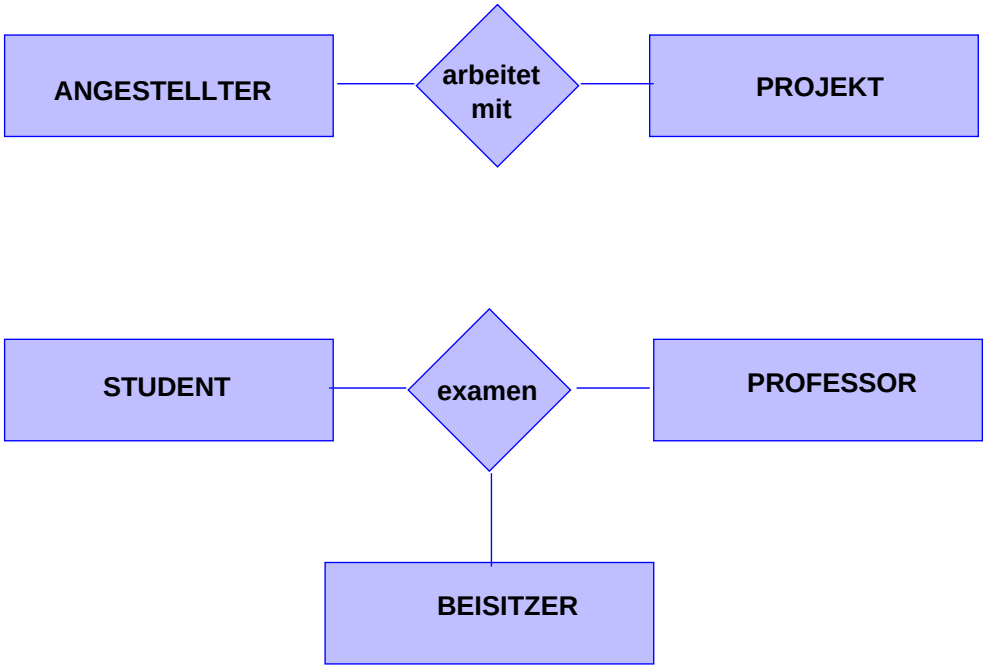


Abbildung 8.2: Darstellung von Beziehungen

Beziehung oder Objekt???

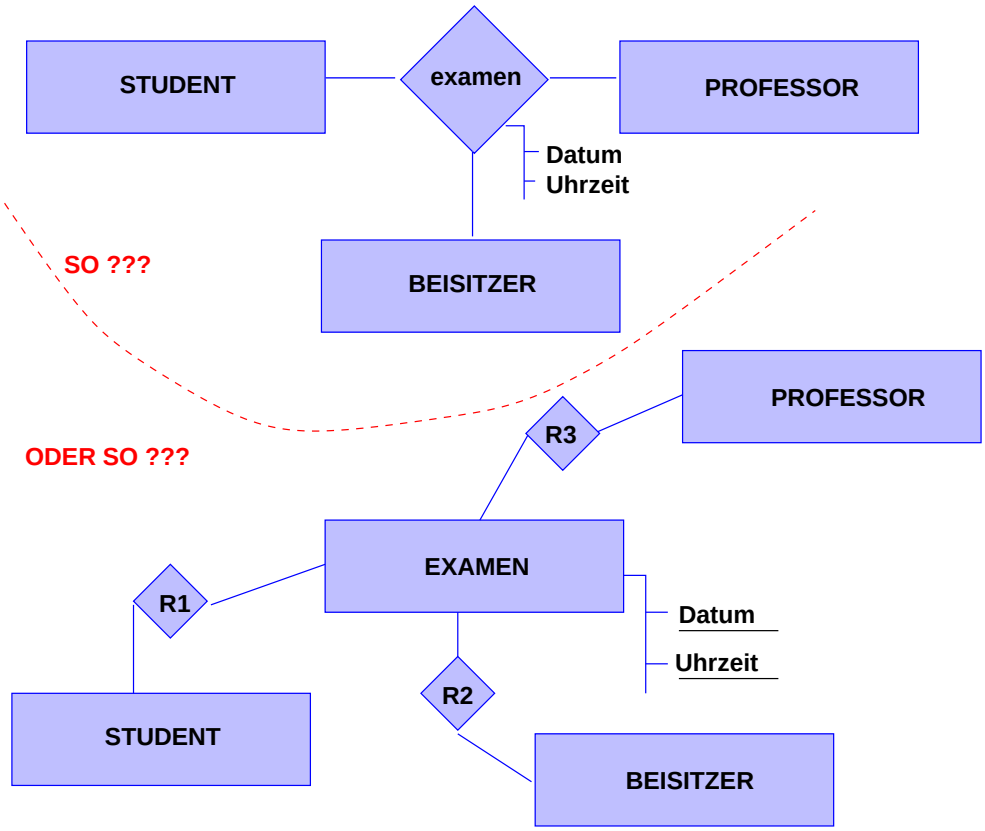


Abbildung 8.3: Beziehung vs. Objekt

8.1.3 Charakterisierung von Beziehungen

Generalisierung, Spezialisierung (“is-a-Beziehung”)

Beispiel
Lohn- wie Gehaltsempfänger sind Angestellte, sie unterscheiden sich nur durch bestimmte Eigenschaften, haben aber viele Attribute (Personalnummer, Name, Eintrittsdatum, Adresse, ...) gemeinsam (disjunkte is-a-Beziehung).

Grafische Darstellung:

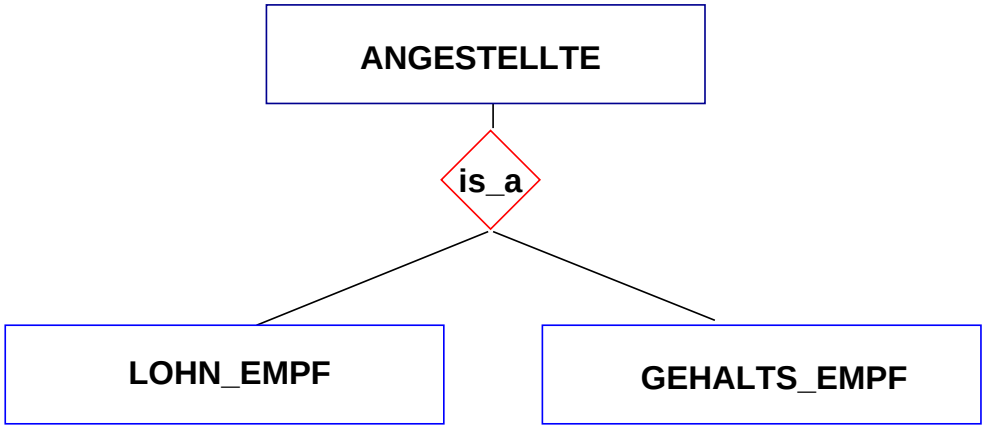


Abbildung 8.4: is-a-Beziehung

Komplexitätsgrade, (min/max-, 1-c-m-mc-Notation) → Integritätsbedingung

- Ein konkreter Angestellter arbeitet maximal in 3 Projekten mit, in einem Projekt arbeiten mindestens 1 und maximal 20 Angestellte mit
- Ein konkreter Angestellter kommt höchstens dreimal in der Beziehung 'arb_in' vor, ein konkretes Projekt kommt dort mindestens einmal, höchstens jedoch zwanzigmal vor

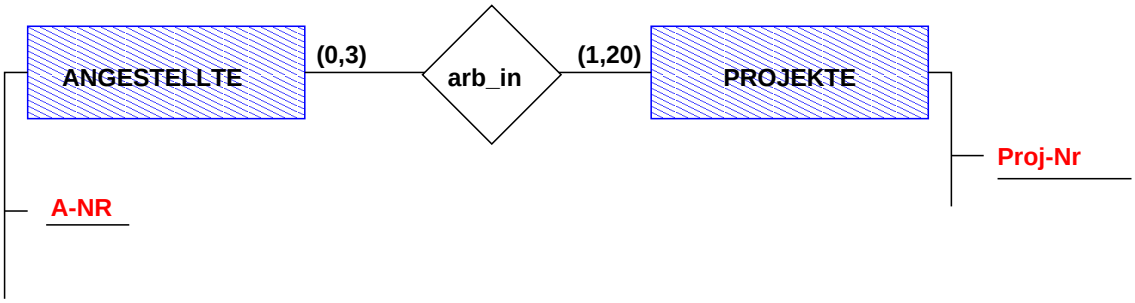


Abbildung 8.5: min-max-Komplexitätsgrade

Sei R eine Beziehung, E eine beteiligte Objektklasse. Dann gilt:

kgrad(E, R)	:	ein konkretes Objekt der Klasse E kommt in
1	:	. . . genau 1 (min=1, max=1)
c	:	. . . höchstens 1 (min=0 und max=1)
m	:	. . . mindestens 1 (min=1 und max bel. groß)
mc	:	. . . beliebig (min=0 und max bel. groß)

konkreten Beziehung(en) vom Typ R vor

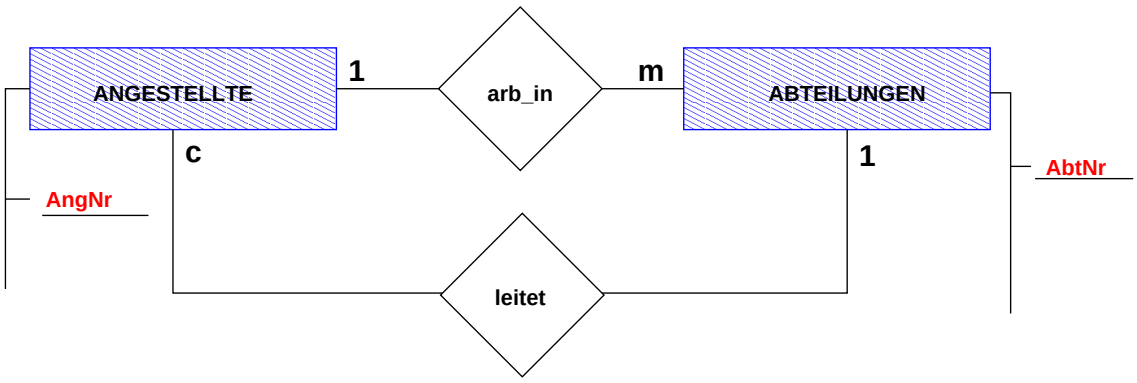


Abbildung 8.6: 1-c-m-mc-Notation

Eine Beziehung (Beziehungstyp) R zwischen z.B. den Entity-Typen A, B, C kann aufgefaßt werden als

$$\mathbf{R} \subseteq \mathbf{A} \times \mathbf{B} \times \mathbf{C}$$

Damit lassen sich die o.a. Komplexitätsgrade auch wie folgt definieren:

$$\text{kgrad(A,R)=(min, max)}$$
$$\Leftrightarrow$$
$$\min \leq \left|\{(a, x, y) \in R : \text{ mit } x \in B,\ y \in C\}\right| \leq \max, \forall\, a \in A$$
(Zu jedem Zeitpunkt, versteht sich!)

Die Zuordnung der Komplexitätsgrade im ER-Diagramm erfolgt hier durch Angabe *beim* Entity-Typ – bei zweistelligen Beziehungen, bei denen man die Relation als „gerichtet“ ansehen kann, wird dies oft „vertauscht“ angegeben:

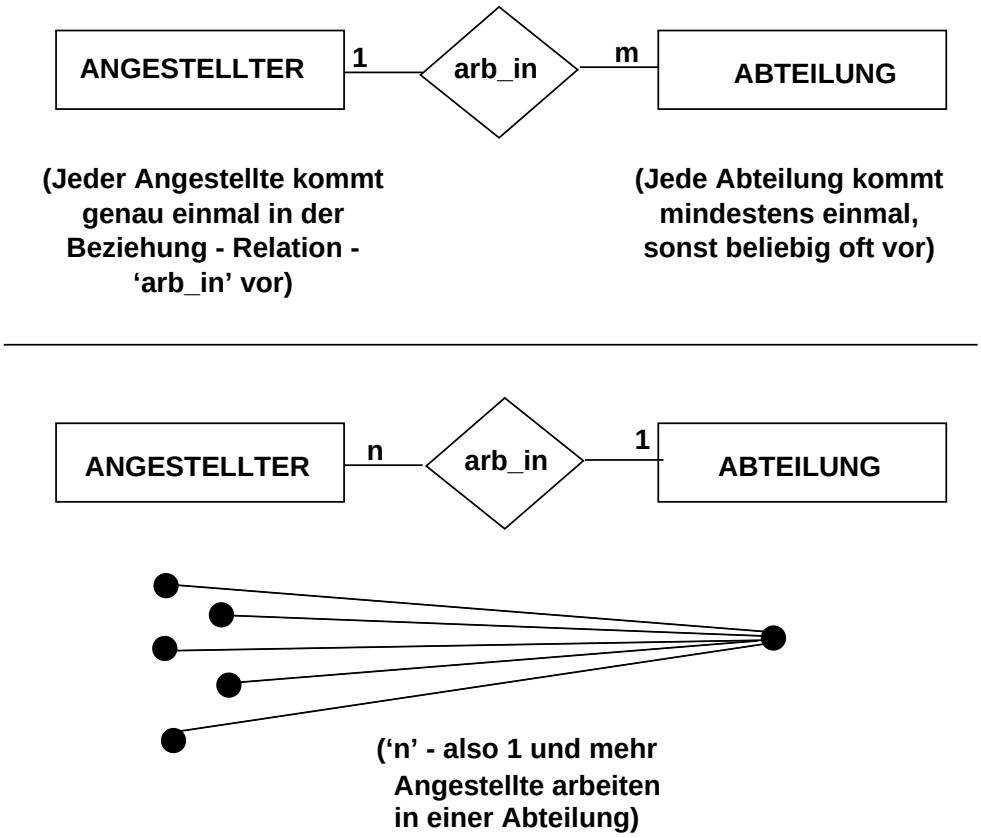


Abbildung 8.7: „Verdrehte“ Notation bei 2-stelligen Beziehungen

8.1.4 IE-Notation

IE – Information Engineering von James Martin, *Information Engineering: Introduction*, Prentice Hall 1989

Beispiel 1:

Dozenten, Diplomanden: Dozenten betreuen keinen oder beliebig viele Diplomanden, ein Diplomand hat genau einen Dozenten als Betreuer:

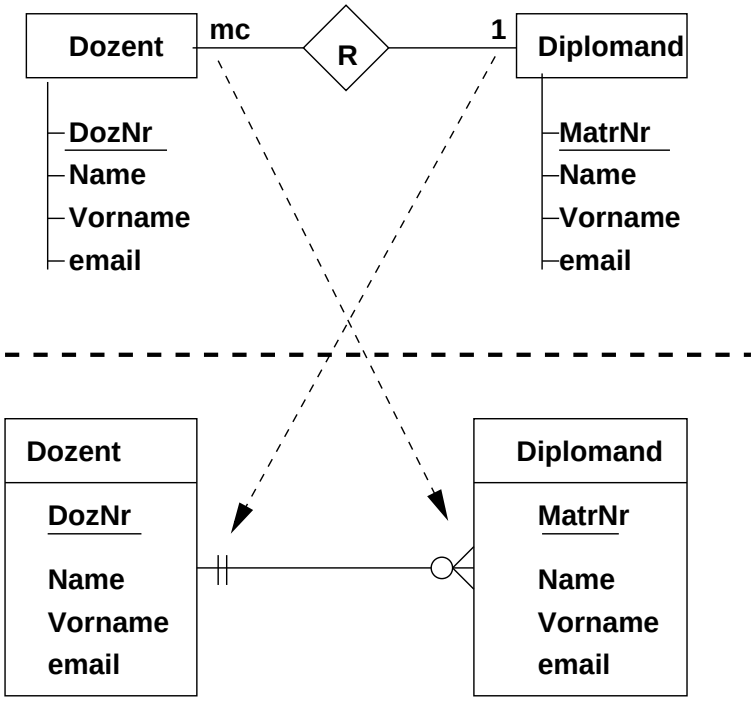


Abbildung 8.8: IE-Notation – Beispiel 1

Im oberen Diagramm ist die oben beschriebene Notation dargestellt, im unteren Teil die IE-Notation. Zu beachten ist, dass die Komplexitätsgrade „vertauscht sind“, d. h. vom Entity aus gesehen ist die Angabe immer beim „Gegenüber“.

Die restlichen Komplexitätsgrade:

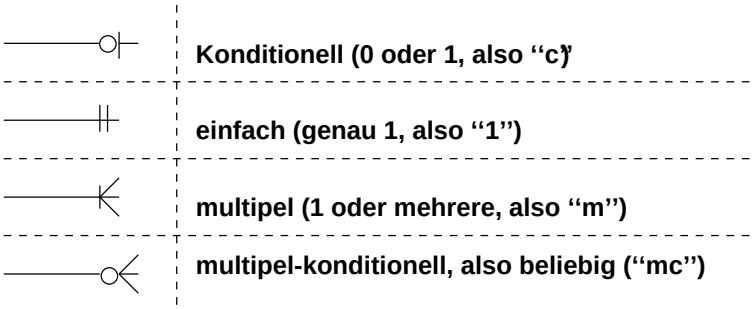


Abbildung 8.9: Komlexitätsgrade in der IE-Notation

Darstellung einer rekursiven Beziehung

Beispiel 2:

Vertriebseinheiten (z.B. Makler, Bezirksdirektionen, Filialdirektionen) stehen miteinander in verschiedenen Beziehungen, so dass beispielsweise Makler zu verschiedenen Bezirksdirektionen und einer Bezirksdirektion mehrere Makler zugeordnet werden können.

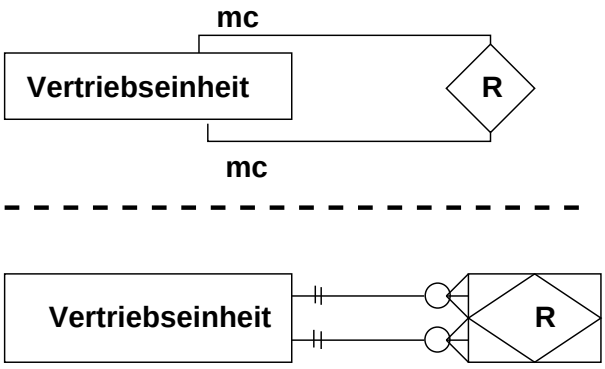


Abbildung 8.10: IE-Notation einer rekursiven Beziehung

Beispiel 3: Personen können verheiratet sein

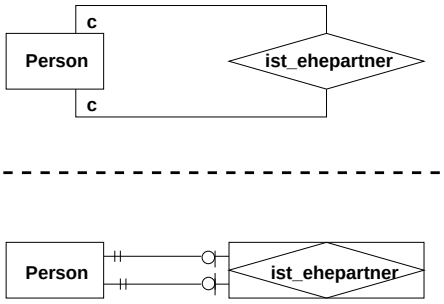


Abbildung 8.11: IE-Notation einer rekursiven Beziehung

Mehrstellige Beziehungen

Beispiel 4: In einem Projekt können bestimmte Bauteile nur von bestimmten Lieferanten benutzt werden; jedes Bauteil wird insgesamt höchstens einmal verwendet; von einem Lieferanten können verschiedene Bauteile geliefert werden, auch keine; in einem Projekt wird mindestens ein Bauteil verwendet.

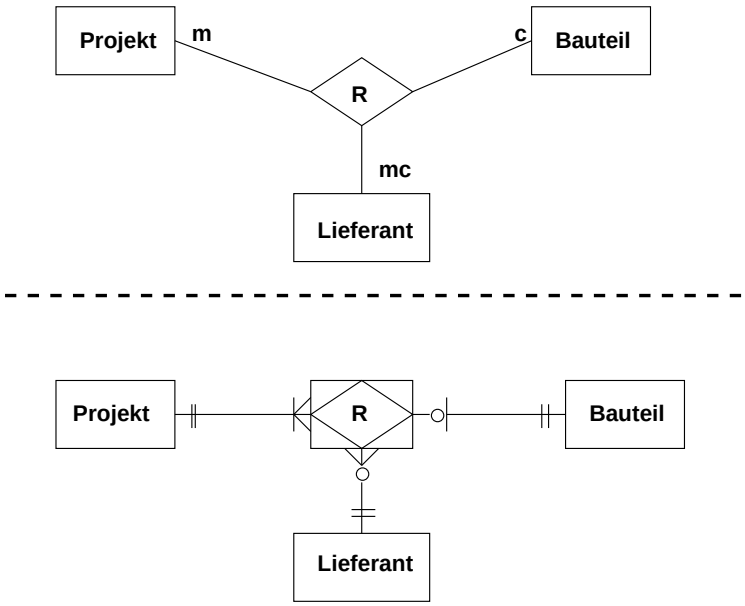


Abbildung 8.12: IE-Notation von mehrstelligen Beziehungen

Beispiel:

Gegeben sei folgende Situationsbeschreibung:

- Ein **Student** kann beliebig viele **Lehrveranstaltung**en, auch keine, besuchen.
- Eine Lehrveranstaltung muss von mindestens einem Studenten besuch werden, ansonsten fällt sie aus.
- Ein **Dozent** kann beliebig viele Lehrveranstaltungen anbieten, auch keine (Forschungssemester!).
- Jede Lehrveranstaltung wird von genau einem Dozenten gehalten.
- Dozenten sind **Fachgebiet**en zugeordnet; jeder Dozent gehört zu genau einem Fachgebiet, in jedem Fachgebiet gibt es mindestens einen Dozenten.
- Dozenten können mehrere **Assistent**en haben; jeder Assistent gehört zu genau einem Dozenten.
- Studenten werden von mindestens einem Dozenten geprüft, nicht jeder Dozent hat Studenten geprüft.
- Es gibt weitere **Mitarbeiter**; diese können zu einem Fachgebiet gehören, wenn ja, so zu genau einem; jedes Fachgebiet hat genau einen Mitarbeiter.
- Dozenten können **Lehrbücher** empfehlen, ein und dasselbe Lehrbuch kann auch von mehreren Dozenten empfohlen werden.

Auf die Angabe von Attributen wurde bewusst verzichtet.

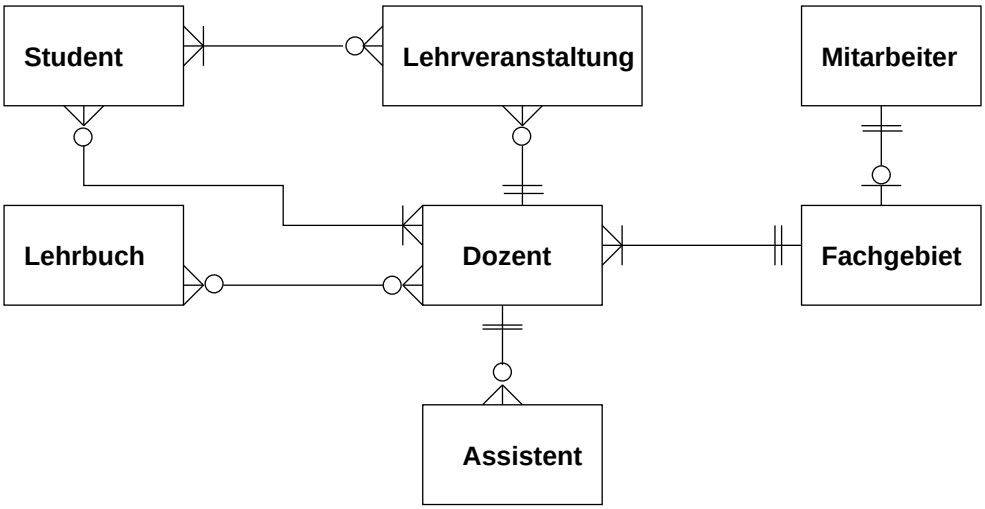
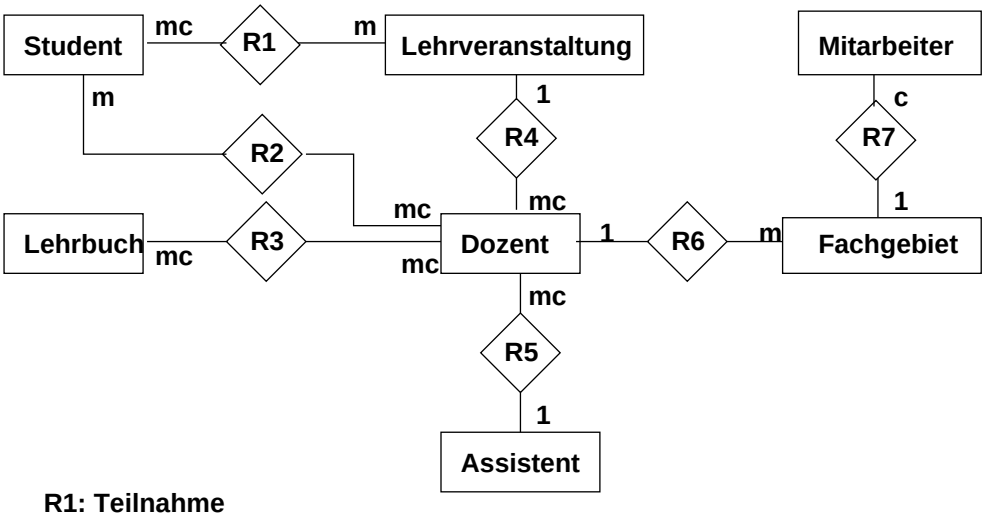


Abbildung 8.13: Beispiel in IE-Notation



R1: Teilnahme
R2: Prüfung
R3: Empfehlung
R4: Durchführung **R6: Fachvertretung**
R5: Zusammenarbeit **R7: Betreuung**

Abbildung 8.14: Beispiel in 1,c,m,mc-Notation

8.1.5 Existenzabhängige (*weak*) Entities

Können nur existieren, wenn andere Entities vorhanden sind

Beispiel:
Labordaten (oder Untersuchungsdaten) gehören zu einem Patienten, existieren also nur, wenn dazu ein Patient existiert

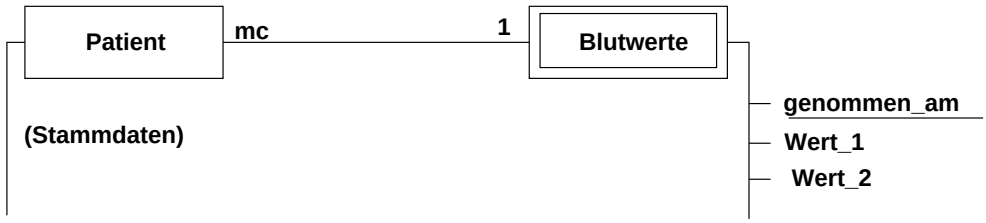


Abbildung 8.15: Existenzabhängige Entities

8.1.6 Integritätsbedingungen

→ Komplexitätsgrade, Wertebereichseinschränkungen, Schlüsseleigenschaft

Allgemein sind Integritätsbedingungen durch ein Tupel (O, B, A, R) mit

O	betroffene Objekte
B	Bedingung (logischer Ausdruck, ASSERTION)
A	Auslöseregel (wann soll B geprüft werden)
R	Reaktionsregel (was soll bei Verletzung von B getan werden

zu beschreiben. Zu unterscheiden sind weiter statische (Zustandsbedingungen) und dynamische (Zustandsübergangsbedingungen).

Operationen: Weiterhin sind die zulässigen Basisoperationen auf den Objekten (ändern, löschen u.dgl.) mit den Objekten zu definieren.

8.1.7 Normalisierung, Normalformen

→ Regeln für die Gestaltung von Objektklassen (Entity-Typen) und Tabellen

1. Normalform Jedes Attribut ist elementar, besitzt also keine relevante Substruktur

Beispiel:
Attribut 'Hobbies' von Objektklasse (Tabelle) 'Mitarbeiter'

In Tabellenform:

ANGESTELLTE	PersNr	...	Hobbies
	12023	irgendwas	Schwimmen, Tanzen
	12123	was anderes	Reisen, Segeln
	14711	egal was	

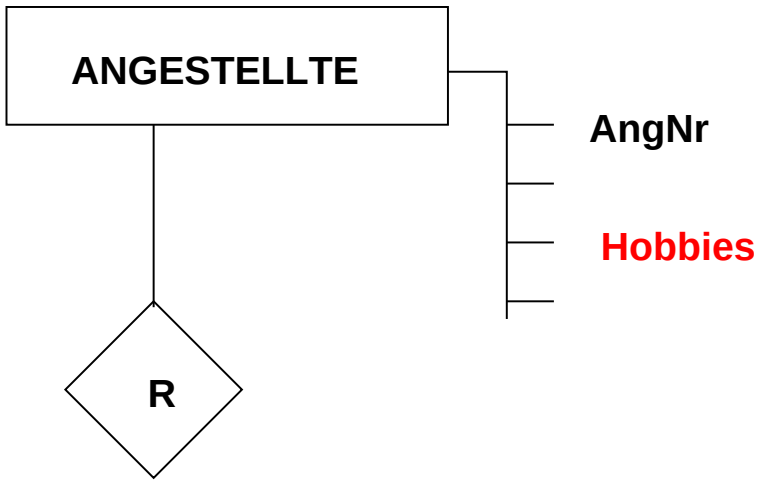


Abbildung 8.16: Erste Normalform – Problem

Lösung:

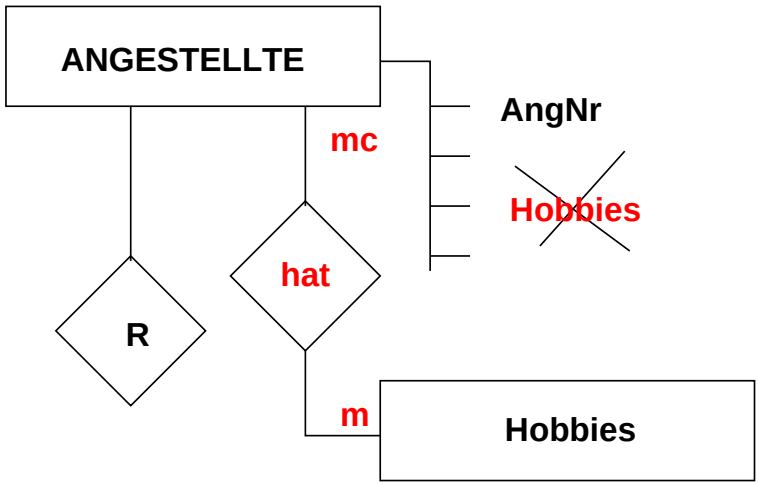


Abbildung 8.17: Erste Normalform – Lösung

In Tabellenform:

ANGESTELLTE	PersNr	...
	12023	irgendwas
	12123	was anderes
	14711	egal was

HOBBIES	Bezeichnung
	Schwimmen
	Tanzen
	Reisen
	Segeln

ANG_hat_HOBBIES	PersNr	Hobby
	12023	Schwimmen
	12023	Tanzen
	12023	Reisen
	12123	Segeln

Funktionalen Abhängigkeit:

Definitionen:

Sei R: rel(U) gegeben, $A, B \subseteq U$ (Attributkombinationen).

Dann heißt B *funktional abhängig von* A, i.Z. $A \rightarrow B$,

$:\Leftrightarrow$ Zu jedem Zeitpunkt t und für je zwei $r_1, r_2 \in R^t$ mit $r_1.A = r_2.A$ ist auch $r_1.B = r_2.B$ (A-Wert bestimmt B-Wert).

B heißt *voll funktional abhängig* von A, i.Z. $A \twoheadrightarrow B$,

$:\Leftrightarrow$ Für alle A' mit $\emptyset \neq A' \subset A$ folgt: $A' \not\rightarrow B$.

$S \subseteq U$ heißt **Schlüssel** genau dann, wenn gilt: $S \twoheadrightarrow U$.

Anmerkung:
Einzelne Attribute können nach dieser Definition durchaus von einer Teilmenge von S abhängig sein (siehe 2. Normalform).

Attribute, die in einem Schlüssel vorkommen, heißen *Schlüsselattribute* (identifizierende Attribute), alle anderen *Nicht-Schlüsselattribute* (beschreibende Attribute).

Anmerkung:
Funktionale Abhängigkeiten kommen wie die Attribute aus der jeweiligen Realität! Sie stellen, falls gegeben, ebenfalls Integritätsbedingungen dar!

2. Normalform

1. Normalform und:

Jedes beschreibende Attribut (Nicht-Schlüsselattribut) bezieht sich auf das Objekt als Ganzes und nicht auf Teile davon ('als Ganzes': beschreibt nicht „Teile des Schlüssels"), d.h.:

Jedes Nicht-Schlüsselattribut ist von jedem Schlüssel *voll funktional* abhängig!

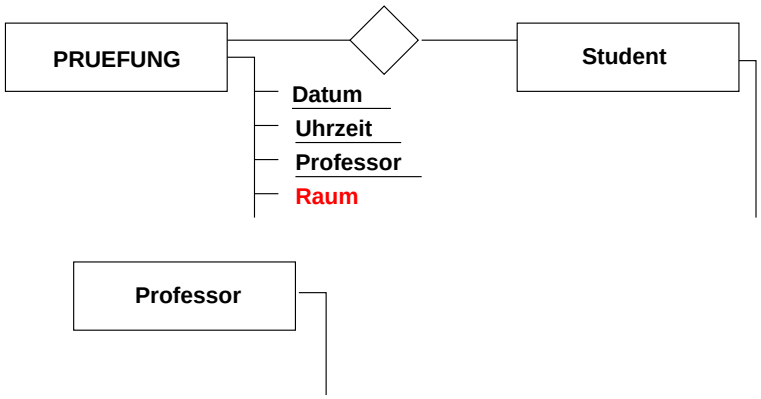


Abbildung 8.18: Zweite Normalform – Problem

Unterstellt sei in Abb. 8.18 (S. 210), dass der Raum für die Prüfung grundsätzlich das Dienstzimmer des Professors sei.

Probleme:

- Da es Professoren gibt, die sehr viele Prüfungen abnehmen, wird deren Raum(-Nummer) sehr oft in 'Prüfungen' vorkommen (Redundanz); wird ein Professor in ein anderes Dienstzimmer ziehen, was dann?
- 'Raum' beschreibt (in diesem Fall zumindest) nicht primär die Prüfung, sondern nur einen Teil davon ('Professor').
  Lösung: Attribut 'Raum' nur bei Entity-Typ/Tabelle 'Professor' führen!

3. Normalform

1. Normalform und

Jedes beschreibende Attribut bezieht sich unmittelbar und nicht nur mittelbar auf das Objekt, d.h.:

> Es gibt keine (nicht-trivialen) transitiven Abhängigkeiten von Nicht-Schlüsselattributen zu einem Schlüssel!

Beispiel:
Objekt: Abteilung
Attribute: AbtNr, Bezeichnung, . . . , GebNr, HM (Hausmeister) -
Schlüssel: AbtNr

Unterstellt sei, dass ein Gebäude mehrere Abteilungen „beheimatet“, dass eine Abteilung sich nicht auf mehrere Gebäude verteilt, dass es weiter zu jedem Gebäude nur einen Hausmeister gibt:

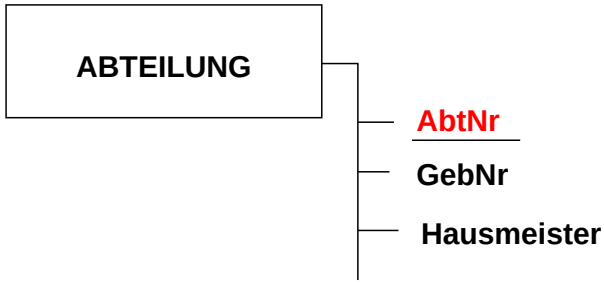


Abbildung 8.19: Dritte Normalform – Problem

ABTEILUNG	AbtNr	Bezeichnung	GebNr	HM
	E23	Microelektronik	23	Meier
	E24	Feinmechanik	25	Huber
	E27	Halbleiter	23	Meier

Probleme:

- neuer Hausmeister für Gebäude 23 → viele Änderungen
- neues Gebäude, Hausmeister ist bekannt, noch keine Abteilung eingezogen → Zuordnung Gebäude Hausmeister nicht möglich
- Verletzung der 3. Normalform: 'Hausmeister' bezieht sich nur mittelbar auf 'Abteilung'!

Lösung:

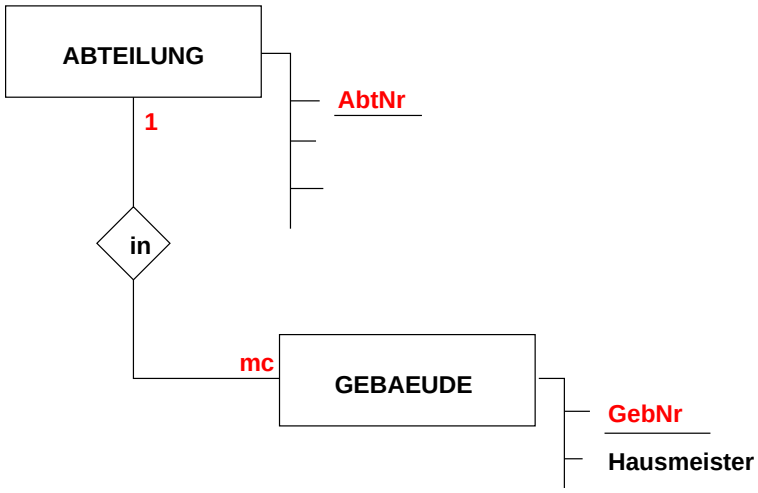


Abbildung 8.20: Dritte Normalform – Lösung

Abt	AbtNr	Bezeichnung	GebNr
	E23	Mikroelektronik	23
	E24	Feinmechanik	25
	E27	Halbleiter	23

Geb_HM	GebNr	Hausmeister
	23	Meier
	25	Huber
	27	

8.1.8 Umsetzen in Tabellen

Ausgangspunkt: konzeptionelles Modell beschrieben durch E/R-Diagramm

- Einführen künstlicher Schlüssel wg. Performance
  Personalnummer, Artikelnummer (keine „natürlichen“ Attrribute)
- ‘Kästchen’ (Objekt-, Entity-Typen) werden Tabellen

Beziehungen:

- ein Komplexitätsgrad = 1:

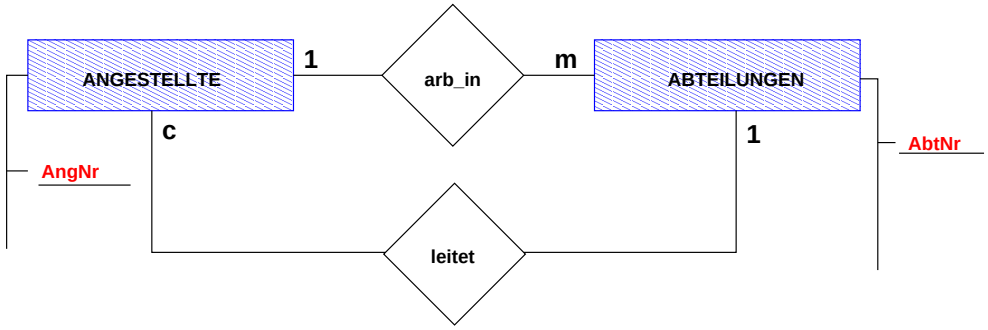


Abbildung 8.21: Vom ER-Diagramm zu Tabellen

arb_in Angestellte	AngNr	*weitere*	AbtNr

leitet Abteilung	AbtNr	*weitere*	AngNr

- kein Komplexitätsgrad = 1

  Beziehung wird zur Tabelle, deren Attribute die Schlüssel der beteiligten Objektklassen sind

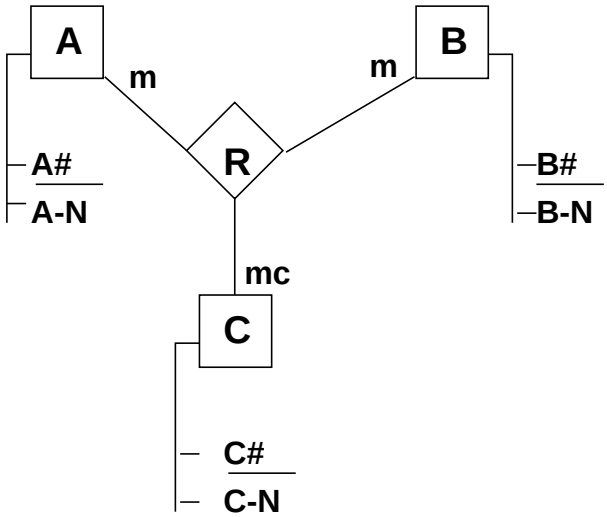


Abbildung 8.22: Vom ER-Diagramm zur Tabelle

R als Tabelle:

R	A#	B#	C#

8.2 Funktions-Diagramme

Ausgehend von Hauptfunktionen (Klassen von zusammengehörenden Funktionen) werden diese sukzessive hierarchisch zerlegt bis auf eine Ebene von Detailfunktionen, die dann direkt verbal oder z.B. in **Pseudo-Code** ("formalisierte Umgangssprache") definiert werden.

Beispiel:

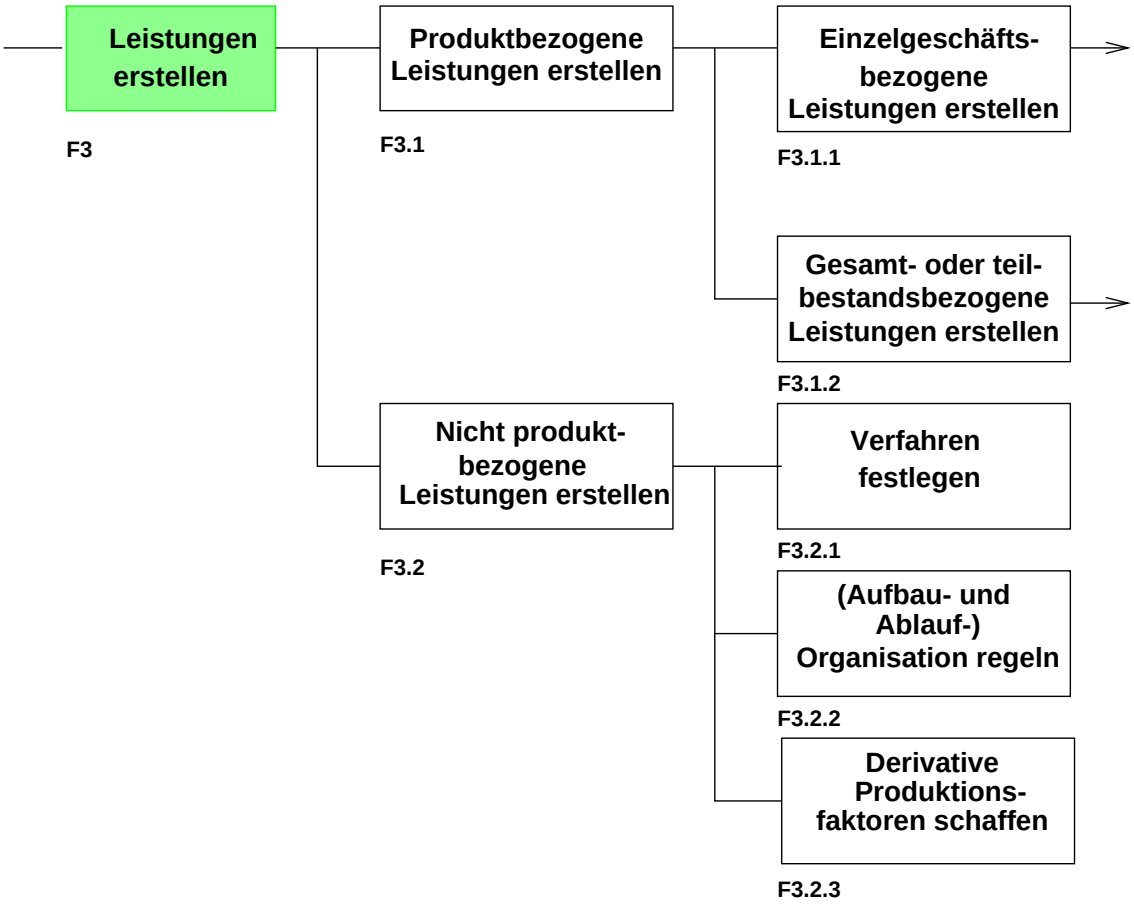


Abbildung 8.23: Funktions-Hierarchie

Der Bezug zu den in die Funktionen eingehenden bzw. von diesen erzeugten Daten wird hier nicht oder nur auf der Ebene der Detail-Beschreibungen gegeben. Hier setzen die sog. **HIPO-Diagramme** (HIPO – *Hierarchy of Input-Process-Output*). *Hierarchy* steht hier wieder für hierarchische Zerlegung der Funktionen, *IPO* steht für eine formalisierte Beschreibung der Input-Output-Wirkung der Funktionen (Detaildiagramme).

Beispiel für ein Detaildiagramm:

Input	**Process**	**Output**
Kundendaten + Artikeldaten + aktuelles Datum	Rechnung erstellen (F7.4.2)	Rechnung
...	...	...

8.3 Entscheidungstabellen

Zur kompakten Beschreibung von Entscheidungssituationen

Beispiel:

Ein Sachbearbeiter in einer Bank soll bei der Einlösung eines Schecks folgende Regeln beachten:

(1) Wenn die vereinbarte Kreditgrenze des Ausstellers überschritten wird, das bisherige Zahlungsverhalten aber einwandfrei war und der Überschreitungsbetrag kleiner als €200,- ist, dann soll der Scheck eingelöst werden.

(2) Wenn die Kreditgrenze überschritten wird, das bisherige Zahlungsverhalten einwandfrei war, aber der Überschreitungsbetrag über €200,- liegt, so soll der Scheck eingelöst und dem Kunden sollen neue Konditionen vorgelegt werden.

(3) War das Zahlungsverhalten nicht einwandfrei, wird der Scheck nicht eingelöst!

Der Scheck wird eingelöst, wenn der Kreditbetrag nicht überschritten ist.

Elementar-Bedingungen:

B1 Kreditgrenze ist überschritten

B2 Zahlungsverhalten ist einwandfrei

B3 Überschreitungsbetrag kleiner als €200,-

Aktionen:

A1 Scheck einlösen

A2 Scheck nicht einlösen

A3 Neue Konditionen vorlegen

A4 Unmögliche Aktion

Beispiel: B1 wahr UND B2 = wahr UND B3 = wahr, dann A1

Kompakte Darstellung als Tabelle:

	R1	R2	R3	R4	R5	R6	R7	R8
B1	W	W	W	W	F	F	F	F
B2	W	W	F	F	W	W	F	F
B3	W	F	W	F	W	F	W	F
A1	X	X			X		X	
A2			X	X				
A3		X						
A4						X		X

Komprimierte (Optimierte) Tabelle:

	R1	R2	R3/4	R5/7
B1	W	W	W	F
B2	W	W	F	-
B3	W	F	-	W
A1	X	X		X
A2			X	
A3		X		

8.4 Petri-Netze

8.4.1 Grundlagen

Stark vereinfacht (für Details: [?], [?]):

„Ein System besteht aus zwei Arten von Komponenten:

- passive Systemkomponenten (Bedingungen, Zustände, Daten, Ereignisse, ...) → „Stellen“
- aktive Systemkomponenten (Funktionen, Arbeitsschritte, Vorgänge, Veränderungen,...) → „Transition“

Aktive Komponenten hängen unmittelbar nur von passiven Komponenten ab und wirken unmittelbar nur auf passive Komponenten“!

Petri-Netz – Definition:

Ein *Petri-Netz* ist ein Tripel $P = (S, T, K \subseteq S \times T \cup T \times S)$ mit S endliche, nichtleere Menge von Stellen, T endliche, nichtleere Menge von Transitionen und K die Menge der gerichteten Kanten.

8.4.2 Kommunikationsnetze

Darstellung der Kommunikation von Funktionen (modelliert als Transitionen) über Daten (modelliert als Stellen).

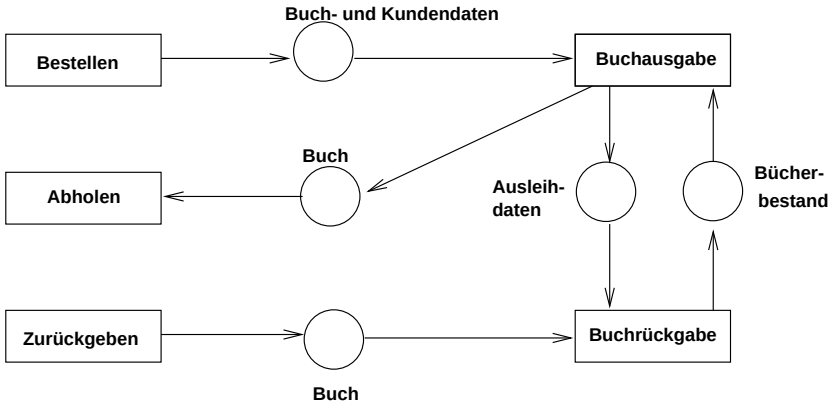


Abbildung 8.24: Petri-Netz – Bibliothek

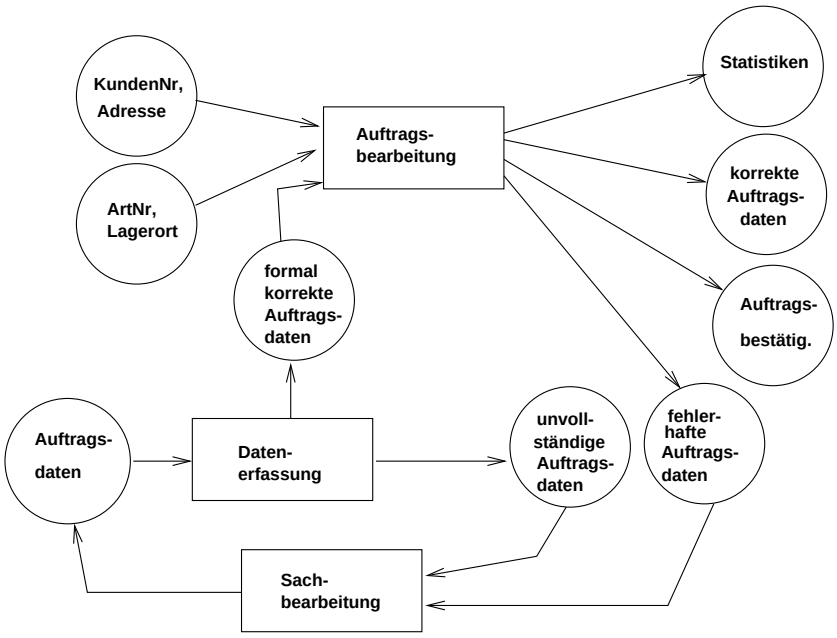


Abbildung 8.25: Petri-Netz – Auftragsbearbeitung

8.4.3 Dynamische Netze

Bedingungs-/Ereignisnetz:

- Markierungsfunktion: $m : S \longrightarrow FALSE, TRUE$
  In der grafischen Darstellung lässt sich die Tatsache, ob eine Stelle markiert ist, durch einen **Token** auf der Stelle ausdrücken.

- Schaltbereitschaft einer Transition:

Regel 1 Eine Transition $t \in T$ ist schaltbereit, g.d., w. gilt: jede Eingangstelle s von t ist markiert, d.h. $m(s) = TRUE$

oder

Regel 2 Eine Transition $t \in T$ ist schaltbereit, g.d., w. gilt: jede Eingangstelle s von t ist markiert, d.h. $m(s) = TRUE$, und jede Ausgangsstelle s' von t ist **nicht** markiert, d.h. $m(s') = FALSE$.

- Eine schaltbereite Transisition schaltet, indem **atomar** (unteilbar) und **zeitlos** alle Eingangsstellen auf FALSE und alle Ausgangsstellen auf TRUE gesetzt werden.

Konflikt: Zwei schaltbereite Transitionen stehen in Konflikt, wenn das Schalten der einen der anderen die Schaltbereitschaft nimmt.

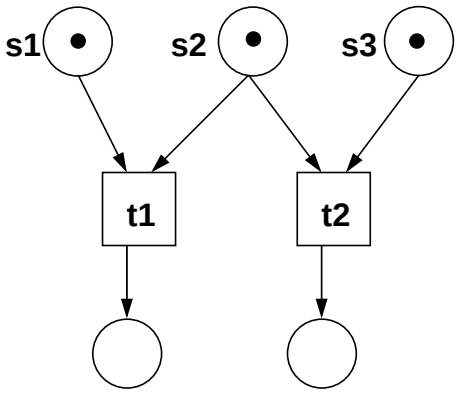


Abbildung 8.26: Petri-Netz – Konflikt

Bespiel: Synchronisation von Prozessen, die lesend und schreibend auf einen Datensatz zugreifen wollen

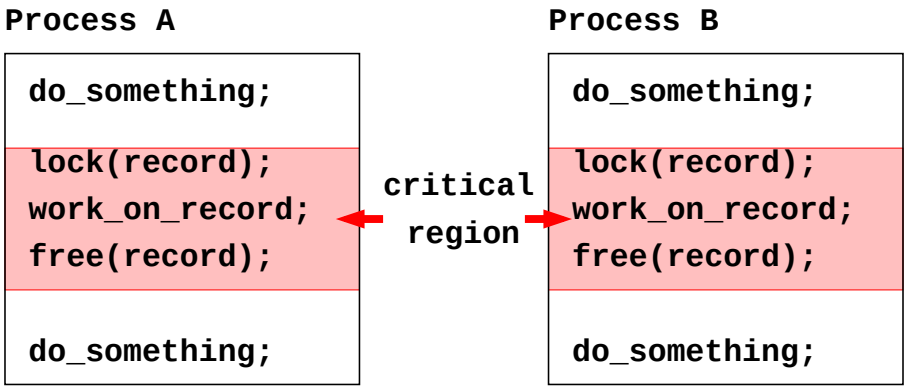


Abbildung 8.27: Synchronisation zweier Prozesse

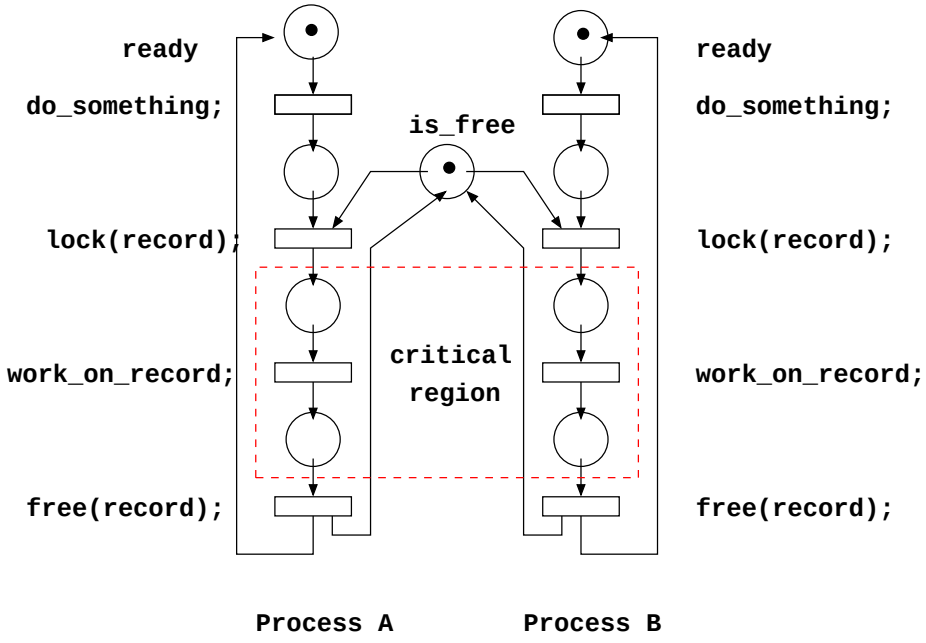


Abbildung 8.28: Synchronisation zweier Prozesse – Petri-Netz

Bei der Modellierung paralleler Abläufe gibt es ausgehend von einer vorgegebenen Anfangsmarkierung diverse Fragestellungen:

- **Lebendigkeit:** Welche Transitionen können durch Schalten erreicht werden?
- **Deadlock:** Wird durch Schalten von Transitionen ein Zustand erreicht, in dem "nichts mehr geht" (keine Transition mehr schaltbereit ist)? Solche Deadlocks können durchaus auch gewollt sein (sicherer Zustand).

Beispiel

In ein System gehen Informationen in Form von Telegrammen ein. Abhängig von einer Kennung wird die Information wie folgt weiterverarbeitet:

- Kennung = 1:
  In Datenbestände 'X' und 'Y' einsortieren (dazu müssen sowohl 'X' wie 'Y' gesperrt werden (Synchronisation).
- Kennung = 2:
  Informationen am Bildschirm darstellen - dazu werden aber Informationen aus 'X' und 'Y' benötigt (also ebenfalls Sperren nötig, um gleichzeitiges Verändern von 'X' und 'Y' verhindern).
- Kennung = 3:
  Sowohl Einsortieren wie auch Darstellen

Petri-Netz:

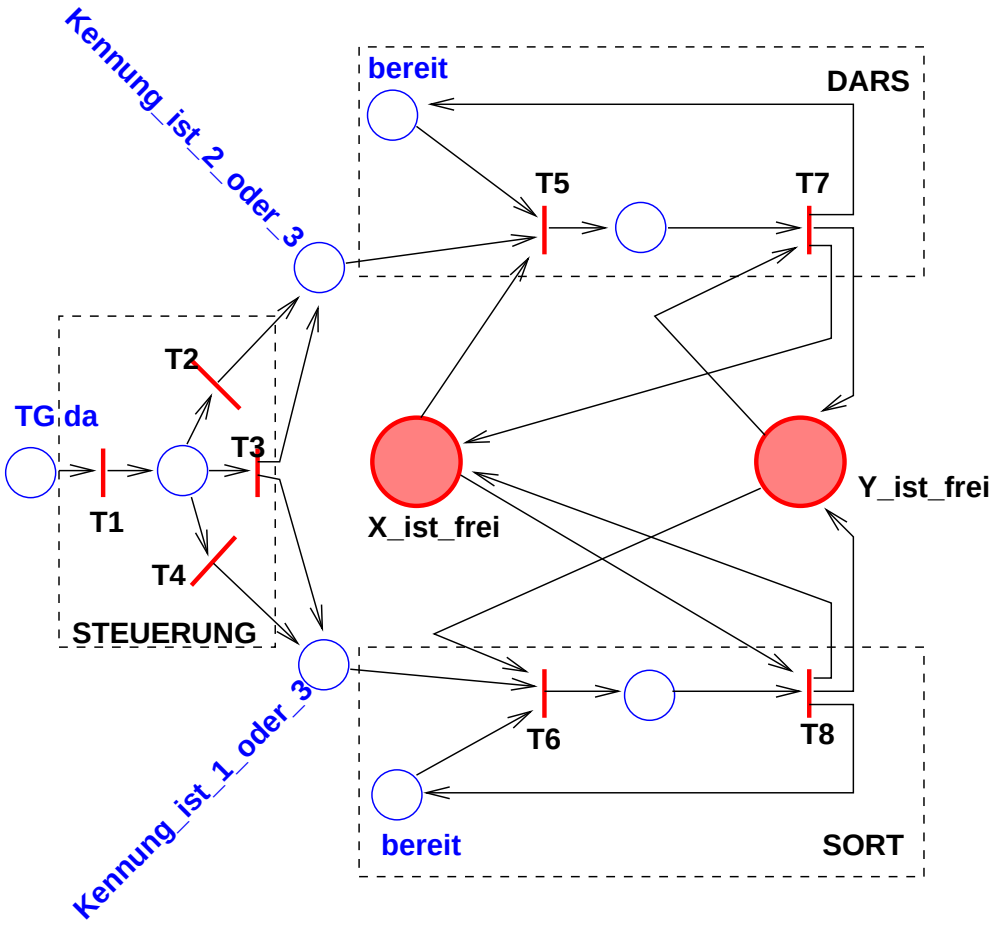


Abbildung 8.29: Petri-Netz – Multitasking I

Wenn man dieses System bzgl. der enthaltenen Kausallogik (ohne Berücksichtung von Zeit beim Schalten von Transitionen) simuliert, so erhält man die Abb. 8.30 (S. 224) dargestellte Situation (Anfangsmarkierung).

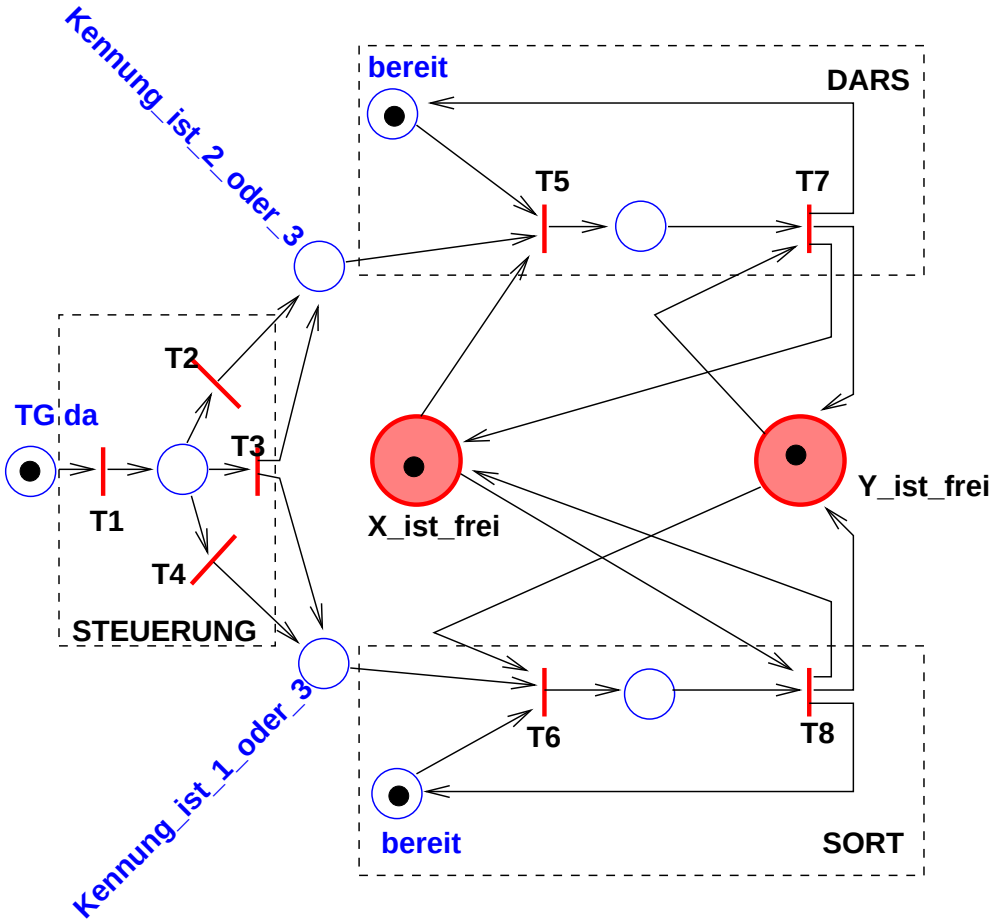


Abbildung 8.30: Petri-Netz – Multitasking II

8.4.4 Weitere Petri-Netz-Typen

Es gibt eine ganze Reihe von Modifikationen dieser Art von Petri-Netzen (siehe z. B. [?], [?]):

- Stellen können als Zählstellen betrachtet werden, also mehrere Tokens aufnehmen – an den Kanten werden ganze Zahlen angebracht, die einen Kantenfluss definieren – die Anzahl der Tokens auf einer Stelle ist beschränkt (Kapazität) – entsprechend wird die Schaltregel modifiziert **Stellen-Transitions-Netz**
- Tokens werden durch Symbole unterschieden, an den Kanten wird auf die nun unterscheidbaren Tokens mit Prädikaten eingegangen (**Prädikaten-Netz**).
- und weitere mehr

Beispiel: Ein System aus 3 parallelen Prozessen muss bezüglich des Zugriffs auf eine Datei wie folgt synchronisiert werden:

- Gleichzeitiges Lesen mehrerer Prozesse ist erlaubt.
- Sobald ein Prozess schreibt, kann kein anderer mehr lesend oder schreibend zugreifen.

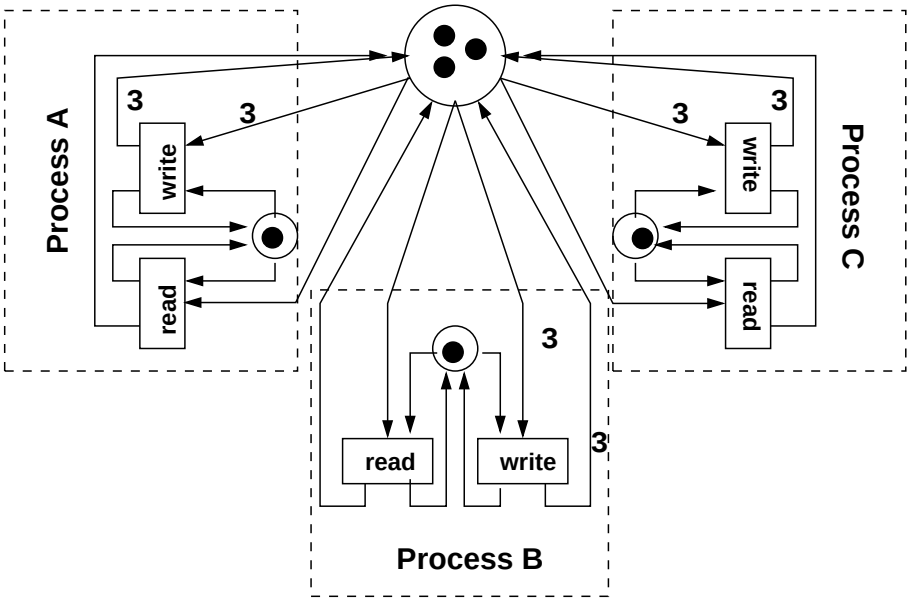


Abbildung 8.31: Stellen-Transitions-Netz

Kapitel 9

IT-Projekte

9.1 Vorbemerkungen

Informationstechnik-(IT-)Projekte:

(1) Auswahl, Anpassung und Einführung von Standardsoftware (Hardware, Software) in eine betriebliche Organisation

(2) Entwicklung von Individualsoftware (Auftragsentwicklung)

(3) Entwicklung von Software-Produkten (Standard-Software)

(4) Entwicklung von software-gesteuerten Produkten (*embedded systems*, z. B. elektronische Geräte in Automobilen)

↪ primär (1) und (2)

Zur Erinnerung:

- immaterielles Produkt
- kein Verschleiß, aber „Alterung“ (Anforderungen ändern sich)
- im Prinzip Unikat
- schwer zu „vermessen“
- im Prinzip keine physikalischen Grenzen (menschlicher Geist!)
- leicht(!) und schnell(!) änderbar (im Prinzip nur Textverarbeitung)

Management von IT-Projekten:

- Aufwandsschätzung (Zeit, Kosten, Ressourcen) – Erfahrung, *Function-Point-Methode*
- Planung (Strukturierung der Vorgehensweise, Zeitplanung, Ressourcen-Einsatz-Planung – Werkzeuge, Hilfsmittel, Personal, . . .)
- Risiko-Management („Was passiert, wenn . . . “)
- Kontrolle
- Steuerung und Koordination

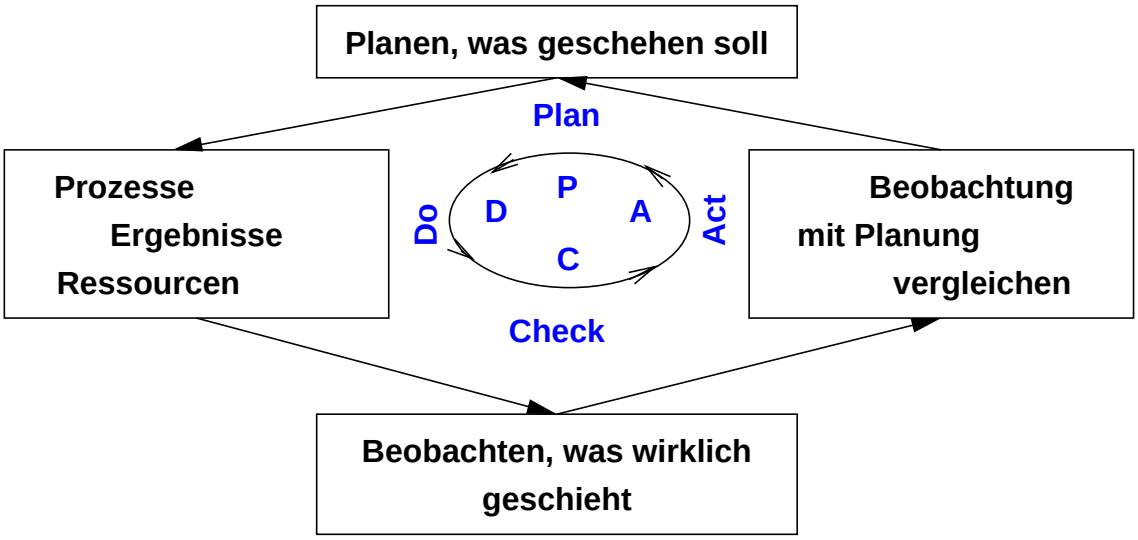


Abbildung 9.1: Projekt-Management

Besonders bei IT-Projekten – **Stör-/Einflussfaktoren:**

- eingeengte, eindimensionale Wahrnehmungen (Ideologien: *Java* ist besser als ...)
- erkennbar unrealistische Termine

  (Projektleiter erhält vom Vertrieb den dem Kunden versprochenen Termin für die Auslieferung – er weiß das heutige Datum, dazwischen muss das Projekt abgewickelt werden – er zerlegt diesen Zeitraum in Abschnitte, setzt also sog. Meilensteine für zu erledigende Teilaufgaben, baut aus Sicherheitsgründen „Puffer" ein – diese Abschnitte reicht er zur Feinplanung an Teil-Projektleiter weiter – diese gehen nach dem selben Muster vor – der Sachbearbeiter erhält Vorgaben, von denen er klar weiß, dass er diese nicht oder zumindest nicht mit der nötigen Sorgfalt / Qualität erfüllen kann)

- geringe Projektidentifikation (zu viele andere parallel auszuführende, „wichtiger" Arbeiten)
- geringe Konfliktbereitschaft (Software-Entwicklung verlangt intensive Kommunikation und damit auch Diskussion!)
- mangelhafte Problemlösungsfähigkeit (ganzheitliches, auch abstraktes Denken geringe Fähigkeit, konzeptionell zu denken)
- hohe Fehlerquoten
- Mängel im „Vorleben von Qualität und Termintreue" bei Vorgesetzten
- unklare Aufgabenstellung
- ständiges reagieren statt agieren
- Verschweigen von Problemen (Software ist immateriell, Probleme sind selten „physikalisch" erkennbar)
- „Denken in Positionen"
- enges Verantwortlichkeitsdenken („Nicht mein Problem")
- falsche Einstellung zum „Fehler" (stillschweigendes Beheben der Fehler, ohne nach den Ursachen zu fragen)
- und andere mehr.

Betrachtung der Zusammenhänge:

- Wechselwirkung (gleich- / entgegengerichtet)
- Intensität (stark, schwach)
- zeitlicher Wirkungsverlauf

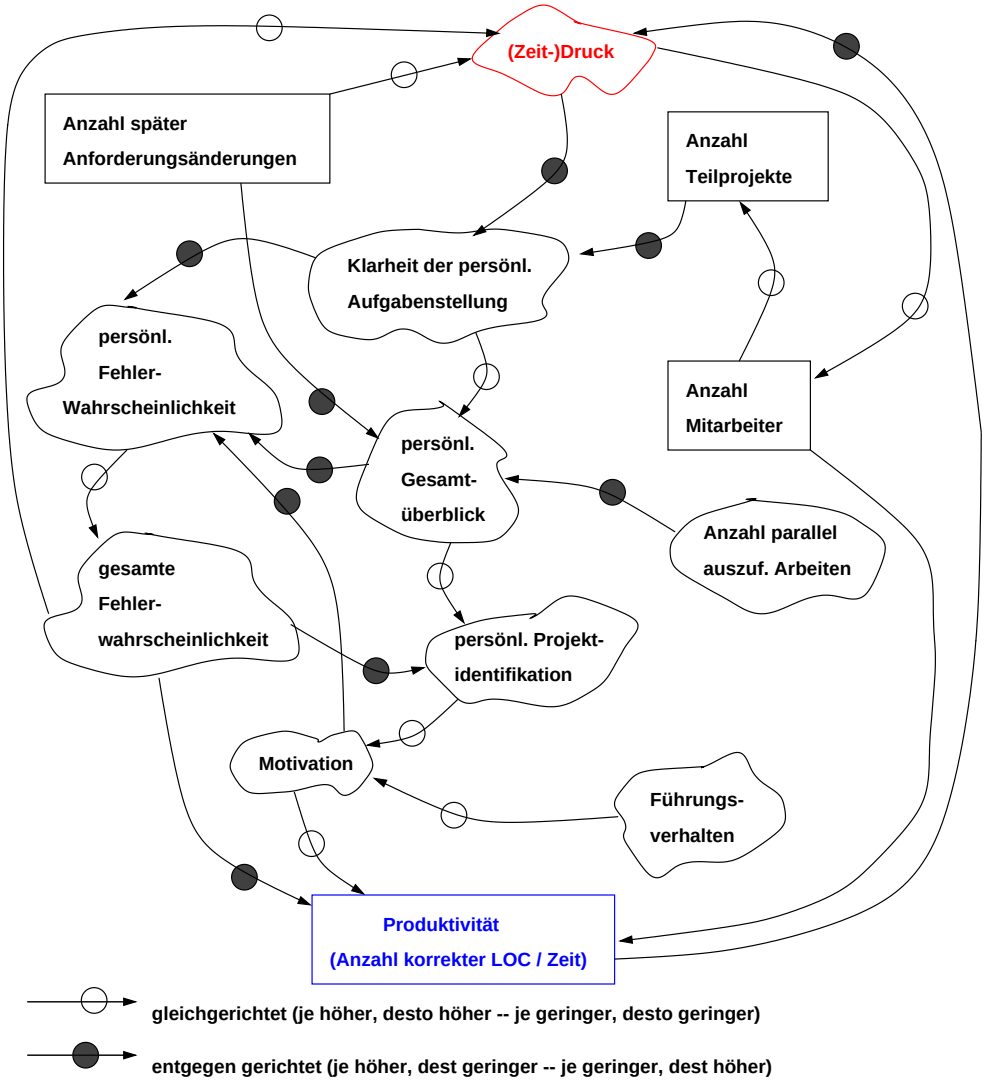


Abbildung 9.2: Wirkungsdiagramm

9.2 Projekt-Organisation

9.2.1 Projektabwicklung in der Linienorganisation

- Das Projekt wird in den bereits bestehenden Fachabteilungen abgewickelt.
- Die Leitung erhält diejenige Abteilung zugewiesen, auf die der größte Teil der Projektarbeit entfällt.
- Der betreffende Abteilungsleiter nimmt die Aufgaben des Projektleiters in Personalunion wahr.
- Für die Projektabwicklung sind keine wesentlichen organisatorischen Veränderungen im Unternehmen erforderlich!
- Geeignet für Projekte, die überwiegend in einer einzigen Fachabteilung abgewickelt werden können!

9.2.2 Stabs-Projektorganisation

auch Einfluss-Projektmanagement

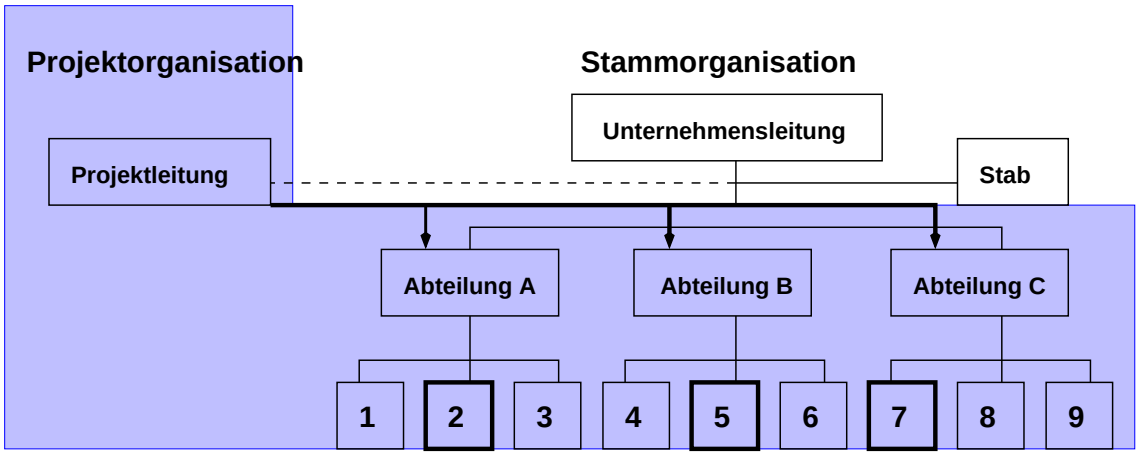


Abbildung 9.3: Stabs-Projektorganisation

Die Projektleitung erfolgt in den Fachabteilungen, die Projektkoordination wird in den Stäben wahrgenommen; der Projektleiter als Stabsstellenleiter hat nur Informations- und Beratungsbefugnisse. Da den projektbetreuenden Stellen somit die Weisungsbefugnis fehlt, ist der Projekterfolg stark vom Willen aller beteiligten Stellen abhängig.

Aufgaben des Projektleiters

- reine Koordinations- und Informationsfunktion zwischen an den am Projekt beteiligten Stellen und Mitarbeitern
- Planungsaufgaben
- Kontrollfunktion (Termine, Kosten, Qualität)
- Meldung von Planabweichungen an vorgesetzte Stelle
- Entscheidungsvorbereitung / Vorschlag für Maßnahmen an Linieninstanzen

Vorteile:

- Bestehende Organisation wird nicht verändert (geringer organisatorischer Aufwand, viele Projekte können gleichzeitig bearbeitet werden)
- Mitarbeiter verbleiben in ihrer „angestammten“ Abteilung (flexible Auslastung der personellen Kapazität möglich, eher geringe Unsicherheit über „Was nach Projektende?“

potenzielle Probleme

- Projektleitung hat Verantwortung, aber nicht die notwendige Weisungsbefugnis
- nicht lösbare Konflikte muss die Projektleitung „nach oben“ melden (Zeit!)
- Auftraggeber hat in der Projektleitung keinen kompetenten (entscheidungsberechtigten!) Partner, also geringe Kundenorientierung!

Einsatzbereiche

- Projekte mit gut strukturierten Aufgaben
- Projekte, in denen den Mitarbeitern die Vorgehensweise bekannt ist
- kleine, unkritische (Zeit, Risiken im Einsatz) Projekte

9.2.3 Reine Projekt-Organisation

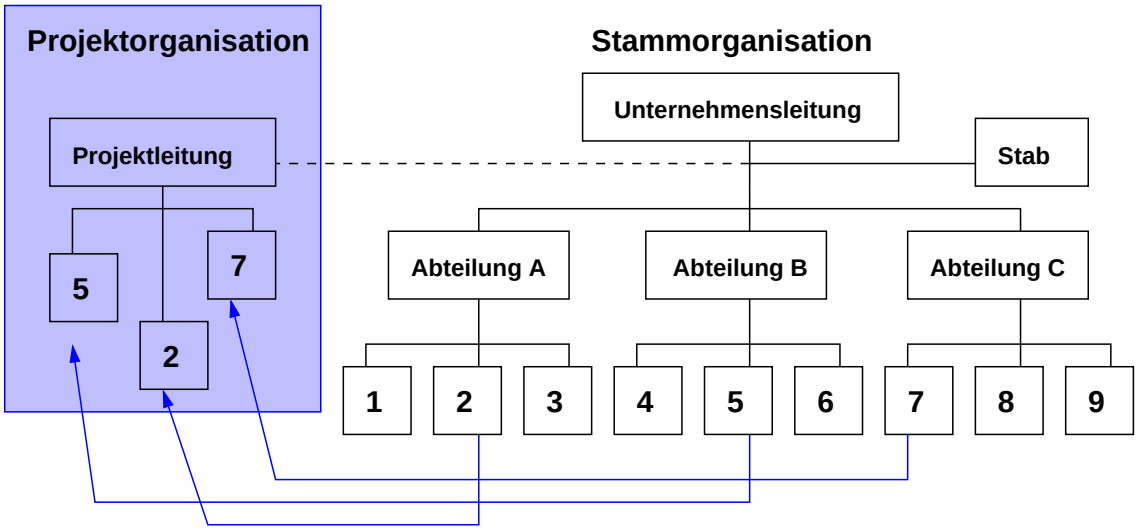


Abbildung 9.4: Task Force

Projekt-Mitarbeiter werden vollständig aus ihrer Abteilung herausgenommen und fachlich wie disziplinarisch der Projektleitung unterstellt (*Task Force Modell*

Vorteile:

- eindeutige Zuordnung von Aufgaben wie Kompetenzen an die Projektleitung
- volle Konzentration **aller** Beteiligten auf das Projekt
- Projektleitung ist klare Schnittstelle zum Projekt für alle Beteiligten (auch Auftraggeber)
- fördert Projekt-Identifikation und damit i.a. Motivation
- ...

(Mögliche) Probleme (der reinen Projekt-Organisation):

- Auslastung der abgeordneten Mitarbeiter in verschiedenen Projektabschnitten unterschiedlich (Spezialisten!)
- qualitative und quantitative Schwächung der abordnenden Abteilung ↪ werden wirklich die Besten abgeordnet?
- die in der abordnenden Abteilung verbleibenden Mitarbeiter werden stärker belastet
- Projektmitarbeiter werden vom Geschehen in ihrer Abteilung abgeschnitten (Rückkehr?)
- ...

Einsatzbereich(e):

- Projekte mit hohem Schwierigkeitsgrad
- Projekte mit hohem Risiko (Haftung?)
- „sehr große" Projekte
- ...

9.2.4 Matrix-Projektorganisation

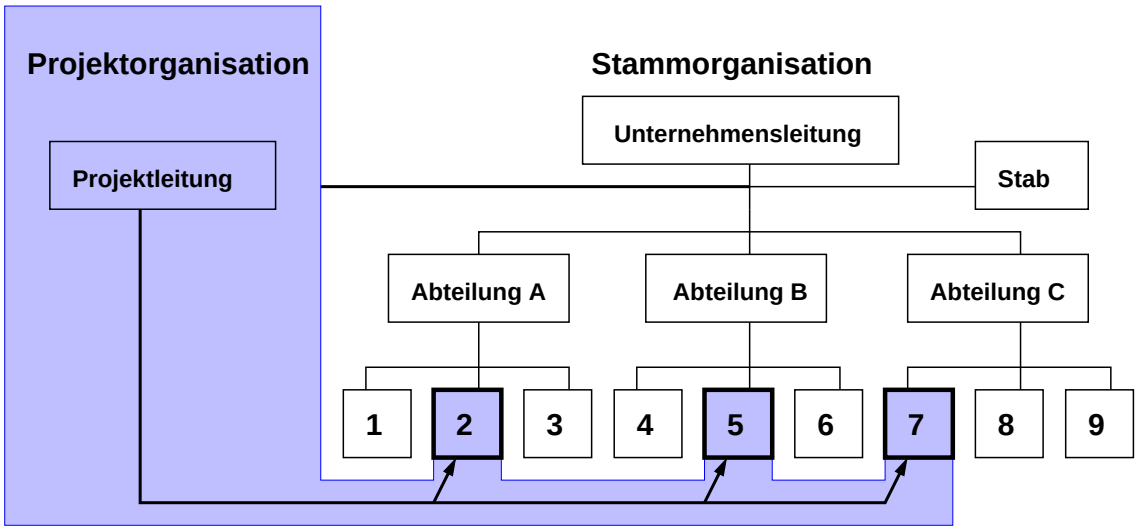


Abbildung 9.5: Matrix-Organisation

Projekt-Mitarbeiter sind fachlich wie disziplinarisch weiter dem Abteilungsleiter unterstellt und nur zeitweise ins Projekt „abgestellt“.

Die Matrix-Organisation ist somit in ihrer Grundstruktur eine zweidimensionale Organisationsform (vertikale Hierarchie, horizontale Projektorganisation). Über die primäre funktionale Gliederung im Unternehmen wird eine nach verschiedenen Projekten geordnete Projektebene gelegt. Jede Organisationseinheit ist zwei Instanzen unterstellt, dem Fachabteilungs und dem Projektleiter.

Die Position des Projektleiters wird in der Regel auf der Hierarchiestufe der Fachabteilungsleiter verankert. Es ist notwendigerweise eine Kompetenzaufteilung zwischen funktionalem und projektbezogenem Leitungssystem zu entwickeln.

„Punkte des beabsichtigten Konflikts“

- Mitarbeiter unterliegen einer „Doppel-Unterstellung“ (disziplinarisch ihrem Abteilungsleiter, fachlich dem Projektleiter)

  Wichtig:

  - Konflikte möglichst früh offenlegen
  - rechtzeitig Lösungen finden

- von allen Beteiligte ist gut ausgeprägtes Führungsverständnis und ebenso Konfliktlösungsfähigkeit gefordert
- weder Abteilungsleitung noch Projektleitung entscheiden / handeln alleine
  - Zwang zur Gemeinsamkeit kann zu besseren Lösungen führen
  - klare Kompetenzaufteilung zwischen Projekt- und Abteilungsleitung ist unabdingbar
- Entscheidungs- und Weisungsbefugnis des Projektleitung sind auf Dauer und Umfang des Projekts beschränkt

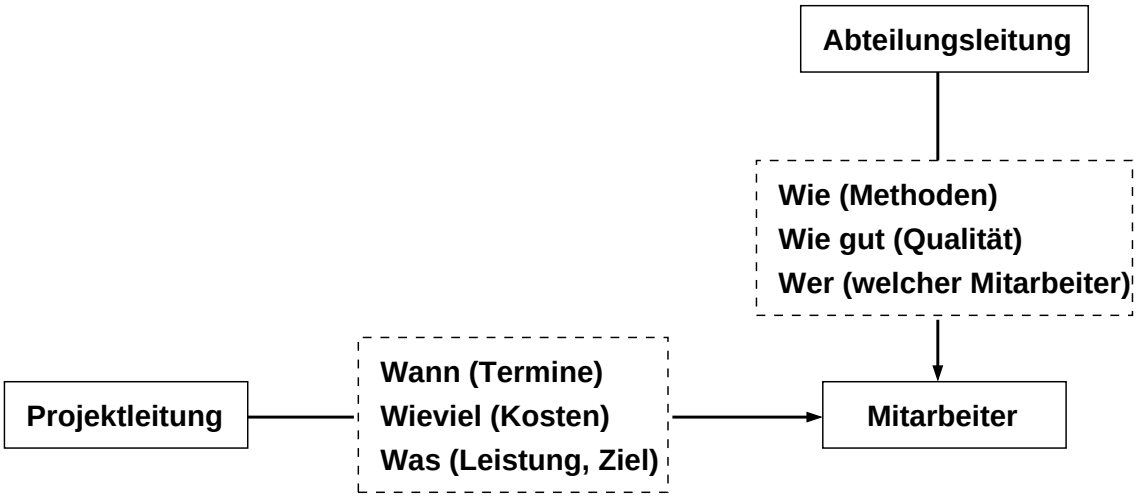


Abbildung 9.6: Kompetenzaufteilung in der Matrix-Organisation

Vorteile:

- Projektziele können von Projektleitung selbständig und direkt verfolgt werden
- flexibler Einsatz von Ressourcen (insb. Personal)
- gezielter Einsatz von Spezialisten
- ...

potenzielle Probleme:

- Verunsicherung des Linienvorgesetzten wegen Verzicht auf Auschließlichkeitsansprüche
- Mehrfachunterstellung der Mitarbeiter (je mehr Projekte desto mehr Chefs)
- Kompetenzkonflikte
- Mitarbeiter spielen Vorgesetzte gegeneinander aus
- ...

Einsatzgebiete:

- Projekte, an denen viele Abteilungen beteiligt sein müssen (hoher Koordinationsaufwand!)
- Viele parallele Projekte
- Projekte mit starker Marktorientierung (Mitarbeit von Vertrieb, Marketing, Kundenservice, ...)

9.2.5 „Chief-Programmer“-Organisation

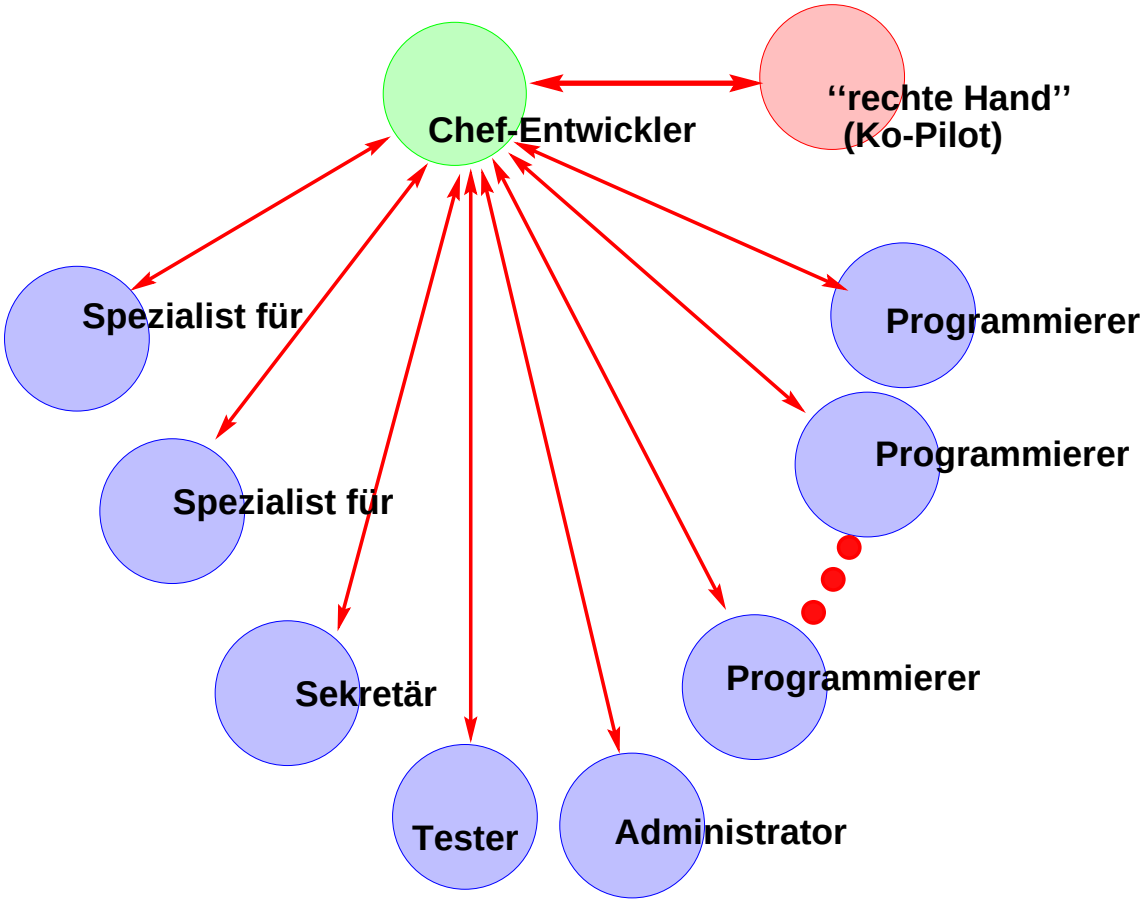


Abbildung 9.7: Chief Programmer Team

Motivation:

- Untersuchungen zeigen, dass es erhebliche Unterschiede in den „Fähigkeiten“ (Produktivität, Qualität der Ergebnisse) von Programmierern / Software-Entwicklern gibt
- Untersuchungen zeigen, dass in „normalen“ Projektorganisationen sehr viel „Leerlauf“ gibt
- Die besten Software-Produkte (Spiele) kommen von kleinen Teams
- Erfahrene Teams sind eine äußerst wertvolle Ressource
- Erfahrene Programmierer sind selten gute Manager
- **Weniger leisten oft mehr:**

Ein einfaches mathematisches Modell

Sei n die Zahl der Mitarbeiter in einem Projekt – wir nehmen an, die Projektdauer T ist von n abhängig, also

$$T = T(n)$$

In einem Projekt findet Leistung statt, die unmittelbar in das Ergebnis eingeht – nennen wir es *Nutzleistung* und bezeichnen die dadurch beschriebene Projektdauer mit $T_N(n)$

In einem Projekt findet aber auch Leistung statt, die allenfalls mittelbar in das Ergebnis eingeht (Kommunikation) – nennen wir es *Blindleistung* und bezeichnen die dadurch beschriebene Projektdauer mit $T_B(n)$

Dann gilt:

$$T(n) = T_N(n) + T_B(n)$$

Die Steigerung der durch die *Blindleistung* bedingten Projektdauer bei Hinzunahme weiterer Mitarbeiter sei proportional zur Anzahl der neuen Mitarbeiter und zur bisherigen Projektdauer:

$$T_B(n+h) - T_B(n) \sim h \cdot T_B(n)$$

also

$$\lim_{h \to 0} \frac{T_B(n+h) - T_B(n)}{h} = a \cdot T_B(n)$$

also

$$T'_B(n) = a \cdot T_B(n)$$

Diese Gleichung hat die Lösung

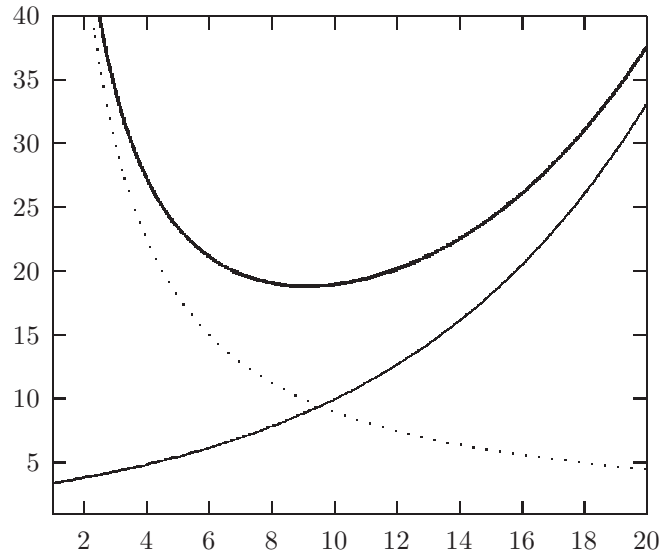
$$T_B(n) = b \cdot e^{a \cdot n}$$

Die durch die *Nutzleistung* bedingte Projektdauer kann nach dem Motto „je mehr desto schneller“ beschrieben werden durch

$$T_N(n) = \frac{d}{n}$$

Die Konstanten a, b, und d beschreiben Einflussfaktoren auf die Projektdauer wie Ausbildung und Erfahrung der Mitarbeiter, Projekt-Komplexität a.a.m.

Prinzipieller Verlauf:



Es gibt also ein gewisses Optimum!

Mit diesem Modell läßt sich sehr leicht auch das **Brooks'sche Gesetz**

– Adding manpower to a late project makes it later –

herleiten!

Lösungsansatz beim Chief Programmer Team:

- Um einen Spitzenprogrammierer herum wird ein Team gebildet, mit dem Ziel, dessen Produktivität zu optimieren!
- Zusammensetzung:
  - **senior people** – Erfahrene Mitarbeiter als Chef-**Entwickler** und Co-Pilot
  - **junior people** – Weniger erfahrene Mitarbeiter als Tester, Mit-Programmierer („Lehrlinge und Gesellen“)
  - **support staff** – Pflege und Verwaltung der Bibliothek, Versionsverwaltung, Dokumentation, Administration, ...
- Einige (nicht der Chef-Programmierer) sollten in mehreren Projekten arbeiten!

Vorteile:

- saubere, klare Verantwortung, Autorität
- erhebliche Produktivitätssteigerung
- Technische Karriere-Wege
- leichter, ein klares, konsistentes Systemkonzept sicherzustellen
- hilft, den sog. *burnout* zu vermeiden (?!)
- ...

Woran scheitert es in der Praxis?

- fehlender Mut und Einfallsreichtum von Managern (Prinzip Lemminge: Machen was die andern machen!)
- fehlende „Programmier-**Meister**“
- kurzfristiges Denken – Heranbilden von Meistern dauert nun mal!
- ...

9.3 Schichten-Modell der Anwendungsentwicklung

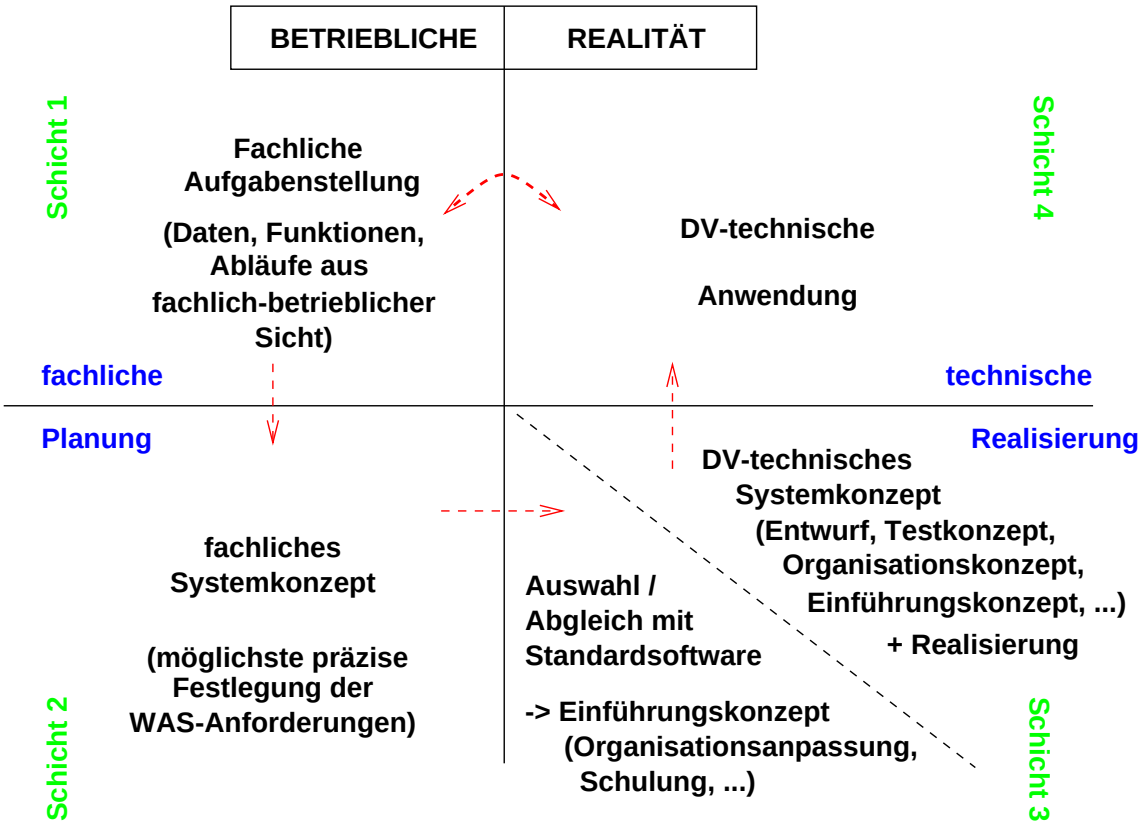


Abbildung 9.8: Schichtenmodell

Wichtig: Entwicklungsergebnisse müssen in irgendeiner Form fixiert werden

↪ definierten Bezugspunkt für weitere, darauf aufbauende Aktivitäten

↪ zugänglich für Qualitätsprüfung

Dazu bedarf es *Beschreibungsmittel* (Sprachen), die vorgegebene *Methoden* und *Prinzipien* unterstützen und selbst durch *Werkzeuge* unterstützt werden.

Einige grundlegende Prinzipien

- Konstruktive Voraussicht — ein System lebt länger als man glaubt und es passiert mehr damit als man glaubt!
- Abstraktionsstufen mit schrittweiser Verfeinerung (Konkretisierung) — „divide et impera“
- Lokalität — das zusammenfassen und zusammen beschreiben, was zusammengehört!
- Information Hiding — Realisierungsdetails verbergen und nur wohldefinierte, explizite Schnittstellen bereitstellen

Methoden:

- „planmäßig angewandte, nachvollziehbare Vorgehensweisen zur Erreichung festgelegter Ziele“)
- diese basieren in der Regel auf Prinzipien und sind mit gewissen Formalismen (Beschreibungsmitteln) verknüpft

Die abgestimmte Kombination einzelner konstruktiver Elemente (Prinzipien, Methoden, Formalismen) wird als **Technik** bezeichnet.

9.4 Querschnittsaufgabe: Konfigurationsmanagement

9.4.1 Aufgabenstellung

Während einer Software-Entwicklung (aber genauso bei der Pflege!) fallen viele Informationen zum eigentlichen Programm an, die in irgendeiner Form fixiert werden müssen – sei es

- für die **Kommunikation** zwischen den Beteiligten,
- für die inhaltliche **Projektkontrolle** (Status, . . .),
- für die **kaufmännische Abwicklung** (Geld- / Leistungs-Fluss),
- oder für die Unterstützung der späteren **Pflege** (*Patches*) resp. des späteren Betriebs (Erzeugen / Installieren).

Diese komplexe Struktur wie auch die Menge der Informationen gilt es „zu beherrschen“:

- inhaltlich:
  Konsistenz, Vollständigkeit, Korrektheit, . . .
- formal:
  Identifikation, Zusammenhang, Aktualität, Änderungsweg, . . .

Bildlich ausgedrückt:

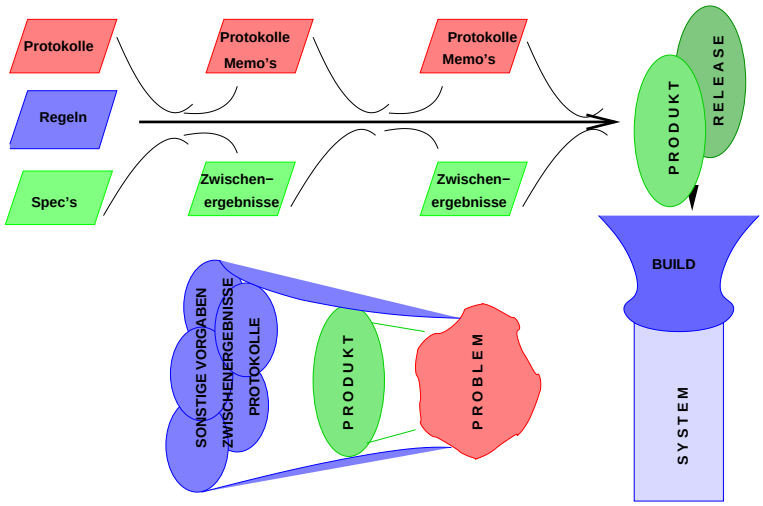


Abbildung 9.9: Informationsanfall

Klassisch: Stücklisten

- PDM – Product Data Management / PLM – Product Lifecycle Management

  > PDM is the discipline of controlling the evolution of a **product design** and all related product data during the full product life cycle, historically with the focus upon hardware product design. The term PDM includes the overall description of the topic and the methodology, while a PDM system is a tool you use for managing the **data and the processes** you have decided to use it for.

- SCM – Software Configuration Management

  > SCM is the discipline of controlling the evolution of a software product, with emphasis on the development phase.

  jeweils aus: Product Data Management and Software Configuration Management – Similarities and Differences. Hrsg.: Association of Swedish Engineering Industries, 2001

PDM und SCM haben sich unterschiedlich entwickelt, sind häufig aber nicht (mehr) zu trennen

---

Wo ist das Problem?

Besonderheiten von Software im Vergleich zu anderen technischen Produkten:

- **immaterielles Produkt**
- kein Verschleiß, aber „Alterung“ (Anforderungen ändern sich)
- im Prinzip Unikat
- **schwer zu „vermessen“**
- im Prinzip keine physikalischen Grenzen (menschlicher Geist!)
- **leicht(!) und schnell(!) änderbar (i. P. Textverarbeitung)**

Arbeiten mehrere / viele Entwickler an einem System – das dürfte die Regel sei – so tauchen bei nicht kontrollierten / nicht abgestimmten Änderungen u. a. folgende Probleme auf:

- Simultane Änderungen:
  Änderungen des Einen machen Änderungen des Anderen zunichte
- *Shared Code*
  Änderungen in einem Code, der von mehreren geteilt wird, werden nicht von allen Beteiligten berücksichtigt
- *Common Code*
  Änderungen an Programmfunktionen oder Datenstrukturen, die von mehreren benutzt werden, „sprechen sich nicht herum“

- Versionsüberblick
  Größere Programme existieren zu bestimmten Zeitpunkten in mehreren Versionen: eine beim Kunden, eine im Testlabor, eine in der Entwicklung – entsprechend müssen auftretende Änderungen berücksichtigt werden können.
- Variantenüberblick
  Kundenspezifische Varianten können zu noch größeren „Übersichts"-Problemen führen. Oder: „Welcher Kunde hat welches Release"?
- „Falsche" Änderungen müssen rückgängig gemacht werden – Wie war der Zustand zuvor?
- Wo – auf welchem Rechner / in welchem Dateisystem – liegen z.B. die aktuellen Testdaten?
- Regressionstests
  Welche Testdaten werden zu welcher Programmversion bei Testwiederholungen benötigt?
- Dokumentation vs. Dokumentation vs. Programm
  Nicht mehr aktuelle Dokumentation kann der „Tod" der Dokumentation sein!
- ...

Schlüsselfragen:

- Was soll überhaupt als kleinste, identifizierbare und austauschbare Einheit betrachtet werden? Eine Prozedur? Eine Quelldatei? Eine Seite Text? Ein Datei mit Testdaten?
- Aus welchen Einheiten besteht das System zur Zeit? (Identifikation der konstituierenden Elemente)?
- Wer ist für welche Einheit verantwortlich? Was sind deren sonstigen, für das Management wichtigen Charakteristiken?
- Wie ist der strukturelle Zusammenhang zwischen diesen Einheiten (horizontal, vertikal)?
- Wie ist der aktuelle Zustand / die „Historie" der Einheiten / des Ganzen?
- Wie können Änderungen kontrolliert werden, so dass sie **gewollt** sind, von **allen** Beteiligten registriert werden, in ihrer Auswirkung auf die anderen Teile wie auf das Ganze beherrscht werden können, ggf. rückgängig gemacht werden können?
- Wer darf wann was wie ändern?
- Wie kann sichergestellt werden, dass die Qualität mit einer Änderung zumindest nicht verschlechtert wird?
- Wie wird das System für den Kunden X erzeugt?
- Welche Patches sind wo warum eingespielt?
- ...

9.4.2 Die Säulen des SCM

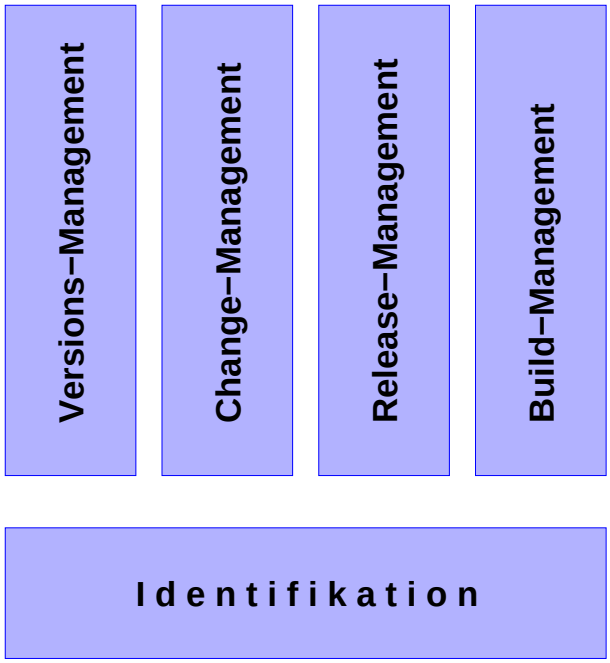


Abbildung 9.10: Säulen des SCM

9.4.3 Begriffe nach IEEE

configuration

(1) *The arrangement of a computer system or network as defined by the nature, number, and the chief characteristics of its functional units. More specially, the term configuration may refer to a hardware configuration or software configuration. (ISO)*

(2) *The requirements, design, and implementation that define a particular version of a system or system component.*

(3) *The functional and / or physical characteristics of hardware / software as set forth in technical documentation and achieved in a product (DoD-STD 480A)*

Konfiguration:

- die zu einem konkreten Zeitpunkt existierende Menge der die Software / das **System konstituierenden Objekte** (Daten- / Funktionskomponenten) sowie
- andere Informationskomponenten (z.B. Testdaten, Build-Anweisungen, Installationsanweisungen) bezogen auf eine bestimmte Abstraktionsstufe sowie
- deren **struktureller Zusammenhang** (vertikal wie horizontal) und
- ihre **Einbettung in das Ganze**

Die Einheiten der Konfiguration (Objekte, *units*, *items*) sollten auch unter Änderungs-Gesichtspunkten definiert sein!

Was im Einzelfall eine Einheit ist, hängt von der jeweiligen Sicht ab:

- für den Anwender das Betriebssystem als Ganzes
- für den Hersteller die einzelnen Module dieses Betriebssystems

configuration item

(1) *A collection of hardware or software elements treated as a unit for the purpose of configuration management*

(2) *An aggregation of hardware/software, or any of its discrete portions, that satisfies an end use function and is designated for configuration management. Configuration items may vary widely in complexity, size, and type from an aircraft, electronic or ship system to a test meter or round of ammunition. During development and initial production, configuration items are only those specification items that are referenced directly in a contract (or an equivalent in-house agreement). During the operation and maintenance period, any reparable item designated for separate procurement is a configuration item. (DoD-STD 480 A).*

configuration management

(1) *The process of* **identifying and defining** *the configuration items in a system,* **controlling** *the release and change of the items throughout the system life cycle,* **recording and reporting** *the status of configuration items and change requests, and* **verifying** *the completeness and correctness of configuration items. See also change control, identification, configuration control, configuration status accounting, configuration audit.*

(2) *A discipline applying technical and administrative direction and surveillance to*

(a) identify and document the functional and physical characteristics of a configuration item,

(b) control changes to those characteristics, and

(c) record and report change processing and implementation status.

(DoD-STD 480 A)

Massnahmen zum SCM:

- Identifikation
  Feststellung der Konfiguration durch Auflistung der konstituierenden Einheiten und ihrer Charakteristiken

Nach [IEEE]:
configuration identification

(1) *The process of designating the configuration items in a system and recording their characteristics.*

(2) *The approved documentation that defines a configuration item.*

(3) *The current approved or conditionally approved technical documentation for a configuration item as set forth in drawings and associated lists and documents referenced therein. (DoD-STD 480A)*

- **Kontrolle**
  Angabe des Verfahrens und der Schritte, um Änderungen durchzuführen

Nach [IEEE]:
configuration control

(1) *The process of evaluating, approving or disapproving, and coordinating changes to configuration items after formal establishment of their configuration identification.*

(2) *The systematic evaluation, coordination, approval or disapproval, and implementation of all approved changes in the configuration of a configuration item after formal establishment of its configuration identification. (DoD-STD 480A)*

- **Überwachung**
  Soll-Ist-Vergleich der Konfiguration

Nach [IEEE]
configuration audit

The process of verifying that all required configuration items have been produced, that the current version agrees with specified requirements, that the technical documentation completely and accurately describes the configuration items, and that all change requests have been resolved.

- **Status-Verfolgung**
  Aufzeichnen der Veränderung der Konfiguration

Nach [IEEE]
configuration status accounting

The recording and reporting of the information that is needed to manage a configuration effectively, including a listing of the approved configuration identification, the status of proposed changes to the configuration, and the implementation status of approved changes. (DoD-STD 480A)

Wichtig: diese Maßnahmen müssen **vor** Beginn der Entwicklung implementiert werden – Ihre Effizienz hängt entscheidend vom Entwicklungsvorgehen und den einzusetzenden Tools ab. Wegen des Umfangs der Maßnahmen ist anzustreben, möglichst viel davon zu automatisieren.

Viele der Aufgaben des SCM sind weitgehend „buchhalterischer“ Natur, nichtsdestoweniger aber immens wichtig.

9.4.4 Identifikation

- Eindeutige Bezeichnung aller existierender “Items”
- Angabe der für die jeweiligen Ziele (z.B. Änderung, Pflege, Fortschrittskontrolle, u.dgl.m.) notwendigen Charakteristika dieser Items
- Einbettung der Items in das “Ganze” (z.B. welche Phase, andere zu diesem Item gehörende Items ↪ Schnittstellen, Umgebung vertikal wie horizontal, *Makefiles*)

Auf der Ebene des Quellcodes i.a. geregelt

↪ Dokumentation???

Quellcode ist (auch) Dokumentation

↪ Prinzipien des Programmierens (strukturiert, modular, objektorientiert) übertragen auf die Erstellung der Dokumentation

NB.: Dokumentation darf nicht Selbstzweck sein, sondern muss einem Zweck **jetzt** (Arbeitsmittel, Kommunikation, Kontrolle) und **später** dienen!

Dokumentationsstruktur ↪ *divide et impera*

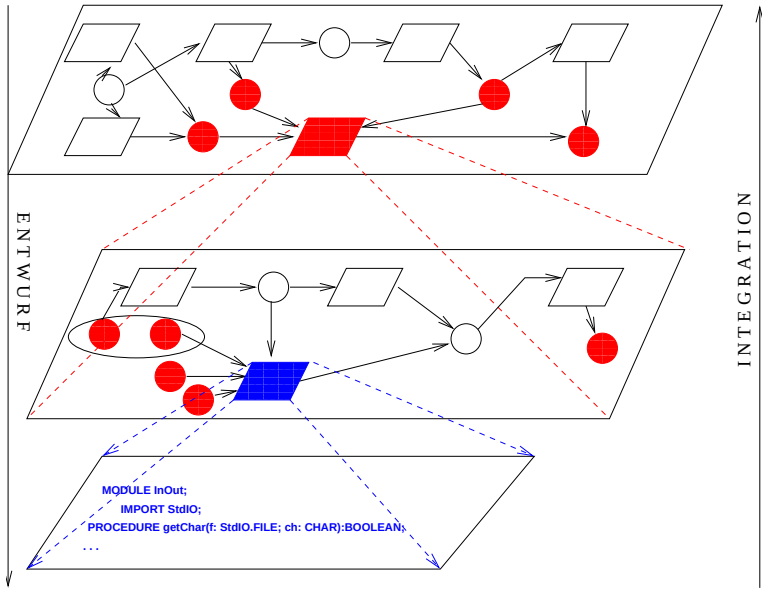


Abbildung 9.11: Dokumentationsstruktur

Zu dieser Hierarchie (Systemzerlegung) – 1. Dimension – und dem phasenweisen Vorgehen – 2. Dimension – sind zu der Grafik (Typ: RELation) jeweils weitere Beschreibungsaspekte (abhängig vom Ziel der Dokumentation) notwendig:

- Textuelle Ergänzungen zur Erfassung der fehlenden Semantik (Typ: TXT)
- Aussagen zu „Hintergründen“, zu wesentlichen Entscheidungen, „Begründungen“ (Typ: MOTivation)
- Aussagen zur Testvorstellungen resp. zum Test (Typ: TST)
- Prüfberichte (Typ: QS - hier aus Platzgründen weggelassen)
- Konfigurationsinformationen (Zustandsübergänge) (Typ: KFK)

Dies ist abhängig von den Randbedingungen und Zielsetzungen des Projekts: Wer macht die Pflege? Wie hoch sind die „Einsatzrisiken“? ...

Allgemeines Dokumentenmodell:

___Phase EBENE__	LB	GE	FE	IP	IG
System	REL, TXT TST, KONF	REL, TXT TST, MOT KONF			SRC, OBJ LNK, TST KONF
Produkt	REL,TXT TST, KONF	REL, TXT TST, MOT KONF			SRC, OBJ LNK, TST KONF
Komp.			SPEC, MOT TST, KONF	SRC, OBJ LNK, TST KONF	
Modul			SPEC, MOT TST, KONF	SRC, OBJ LNK, TST KONF	

Kürzel	Begriff / Erläuterung	Kürzel	Begriff / Erläuterung
LB	Leistungsbeschreibung	GE	Grobentwurf
FE	Feinentwurf	IP	Implementierung
IG	Integration	REL	Relation (Grafik)
TXT	textuelle Beschreibung	TST	Testaussagen
MOT	Hintergründe, Motivation	KONF	Konfigurationsdokument
SPEC	z.B. Pseudo-Code	SRC	Quellcode
OBJ	Objektcode	LNK	Lade-/Bindeanweisungen

Umsetzung im Rechner

↪ Hypertext (html, xml): Hierarchie über Links

↪ hierarchisches Dateisystem: Pfadnamen als Dokumentenbezeichner

KONF-Dokumente (eine Art *logfile*), z.B.: für Projekt / System mit Bezeichnung *ABC*

Dok.:	Zustand	Datum	Bericht
ABC/LB/rel.001	ROH	2.9.2000	
ABC/LB/text.001	ROH	2.9.2000	
ABC/LB/rel.001	UEB	12.9.2000	
ABC/LB/text.001	UEB	12.9.2000	
ABC/LB/rel.001	ANN	12.9.2000	ABC/LB/qs.rel.001
ABC/LB/text.001	ZUR	12.9.2000	ABC/LB/qs.text.001
ABC/LB/text.001	ROH	13.9.2000	ABC/LB/kmb.text.001
ABC/LB/rel.001	ANT	13.9.2000	ABC/LB/aend.rel.001
ABC/LB/rel.001	REV	14.9.2000	ABC/LB/kmb.rel.001
ABC/LB/rel.002	ROH	14.9.2000	
ABC/LB/rel.002	UEB	22.9.2000	
ABC/LB/text.001	ANN	24.9.2000	
ABC/LB/rel.002	ANN	24.9.2000	

9.4.5 Change-Management

NB: Dieser Begriff wird im betriebswirtschatlichen Kontext auch anders verwendet – Management betrieblicher Veränderungen!

Zustandsmodell für die Konfigurationsobjekte:

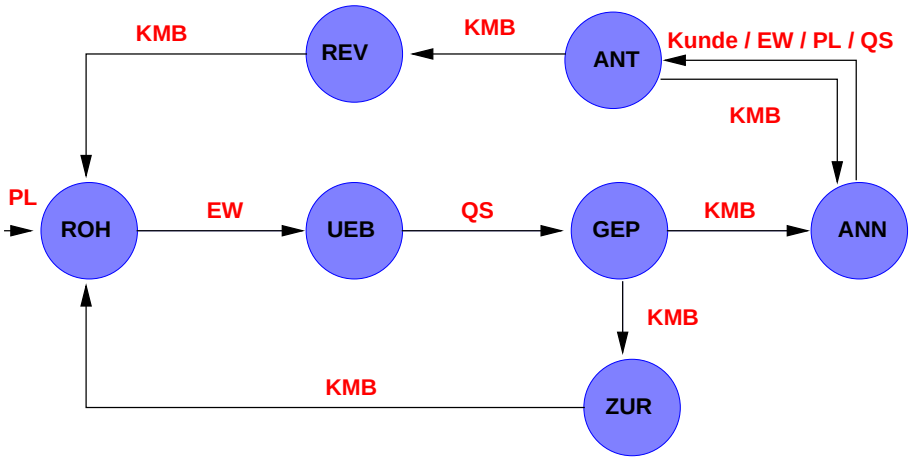


Abbildung 9.12: Zustandsmodell

ROH	in freier Bearbeitung	UEB	übergeben an QS, gesperrt
GEP	geprüft durch QS	ANN	angenommen u. „eingefroren“
ZUR	zurück zur Überarbeitung	ANT	Änderungsantrag liegt vor
REV	Revision (was, wie, was noch ändern)		

9.4.6 Release-Management

- "Release ist eine Beta, die aus dem Labor entkommen ist"
- oder: Jedes Software-Paket, das an den Kunden geliefert wird, ist ein Release
- Benennung und Archivierung notwendig für Wartung

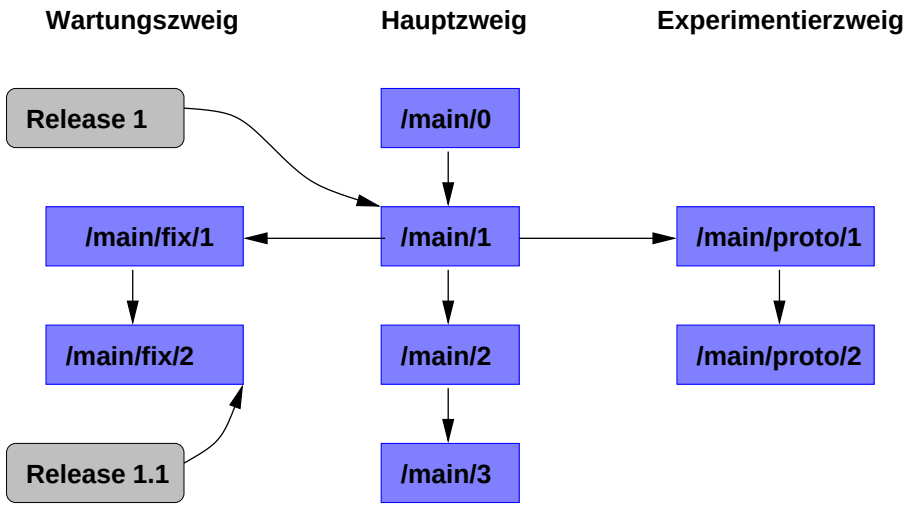


Abbildung 9.13: Release

9.4.7 Build-Management

viele (einzelne) Quellen, Bibliotheken u.dgl. → auf bestimmter Hardware in bestimmter Betriebssystem-Version lauffähige Software

- trivial: Übersetzung **aller** Quellen (Zeit, Ressourcen)
- besser: nur tatsächliche geänderte Teile übersetzen
  - notwendig: Beschreibung der Abhängigkeiten
    - Wen benutze ich?
- Aufrufe
- includes
- Vererbung
- ...
    - Wer benutzt mich?
- entscheidend für Build-Prozess
- schwieriger zu finden
  - Beispiele:
    - *ifdef*-Anweisungen (bedingte Übersetzung)
    - *make* (Unix, Windows)
    - *ant* (Java)

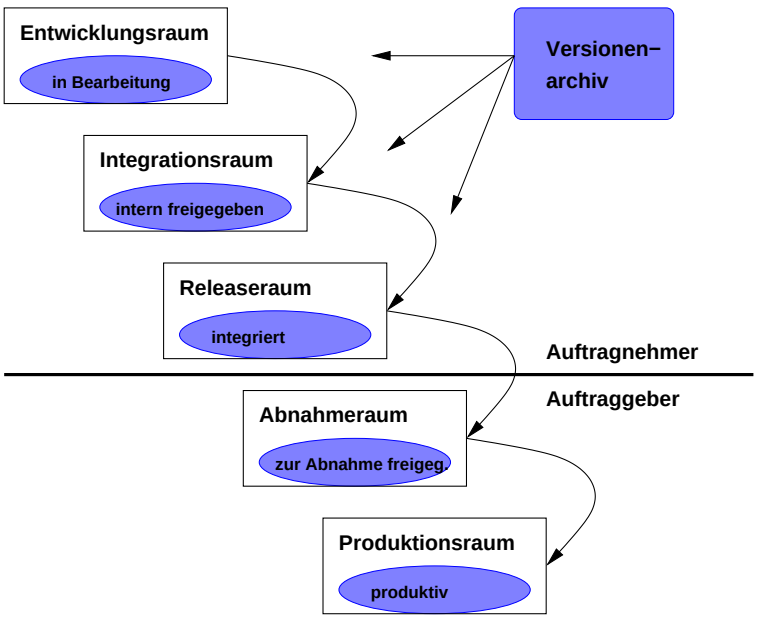


Abbildung 9.14: Raumkonzept

9.4.8 Konfigurationmanagement-Plan

siehe z.B. [Humphrey89], [IEEE83a]

1. Übersicht

   - Generelle Zielsetzung
   - Systemübersicht
2. Organisation
   - Verantwortlichkeiten
   - Zusammensetzung des Konfigurationsmanagementboards (KMB)
   - Entscheidungsregeln
   - Beziehung / Schnittstellen zur QS
3. Methoden
   - Strukturierungsprinzip (Items)
   - Zustandsmodell (Baselines)
   - Identifikationssystem (Nomenklatur)
   - Änderungskontrolle
   - Überwachung
   - Zustandsverfolgung
   - Werkzeuge
4. Prozeduren
   - manuelle Verfahren
   - Formulare, Aufzeichnungswege
5. Umsetzung
   - Personalplan
   - Mittel-/Ressourcenplan
   - Budget
   - Überwachung der Wirksamkeit

9.4.9 Tools

Übersichten:

- www.cmcrossroads.com/bradapp/links/scm-links.html
- www.cmcrossroads.com/yp/Tools/Commercial_CM_Tools/more2.php
- www.linuxmafia.com/faq/Apps/scm.html
- www.faqs.org/faqs/by-newsgroup/comp/comp.software.config-mgmt.html
- http://www.daveeaton.com/scm/CMTools.html

Prozess-basierte SCM Tools

- AccuRev – AccuRev/CM (www.accurev.com)

- Intasoft – Allchange (www.intasoft.net)
- Computer Associates – AllFusion Harvest Change Manager (www3.ca.com/Solutions/Product.asp?ID=255)
- Serena – Change Man (www.serena.com/Products/changeman/home.asp)
- Rational – ClearCase (www-306.ibm.com/software/rational/offerings/scm.html)
- ExpertWare – CMVision (www.cmvision.com)
- Telelogic – Synergy (www.telelogic.com/products/index.cfm)
- Softlab – Enabler (www.softlab.de/sixcms/detail.php?id=186&skip=&_nomail=&_selected_artikel_id=4522)
- Visible Systems – Razor (www.visible.com/Products/Razor/index.htm)
- McCabe&Associates – TRUExchange (www.mccabe.com/true.htm)
- Promergent – XStream (www.promergent.com/products/xstream.html)

Configuration Management and Version Control

- AccuRev – AccuRev/CM (www.accurev.com)
- Cybermation – Alchemist (www.cybermation.com/products/alchemist/)
- BitKeeper – BK/Pro (www.bitkeeper.com/Products.BK_Pro.html)
- Aldon – Lifecycle Manager (www.aldon.com/lm/index.html)
- Reliable Software – Code Co-op (www.relisoft.com/co_op/index.htm)
- Agile Software Corp. – agile cm (www.agile.com/solutions/cm/index.asp)
- Collabnet (/www.collab.net/products/enterprise_edition/collabnet_scm.html)
- Network Concepts, Inc. (www.nci-sw.com)
- ComponentSoftware – CS-RCS Pro, CS-RCS Basic (www.componentsoftware.com)
- Configuration Data Services, Inc. – ECMS (www.configdata.com/config_management.html)
- Quality Software Components – TeamCoherence (www.qsc.co.uk/teamcoherence.htm), Build-Management (www.qsc.co.uk/builder.htm)
- Sun Microsystems – JavaSafe (wwws.sun.com/software/sundev/index.html)
- JSSL – Librarian (www.winlib.com/software.htm)
- Tesseract – Lifecycle Manager (www.tesseract.co.za)
- British Aerospace – LifeSpan (www.lifespan.co.uk)
- Perforce Software – Perforce SCM System (www.perforce.com/perforce/products.html)
- Data Design Systems – PrimeCode (www.datadesign.com/nonstop/solutions/index.html)
- Synergex– PVCS (www.pvcs.synergex.com/Overview/Overview.asp)
- Quma Software, Inc. – QVCS (www.qumasoft.com/)

- Lucent Techn. – Sablime (www.bell-labs.com/project/sablime)
- MKS, Inc. – MKS (www.mks.com)
- Sourcegear – SourceOffSite (www.sourcegear.com)
- SiberLogic – SiberSafe (www.siberlogic.com)
- Microsoft – Visual SourceSafe (msdn.microsoft.com/vstudio/previous/ssafe)
- Interwoven – TeamSite (www.interwoven.com/products/content_management/index.html)
- Burton Systems Software – TLIB (www.burtonsys.com)
- Uni Software Plus – Voodoo Server f. MacIntosh (www.unisoft.co.at/products/voodooserver/index.html)

Public Domain Free Software

- Aegis: http://aegis.sourceforge.net/
- CERN - CMZ: http://wwwcmz.web.cern.ch/wwwcmz/
- CVS (Concurrent Versions System): https://www.cvshome.org/
- DVS (Distributed Versioning System): http://serl.cs.colorado.edu/ serl/cm/dvs.html
- Open Source Software Engineering: http://scm.tigris.org/
- jCVS (Java): http://www.jcvs.org/
- keep-it: http://www.keep-it.com/
- bonsai: http://bonsai.mozilla.org/cvsqueryform.cgi
- PRCS (Project Revision Control System): http://prcs.sourceforge.net/
- RCS (Revision Control System): http://www.gnu.org/software/rcs/rcs.html
- SCCS (Source Code Control System): https://www.cvshome.org/cyclic/cyclic-pages/sccs.html
- **Subversion**: http://svnbook.red-bean.com/svnbook/index.html, http://subversion.tigris.org/project_packages.html

9.5 Entwicklungsmodelle / Ablauforganisation

9.5.1 Einführung

Zentrale Frage:

> Wie geht man vor, um **effizient und effektiv** Software zur Lösung einer gestellten Aufgabe zu entwickeln?

Analoge Fragestellung:

> Wie gehen andere Ingenieur-Disziplinen vor?

Technische Artefakte (Geräte, Maschinen, ...)

- haben ein gewisse Lebensdauer:
  Idee → Konzepte → Konstruktionspläne → Realisierung → Gebrauch →Wartung → Ausmusterung
- werden in einer Reihe von Schritten / durch eine Folge von Aktivitäten entwickelt:
  Machbarkeitsstudie → Entwurf → Realisierung von Komponenten → Zusammenbau → Abnahmeprüfung

Software-**Engineering**: Vorgehensweise zur Entwicklung von Software orientiert sich an der Vorgehensweisen zur Entwicklung von technischen Artefakten

Software Life Cycle (nach IEEE-Std.)

> The period of time that begins when a software product is conceived and ends when the software is no longer available for use. The software life cycle typically includes a concept phase, requirements phase, design phase, implementation phase, test phase, installation and checkout phase, opertation and maintenance phase, and, sometimes, retirement phase.

Software Development Cycle (nach IEEE-Std.)

> The period of time that begins with the decicion to develop a software product and ends with when the software is delivered, This cycle typically includes a requirements phase, design phase, implementation phase, test phase, and, sometimes, installation and checkout phase.

Phasenmodelle vs. Prozessmodelle:

- **Phasenmodell:** Folge (sequentiell und parallel) von Entwicklungsstufen (Entwicklungsphasen, -stadien, Modelle unterschiedlicher Abstraktionsniveaus) bis hin zum fertigen Produkt:

$$m_1; m_2; \ldots$$

- **Prozessmodell:** Folge von Aktivitäten (sequentiell und parallel), die die Anforderungen in das Endprodukt transformieren

$$m_1 \rightarrow m_2 \rightarrow \ldots$$

NB:

- In Prozessmodellen sind meist Aspekte von Phasenmodellen (Definition von Ergebnissen einzelner Aktivitäten) enthalten und umgekehrt!
- reine Phasenmodelle sind ergebnisorientiert (dokumentationsorientiert), reine Prozessmodelle sind aktivitätsorientiert

Lineare Vorgehensmodelle:

- Sequentielles Durchlaufen definierter (verschiedener) Aktivitäten
- Sequentielles Erstellen immer konkreter werdender Modelle (Abstraktionsstufen)

Inkrementelle Vorgehensmodelle

- stufenweise Realisierung: vom Kern über Ausbaustufen zur Gesamtlösung

Iterative Modelle

- Wiederholtes Durchlaufen linearer Modelle

Praxis: meist Mischformen

Wesentlich bei der **Festlegung eines Phasenmodells**:

Anpassung an die gestellte Aufgabe und die Randbedingungen sowie die Definition jeder Phase über das zu erreichende Ziel:

- WAS ist WARUM von WEM in der jeweiligen Phase zu tun, WAS WARUM nicht?
- WAS muss das Ergebnisdokument beinhalten, WAS soll es nicht enthalten?
- WIE ist in der jeweiligen Phase vorzugehen, welche Techniken sind einzusetzen, um das gesetzte Ziel (das WAS) zu erreichen?
- Wie kann die Zielerreichung überprüft („gemessen“) werden?

Jede **Phase** sollte weiterhin definiert sein durch:

- Startkriterien, die erfüllt sein müssen, um die Phase sinnvoll beginnen zu können (vorausgesetzte Informationen, Methoden, Werkzeuge, geschulte Mitarbeiter, . . .),
- Endekriterien, die erfüllt sein müssen, um die Zielerreichung feststellen zu können
- Qualitätskriterien (Checkliste) als Vorgabe und Prüfgrundlage

9.5.2 Ein allgemeines, lineares Phasenmodell

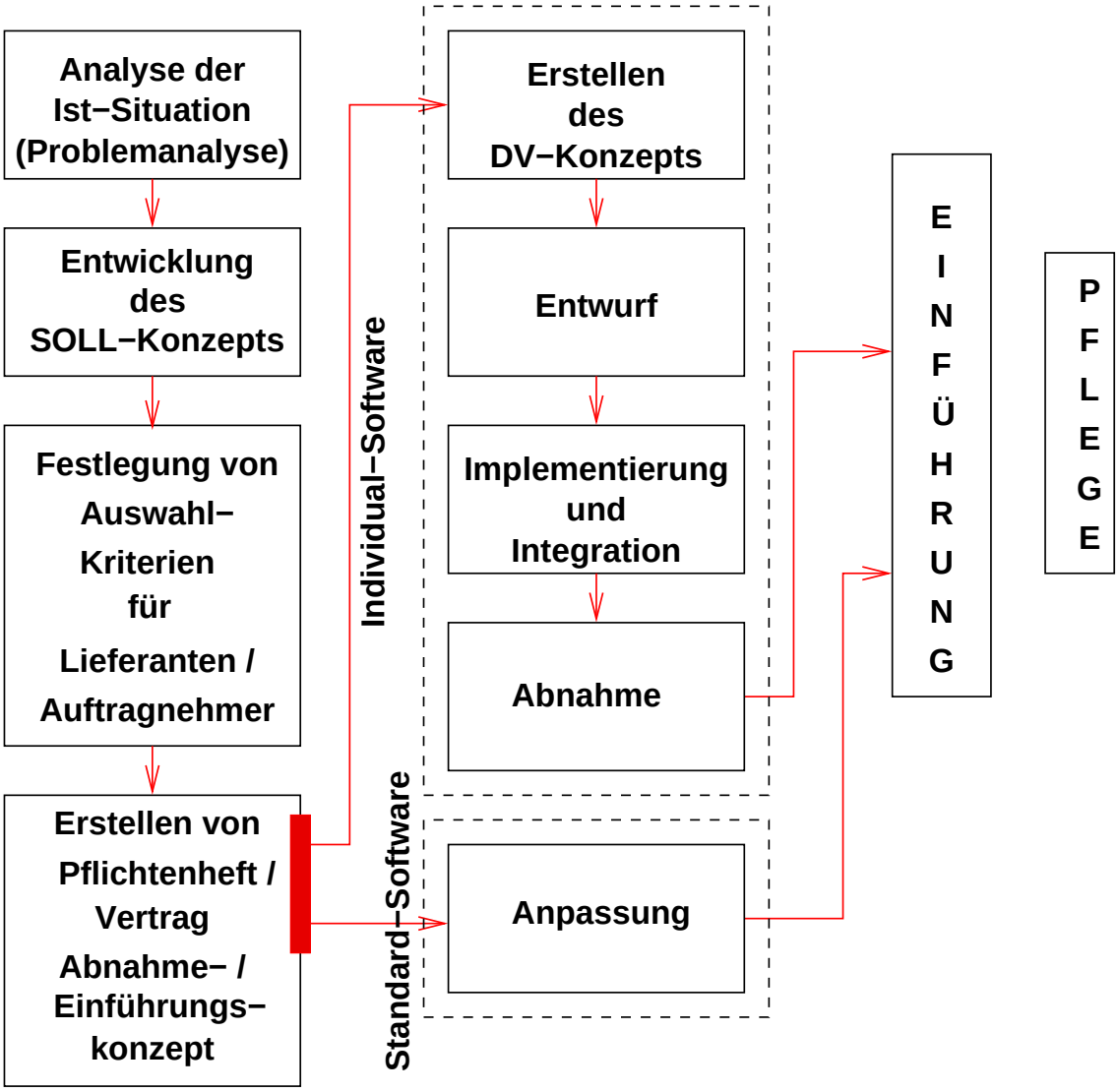


Abbildung 9.15: Allgemeines, lineares Phasenmodell

9.5.3 Das Wasserfall-Modell nach Boehm

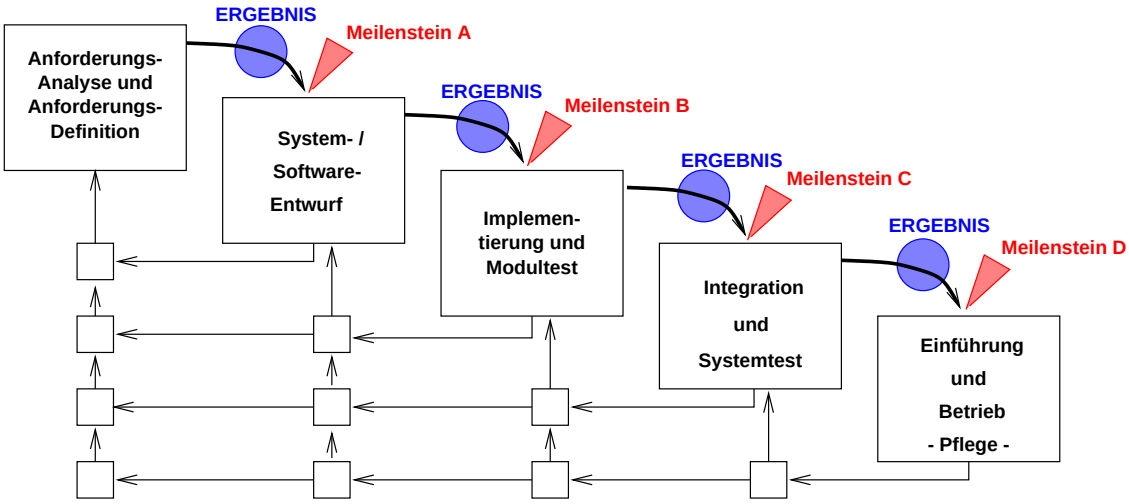


Abbildung 9.16: Boehmsches Wasserfallmodell

- definiert strikt voneinander getrennte Entwicklungsschritte (Phasen)
- diese müssen sukzessive durchlaufen werden („Wasserfall“).
- erst wenn eine Phase „vollständig“ abgeschlossen ist, kann nächste Phase begonnen werden
- werden in einer Phase Fehler entdeckt oder kommen von „außen“ Änderungsanforderungen, so muss in der Phase, in der die Ursache des Fehlers liegt oder die Änderungsanforderung Relevanz hat, erneut begonnen werden und die darauf folgenden Phasen wieder sukzessive durchlaufen werden
- ist im Prinzip „dokumentationsorientiert“, d. h. jede Phase hat ein in Form eines oder mehrerer Dokumente fixiertes Ergebnis, auf dem die nächste Phase definiert aufsetzen muss
- einzelne Phasen können inhaltlich weiter untergliedert werden
- Technik des Prototyping kann integriert werden

Vorteile

- Transparente Strukturierung des Entwicklungsprozesses und damit
- gute Basis für eine erfolgreiche Projektsteuerung und Projektkontrolle (definierte Meilensteine)
- Bei Auftragsentwicklung kann dem Geldfluss ein definierter, prüfbarer Projektfortschritt gegenübergestellt werden.
- Einarbeitung neuer Mitarbeiter leichter möglich, da definierte Einstiegspunkte vorhanden.
- Die inhaltlich definierten Meilensteine bieten **sauberen Ansatzpunkt für QM**: Reviews, Inspektionen oder andere Prüftechniken

Begriffe: (z.B. nach IEEE-Std. 1028)

- **Management-Reviews**: Vergleich Projektstatus mit Projektplan
- **technische Reviews**: Prüfung und Bewertung eines Software-Elements (Entwicklungs- (Teil-) Ergebnis)
- **Software-Inspektionen**: Im Vordergrund steht Suche nach Fehlern in einem Software-Element
- **Walkthrough**: Gruppe geht zusammen mit Autor ein Software-Element Schritt für Schritt durch (z.B. anhand von Testfällen), sammelt Fehler, Änderungs- und Verbesserungsvorschläge
- **Audits**: Überprüfung der Einhaltung von Vorgaben, Standards und Richtlinien

Manchmal auch so definiert:

- Inspektion:
  Eine statische Analysetechnik, die in der visuellen Untersuchung von Entwicklungsprodukten besteht, mit dem Ziel, Fehler, Verletzungen von Entwicklungsstandards oder sonstige Probleme aufzudecken. z.B. Code-Inspektion, Design-Inspektion.
- Review:
  Ein Prozeß oder eine Sitzung, während dessen ein Arbeitsergebnis oder eine Gruppe von Arbeitsergebnissen Projektmitgliedern, Managern, Anwendern, Kunden oder anderen interessierten Parteien zum Zweck der **Kommentierung, Abstimmung oder Zustimmung** vorgelegt wird.

Potenzielle Probleme beim strikten Wasserfallmodell:

- Verlangt flexible und vor allem transparente Dokumentationsstrukturen, da die „Rückwärtspfeile“ in der Praxis häufig durchlaufen werden (müssen).
- Verlangt Entwickler, die sich mehr als Ingenieur denn als Programmierkünstler sehen.
- Verlangt intensive Zusammenarbeit zwischen Nutzern („Problemkennern“) und Entwicklern in der Startphase.
- Der größte Aufwand liegt in der Startphase und es gibt bei den Anwendern in der Praxis wenige, die hierfür so viel investieren wollen.
- Durch die lange Dauer der Startphase besteht durch die „Dynamik“ der Anforderungen die Gefahr, dass die realen und die definierten Anforderungen divergieren.

Schwächen im klassischen Wasserfallmodell:

- Projektmanagement und Konfigurationsmanagement nicht erfasst!
- Testprozess ist an das Ende verlagert
- Im ursprünglichen Modell sind auch QS-Maßnahmen nicht enthalten

---

Wichtige Begriffe: **Lastenheft** und **Pflichtenheft**

Die Aufgabenbeschreibung, d.h. die Definition der Sollanforderungen, wird als **Lastenheft** des Auftraggebers bezeichnet. Der daraufhin durch den Auftragnehmer entwickelte Lösungsansatz wird in einer Leistungs- beschreibung zusammengefasst, die als **Pflichtenheft** bezeichnet wird.

9.5.4 Boehmsches V-Modell

Modifiziertes Modell nach B. Boehm in **V**-Form: Kopplung von Entwicklungs- und QS-Aktivitäten

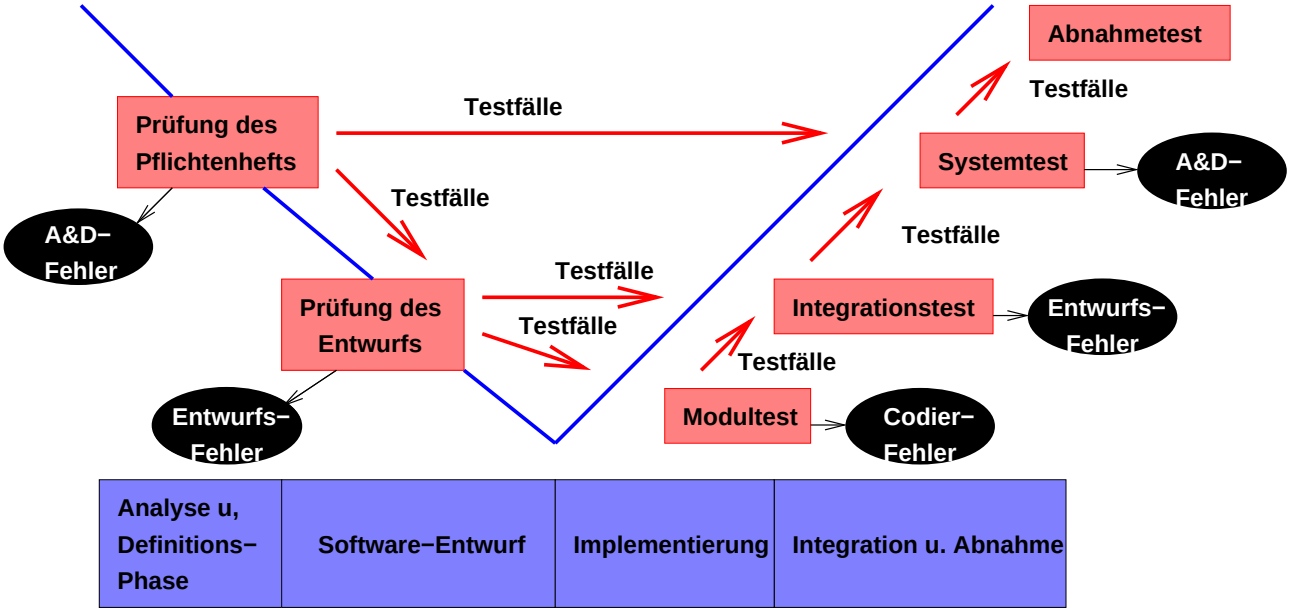


Abbildung 9.17: Boehmsches V-Modell

Validations- (bauen wir das richtige Produkt?) und **Verifikations-** (bauen wir das Produkt richtig?) **getrieben**

Testfälle leiten sich aus Spezifikationen (und anderen Quellen, später mehr) ab:

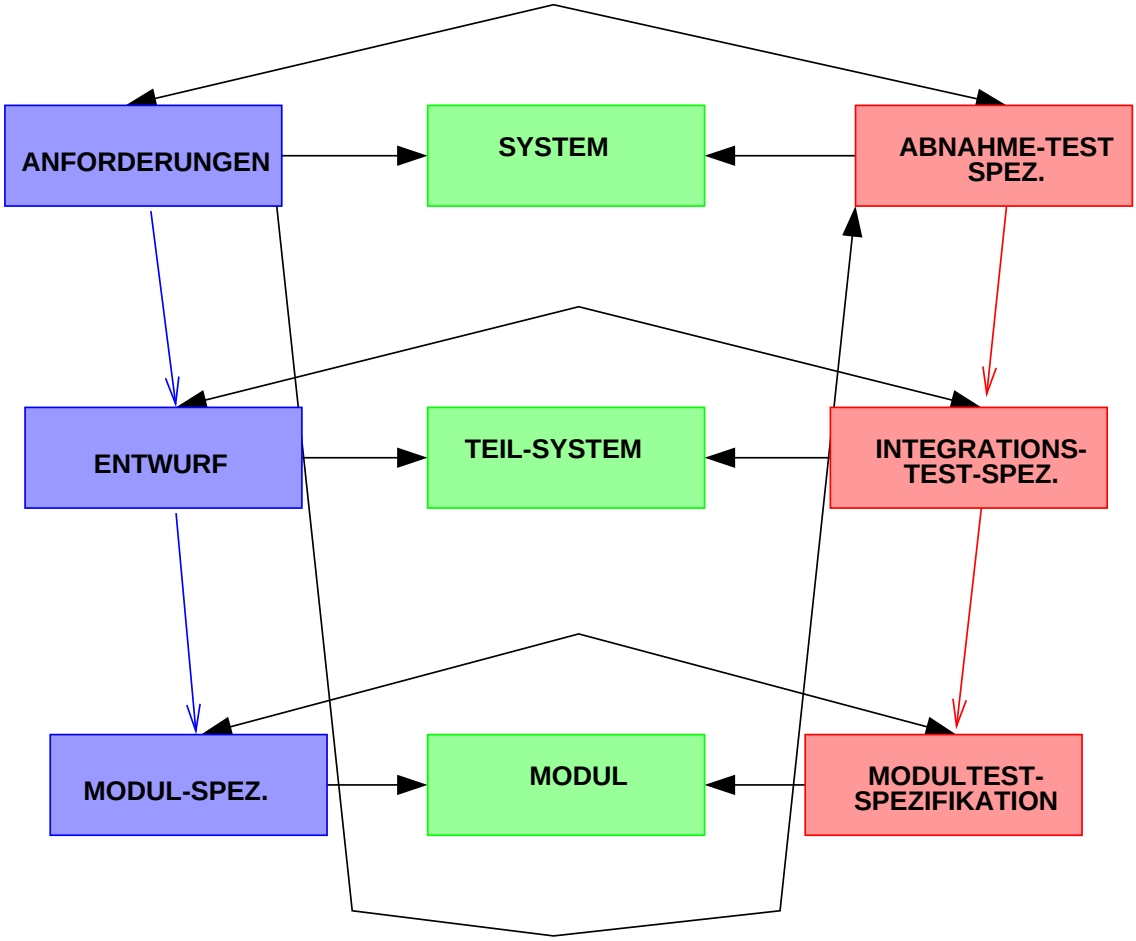


Abbildung 9.18: Testen in Entwicklung integriert

Mit integriertem Testprozess:

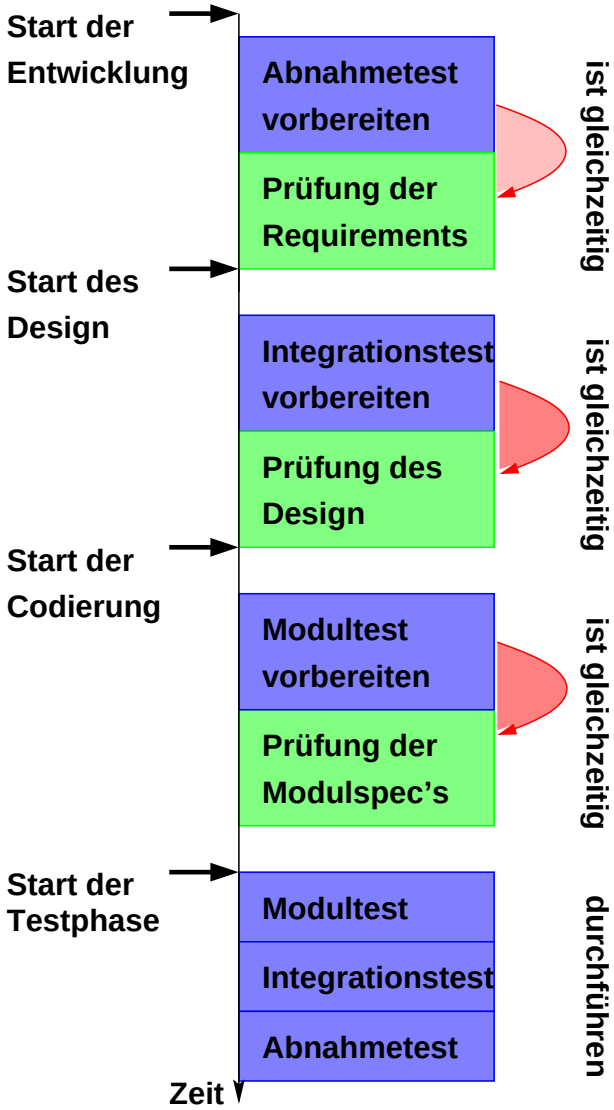


Abbildung 9.19: Integrierter Testprozess

9.5.5 Das V-Modell '97

9.5.5.1 Übersicht

Vorgehensmodell des Bundes(www.v-modell.iabg.de)

- Weiterentwicklung des Boehm'schen V-Modells
- Legt Produkte **und** Aktivitäten des System- / Software-Entwicklungsprozesses fest
- Allgemein gehaltenes Modell, das an Firmen- wie Projekt-Spezifika angepasst werden kann (*Tayloring*)
- **Standard** für Software-Projekte des Bundes

Ziele der Standardisierung:

- Bessere Planbarkeit von IT-Projekten
- Reduktion der Kosten im gesamten Lebenszyklus
- Verbunden mit einer Qualitätsverbesserung
- Grundlegende Verbesserung der Kommunikation zwischen Auftraggeber und Auftragnehmer

Ebenen der Standardisierung:

- Das Vorgehensmodell:
  - „Das Vorgehensmodell beschreibt die Aktivitäten (Tätigkeiten) und Produkte (Ergebnisse), die während der Entwicklung von Software durchzuführen bzw. zu erstellen sind."
  - „Das Vorgehensmodell ist neben dem militärischen Bereich auch für den gesamten Bereich der **Bundesverwaltung** verbindlich und wird von sehr vielen Industriefirmen als Hausstandard zur Softwareentwicklung verwendet."
  - „Das Vorgehensmodell ist ein Prozessmodell, mit dessen Hilfe Projekte gemäß der Norm **ISO 9001** abgewickelt werden können."
- Die Methodenzuordnung
  - „Die Methodenzuordnung legt fest, mit welchen Methoden die Aktivitäten des Vorgehensmodells durchzuführen und welche Darstellungsmittel in den Ergebnissen zu verwenden sind"
- Die Werkzeuganforderungen
  - „Die Funktionalen Werkzeuganforderungen legen fest, welche **funktionalen Eigenschaften** diejenigen Software-Werkzeuge ("tools") aufweisen müssen, die bei der Entwicklung von Software eingesetzt werden sollen."

Die Teilmodelle

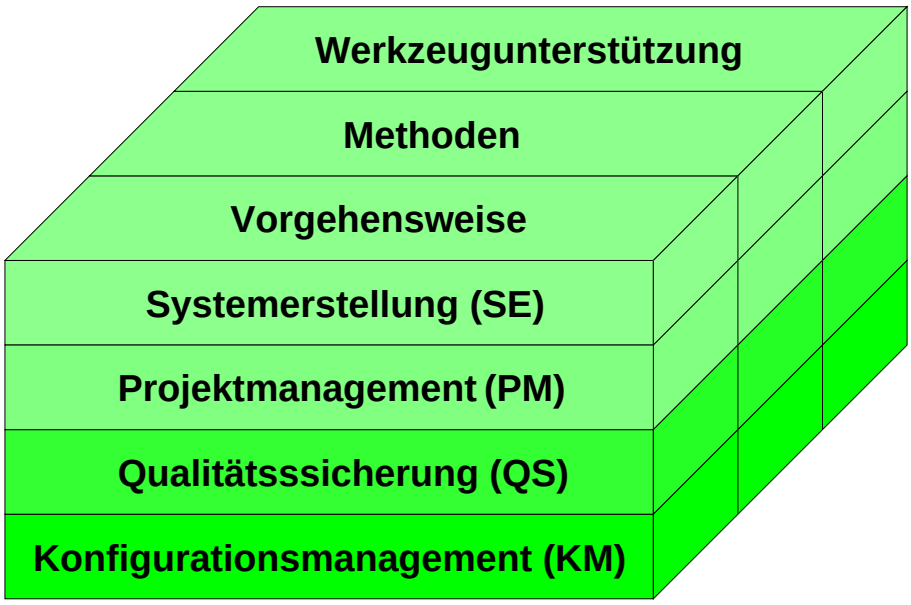


Abbildung 9.20: Teilmodelle des V-Modells

Zusammenspiel der Teilmodelle

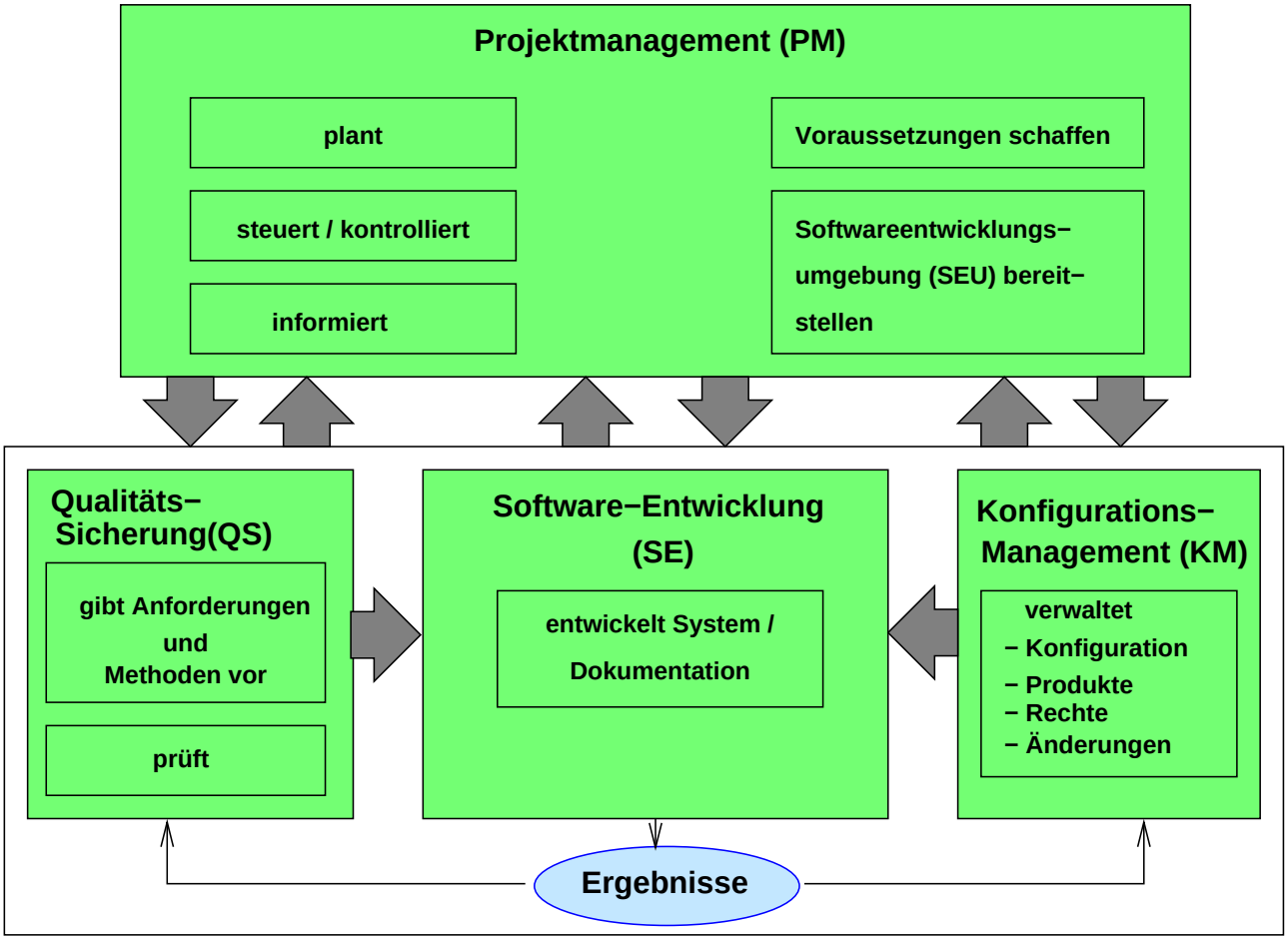


Abbildung 9.21: Zusammenspiel der Teilmodelle

Aktivitäten und Produkte

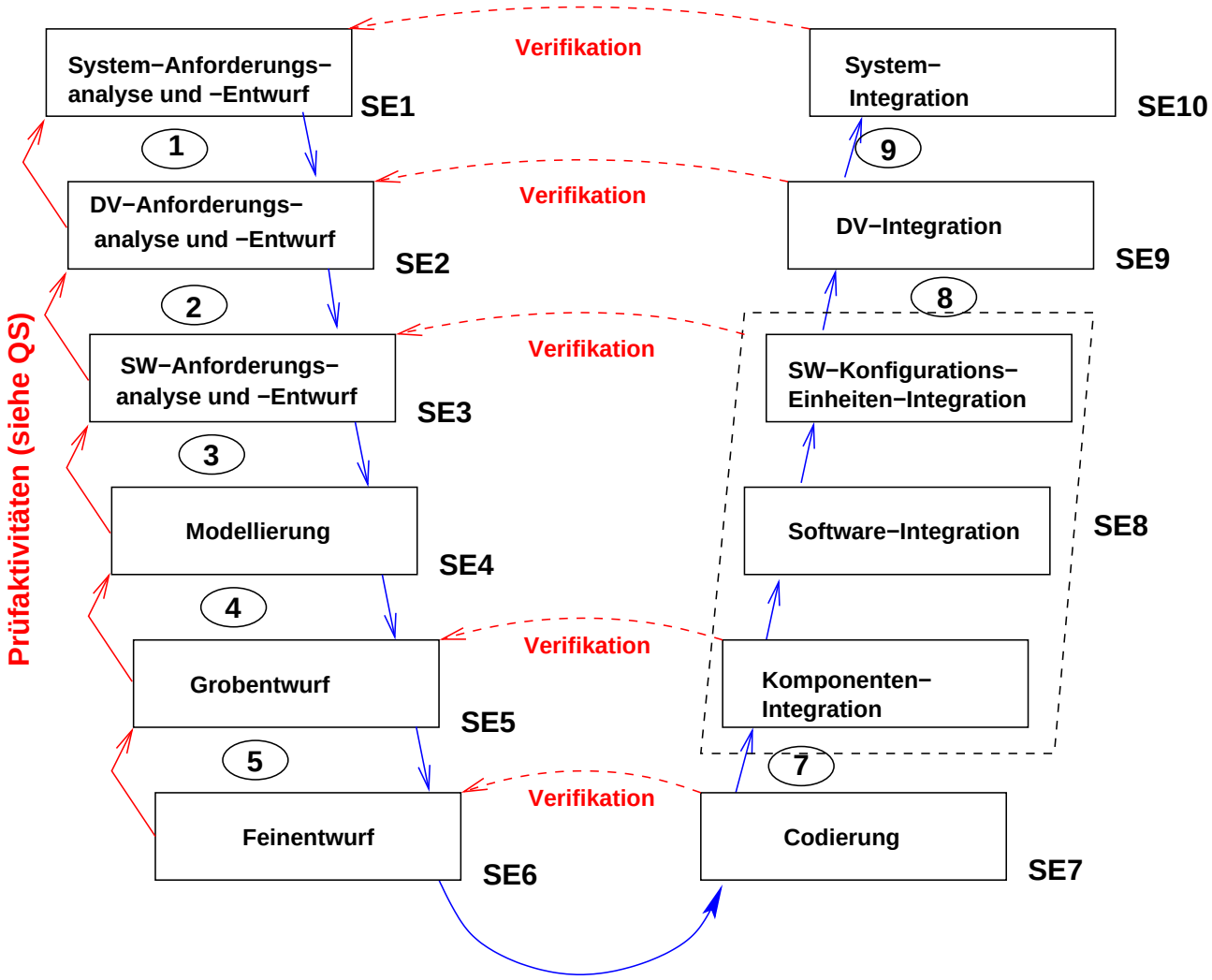


Abbildung 9.22: Aktivitäten und Produkte

	Tätigkeit		Ergebnisse
SE1	Beschreibung der Anforderungen an das zu erstellende System und seine Umgebung und Strukturierung des Systems in seine DV- und Nicht-DV-Segmente	1	Systemanforderungen Systementwurf Systemintegrationsplan
SE2	Beschreibung der Anforderungen an ein DV-Segment und seine Umgebung und Strukturierung des Segments in seine SW- und HW-Konfigurationseinheiten	2	DV-Anforderungen DV-Entwurf DV-Integrationsplan
SE3	Beschreibung der Anforderungen an eine SW-Konfigurationseinheit und ihre Umgebung	3	SW-Anforderungen
SE4	Strukturierung, Beschreibung des Zusammenspiels aller Funktionen und Daten hinsichtlich Zweck, **Kritikalität** und Struktur, Inhalt und Beziehungen (Daten)	4	Modelldokumente
SE5	Strukturierung der SW-Konfigurationseinheiten hinsichtlich notwendiger/möglicher Parallelverarbeitung, Zerlegung in SW-Komponenten und -Module und Spezifikation des Zusammenspiels / der Schnittstellen von Komponenten und Modulen	5	SW-Architektur Schnittstellenentwurf SW-Konfigurationseinheiten, Integrationsplan
SE6	Beschreibung der Komponenten und Module hinsichtlich der softwaretechnischen Realisierung ihrer Funktionen, der Datenhaltung und Fehlerbehand-	6	Datenkatalog SW-Entwurf

Definierte Produkt-Zustände (KFM)

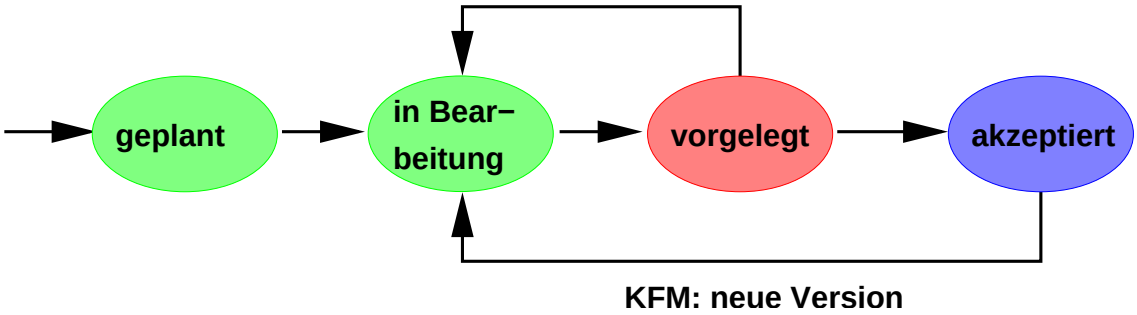


Abbildung 9.23: Dokumentzustände im V-Modell

- geplant: in der Planung enthalten
- in Bearbeitung: unter Kontrolle des Entwicklers
- **vorgelegt**
  - aus Entwicklersicht fertig
  - dem Konfigurationsmanagement unterworfen
  - einer Qualitätsprüfung vorgelegt
- **akzeptiert**
  - von der QS überprüft und freigegeben
  - jede Änderung führt zu einer neuen Version

Zuordnungstabellen für das Submodell QS

Aktivität QS 1.1: QS-Plan erstellen

Aktivität QS 1.2: Prüfplan erstellen

Produkt: Prüfplan	
2 Prüfgegenstände und Qualifikationserfordernisse	
3 Aufgaben und Verantwortlichkeiten	
4 Zeitplan	(BALK / NPT)

Aktivität QS 2.1: Prüfmethoden und -kriterien festlegen

Aktivität QS 2.2: Prüfumgebung definieren

Aktivität QS 2.3: Prüffälle festlegen

Produkt: Prüfspezifikation	
5 Prüffälle	
5.1 Prüffallbeschreibung	
- für System, Segment, SW-Einheit	BBTE + FMEA
- für SW-Komponente, SW-Modul, Datenbank	BBTE + WBTE
- für Betriebsinformationen	BBTE
5.2 Abdeckungsmatrix	

Aktivität QS 2.4: Prüfprozedur erstellen

Aktivität QS 3: Prozessprüfung von Aktivitäten (AUD)

Aktivität QS 4.1: Prüfbarkeit feststellen

Prüfgegenstand	
alle V-Modell-Produkte auÃ§er:	
- Projektplan,	
- Projekthandbuch,	REV + STAT
- Projekthistorie,	
- QS-Plan	

Aktivität QS 4.2: Produkt inhaltlich prüfen

Prüfgegenstand	
Anwenderforderungen	REV + T + STAT + SIMU
Technische Anforderungen	REV + T + STAT + SIMU
Systemarchitektur	REV + T + STAT
SW-Architektur	REV + T + STAT
SW-Entwurf	REV + T + STAT
Implementierungsdokumente	REV + STAT
SW-Modul, Datenbank, SW-Komponente, SW-Einheit, Segment, System	T + SIMU
Integrationsplan	REV
Betriebsinformationen (Anwendungshandbuch, Diagnosehandbuch, ...)	REV + T
Datenkatalog	REV + STAT
Schnittstellenübersicht	REV + STAT
Schnittstellenbeschreibung	REV + STAT
Prüfplan	REV
Prüfspezifikation	REV
Prüfprozedur	REV + T
KM-Plan	REV
Konfigurations-Identifikationsdokument (System-, SW-, HW-)	REV
Änderungsauftrag	REV
Änderungsmitteilung	REV

Aktivität QS 5: QS-Berichtswesen

Erläuterungen:

- Die Methode **BALK** wird in der Aktivität QS 1.2 im Sinne eines Terminplans und im Sinne eines Ressourcenplans verwendet. In beiden Fällen ist die Methode BALK für die Erfüllung der Produktanforderungen ausreichend.
- Die "Netzplan-Technik" (**NPT**) ist ein auf der Graphentheorie basierendes Verfahren zur Analyse, Beschreibung, Strukturierung, Planung, Kontrolle und Steuerung von Projekten und Abläufen, wobei Zeit, Kosten, Einsatzmittel und weitere Einflussgrößen berücksichtigt werden können.
- **BBTE** (BlackBoxTestfallEntwurf) in Aktivität QS 2.3 "Prüffälle festlegen"

  Mittels der Methode BBTE werden Testfälle für die folgenden Prüfgegenstände festgelegt: SW-Modul, Datenbank, SW-Komponente, SW-Einheit, Segment, System, Informationen zum Anwendungshandbuch, zum Diagnosehandbuch, zum Betriebshandbuch.

  Testfälle für die Betriebsinformationen sind in der Regel Funktionsabdeckungstestfälle. Mit den Testfällen sollen Abweichungen der beschriebenen Betriebsinformationen vom tatsächlich realisierten System aufgedeckt werden.

  Für die PrüfgegenstÃd'nde SW-Komponente, SW-Modul und Datenbank wird das Teilprodukt "Prüffallbeschreibung" nur durch die ergänzende Anwendung der Methode **WBTE**

(WhiteBox-Testfall-Entwurf) abgedeckt. Für SW-Einheit, Segment und das System wird das Teilprodukt "Prüffall- beschreibung" nur durch die ergänzende Anwendung der Methode **FMEA** (im Falle hoher Zuverlässigkeitsanforderungen) vollständig abgedeckt.

- **FMEA** in Aktivität QS 2.3 "Prüffälle festlegen"
  Mittels der Methode **FMEA** (Fehler-Möglichkeiten- und Einfluss-Analyse) werden im Falle hoher Anforderungen an die Zuverlässigkeit Testfälle für die folgenden Prüfgegenstände festgelegt: SW-Einheit/HW-Einheit, Segment, System. Auf der Segment- und Systemebene sind neben Funktionsaspekten auch Umweltaspekte zu berücksichtigen.

  Das Teilprodukt "Prüffallbeschreibung" wird nur durch die ergänzende Anwendung der Methode **BBTE** vollständig abgedeckt.

- **AUD** (Audit) in Aktivität QS 3 "Prozessprüfung von Aktivitäten"
  Die Prüfung von Aktivitäten ist anhand spezifisch festgelegter Prüfkriterien durchzuführen. Auch wenn nur schwerpunktmäßig Aktivitäten geprüft und bewertet werden, so ist vom Prozess als Ganzem auszugehen. Auf Verbesserungsmöglichkeiten ist hinzuweisen. Einzuleitende Maßnahmen sind vorzuschlagen. Die Methode AUD deckt die Aktivität QS 3 "Prozessprüfung von Aktivitäten" vollständig ab.

- **REV** in Aktivität QS 4.1 "Prüfbarkeit feststellen"

  Die Methode REV wird zur Feststellung der Prüfbarkeit sämtlicher Prüfgegenstände angewendet. Die Prüfung ist anhand spezifisch festgelegter Prüfkriterien durchzuführen.

  Die Methode deckt die Feststellung der Prüfbarkeit vollständig ab. Allerdings empfiehlt sich zur Reduzierung des Aufwandes im Falle von Prüfgegenständen, die nach einem vorgegebenen Formalismus aufgebaut sind, zusätzlich der Einsatz der Methode **STAT**. Die Menge der zu berücksichtigenden Aspekte wird dann wesentlich geringer ausfallen.

- **STAT** in Aktivität QS 4.1 "Prüfbarkeit feststellen"

  Die Methode STAT wird zur Feststellung der Prüfbarkeit von Prüfgegenständen angewendet, die nach einem vorgegebenen Formalismus aufgebaut sind. Sie deckt die Feststellung der Prüfbarkeit nur in Kombination mit der Methode REV vollständig ab. Allerdings wird bei vorausgegangener statischer Analyse die Menge der in einem Review zu berücksichtigenden Aspekte wesentlich geringer ausfallen.

- Ziel des "Testens" (**T**) ist das Aufdecken von Fehlern sowie der Nachweis der Erfüllung spezifizierter Anforderungen.

- Ziel einer Simulation (**SIMU** ist die Bewertung der Realisierbarkeit von Anforderungen durch Sichtbarmachen der Auswirkungen auf Leistungsparameter und Systemverhalten unter dynamischen Aspekten.

  Die dynamischen Auswirkungen werden durch Einspielen eines operationellen Szenarios oder durch eine Folge von Ereignissen in das Modell erzeugt bzw. geschätzt. Der Einsatz der Simulationsmethode ist insbesondere zweckmäßig zur Bewertung folgender Eigenschaften:

  - Erfüllung der Qualitätsanforderungen
  - Antwortverhalten für spezifische Eingabedaten
  - CPU-Nutzung
  - Speichernutzung/-Kapazität
  - Erfüllung von Bedienungs-/Einsatzzeitzwängen
  - Mensch/Maschine-Zusammenspiel und Antwortverhalten

Anm. zu FMEA:

Ziel und Zweck

Die “Failure Mode Effect Analysis” (**FMEA**) ist eine Methode zur Identifikation von potentiellen Fehlerarten zwecks Bestimmung ihrer Auswirkungen auf den Betrachtungsgegenstand (System, Segment, SW-/HW-Einheit) sowie zur Klassifizierung der Fehlerarten hinsichtlich **Kritikalität** oder Hartnäckigkeit.

Es sollen Fehler vermieden und damit Entwurfsschwächen aufgedeckt werden, die beim Auftreten eine Gefährdung oder Verlust des Systems/Software und/oder eine Gefährdung der damit im Zusammenhang stehenden Personen bewirken würden.

Die Methode FMEA soll ferner Ergebnisse für korrektive Maßnahmen liefern sowie zur Bestimmung von Testfällen und Bedienungs- und Einsatzzwängen des Systems/Software dienen.

Funktioneller Ablauf von FMEA:

Das Grundprinzip besteht darin, dass in der Funktionshierarchie und in der Programmlogik systematisch (funktional und zeitlich) nach definierten Erfolgs- oder Fehlerkriterien gefragt wird: **was passiert, wenn?** Diese Analyse und Auswertung ist für alle Betriebsphasen und Bedienungsmöglichkeiten durchzuführen.

Der Ablauf der Methode FMEA besteht aus folgenden Hauptschritten:

- FMEA-Planung zur Identifikation der FMEA-Ziele und -Ebenen
- Festlegung spezifischer Verfahren, Grundregeln und Kriterien für die Durchführung der FMEA
- Analyse des Systems hinsichtlich Funktionen, Schnittstellen, Betriebsphasen, Bedienungsarten und Umwelt
- Entwurf und Analyse von Funktions- und Zuverlässigkeits-Blockdiagrammen oder Fehlerbaum-Diagrammen zur Darstellung der Abläufe, Zusammenhänge und Abhängigkeiten
- Identifikation von potentiellen Fehlerarten
- Bewertung und Klassifizierung der Fehlerarten und ihrer Folgen
- Identifikation von Maßnahmen zur Fehlervermeidung und Fehlerkontrolle
- Bewertung der Auswirkungen von vorgeschlagenen Maßnahmen
- Dokumentation der Ergebnisse

Sicherheit und Kritikalität

(aus "Allgemeinem Umdruck AU 250: Teil 3 Handbuchsammlung
`www.informatik.uni-bremen.de/uniform/gdpa/part3_d/p3si.htm`)

Behandlung der Sicherheit

Kriterienkataloge zur Sicherheit fassen jeweils Mengen von Prüfkriterien an den Sicherheitsanteil eines Systems in Abhängigkeit der **sicherheitsrelevanten Einstufung** zusammen. Diese Einstufung bestimmt die **Evaluierung** unter den Aspekten Wirksamkeit und Korrektheit.

Aus Sicht des Entwicklungsprozesses sind daraus abzuleiten:

- Anforderungen an die Existenz bestimmter Produkte, deren Inhalt und Eigenschaften
- Anforderungen an die Prüfung dieser Produkte (Art, Umfang, Tiefe, ...)
- Anforderungen an Methoden/Werkzeuge der Entwicklung
- Anforderungen an die Rollenverteilungen und die damit verbundenen organisatorischen Maßnahmen

Behandlung der Kritikalität

Unter „Kritikalität" wird im V-Modell die **Bedeutung** verstanden, die einem **Fehlverhalten** einer Betrachtungseinheit beigemessen wird. Es gibt sowohl physikalische Betrachtungseinheiten (die Produkte System, Segment, SW-Einheit/HW-Einheit, SW-Komponente, SW-Modul, Datenbank) als auch logische Betrachtungseinheiten (Funktionen: Systemfunktion, Segmentfunktion, Einheiten-Funktion).

Zur **Einstufung** der Kritikalität (**im Submodell SE**) ist nicht allein die physikalische Betrachtungseinheit, sondern auch die von ihr berührte Umgebung (Auswirkungen des Fehlverhaltens) zu berücksichtigen.

Kritikalitätsstufen: allgemeine Festlegung für technische Systeme

Kritikalität	Auswirkungen des Fehlverhaltens
hoch	Fehlverhalten kann zum Verlust von Menschenleben führen
mittel	Fehlverhalten kann die Gesundheit von Menschen gefährden oder zur Zerstörung von Sachgütern führen
niedrig	Fehlverhalten kann zur Beschädigung von Sachgütern führen, ohne jedoch Menschen zu gefährden
keine	Fehlverhalten gefährdet weder die Gesundheit von Menschen noch werden Sachgüter beschädigt

Bei Software muss die Festlegung der Kritikalität von deren **Einsatzzweck** abhängig gemacht werden. Sie soll, ebenso wie die Festlegung der Anzahl der Stufen, immer **projektspezifisch** durch eine Abschätzung der direkten und indirekten Auswirkungen eines mÃűglichen Fehlverhaltens erfolgen.

Kritikalitätsstufen: allgemeine Festlegung für administrative Systeme

Kritikalität	Auswirkungen des Fehlverhaltens
hoch	Fehlverhalten macht sensitive Daten unberechtigten Personen zugänglich, verhindert administrative Vorgänge (Gehaltsauszahlung, Mittelzuweisung, ...) oder führt zu gravierenden Fehlentscheidungen infolge fehlerhafter Daten.
niedrig	Fehlverhalten, das zum Ausfall von Plandaten, zu Abflugverzögerungen, ... führen kann.
keine	alle übrigen Arten von Fehlverhalten.

Kritikalitäten-/Funktionen-Matrix

Kritikalitätsstufe \ Funktion	F1	F2					Fn
hoch	X		X				
mittel							
niedrig		X					
keine							X

Abbildung 9.24: Kritikalitäten-/Funktionen-Matrix

Vererbung der Kritikalität:

Mit zunehmender Verfeinerung der Betrachtungseinheiten werden auch die Kritikalitätseinstufungen vererbt:

- vom System auf die Systemfunktionen (Produkt → Funktion),
- von den Systemfunktionen auf Segmente (Funktion → Produkt),
- vom Segment auf die Segmentfunktionen und
- von hier ab über die SW-/HW-Einheiten zu den Funktionen der SW-/HW-Einheiten bis auf die SW-Komponenten, SW-Module und Datenbanken vererbt!

Aus **Kostengründen**: so wenig kritische Funktionen wie nötig bei gleichem Systemverhalten!

Regel R1: (Vererbungsregel)

R1a	Produkt → Funktion Mindestens eine der beim Verfeinerungprozess aus einem Produkt erzeugten Funktionen muss eine gleich hohe Kritikalitätsstufe wie das Produkt selbst besitzen.
R1b	Funktion → Produkt Bei der Zuordnung der Funktionen zu den Produkten muss mindestens ein einer Funktion zugeordnetes Produkt eine gleich hohe Kritikalitätsstufe haben wie die Funktion selbst.

R1 stellt sicher, dass eine einmal festgelegte Kritikalitätseinstufung eines Produkts im Verlaufe des Verfeinerungsprozesses in mindestens einem Verfeinerungszweig erhalten bleibt.

Sie soll dazu veranlassen, die Kritikalitätseinschätzung beim Entwurf möglichst genau zu lokalisieren.

Regel R2: (Kritikalität abhängiger Funktionen)

R2a	Besteht zwischen zwei Funktionen eine einseitige Abhängigkeit, d. h. eine Funktion nutzt Leistungen der anderen, so muss die Kritikalitätsstufe der beeinflussenden (benutzten) Funktion mindestens ebenso hoch sein wie die der nutzenden Funktion.
R2b	Beeinflussen sich zwei Funktionen wechselseitig, müssen beide dieselbe Kritikalitätseinstufung haben.

Regel R2 soll dazu veranlassen, die Abhängigkeiten zwischen den Entwurfseinheiten zu minimieren. Dies entspricht einem wesentlichen **Entwurfsprinzip** für sichere, zuverlässige und wartbare Systeme.

Massnahmen zur Abwehr der Auswirkung von Fehlverhalten

- Konstruktive Maßnahmen:
  Entwicklung von eigensicheren bzw. fehlertoleranten Funktionseinheiten, Konfigurierung von redundanten oder diversitären Funktionseinheiten

  Arbeiten mit **assertions**, **exceptions**, unabhängige Entwicklung mehrerer Versionen derselben Einheit (**diversitäres Vorgehen**) und "paralleler" Einsatz dieser Versionen (**Redundanz**)
- Analytische Maßnahmen:
  Durchführung umfangreicher Verifikations- und Validationsmaßnahmen

Kritikalitäten-/Methoden-Matrix (im QS-Plan) – Beispiel

Analytische QS−Methoden	Kritikalitätsstufe			
	hoch	mittel	niedrig	keine
Walkthrough	●	●	●	●
Aktivitäten−Audit nach QS−Plan	●	●	●	
Durchschnittl. C1−Abdeckung mind. 90%	●	●	●	●
Durchschnittl. C2−Abdeckung mind. 90%	●	●	●	
Informelle Prüfung gem. Prüfspezifikation	●	●		
Korrektheitsbeweis Code vs. Spezifikation	●			
Stat. Analyse des Codes bzgl. Einhaltung von Standards	●	●	●	●
Simulation	●			

Abbildung 9.25: Kritikalitäten-/Methoden-Matrix

9.5.5.2 Das V-Modell: QS-Plan

aus: `www.informatik.uni-bremen.de/uniform/gdpa/vmodel_d/d-qaplan.htm`

1. Allgemeines
2. Qualitätsziele und Risiken im Projekt
    - 2.1 Qualitätsziele für Produkte und Prozesse
    - 2.2 Qualitätsrisiken
    - 2.3 Maßnahmen aufgrund der Qualitätsziele und -risiken
3. QS-Maßnahmen gemäß Kritikalität und IT-Sicherheit
    - 3.1 Verwendete Richtlinien oder Normen
    - 3.2 Einstufungsbedingte QS-Maßnahmen
4. Entwicklungsbegleitende Qualitätssicherung
    - 4.1 Zu prüfende Produkte
    - 4.2 Zu prüfende Aktivitäten
5. Spezifische Kontrollmaßnahmen
    - 5.1 Eingangskontrolle von Fertigprodukten
    - 5.2 Kontrolle von Unterauftragnehmern
    - 5.3 Ausgangskontrolle der Softwarebausteine
    - 5.4 Änderungskontrolle
    - 5.5 Kontrolle von Bearbeitungskompetenzen
    - 5.6 Kontrolle des Konfigurationsmanagements

9.5.5.3 Das V-Modell: Prüf-Plan

definiert

- die Prüfgegenstände
- die Aufgaben und Verantwortlichkeiten bei den Prüfungen
- die zeitliche Planung sowie
- die für die Durchführung erforderlichen Ressourcen

Welche Produkte und Aktivitäten sind in welchem Zustand wann, von wem und womit zu prüfen?

Orientiert an den Projektgegebenheiten wird entweder ein – für alle Prüfungen verbindlicher – Prüfplan erstellt,

oder es erfolgt eine zweckmäßige Aufteilung in mehrere, physikalisch getrennte Prüfpläne, die jeweils Teilmengen der Prüfungen umfassen – z.B. anhand von Teilprojekten oder anhand der Bezugsprodukte (System, Segment, SW-Einheit/HW-Einheit)

Inhalt des Dokuments:

1. Allgemeines
2. Prüfgegenstände und Qualifikationserfordernisse
3. Aufgaben und Verantwortlichkeiten
4. Zeitplan
5. Ressourcen
   5.1 Prüfumgebung
   5.2 Weitere Ressourcen

9.5.5.4 Das V-Modell: Tailoring

Ziele des Tayloring:

- Vemeidung „unnötiger“ Dokumente / Vermeidung der Aufnahme „unnötiger“ Informationen in die Dokumentation
- Vemeidung „unnötiger“ Aktivitäten („Vergolden“, basierend auf Erfahrungen des Teams. . .)
- Sicherstellen, dass alle projektrelevanten Dokumente berücksichtigt und alle relevanten Aktivitäten auch tatsächlich durchgeführt werden

Unabdingbar: Mitwirkung des QM

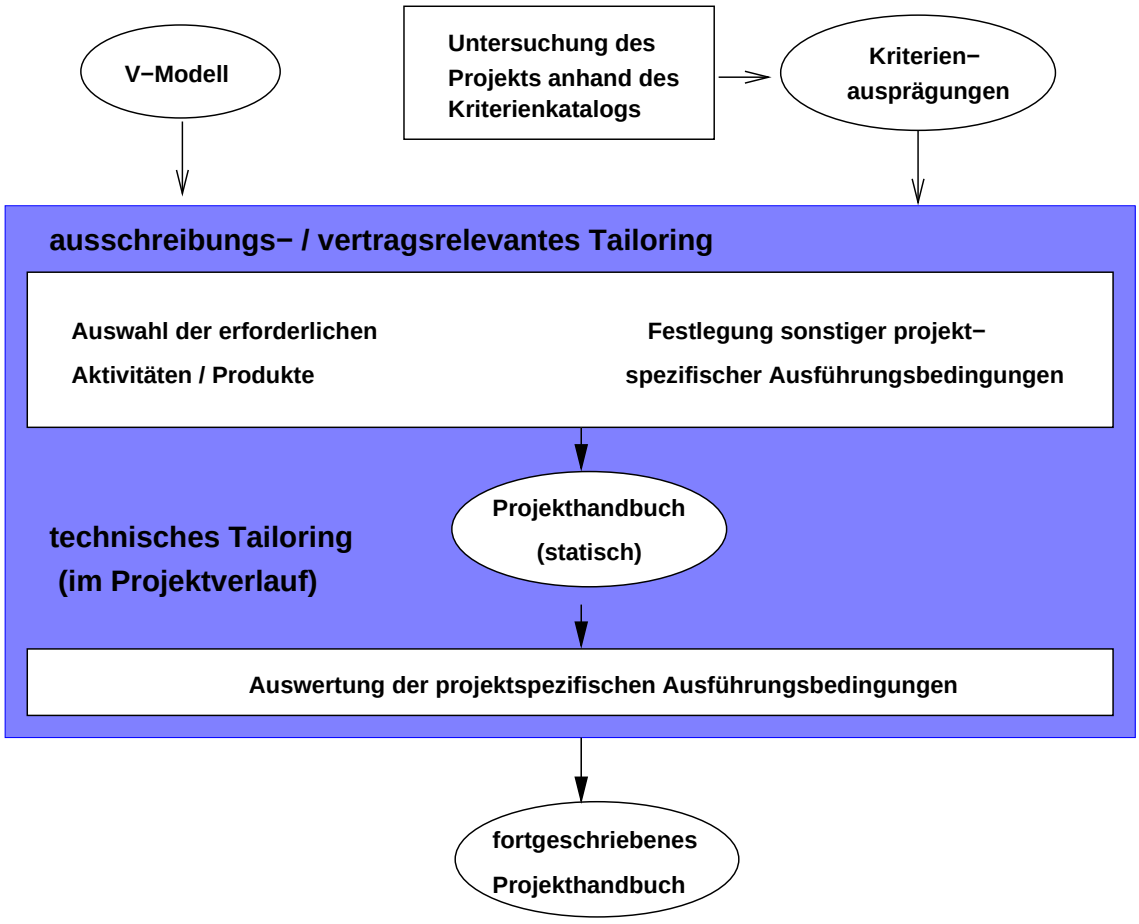


Abbildung 9.26: Tailoring

Ausschreibungsrelevantes Tayloring:

- **Vor** Beginn des (eigentlichen) Projekts / **vor** Vertragsabschluss durchgeführt
- Auswahl der für das Produkt erforderlichen Aktivitäten und Dokumente (z.B. wer macht die Pflege?)
- Festlegung von Bedingungen, unter denen bestimmte Aktivitäten / Dokumente (Inhaltspunkte) im Projektverlauf entfallen können

Technisches Tayloring:

- kontinuierlich während der Projektabwicklung durchgeführt
- Auswertung der im Projekthandbuch festgeschriebenen Auswertungsbedingungen
- Entscheidung darüber, welche Aktivitäten durchzuführen sind (z.B. welche Dokumente sind per Inspektion zu prüfen?)

9.5.5.5 V-Modell '97: Vorteile und Probleme

Vorteile:

- **Integration von Systementwicklung, Qualitäts-, Konfigurations- und Projekt-Management**
- Generisches Modell mit definierten Anpassungs-Möglichkeiten
- Standardisierte Abwicklung von Projekten zur Systemerstellung
- Gut geeignet für große Systeme, für kritische Systeme, für langlebige Systeme, für große Projekte (Zahl der Mitarbeiter, Laufzeit)
- Technik des Prototyping kann integriert werden

Potenzielle Probleme:

- Gefahr von „Software-Bürokratie“ für kleinere / mittlere / unkritische / . . . Projekte / Produkte
- Schwierigkeit der vollständigen Definition der Anforderungen bei komplexen oder innovativen Systemen
- Schwierigkeit einer vollständigen Projektplanung (Einbeziehung und Bewertung von Alternativen)
- Ohne geeignete Werkzeugunterstützung schwer handhabbar
- . . .

Neu: V-Modell XT

wird hier nicht weiter behandelt, siehe z.B.

A. Rausch, M. Broy: Das V-Modell XT - Grundlagen, Erfahrungen und Werkzeuge. dpunkt.verlag, 2005

9.5.6 Potenzielle Probleme bei Phasenmodellen

- Wechselseitige Koordination und Kommunikation zwischen Auftragnehmer / Entwicklern und Auftraggebern / Anwendern (Benutzern?) nur in der Anforderungsphase und dann erst wieder in der Abnahmephase
- häufiger / langer Ausschluss des Benutzers

Adressierbar durch

→ Technik des **Prototyping**

→ **eXtreme Programming**

- Schwierigkeit, (die Anforderungen an) ein komplexes System vollständig zu spezifizieren
  - technische Unklarkeiten
  - organisatorische Unklarheiten (die "Falschen" entscheiden)
  - "diffuse" Vorstellungen des Auftraggebers
  - Überschätzung des Auftragnehmers

Adressierbar durch

→ Technik des **Prototyping**

→ **inkrementelle Entwicklung (evolutionäre Entwicklung)**

→ **iterative Entwicklung (wiederholtes Durchlaufen eines Phasenmodells)**

→ **eXtreme Programming**

- Hoher Zeitaufwand, lange Projektlaufzeit

NB:

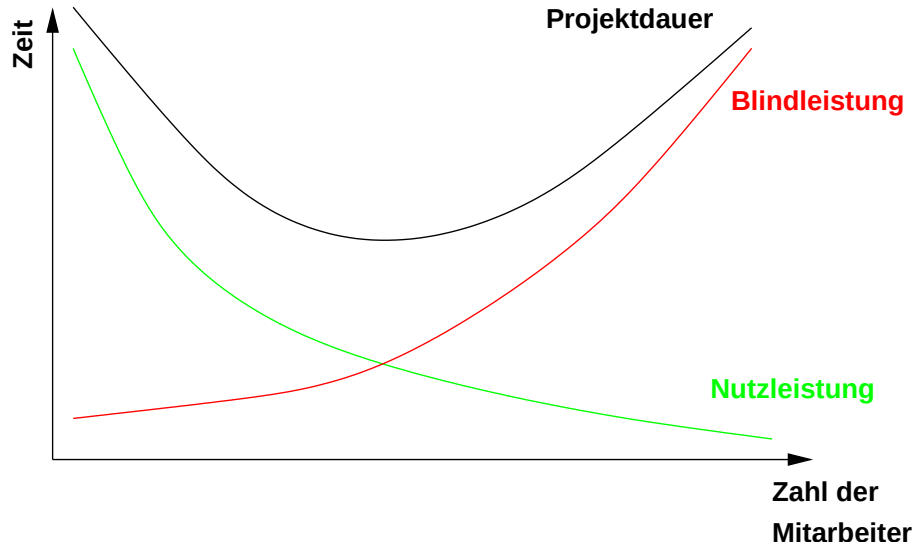


Abbildung 9.27: Projektdauer in Abhängigkeit der Mitarbeiterzahl

Adressierbar durch

→ **Concurrent Engineering**

→ **Einsatz von Generatoren**

- Garantierte „Korrektheit“ der Software

Adressierbar durch

→ **Einsatz formaler Methoden (Bsp.: Programmverifikation)**

- Schwierigkeit der vollständigen Projektplanung,
  Einbeziehung von Alternativen

Adressierbar durch

→ **Boehm’sches Spiralmodell**

9.5.7 Die Technik des Prototyping

Prototyp (nach IEEE-Standard)

> A preliminary type, form, or instance of a system that serves as a model for later stages or for the final, complete version of the system

Prototyping (nach IEEE-Standard)

> A hardware and software development technique in which a preliminary version of part or all of the hardware or software is developed to permit **user feedback, determine feasability**, or investigate timing or other issues in support of the development process

Rapid Prototyping (nach IEEE-Standard)

> A type of prototyping in which emphasis is placed on developing prototypes **early** in the development process to permit **early feedback and analysis** in support of the development process

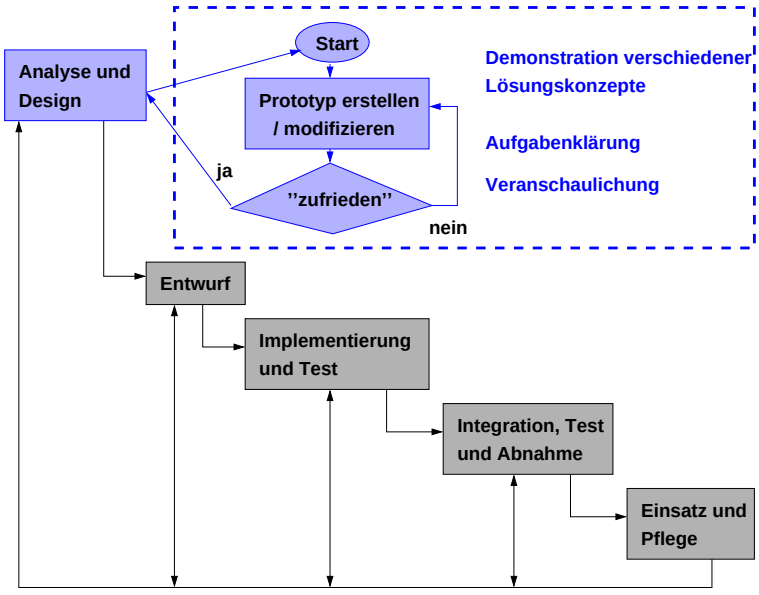


Abbildung 9.28: Exploratives Prototyping

Evolutionäre Software-Entwicklung, Pilotsystem wird in mehrerer Zyklen zur „Produktreife“ gebracht

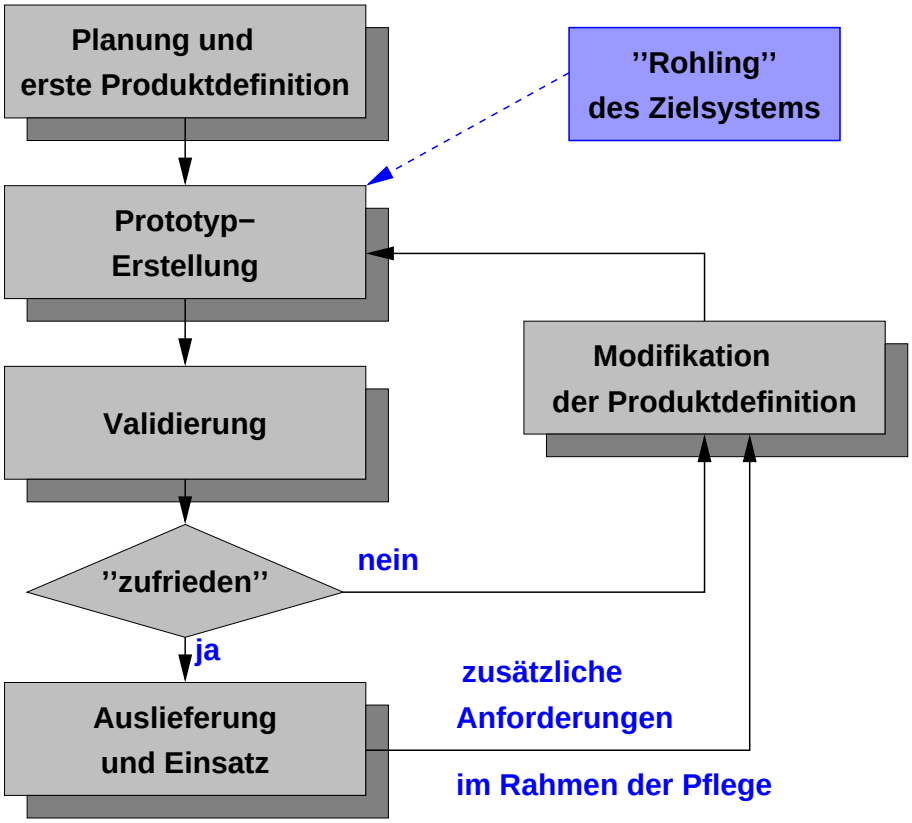


Abbildung 9.29: Evolutionäre SW-Entwicklung

Arten von Prototypen, unterschieden nach ihrem Zweck:

- **Exploratives Protoyping** (meist für GUI, schwierige Aspekte in der Funktionalität) – s.o.
- **Pilotsystem** (evolutionäre Software-Entwicklung) – s.o.
- **Demonstrationsprototyp**
  - zur Akquisition
  - liefert ersten Eindruck des geplanten Systems
- **Labormuster**
  - zur Studie von Konstruktionsalternativen
  - zur Demonstration der technischen Machbarkeit des Produktmodells

---

Prototyping: Vorteile

- Verringerung der **Entwicklungsrisikos** / der **Entwicklungskosten** weil **frühzeitige** Entdeckung von Missverständnissen, Inkonsistenzen und Unvollständigkeiten
- Stärkere Einbindung von Auftraggebern und **späteren Benutzern**
- mit geeigneten Tools – drag & drop, copy & paste, Visual Studios – schnell zu erstellen

Prototyping: Probleme

- Zusätzliche Investition
- Unterscheidung zwischen Wesentlichem und Unwesentlichen oft schwierig
- Beschränkungen und Grenzen des Prototypen oft nicht bekannt
- Gefahr des „Aufbohrens“ und damit der „Flickschusterei“ (**software development by stepwise debugging**)

9.6 Flexibilität als neue Herausforderung: Agile Ansätze

9.6.1 Motivation

Entwicklung von Software z.Bsp. im Automotive Bereich findet statt in "hochdynamischen Umgebungen":

- schnelle Innovationen / kurze Innovationszyklen
- flexible Kunden-/Lieferantenbeziehungen
- Unklare Anforderungen zu Beginn einer Entwicklung
- Viele Änderungen in den Anforderungen während der Entwicklung
- ...

Software-Entwicklung als Transformationsprozess:

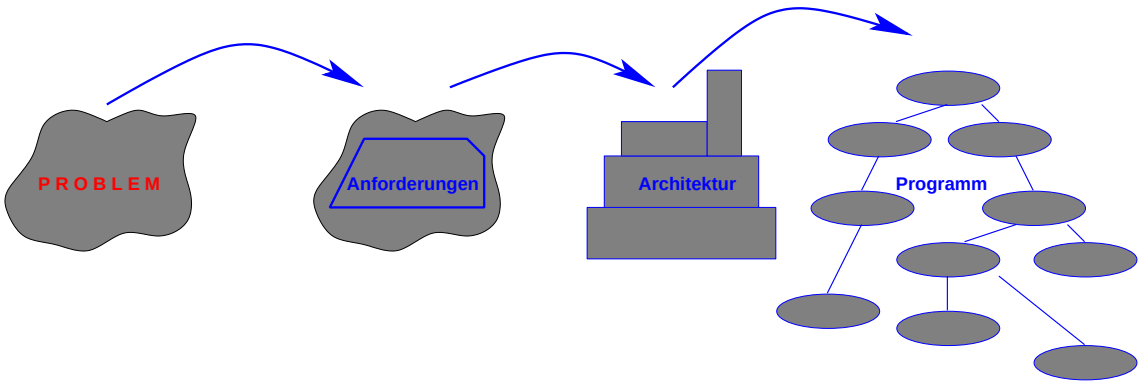


Abbildung 9.30: Transformation im Mittelpunkt

Wasserfallmodell - etwas anders betrachtet:

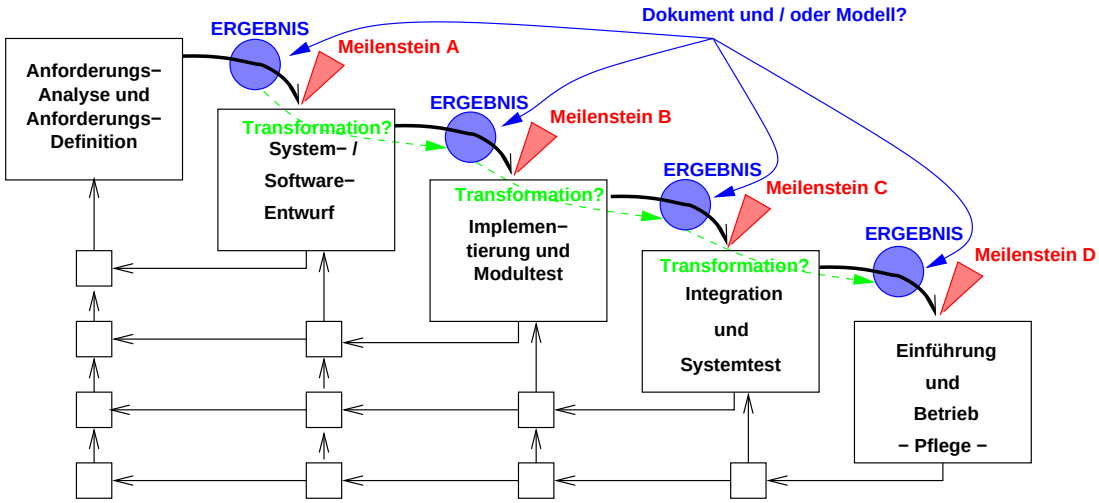


Abbildung 9.31: Modellbasiertes Wasserfallmodell

9.6.2 V-Modell XT

Vorgehensbausteine und ihre Bestandteile:

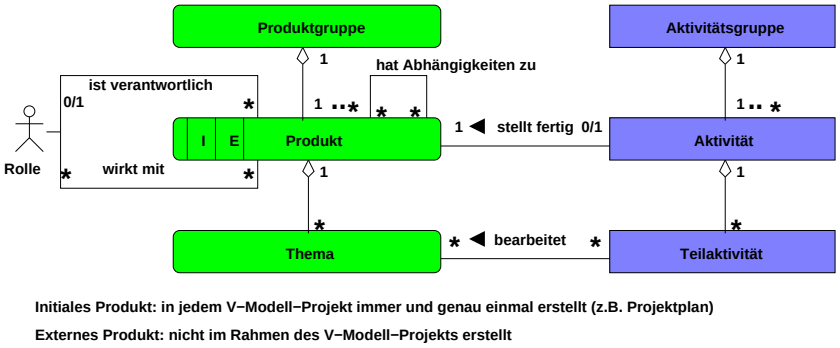


Abbildung 9.32: Bausteine

Wesentliches Prinzip: Ziel- und ergebnisorientierte Vorgehensweise

- Produkte stehen im Mittelpunkt – sie sind die zentralen Projektergebnisse
- Projektdurchführungsstrategien und Entscheidungspunkte geben die Reihenfolge der Produktfertigstellung und somit die grundlegende Struktur des Projektverlaufs vor
- Die detaillierte Projektplanung und -steuerung wird auf der Basis der Bearbeitung und Fertigstellung von Produkten durchgeführt
- Für jedes Produkt ist eindeutig eine Rolle verantwortlich, und in einem konkreten Projekt dann eine dieser Rolle zugeordnete Person oder Organisationseinheit
- Die Produktqualität ist durch definierte Anforderungen an das Produkt und explizite Beschreibungen der Abhängigkeiten zu anderen Produkten überprüfbar

9.6.3 Generatoreneinsatz

Entwicklung einer abstrakten Lösung und ggf. parallel dazu eines problem- oder zielmaschinenspezifischen Generators

Beispiele:

- Parser-Generatoren (lex, yacc, flex, bison & Co., siehe z.B. http://dinosaur.compilertools.net/)
- Code-Erzeugung aus grafischen / textuellen Modellen (CASE-Tools):
  - MSC (Message Sequence Charts, **ITU-Standard Z.120**[1]) zur automatischen Generierung von SDL-Skeletten (Specification and Description Language, ITU-Standard Z.100)
- VHDL (Very High Speed Integrated Circuit Hardware Description, ANSI/IEEE Standard 1076-1993)
- Statechart-Modelle
- UML Sequenz-Diagramme
- Matlab und Simulink
- neuere OpenSource-Entwicklungen wie **SystemC**
- ...

zu Tools siehe z.B. http://www.incose.org/tools/ieee1220tax/synthesis.html (INCOSE International Council on Systems Engineering, IEEE-1220 SE Tools Taxonomy - Synthesis Tools[2]

[1] The ITU (International Telecommunication Union), headquartered in Geneva, Switzerland is an international organization within the United Nations System where governments and the private sector coordinate global telecom networks and services. See http://www.itu.int/home/

[2] Contact us at info@incose.org Copyright 1998-2000 International Council on Systems Engineering Last Modified: March 15, 2002

9.6.4 Agilität als neue Herausforderung

Was ist agil?

Ein adaptiver Ansatz zur Lösung von Aufgaben, der streng fokussiert ist auf

- Kommunikation
- Zusammenarbeit / Kooperation
- Auslieferung
- Änderung

Das agile Manifest (siehe http://agilemanifesto.org/)

Individuals & Interactions	over	Processes & Tools
Working Software	over	Comprehensive Documentation
Customer Collaboration	over	Contract Negotiation
Responding to Change	over	Following the Plan

That is, while there is value in the items on the right, we value the items on the left more.

Einige Prinzipien (frei nach http://agilemanifesto.org/principles.html)

Ein agiler Prozess ...

- ...ist bestrebt, den Kunden durch frühzeitige und ständige Auslieferung von brauchbarer Software entsprechend seiner Prioritäten zufriedenzustellen
- ...hat nichts gegen Änderungen in den Anforderungen, auch in späten Phasen der Entwicklung (flexibel das Pferd umspannen zum Wohl des Wettbewerbsvorteils des Kunden)
- ...liefert lauffähige Software, die "tut", häufig aus, zwischen wenigen Wochen und wenigen Monaten, je schneller desto besser
- ...verlangt, dass Entwickler und Anwender / Nutzer ständig das ganze Projekt hindurch zusammenarbeit
- ...braucht motivierte Perönlichkeiten – gib ihnen Umgebung und Ausstattung, die sie brauchen, und vertraue ihnen, dass sie es schaffen
- ...fördert direkte Kommunikation als die effizienteste und effektivste Form der Übermittlung von Informations ins Team und innerhalb des Teams
- ...benutzt Software "die tut" als primäres Fortschrittsmaß
- ...fördert nachhaltige Entwicklung (*value based software engineering*, alle sog. Stakeholder ziehen am selben Strick und sind im Boot)

- ... zollt permanent Aufmerksamkeit der technischen Exzellenz und achtet auf gutes Design
- ... fordert Einfachheit
- ... vertraut auf selbst-organisierende Teams
- ... bittet das Team, regelmäßig zu reflektieren, um stets besser zu werden
- ...

siehe auch http://www.agilealliance.com/home/)

9.6.5 eXtreme Programming

spezielle Aufbau-Organisation, vor allem aber spezielle Ablauf-Organisation, aber auch Entwicklungs-Methode

siehe z. B.: **http://www.extremeprogramming.org**, [Beck99], [Beck00]

Einige Schlüssel-Praktiken

- **Das TEAM**
  - Alle, die zum Projekt etwas beitragen, sind integraler Bestandteil des Teams!
  - Sollte nicht zu groß sein: ≤ 12
  - der „Kunde“ als Mitglied: der, der am besten weiß, was die tatsächlichen Anforderungen sind, der Prioritäten setzen kann, der das Projekt **steuert**
  - Programmierer, die **paarweise** arbeiten
  - Tester, die dem „Kunden“ helfen, den Abnahmetest vorzubereiten
  - Analytiker, die dem „Kunden“ helfen, die Anforderungen festzulegen
  - Ein **Coach**, der dem Team hilft, auf „Kurs“ zu bleiben
  - Manager, die dem Team den Rücken freihalten, Ressourcen beschaffen, Aktivitäten koordinieren, die Kommunikation nach außen besorgen
  - **NB:** Diese Rollen müssen nicht notwendig jeweils von einem Individuum wahrgenommen werden – gefragt sind Generalisten mit speziellen Fähigkeiten (keine Spezialisten im Sinne von „Fachidioten“)!
- **Offene Arbeitsumgebung**
  - Absolute räumliche Nähe aller Teammitglieder
  - offene Arbeitsplätze
  - Kommunikationsunterstützende Ausstattung (Flipcharts, Wandtafeln, grosse Arbeitstische, so dass zwei bequem gleichzeitig an einem Computer arbeiten können)

Prozessbeschreibung

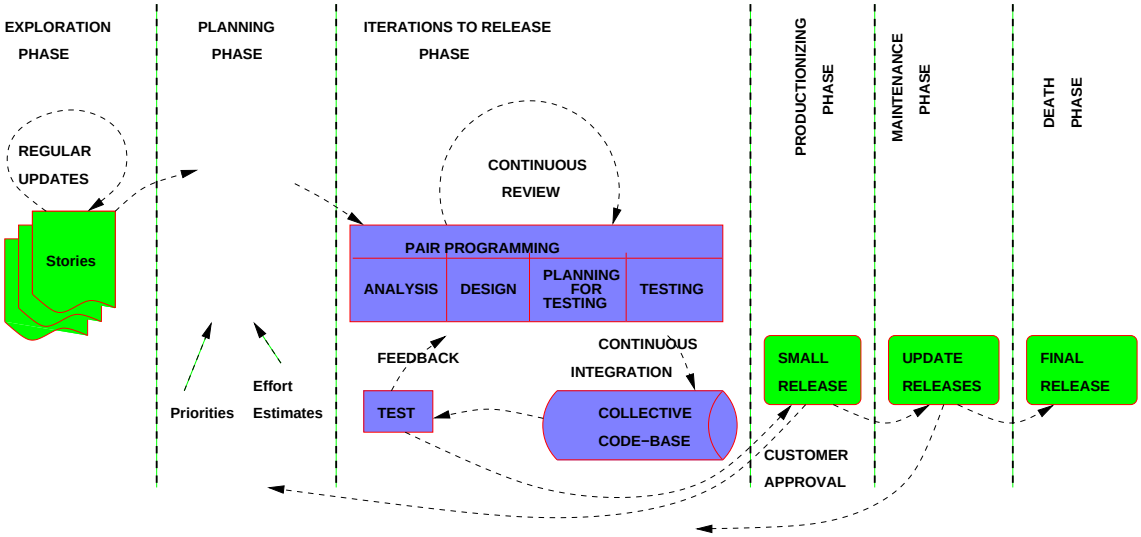


Abbildung 9.33: eXtreme Programming – Prozess

- **Planen:**

  Was wird zu einem definierten Zeitpunkt fertig sein?
  Was muss als nächstes gemacht werden?

  - **Release Planung — kurze Release Zyklen:**
    - Der „Kunde“ präsentiert den Entwicklern die notwendigen Features, diese schätzen den Aufwand
    - Mit diesem Aufwand und der Kenntnis der Bedeutung dieser Features legt der „Kunde“ den Projektplan vor
    - je weiter das Projekt fortschreitet, desto genauer werden die Pläne
    - nach ein bis drei Monaten wird das System an die wirklichen Endanwender ausgeliefert, damit der „Kunde“ wichtiges Feedback für die weitere Entwicklung erhält
  - **Iterations-Planung – kurze Entwicklungsschritte:**
    - Einzelne Entwicklungsschritte sollten nur wenige (2?) Wochen dauern
    - Ergebnis ist jeweils lauffähige und brauchbare Software
    - Detail-Planung erfolgt also auf 2-Wochen-Ebene

- **Einfacher Design:**
  - Nicht *high sophisticated*, sondern immer passend zu aktuellen Funktionalität
  - wird ständig fortgeschrieben, ggf. vollständig erneuert (Stichwort: **Refactoring**)
  - kein Design “ein-für-allemal”

- **Paarweises Programmieren:**
  - Jedes Stück Software wird von zwei Programmierern „Seite-an-Seite“ geschrieben!

  - Vieraugenprinzip: dadurch automatisches Review
  - Lerneffekte nicht zu unterschätzen
  - „Ausfallsicherheit“
- **Frühe / ständige Tests**
- **Permanente Systemintegration**
- **Der Code gehört allen!**
- **Standards**

Anm.: Dies sind nur einige Aspekte, die XP kennzeichnen!

12 häufige Fehler bei der Einführung / Durchführung von XP

nach: S. Roock, H. Wolf: Fünf Jahre XP: Wie erwachsen sind wir geworden. in: OBJECTspectrum, Nr. 4, Juli/August 2005

- Agil = Schwammig / beliebig

  „Wir sind ja so agil! Und agil heißt doch, dass man schnell auf Änderungen reagiert. Also, da hätten wir hier schnell mal ein paar neue Anforderungen. Das schaffen Sie schon. Bis morgen, ach, übermorgen reicht auch noch.“

  - falsches Verständnis von Flexibilität
  - eine Iteration sollte man vorausdenken können
  - Agiles Vorgehen bedeutet, äußerst diszipliniert vorzugehen!
- Kommunikationsprobleme

  „Wir schaffen das schon. Ich will keine schlechten Nachrichten hören!“

  - Kommunikationsprobleme sind meist qualitativer und nicht quantitativer Natur!
  - Notwendig: klare, transparente Kommunikationsstrukturen:
    * Wer nimmt welche Aufgaben wahr?
    * Wer ist wofür verantwortlich?
    * Wer hat welche (Entscheidungs-) Kompetenzen?
    * Wer muss dann wann zu welchem Zweck befragt werden?
    * Wer braucht deshalb welche Informationen zu welcher Zeit?
  - Kommunikation muss offen und ehrlich sein!
    IEEE Software Engineering Code of Ethics and Professional Practice, 1999
    www.computer.org/cerification/ethics.htm

- Überanpassung

  „Wir haben ja eigentlich immer schon agil Software entwickelt." „Gerade diese eine Technik kann man bei uns nicht einsetzen." „Technik XYZ machen wir nicht. Wir machen ABC, das ist ja fast das Gleiche".

  - Häufiges „Das macht eigentlich keinen Unterschied" macht in der Summe sehr wohl einen Unterschied!
  - Wenn sich etwas ändern soll, muss man etwas ändern!

- Kein Tracking / Controlling

  „Tracking bringt uns nichts. Das macht uns auch nicht schneller."

  - Tracking ist Voraussetzung, Planabweichungen zu erkennen
  - Gerade agile Methoden erlauben eine Reaktion auf Planabweichungen!
  - *Wir sollten lieber wissen wo wir stehen, als mit Höchsgeschwindigkeit in eine eliebige Richtung vorzupressen*

- Wir stellen uns dumm

  „Wir brauchen uns nicht um XYZ zu kümmern, das wird der XP-Prozess schon für uns richten."

  - Risikomanagement?!
  - *Vorgehen nach XP bedeutet, im Brain-On-Modus zu arbeiten*

- Naives Design

  „Wir machen das was im Moment den geringsten Aufwand verursacht!"

  - *Mache es so einfach wie möglich, aber nicht einfacher* (Albert Einstein)
  - Einfaches Design bedeutet nicht No-Design!

- Einführung wird unterschätzt
- Teamgröße skaliert nicht
- Falsche Retrospektiven
- Unechter Kunde
- Grandioser Prozess
- Kunde unterliegt Entwicklern

9.6.6 Scrum, Crystal & Co.

Scrum

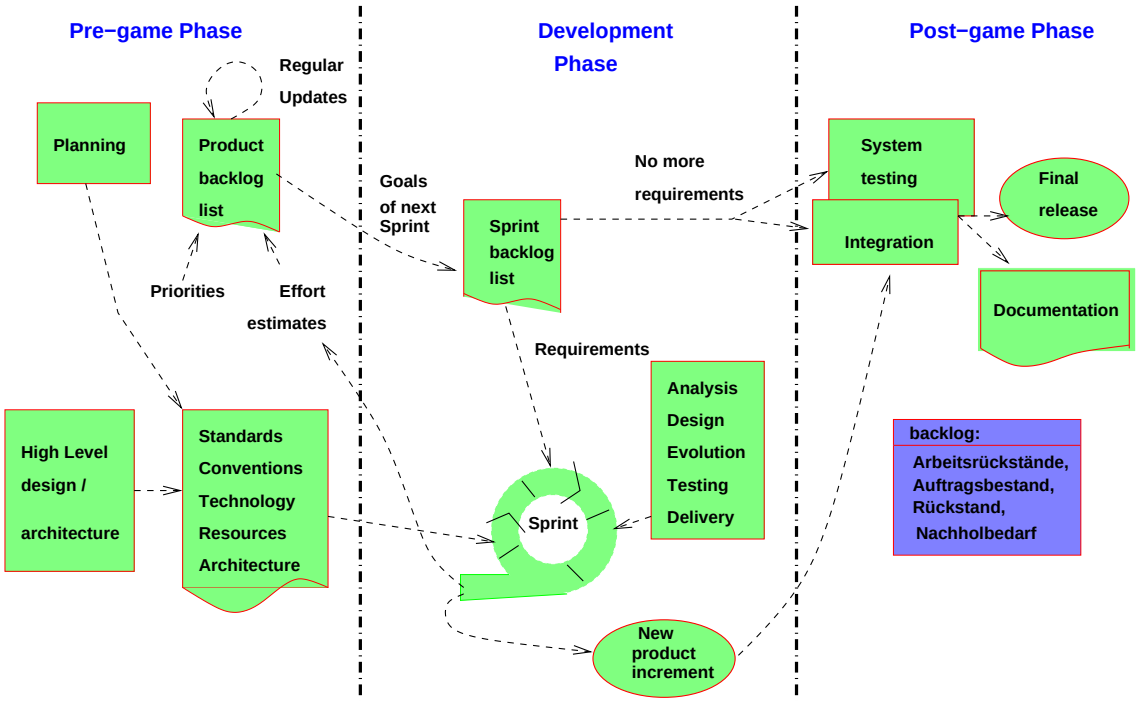


Abbildung 9.34: Scrum – Übersicht

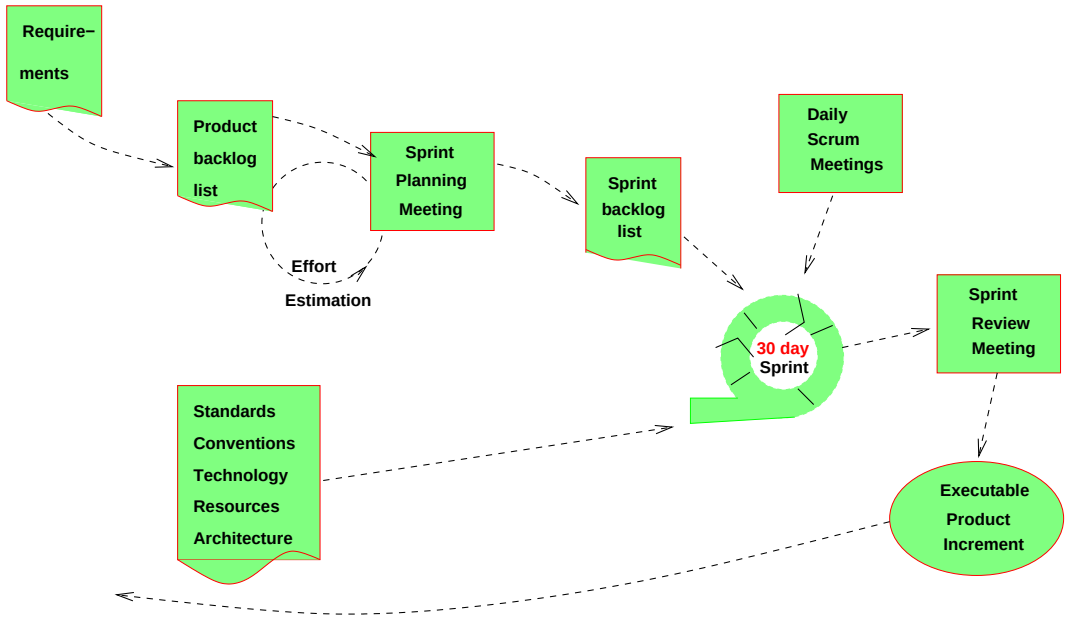


Abbildung 9.35: Scrum – die eigentliche Entwicklung

Familie der Crystal Methoden (siehe z.B. [Cockburn03])

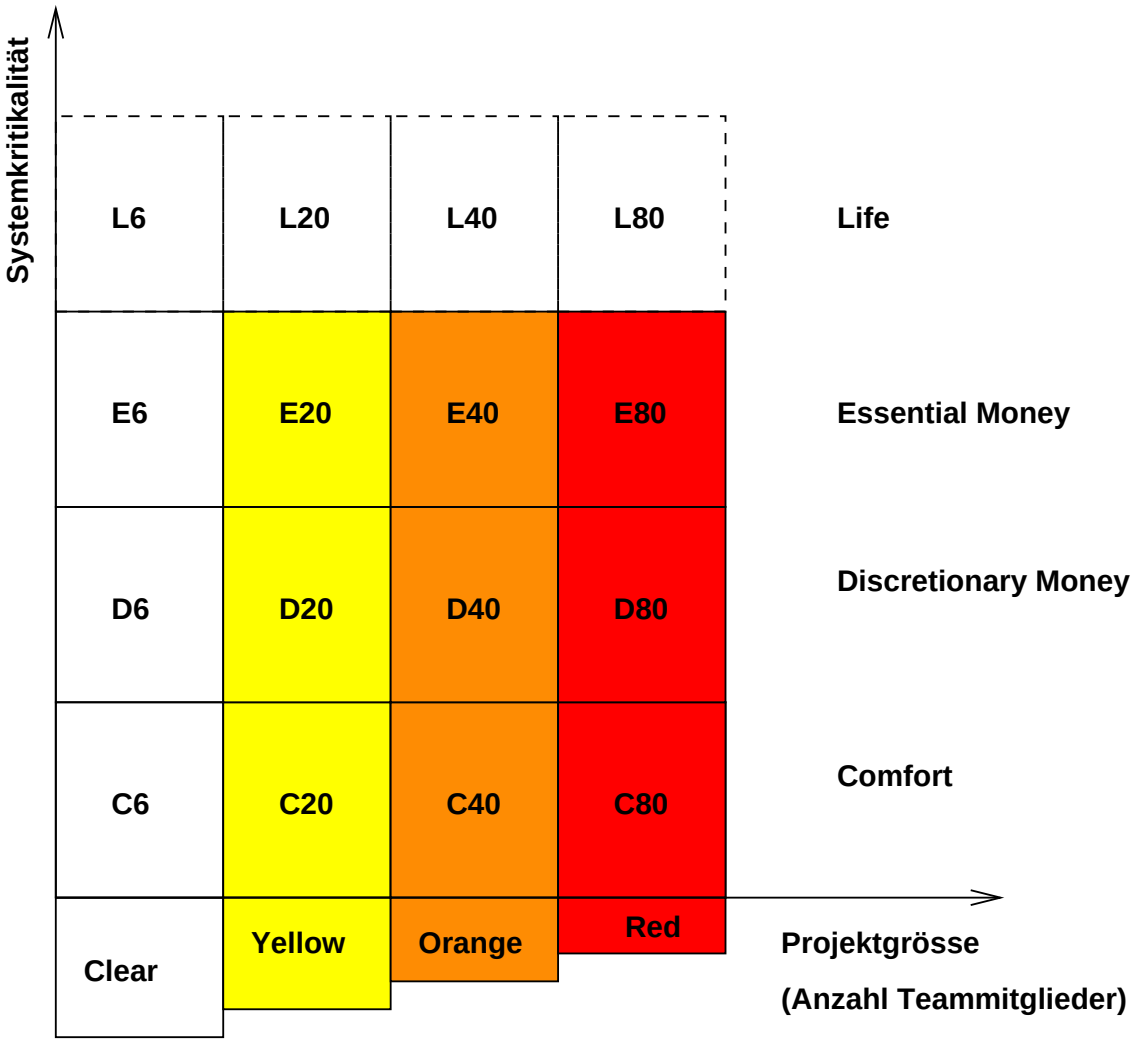


Abbildung 9.36: Familie der Crystal Methoden

Für die Prozesse *Chrystal Clear* und *Chrystal Orange* sind folgende Prinzipien definiert:

- Inkrementelle Entwicklung
- Fortschrittskontrolle durch Meilensteine, die auf Entscheidungen und Softwareauslieferungen basieren und sich weniger auf Dokumentation beziehen
- der Kunde wird direkt in das Projekt involviert
- vollautomatische Regressionstest der ("früheren") Funktionalität
- Teambesprechungen vor, nach und vorzugsweise während eines Inkrements

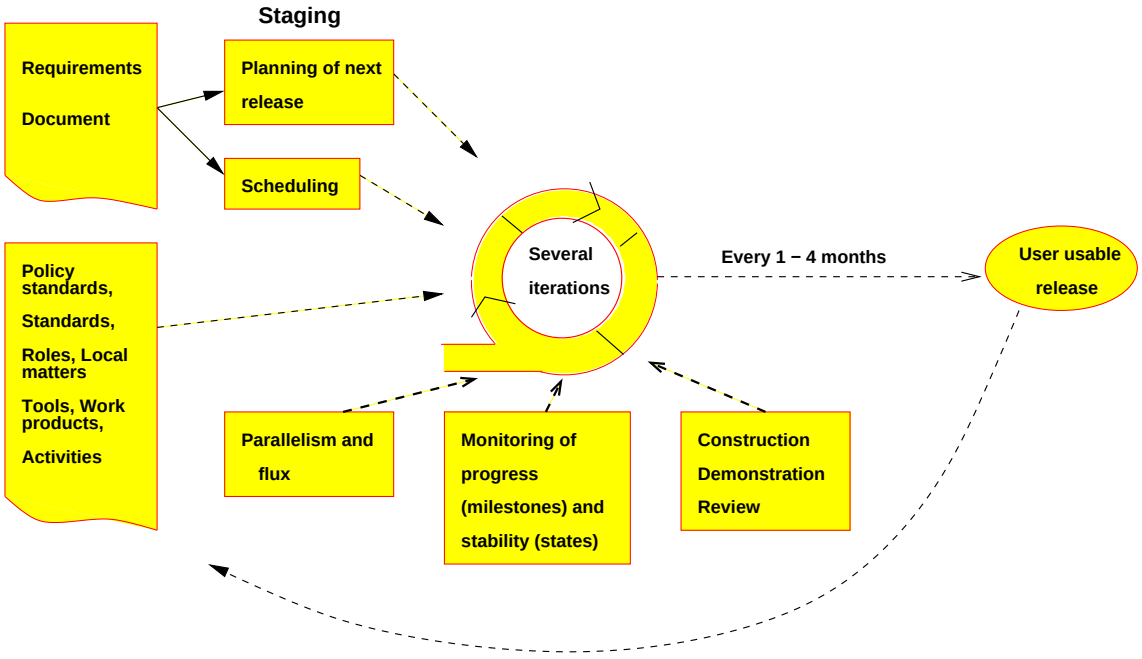


Abbildung 9.37: Ein Inkrement des Chrystal Orange Prozesses

Für alle Chrystal Prozesse sind diverse Praktiken und Aktivitäten definierbar:

- *Staging*

  Planen des nächsten Inkrements – Team wählt für jedes Inkrement zu implementierende Anforderungen aus – plant, welche Features in diesem Inkrement realisiert werden können

- *Revision and Review*

  Jedes Inkrement beinhaltet mehrere Iterationen – jede Iteration umfasst: Konstruktion, Demonstration und Review der Features dieses Inkrements

- *Monitoring*

  Überwachung von Fortschritt bezgl. Arbeitsergebnisse und von Stabilität des Systems – Meilensteine: Start, Review 1, Review 2, Test, Lieferung des Systems – Stabilitätsetappen: stark schwankend, schwankend, stabil genug für Review

- *Parallelism and flux*[3]

  Ergibt Überwachung der Stabilität „stabil genug für Review“, so kann mit der nächsten Aufgabe begonnen werden – in Chrystal Orange: die verschiedenen Teams können mit maximaler Parallelität arbeiten

- *Methodology-tuning technique*

  Basistechnik der Chystal Familie – anhand von Projekt-Interviews und Workshops werden spezifische Prozesse für jedes individuelle Projekt kreiert – in jedem Inkrement kann das Wissen aus dem vorhergehenden Inkrement zur Verbesserung genutzt werden

- *Reflection workshops*

  Crystal Clear / Orange: vor jedem und nach jedem Inkrement eine „Besinnung“ (kritische Vor- und Rückschau)

- ...

nach der Diplomarbeit von S. Przewoznik: Flexibilität bei Definition und Anwendung verschiedener Prozessmodelle. Uni Ulm, 2005

[3] Fluss

9.7 Modellbasierte / Modellgetriebene Entwicklung

Modellbegriff:

aus der als Dissertation an der Uni Ulm eingereichten Arbeit von Mario Jeckle, der vor Abschluss des Promotionsverfahrens tödlich verunglückte:

- stereometrisches Vorbild des endgültigen Werkes
- Muster, Entwurf (für eine (serienweise) Herstellung), Beispiel im Speziellen, Bezeichnung einer (Holz-)Form zur Herstellung einer Gußform
- Abbild der Natur unter Hervorhebung von als wesentlich erachteter Eigenschaften unter Außerachtlassung (im Sinne der Vereinfachung) als nebensächlich angesehener Aspekte
- in verkleinertem, natürlichen oder vergrößertem Maßstab ausgeführte räumliche Abbilder
- als Einzelstück gefertigtes Kleidungsstück
- (Natur-)Objekt oder -Gegenstand, der als Vorbild künstlerischer Gestaltung dient
- Entwurf- oder Ausführungsart
- Bereich, dessen Mitglieder und deren Verknüpfungen eine durch Axiome beschriebene abstrakte Struktur besitzen
- konstruiertes, vereinfachtes Abbild des tatsächlichen Wirtschaftsablaufs

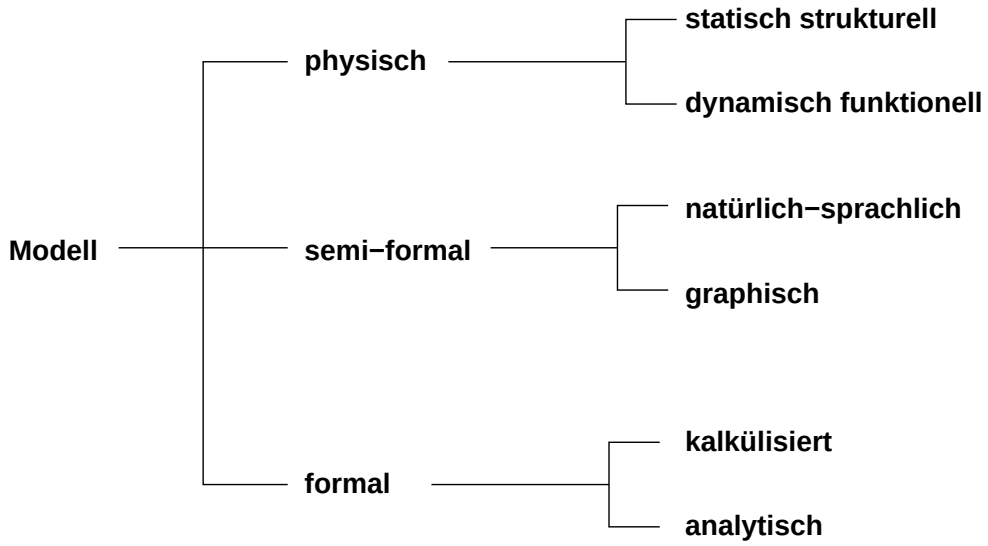
Begriff **Model**:

- Halbmesser des unteren Teils einer antiken Säule
- Hohlform für die Herstellung von Gebäck oder zum Formen von Butter
- Erhabene Druckform für Stoff- und Tapetendruck
- Strick- und Wirkmuster

Im Englischen:

- *a representation or pattern in miniature, in three dimensions, of something to be made on a larger scale*
- *a figure in clay, plaster etc. for execution in durable material*
- *a thing or person to be represented by a sculptor or painter*
- *one employed to pose as subject to an artist*
- *one employed to wear clothes to display their effect*
- *a standard, an example regarded as a canon of artistic execution*
- *a particular style or type*
- *a set of postulations, mathematical equations*
- *worthy of imitation*

Klassifikation von Modellen



praktisch: aggregierte Modelle, die nicht mehr genau einem Modelltyp zuzuordnen sind

Abbildung 9.38: Modell-Klassifikation

Modell – kurz und knapp:

- Ein **Modell** ist ein System, das ein anderes (reales) System abbildet.
- Eine **Modellbildung** erfordert geeignete Modellierungs**methoden**.
- Eine **Methode** ist ein Vorgehensprinzip zur Lösung von Aufgaben.
- Eine Modellierungsmethode umfasst Konstrukte und eine Vorgehensweise, die beschreibt, wie diese Konstrukte wirkungsvoll bei der Modellbildung anzuwenden sind (**Meta-Modell**).
- Konstrukte umfassen die Elemente einer „Beschreibungssprache“ und die Regeln für die Verknüpfung der Elemente
  (**Meta-Modell: Syntax und Semantik**).
- **Modell-Merkmale:** das Abbildungsmerkmal, das Verkürzungs-/Abstraktions-Merkmal, das pragmatische Merkmal
- Das **Abbildungsmerkmal** kennzeichnet das Modell als Abbild des realen Systems. Man unterscheidet zwischen isomorphen (strukturgleichen) und homomorphen (strukturähnlichen) Modellen.
- Das **Verkürzungsmerkmal** ergibt sich daraus, dass das Modell nur die Elemente und Relationen des realen Systems abbildet, die dem Modellbildner (subjektiv) als relevant erscheinen.
- Das **pragmatische Merkmal** (Pragmatik: Ziel- / Zweckorientierung) bezeichnet Ziel und Zweck der Modellbildung.
  - Beschreibungsmodelle (z.B.: Ergebnis der Anforderungsanalyse ist eine Anforderungsdokument, das das “Modell” aus Sicht der Anforderungen beschreibt – Begriff: **Schema**)
  - Bezugsmodell (z.B. ein Prototyp)
  - Erklärungsmodell (z.B. ein Simulationsmodell) und
  - Entscheidungsmodelle (Mathematisches Modell zur Restfehlerprognose)

Allgemein:

- Zweck von **Beschreibungsmodellen** ist die Informationsgewinnung über die Beschaffenheit eines Systems in einem als statisch angenommenen Zustand durch eine Beschreibung des Systems und seiner Entscheidungssituationen. Es kann als Bezug / Ausgangspunkt für weitere Modelle sein (↪ Software-Entwicklung).
- **Erklärungsmodelle** sollen reale Erscheinungen eines Systems erklären und prognostizieren, also Aussagen über künftige Systemzustände ermöglichen.
- **Entscheidungsmodelle** sollen bestimmte Handlungsmassnahmen aus vorgegebenen Zielsetzungen, Randbedingungen und Entscheidungsvariablen ableiten.

Meta

- **Meta-**Modelle:

  Modelle zur Formalisierung von Modellen bzw. Modelle zur strukturellen Beschreibung von Modellierungssprachen
- **Meta-**Sprache:

  Sprache zur formalen Beschreibung formaler Sprachen

  Beispiel: EBNF ist eine Sprache zur Beschreibung der Grammatik von Programmiersprachen

Konzeptuelle Modellierung

- konzeptuell (manchmal im Deutschen auch *konzeptionell*
  - von lat. *concipere* – „zusammenfassen, begreifen, sich vorstellen“
  - eng. *conceptual*
- **Konzeptualismus**:

  Theorie der *Universalien*[4], der zufolge abstrakte – von den realen oder gedachten Dingen losgelöste – allgemeine Ideen, *konzepte* genannt, existieren. Diese Konzepte können **Eigenschaften** oder **Beziehungen** sein!
- **Konzeptuelles Schema:**

  ein Modell, welches den relevanten Informationsbereich (auch Miniwelt oder *Universe of Discourse*) in **Struktur** und **Inhalt** beschreibt.

[4] Allgemeinbegriff, nach dem lat. *universalis* – „zur Gesamtheit gehörend, allgemein“; gebildet aus lat. *universus* – „in eins gekehrt, zu einer Einheit zusammengefasst; ganz, sämtlich“

- **Logisches Schema**

  Ein **paradigmenspezifisches**[5] Modell, das aus einem konzeptionellen Modell abgeleitet wurde (direkt oder über daraus abgeleitete logische Schemata).

- **Physisches Schema**

  Ein implementierungsspezifisches Modell, das aus einem logischen Schema abgeleitet (direkt oder über daraus abgeleitete physische Schemata).

Allgemeine Drei-Schema-Architektur (nach Jeckle)

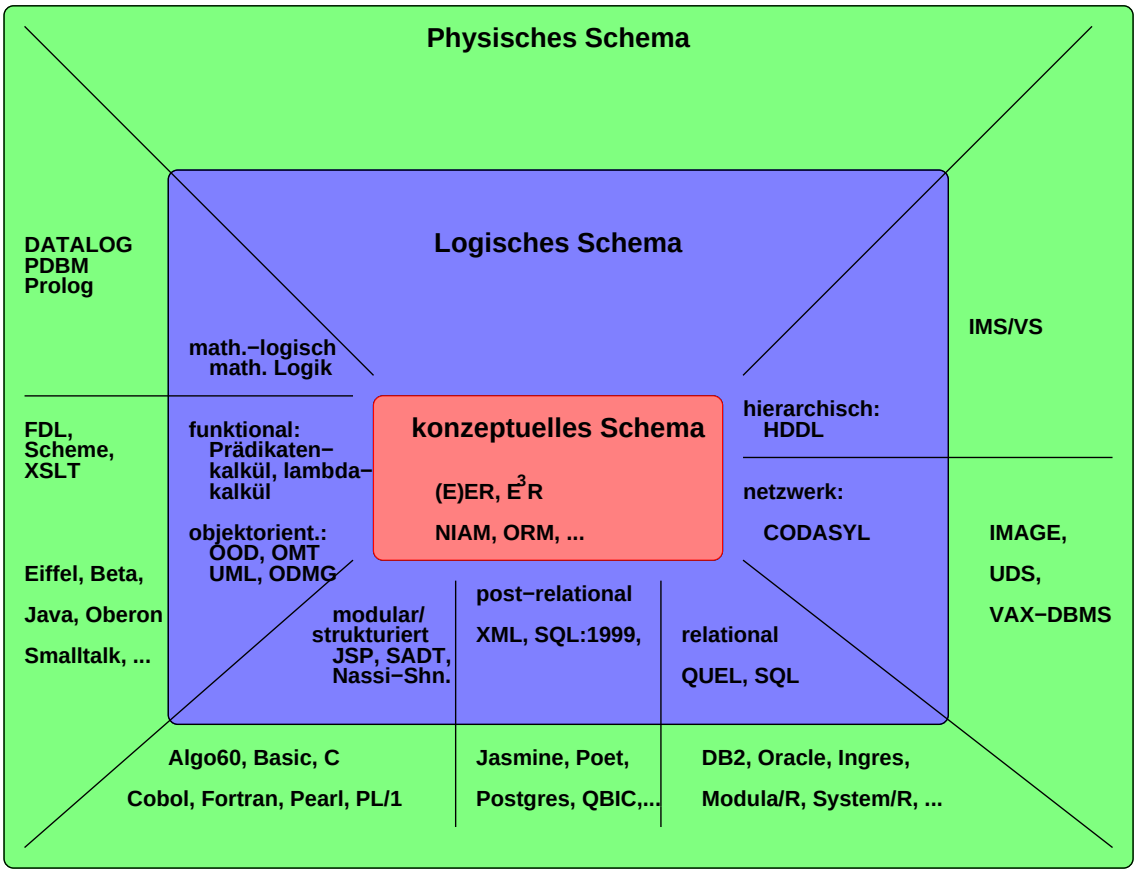


Abbildung 9.39: Drei-Schema-Architektur

[5] Paradigma: lat., aus dem griechischen kommend – „vorzeigen“, „sehen lassen“. Nach Platon die ewigen unveränderlichen Urbilder der sinnlich wahrnehmbaren Dinge. Allgemein: Beispiel, Muster, Erzählung, Geschichte mit modellhaftem Charakter. In der Wissenschaftstheorie: **ein Denkmuster, das eine Weltsicht prägt**

Modellbasierte Software-Entwicklung (MBSD)

- Entwicklung einer Folge $m_1; m_2; m_3; \ldots m_n$ von Modellen
- m_1 könnte sein: Machbarkeitsstudie, Lastenheft, Anforderungsdefinition
- m_n soll sein das lauffähige operative System
- jedes Modell ist Ergebnis eines Denkprozesses und baut gedanklich auf dem vorhergehenden Modell (falls vorhanden) auf ("gedankliche" Modeltransformation)

Modellgetriebene Software-Entwicklung (MDSD)

- Entwicklung einer Folge $m_1; m_2; m_3; \ldots m_n$ von Modellen
- m_1 und m_n wie oben
- der Transformationsprozess ist stark formalisiert und möglichst weitgehend automatisiert, d.h. $m_i \hookleftarrow T(m_{i-1})$

Prinzip des MDA-Prozesses[6]

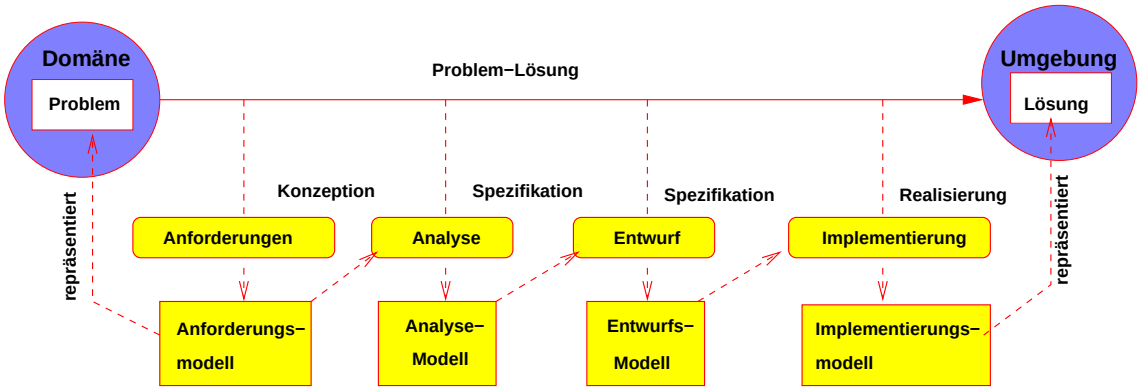


Abbildung 9.40: Systementwicklungsprozess

[6] aus: Modellgetriebene Software-Entwicklung mit der Model Driven Architecture, Diplomarbeit von A. Keis, Uni Ulm

- Historisch: Sonderforschungsbereich **CIP** der TU München Anfang der 70er
  Computer-aided Intuition-guided Programming
- Grundlage der MDSD: Konzept **Model Driven Architecture (MDA)** der **Open Management Group (OMG)**
  - Konsequente Weiterentwicklung der OMG-Vision von **komponentenbasierter Software-Entwicklung**
  - In das Konzept der MDA integrieren sich andere Standards:
    * **UML** Unified Modeling Language (2.0)
    * **XMI** eXtensible Markup Language Metadata Interchange
    * **MOF** MetaObject Facility, speziel **QVT** Query View Transformation
    * **CWM** Common Warehouse Metamodel
- In der aktuellen Version UML 2.0 wird durch die Erweiterung hinsichtlich Metamodellierung und Profilmechanismus der MDA-Ansatz unterstützt. Hierzu wurden und werden diverse Erweiterungen in Form von Plattformprofilen (z.Bsp. UML/Realtime für Echtzeitanwendungen) konzipiert, um der UML immer mehr Plattformen zu erschliessen.
- **UML-Profile** sind der Standardmechanismus zur Erweiterung des Sprachumfangs der UML, um sie an spezifische Einsatzbedingungen (z. Bsp. fachliche oder technische Domänen) anzupassen.
- MOF ist ein Standard der OMG zur Definition von Metamodell-Sprachen. Die MOF ist auf der Meta-Metamodellebene angesiedelt und definiert den Aufbau der Metamodellebene, auf der z.Bsp. UML und CWM liegen. Speziell: **QVT** (Query View Transformation) zur Modelltransformation, siehe z.B. `http://www.omg.org/docs/ptc/05-11-01.pdf` (204 Seiten!)
- XMI ist ein XML-basiertes Austauschformat für UML Modelle; dadurch wird die Interoperabilität von Modellen zwischen verschiedenen Werkzeugen unterschiedlicher Hersteller ermöglicht. So sollen sich mittels der MDA die Modelle zwischen verschiedenen Werkzeugen und Werkzeugkategorien unterschiedlicher Hersteller austauschen lassen.
- CWM ist ein von der OMG entwickeltes und standardisiertes Referenzmodell für den formalen und herstellerunabhängigen Zugriff und Austausch von Metadaten in der Domäne “Data Warehousing”.

Plattform

> A **platform** is a set of subsystems and technologies that provide a coherent set of functionality through interfaces and specific usage patterns, which any application supported by that platform can use without concern for the details of how the functionality provided by that platform is implemented. MDA GUIDE Version 1.0.1 (www.omg.org/docs/)

Beispiele für technologische Plattformen:

- $CORBA^{TM}$ Common Objects Request Broker Architecture
- CORBA Component Model
- $J2EE^{TM}$ Java 2 Enterprise Edition

Beispiele für herstellerspezifische Plattformen:

- CORBA: $Iona\ \ Orbix^{TM}$, $Borland\ \ VisiBroker^{TM}$, ...
- J2EEs: $BEA\ \ WebLogic^{TM} Server$, $IBM\ \ WebSphere^{TM}$, ...
- $Microsoft\ \ .NET^{TM}$

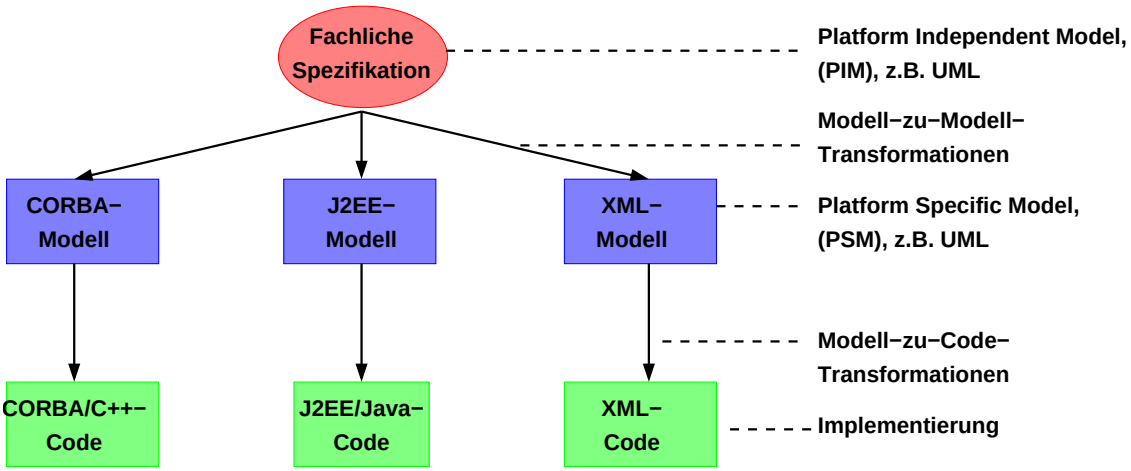


Abbildung 9.41: MDA-Prinzip

Platform Independence

> Eigenschaft eines Modells, unabhängig zu sein von Eigenheiten, Besonderheiten, Eigenschaften oder Fähigkeiten einer Plattform

MDA-Sichtweisen (bei der Modellierung)

- **Computation Independent Viewpoint**
  fokussiert ausschließlich auf die Systemumgebung und die Systemanforderungen – Details der Struktur wie auch der Arbeitsweise des Systems (*computation*) bleiben aussen vor oder sind noch unbestimmt
- **Platform Independent Viewpoint**
  - fokussiert auf die Arbeitsweise des Systems, verbirgt aber sämtliche Details, die notwendig sind für eine spezielle Plattform;
  - geht meist nur bis zu einem bestimmten Grad: “erwartet RMI”, “erwartet bestimmte Tools einer CORBA Platform”
- **Platform Specific Viewpoint**
  kombiniert die plattform-unabhängige Sicht mit einem dezidierten Fokus auf die spezifische Plattform

Computation Independent Model (CIM)

- Modell aus der *Computation Independent* Sicht
- Manchmal als „Domänen-Modell“ bezeichnet
- In der Sprache dessen, der mit der Domäne vertraut ist (z.B. Fachanwender)

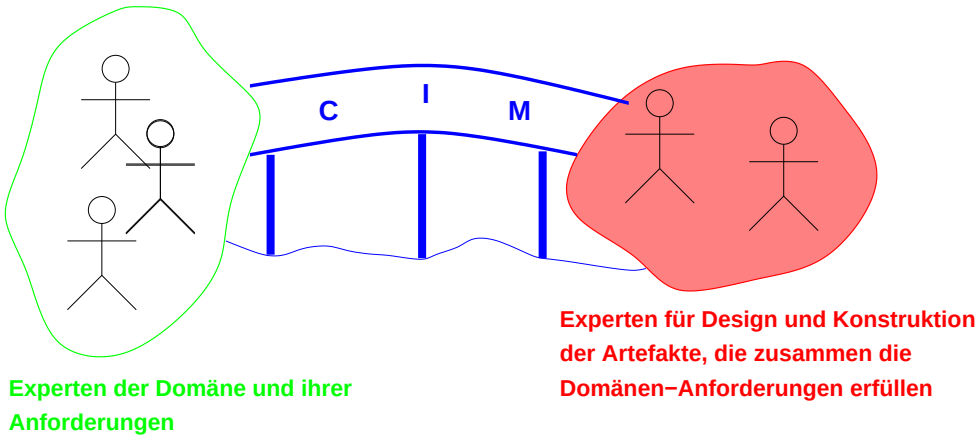


Abbildung 9.42: Computation Independent Model

Platform Independent Model (PIM)

- Modell aus der *Platform Independent* Sicht
- enthält einen festgelegten Grad an plattformunabhängigen Features, so dass es benutzt werden kann auf verschiedenen Plattformen ähnlichen Typs
- z.Bsp. durch Ausrichten an einer technologie-neutralen **virtuellen** Maschine
- **Virtual Machine:** Menge von Elementen und Diensten / Mechanismen (*communication, naming, scheduling, ...*), die unabhängig von einer spezifischen Plattform definiert sind und die plattform-spezifisch auf verschiedenen Plattformen realisiert sind.
- Eine virtuelle Maschine ist selbst eine Plattform – ein "PIM" ist daher spezifisch zu dieser Plattform
- Das Modell ist dennoch ein "PIM" bezüglich der Klasse der verschiedenen Plattformen, auf denen diese virtuelle Maschine realisiert ist

Platform Specific Model (PSM)

- aus der plattform-spezifischen Sicht
- kombiniert die Sepzifikationen im vorausgehenden PIM mit den Details, die angeben, **wie** die spezifische Plattform zu nutzen ist

Platform Model (PM)

- stellt eine Menge von technischen Konzepten bereit,
  - die die verschiedenen Elemente, die eine Plattform konstituieren, und
  - die die Dienste, die eine Plattform bereitstellt,

  repräsentieren **(WAS)**
- stellt zur Benutzung in einem PSM Konzepte bereit, die die verschiedenen Elemente spezifizieren **(WIE)**

Platform Model (PM)

Beispiel: Das *CORBA Component Model* stellt Konzepte wie *EntityComponent*, *SessionComponent*, *ProcessComponent* sowie sog. *Ports* wie

- *Facet* – Interfaces, die von einer Komponente angeboten werden
- *Receptacle* – Schnittstellen, über die eine Komponente auf andere Komponenten zugreifen kann
- *EventSource* – Ereignisproduzent, bietet die Möglichkeit zur Aussendung eines Ereignisses
- *EventSink* – Ereigniskonsument, Möglichkeit zum Empfang von Ereignissen
- und andere

bereit.

Platform Model (PM)

- spezifiziert weiter Anforderungen an die Verbindung und die Nutzung der Plattform-Elemente
- spezifizert die Verbindung einer Anwendung mit der Plattform

Beispiel:

Die OMG hat Teile der CORBA Plattform als *UML Profile for CORBA* spezifiziert. Dieses stellt eine Sprache zur Verfügung zur Spezifikation von CORBA Systemen (siehe www.omg.org/technology/documents/formal/profile_corba.htm)

Modell Transformation[7]

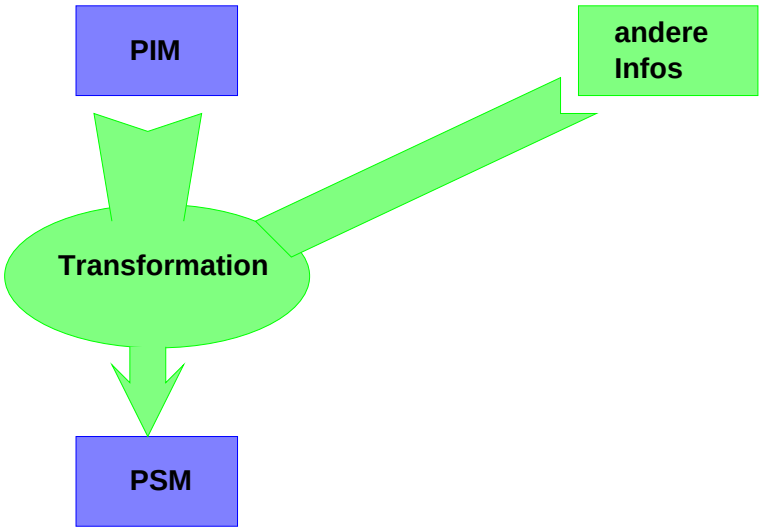


Abbildung 9.43: Modell-Transformation

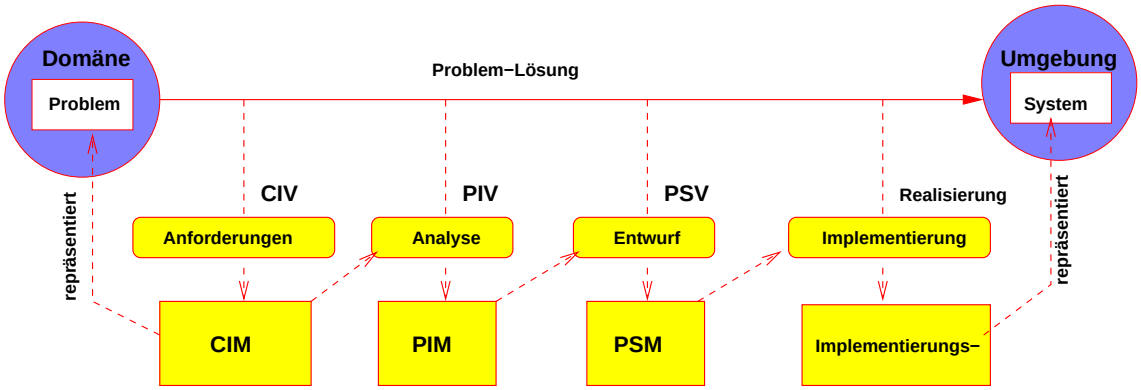


Abbildung 9.44: Grundlegendes Konzept der MDA

[7] aus: Modellgetriebene Software-Entwicklung mit der Model Driven Architecture, Diplomarbeit von A. Keis, Uni Ulm

Beispiel: Bibliothek

Bibliothek – Ausleihe von Büchern und CDs

Das Beispiel

- soll exemplarisch einige Konzepte zeigen
- ist in sich unvollständig (Beispiel)
- soll insbesondere auch die Anforderungen an die Semantikdefinition demonstrieren
- soll überzogene Erwartungen dämpfen
- verwendet bewusst **nicht** UML

Syntax und Semantik des Diagramms sind mathematisch-formal definiert **(Metamodell)** – beispielsweise und vereinfacht:

- Attribute sind Elemente einer Menge (Wertebereich), die Attributnamen sind die Namen dieser Mengen, die Mengen selbst können als formale Sprache definiert werden
- Ein Entitytyp (einfaches Rechteck) ist Teilmenge des kartesischen Produkts seiner Attribute, z.B. $Kunde \subseteq KNr \times Name$
  Anm.: die Reihenfolge der Mengen im kartesischen Produkt sei ohne Bedeutung, die Zeitabhängigkeit wird hier ignoriert
- Eine Relation (Raute) ist ebenso Teilmenge eines kartesischen Produkts, z.Bsp.: $A \subseteq Buch \times Autor$, oder $L \subseteq Kunde \times AusleihObjekt \times ADatum \times BDatum$
- Ein Existenzabhängiger Entityp (Beispiel: *Autor*) ist wie folgt definiert:
  $\forall\, a \in Autor : \exists\, b \in Buch$ mit $(a, b) \in A \subseteq Buch \times Autor$
- **is_a-Beziehung** (XOR):
  $AusleihObjekt \subseteq InvNr \times KaufDatum$
  $Buch \subseteq InvNr \times KaufDatum \times Titel \times ISBN \times Verlag$
  $CD \subseteq InvNr \times KaufDatum \times MusikGes \times Label$
  Sei

  $$Buch.InvNr = \{i | i \in InvNr \wedge \exists b \in Buch : b.InvNr = i\},$$
  $$CD.InvNr = \{i | i \in InvNr \wedge \exists c \in CD : c.InvNr = i\}$$

  dann gilt: $Buch.InvNr \cap CD.InvNr = \emptyset$
- Unterstrichene Attribute bilden einen Schlüssel:
  seien $A_1, A_2, \ldots A_n$ Attribute eines Entitytyps **E** – sie bilden einen einen Schlüssel, g.d., w. gilt:

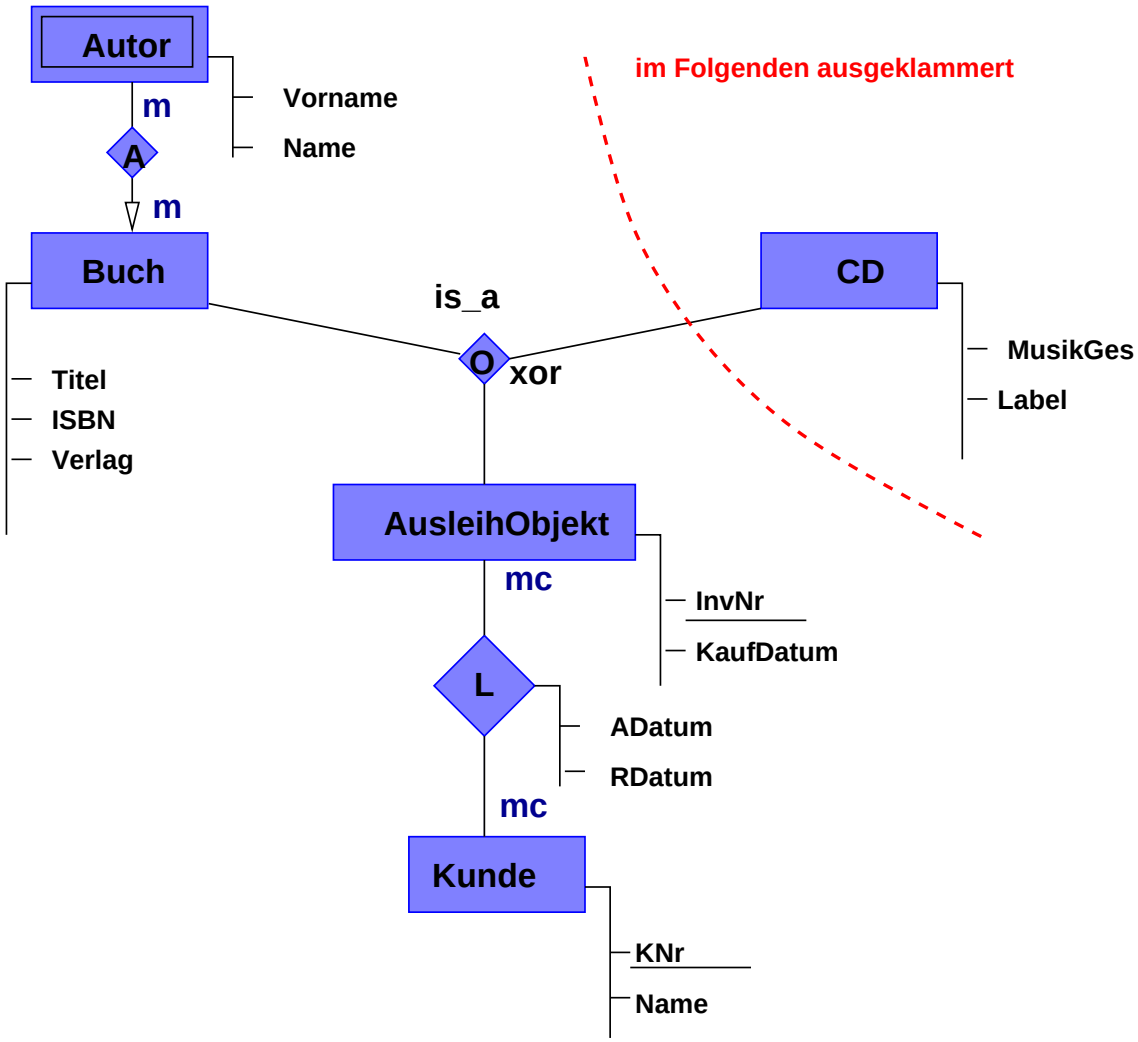


Abbildung 9.45: ER-Diagramm "Bibliothek"

- $\forall e_1, e_2 \in E : e_1.(A_1, A_2, \ldots, A_n) = e_1.(A_1, A_2, \ldots, A_n) \Rightarrow e_1 = e_2$
- keine Teilmenge von $\{A_1, A_2, \ldots, A_n\}$ hat diese Eigenschaft

Komplexitätsgrade

Sei R eine Relation und E ein daran beteiligter Entitytyp:

$$
\begin{array}{lcl}
kgrad(E,R) = 1 & \Leftrightarrow & \forall e \in E : \; |\{r \in R : r.E = e\}| = 1 \\
kgrad(E,R) = c & \Leftrightarrow & \forall e \in E : \; |\{r \in R : r.E = e\}| \leq 1 \\
kgrad(E,R) = m & \Leftrightarrow & \forall e \in E : \; |\{r \in R : r.E = e\}| \geq 1 \\
kgrad(E,R) = m & \Leftrightarrow & \forall e \in E : \; |\{r \in R : r.E = e\}| \geq 0
\end{array}
$$

Anm.: man müsste korrekterweise schreiben: $\forall t : \forall e \in E^t : \ldots$

($\forall t \longrightarrow$ "zu jeder Zeit")

Ergänzende Semantik: **Funktionale Abhängigkeit**

Sei U die Attributmenge eines Entitytyps E, seien $A, B, C \subseteq U$;

- B heißt **funktional abhängig** von B – in Zeichen $A \rightarrow B$, g.d., w. gilt:

$$\forall e_1, e_2 \in E : e_1.A = e_2.A \Rightarrow e_1.B = e_2.B$$

- Es gilt:
  - $A \rightarrow B \ \wedge \ B \rightarrow C \Rightarrow A \rightarrow C$ (Transitivität)
  - $A \subseteq B \ \Rightarrow \ B \rightarrow A$
  - A Schlüssel von $E \Rightarrow \forall u \in U : \ A \rightarrow u$
- Die funktionale Abhängigkeit ist eine **semantische Integritätsbedingung** und kommt aus der "Realität"!
- Damit lassen sich Regeln (Normalformen) für die Gestaltung von Tabellen im Relationalen Modell definieren (**Transformationsregeln**)

Im Beispiel:

- $ISBN \ \rightarrow \ (Titel, Verlag)$
- $(ADatum, InvNr) \ \rightarrow KNr$ (ein Buch kann am selben Tag nicht mehrmals ausgeliehen werden – so jedenfalls in der gegebenen Realität!)
- $(RDatum, InvNr) \ \rightarrow KNr$ (ein Buch kann am selben Tag nicht mehrmals zurückgegeben werden)

Weitere ***Constraints***:

- $ADatum < RDatum$
- $\forall l \in L \subseteq AusleihObjekt \times Kunde \times ADatum \times RDatum = InvNr \times Kaufdatum \times KNr \times Name \times ADatum \times RDatum : l.Kaufdatum < l.ADatum$

Umsetzung in ein PIM:

- Komponentendiagramm (UML) ?

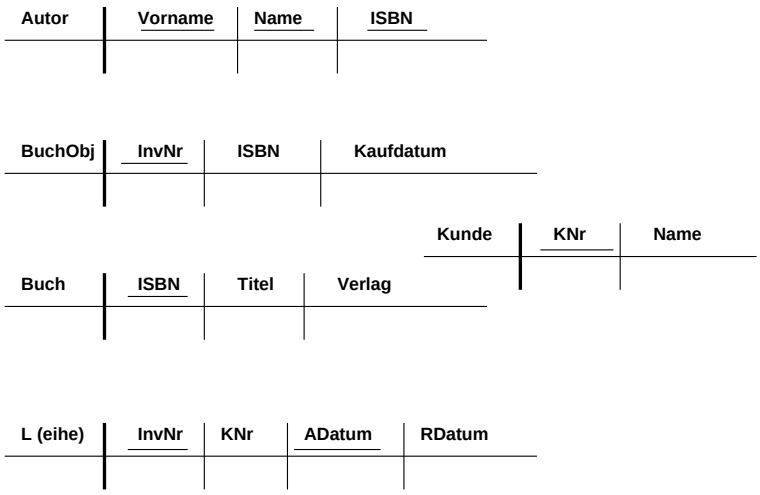


Abbildung 9.46: Tabellen der "Bibliothek"

- Relationales Modell!
  - lässt sich weitgehend automatisieren (Synthesealgorithmus, diverse Gestaltungsmuster (**Design Pattern**)
  - Intuition ist dennoch gefragt
  - es gibt mehrere Möglichkeiten (Nicht-Determinismus)
  - alle Integritätsbedingungen werden übernommen, "zusätzliche" entstehen aufgrund der höheren Ausdruckmächtigkeit der ER-Diagramme

PIM: die Tabellen

Weitere ***Constraints***:

- $BuchObj.ISBN \subseteq Buch.ISBN$
- $Autor.ISBN \subseteq Buch.ISBN$
- $Leihe.InvNr \subseteq BuchObj.InvNr$
- $Leihe.KNr \subseteq Kunde.KNr$

PSM?

create table Kunde …

Bislang: statisches Modell der Objekte

Was fehlt?

- dynamisches Modell (Ablauf)
- Technologie in PIM / PSM: CORBA, Web Services, ...
- GUI
- Der große Bereich der **Design Patterns**!

Bewertung von Modellen: MML Modeling Maturity Levels ([Warmer03])

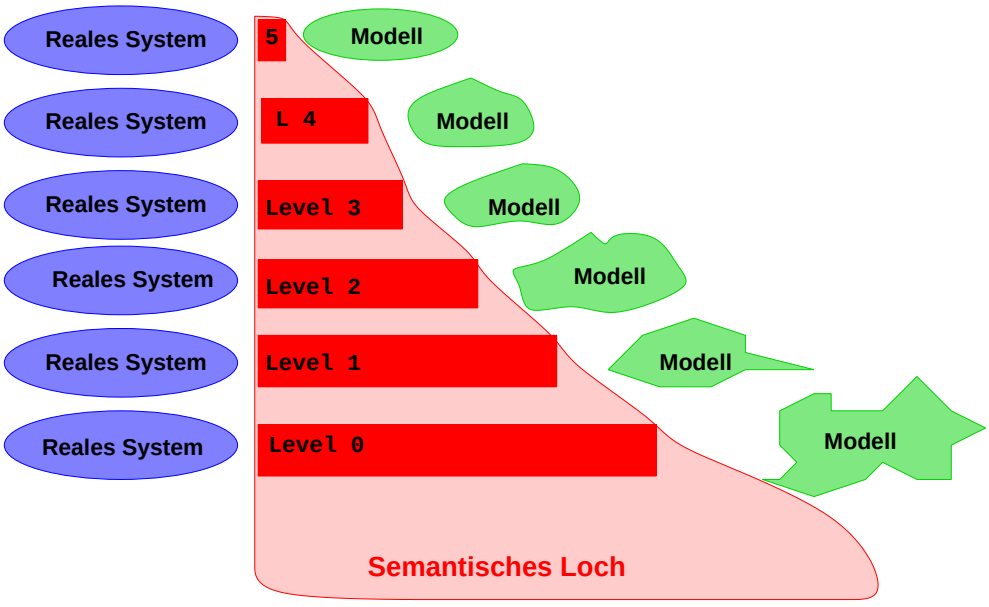


Abbildung 9.47: Modell-Bewertung

- **Level 0: Keine Spezifikation**
  - Spezifikation nur **im Kopf** des Entwicklers
  - Entwicklung auf Basis mündlicher Aussagen
  - fehlende Diskussionsgrundlage – Konsequenz: hohe Konfliktgefahr zwischen Entwicklern oder zum Anwender hin
  - Quellcode ist das einzige formale, fixierte Modell – Konsequenz: Quellcode verstehen nur die Entwickler, da sie alle Architekturentscheidungen **ad hoc** getroffen haben

  - für "sehr kleine" Anwendungen angebracht: 1-2 erfahrene Personen, kurze Laufzeit / Lebenszeit, geringe Kritikalität im Einsatz, kaum Schnittstellen zu anderen Systemen
  - *Chief Programmer Team*???

- **Level 1: Textuell**
  - Dokumente in natürlicher Sprache
  - mehr oder weniger formal, da z.Bsp. nicht festgelegt, wie Anforderungen / Funktionen wie detailliert beschrieben werden müssen
  - Unklare Semantik, Doppeldeutigkeiten, Interpretationsräume
  - Geringe Änderungsfreundlichkeit / Flexibilität
  - Irgendwann ist der Code das einzig aktuelle Dokument / Modell
  - "niedrigstes Niveau" für professionelle Software-Entwicklung

- **Level 2: Text mit Diagrammen**
  - "Ein Bild sagt mehr . . . "
  - Diagramme für Übersichten
  - Diagramme zur Erläuterung von Details
  - keine konsistente Verwendung von Diagrammen
  - Diagramme mit mehr oder weniger klar definierter Syntax und vor allem Semantik

- **Level 3: Modelle mit Text**
  - Das **Modellieren** steht im Vordergrund des Entwicklerdenkens
  - Modelle bestehen aus Diagrammen und / oder textuellen Beschreibungen ohne Mehrdeutigkeiten
  - Text dient zum besseren Verständnis der Zusammenhänge, enthält Motivation
  - die Modelle sind die reale und ausschließliche Repräsentation von Problem und Lösung
  - Modellübergänge sind kaum formalisiert, meist **ad hoc**
  - die Konsistenz der Modelle ist nicht sichergestellt, da oft eine stringente formale Definition der Semantik fehlt
  - der Weg vom abstrakten Modell (Spezifikation) zum realen Modell (Code) ist meist nachvollziehbar, der Weg zurück eher weniger (geringe Änderungsfreundlichkeit)

- **Level 4: Präzise Modelle**
  - Modelle bestehen aus konsistenten, zusammenhängenden und exakt in Syntax wie Semantik definierten Texten / Diagrammen

  - Es gibt eine präzise, eindeutig nachvollziehbare Verbindung zwischen den Modellen (hin zum Code und zurück)
  - Modelle wie Quellcode sind jederzeit konsistent und lassen sich "sehr einfach" auf den aktuellen Stand bringen
  - durch iterative und inkrementelle Entwicklung wird die direkte Transformation der Modelle hin zum Code erleichtert

- **Level 5: Ausschließlich Modelle**
  - Die Modelle bilden eine komplette, konsistente, detaillierte und präzise Beschreibung des Systems: Anwendung wie Lösung
  - die Modelle unterschiedlicher Abstraktionsebenen (CIM / PIM / PSM / Code) sind in sich konsistent und werden durch definierte Transformationsregeln auseinander generiert
  - Analogon: Modell "Quellcode" $\Leftrightarrow$ Modell "Ausführbarer Code" (Transformation durch Compiler)
  - insbesondere der Quellcode wird komplett generiert, der Entwickler hat mit dem Quellcode nichts mehr zu tun

MDA / UML-Tools

Informationsquellen:

- `http://www.omg.org/mda/committed-products.htm`
- `http://www.modelbased.net`
- `http://www.codegeneration.net/` (Übersicht zu allen möglichen Codegeneratoren)

Weitere Quellen:

`ftp.heise.de/pub/ix/ix_listings/2005/05/MDA_Tools_Uebersicht.html`

`http://umt-qvt.sourceforge.net/`

`www.galileocomputing.de/linklisten/gp/Liste-4/`

Anforderungen:

- Unterstützung der kompletten Syntax **und** Semantik **aller** UML Diagramme
- Forward Engineering
- Reverse Engineering (z.B. Generierung von Komponentendiagrammen aus Klassembibliotheken)
- Round-trip Engineering (z.B Synchronisation von Code-Änderungen und Modell-Änderungen)
- Dokumentation
- Versionskontrolle
- Testskript-Generierung
- Integration in IDEs
- Unterstützung von Design Pattern
- ...

Kapitel 10

Qualitätsmanagement (QM)

10.1 Grundlegende Aspekte des QM

QM muss einen Beitrag zur Erreichung der Unternehmensziele leisten, also zu

- Kostensenkung
- Profitverbesserung
- Rentabilitätssteigerung
- Erhöhung der Marktanteile
- Kundenbindung / Langfristige Rentabilität
- Mitarbeiterzufriedenheit / Steigerung der Produktivität
- Vermeidung von Gewährleistungs- / Haftungsansprüchen
- ...

Qualität ist unter wirtschaftlichen und juristischen Aspekten zu sehen und kann (am Markt) nicht Selbstzweck sein!

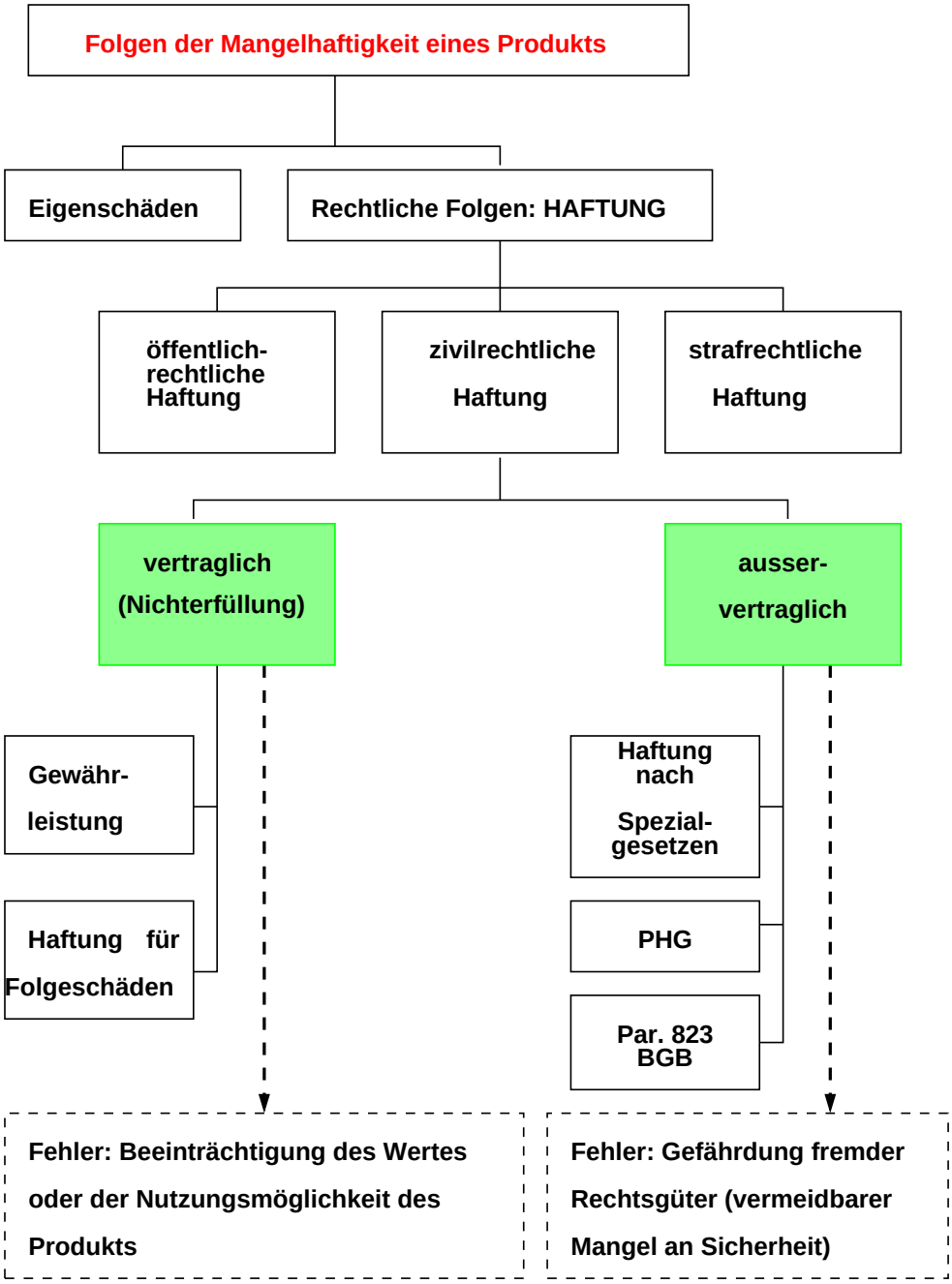


Abbildung 10.1: Haftungsgrundlagen

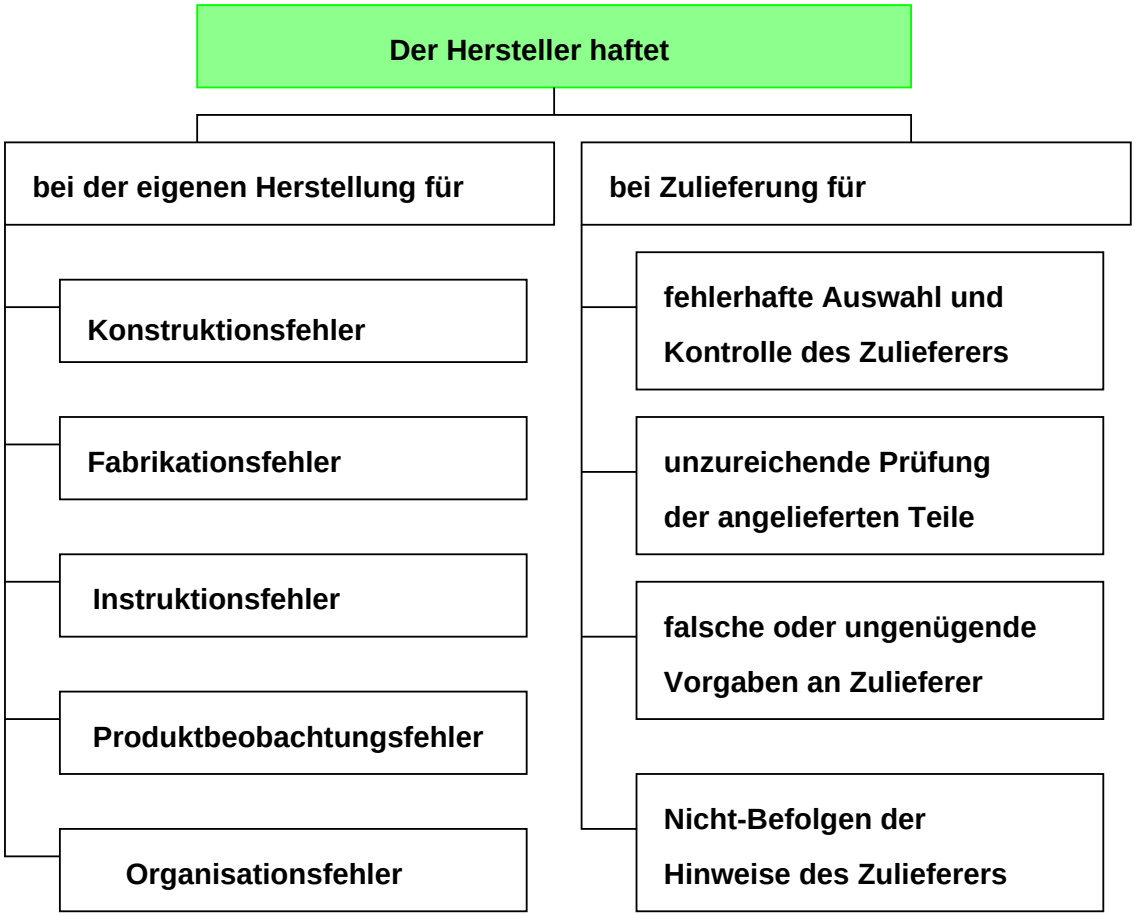


Abbildung 10.2: Haftungsbereiche nach Produkthaftungsgesetz (PHG)

Produkthaftung für Software - Haftungsentlastung durch Qualitätssicherung, Frank A. Koch ([?]):

> ...
> *Hierbei zeigt sich, dass eine Haftungsentlastung des Anbieters vielfach am Fehlen einer geschlossenen Qualitätssicherungskonzeption scheitert, spätestens aber an der unzureichenden und deshalb nicht beweisfähigen Dokumentation zu dieser Qualitätssicherung.*
> ...
> *Der Hinweis, dass sich Fehler von Software niemals vollständig ausschließen lassen, führt zu keiner Haftungsentlastung — weder vertraglich noch aus deliktischer bzw. gesetzlicher Produkthaftung — und kann sogar umgekehrt aufgrund des erhöhten Gefährdungspotentials zu einer Intensivierung der anbieterseitigen Pflicht zur Produktionsabsicherung führen.*

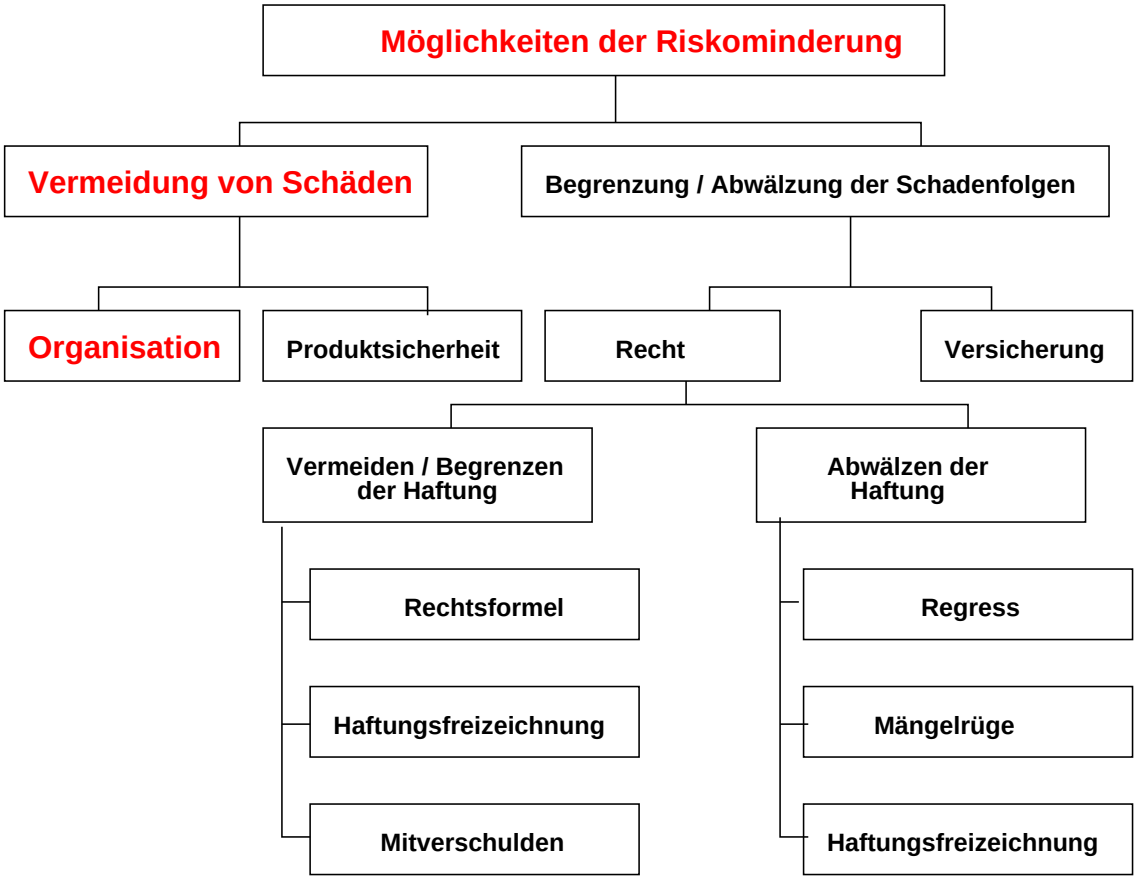


Abbildung 10.3: Risikominderung

Anforderungen an die Organisation:

Die Rechtssprechung des Bundesgerichtshofes zur Organisationsverantwortung bewertet das Erfüllen allgemeiner Sorgfaltspflichten nach den Erkenntnissen der betrieblichen Organisationslehre als Grundlage der Unternehmensorganisation. ([?])

Aufbau-, Ablauforganisation und Stellenbeschreibungen sind sich wechselseitig ergänzende Teile der Unternehmensorganisation. Sie bilden die dokumentierte Grundlage für die rechtlich so bedeutsame 'tatsächliche innerbetriebliche Verantwortung", die nach dem Strafrecht wichtigster Maßstab für das Bewerten von Handeln oder Unterlassen ist. ([?])

Qualitätsorganisation:

Die rechtlichen Anforderungen aus der Produkthaftung sind üblicherweise nur zu erfüllen durch eine ***unternehmensumfassende bereichsübergreifende Qualitätsorganisation****. Sie hat alle Vorgaben, Voraussetzungen, Maßnahmen und Abläufe zu erfassen, in oder bei denen sicherheits- / qualitätsrelevante Funktionen betroffen, ausgeführt oder darauf Einflüsse ausgeübt werden.* ([?])

Qualitätsmanagement

- Bestandteil der Unternehmensführung / Projektleitung
- Aufgaben können delegiert werden – notwendig dazu
  - Fachkompetenz
  - Entscheidungskompetenz
  - Ressourcen
  - Transparente Entwicklungsprozesse
- Muss unter unternehmerischen / wirtschaftlichen Gesichtspunkten gesehen werden
- Ist also nicht Selbstzweck und steht nicht in Konkurrenz zu Kosten / Terminen
  Was können wir uns leisten, was müssen wir uns leisten?

10.2 Software-Qualität

10.2.1 Qualitätsmodell

- nach ASQC (*American Society for Quality Control*), EOQC (*European Organization for Quality Control*), auch DIN 55 350:

  > Qualität ist die Gesamtheit von Eigenschaften und Merkmalen eines Produkts oder einer Tätigkeit, die sich auf deren **Eignung zur Erfüllung gegebener Erfordernisse** beziehen.

- Nach IEEE-Standard 729-1983:

  (1) The totality of features and characteristics of a software product that bear on its ability to satisfy given needs; e.g., conform to specifications.

  (2) The degree to which software possesses a desired combination of attributes.

  (3) The degree to which a customer or user perceives that software meets his or her composite expectations.

  (4) The composite characteristics of software that determine the degree to which the software in use will meet the expectations of the customer.

Ergo: Qualität ist **relativ**, nämlich relativ zu gegebenen Erfordernissen

Diese können sein:

- anwendungsspezifischer, funktionaler Art
- anwenderspezifische, also aus dem Benutzerprofil resultierende Erfordernisse
- Erfordernisse, die sich aus der Einhaltung von Gesetzen oder anderen Vorschriften ergeben (Datenschutzgesetze, Grundsätze ordnungsgemäßer Buchführung, Normen, . . .)

Qualität – Sichtweisen:

Abbildung 10.4: Qualität – Sichtweisen

Noch einmal – die mit Software verbundenen Prozesse:

Entwicklung	Weiter-Entwicklung	Pflege	Prüfung
Benutzung	Operating	Portierung	...

Qualitätsmerkmale ⇒ Eigenschaften,

- die einen bestimmten Prozess
- unter einer bestimmten Sichtweise
- **effektiv und effizient unterstützen**

Beispiel: Gebrauch (Merkmal: Gebrauchstauglichkeit)

- Wie gut lässt sich das Produkt (in seinem aktuellen Zustand) zur vorgesehenen Nutzung verwenden? Wie stark ist die Wirkungssteigerung in der Anwendung?
- Gebrauchstauglichkeit (Brauchbarkeit):
  Eigenschaften der Software, die sich auf die Erfüllung der vorgesehenen, spezifizierten und damit festgeschriebenen Erfordernisse aus Sicht der Nutzung / Benutzung beziehen.
- Software = .?. (Online-Hilfen & Co.)
  Der Umgang ist hier die Benutzung, also umfasst hier der Begriff Software alles, was z.Bsp. der Mensch als Benutzer an Anweisungen und Informationen zur effektiven Benutzung **zur Erreichung eines vorgegebenen Ziels** benötigt!

Beispiel: Pflege (Merkmal: Pflegbarkeit)

- Wie leicht lässt sich das Produkt modifizieren (Fehler und Mängel beseitigen, weiterentwickeln), um die aktuellen, sich gegebenenfalls veränderten Erfordernisse zu erfüllen?
- Pflegbarkeit (Wartbarkeit i.w.S., Entwicklungsfähigkeit)
  Eigenschaften der Software, die das Erkennen von Fehlern und Mängeln sowie deren Ursachen und die Durchführung von Korrekturen sowie die Durchführung von Änderungen zur weiteren Erfüllung geänderter Erfordernisse erleichtern.
- Software = .?. (Testdaten & Co.)
  Der Umgang betrifft hier die Modifikation eines Produktes, daher gehören hier zum Begriff Software alle Anweisungen und Informationen, die dies effektiv ermöglichen.

Beispiel: Portierung (Merkmal: Portabilität)

- Wie leicht lässt sich das Produkt in andere Betriebsumgebungen oder andere Anwendungsgebiete übertragen ?
- Übertragbarkeit (Portabilität i.w.S.)
  Eigenschaften der Software, die die Übertragung in andere Umgebungssysteme oder andere Anwendungsgebiete (technische oder funktionale Einsatzumgebungen) erleichtern.
- Software = .?.
  Der Umgang betrifft hier die Übertragung, daher gehören hier zum Begriff Software alle Anweisungen und Informationen, die dies effektiv ermöglichen.

NB: Qualität $\Longleftrightarrow$ Aufwand

Prozess- und **System**-Anforderungen
$\Downarrow$
Produkt-Anforderungen
$\Downarrow$
Qualitätsmodell hierarchisch strukturierte Merkmale

Gebrauchstauglichkeit:

Anwendungsprozess + Systemumgebung (Benutzerprofil, . . .)
$\Downarrow$
Erlernbarkeit / Handhabbarkeit / Effektivität
Konsistenz der Begriffe, Verständlichkeit der Fehlermeldungen, Gewohntes Look & Feel, . . .

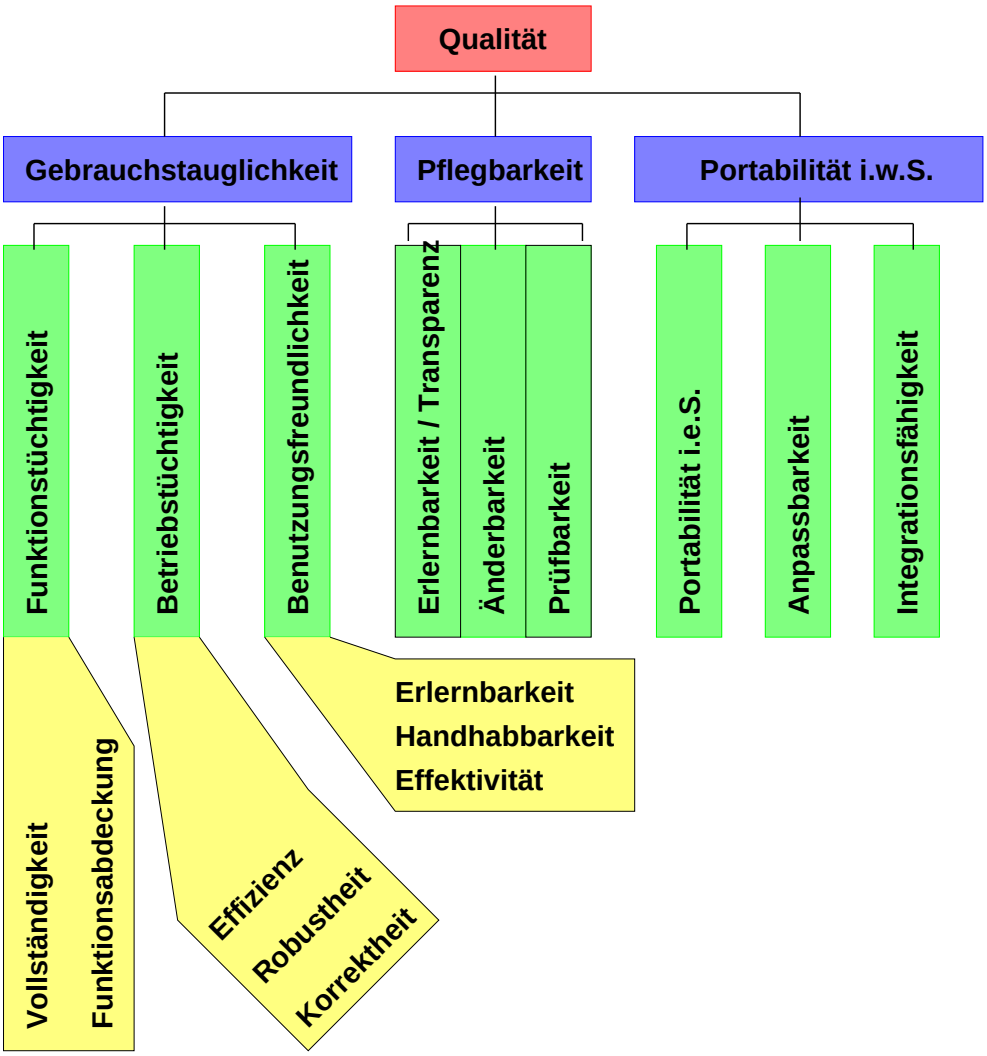


Abbildung 10.5: Ein „Qualitätsmodell“

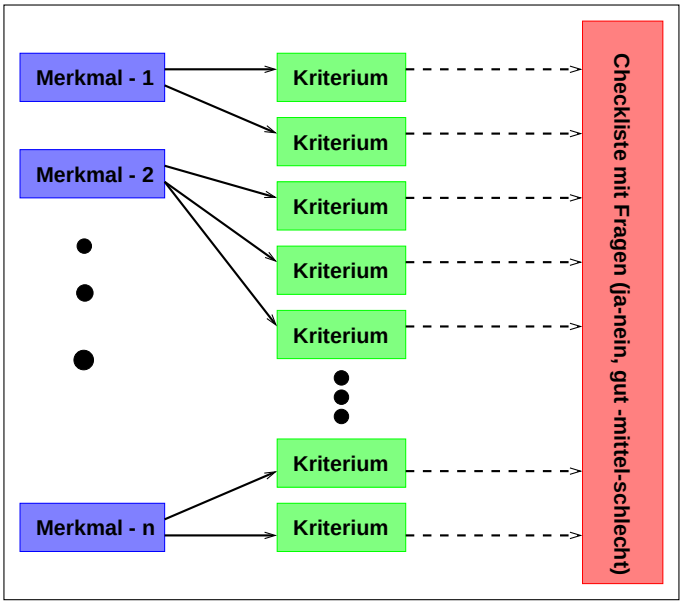


Abbildung 10.6: Qualität – Checklisten

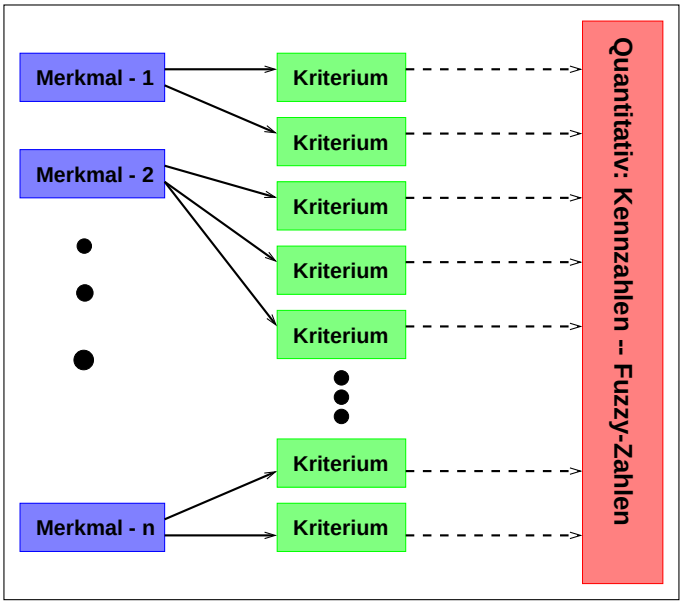


Abbildung 10.7: Qualität – Quantitativ

10.2.2 Quantitative Aspekte

Ziel: quantitative Aussagen (Kennzahlen, Indikatoren für Anomalien, Metriken, Maßzahlen) über

- die Qualität / einzelne Qualitätsmerkmale eines Software- (Zwischen-) Produkts
- die Qualität / einzelne Qualitätsmerkmale des Entwicklungs- / Pflege-Prozesses (schließt Fragen der Produktivität mit ein)

Als Grundlage für

- objektive Bewertung, Feststellung der Zielerreichung / Veränderung, Benchmarking
- Transparenz von (Veränderungs-) Prozessen (*Change Management*
- Entscheidungsfindung (z.B. Inbetriebnahme ja / nein)
- mathematische Modelle (Fehlerprognose, Zuverlässigkeit)
- quantifizierte Vorgaben (Zieldefinition) – **gefährlich!!!**

Begriffe: Maß, Metrik, Kennzahl

Anforderungen:

- **Validität:**
  Macht die Zahl wirklich eine Aussage über das gewollte Merkmal? („je größer die Zahl, desto besser / schlechter die Erfüllung eines Merkmals?“)
- **Verlässlichkeit:**
  Wiederholten „Messungen“ müssen gleiche Ergebnisse liefern!
- **Normierung:**
  Es muss eine eindeutige Abbildung auf eine Skala geben, die einen Vergleichsmaßstab („Urmeter“) darstellt!
- **Objektivität:**
  Die gewonnenen Ergebnisse müssen unabhängig sein von der Person des Prüfers, also frei von subjektiven Einflüssen
- und weitere mehr

Nicht absolut werten, sondern

- im Kontext mit anderen Kennzahlen (eine bestimmte Ausprägung **einer** Kennzahl kann im Prinzip immer erreicht werden, mehrere i.a. nur schwer)
- oder im Vergleich mit Erfahrungswerten
- oder über die Veränderung des Objekts (Programm, Prozess) hinweg („Zahlenreihe“)

interpretieren

Produkt-Kennzahlen – Messen der Programm-Größe

Beispiel 1: **LOC** (Lines Of Code)

Brechnung des GGT (Größter Gemeinsamer Teiler):

```
# include <stdio.h>

int main(){
  int x,y,x0,y0;
  printf("GGT zweier Zahlen bestimmen\n");
  printf("Erste Zahl: \en");
  if( scanf("%d",&x0) != 1 ) exit(1);
  printf("Zweite Zahl: \en");
  if( scanf("%d",&y0) != 1 ) exit(2);
  if( (x0 <= 0) || (y0 <= 0) ) exit(3);
  x = x0; y = y0;
  while(x != y)  if(x>y) x = x - y; else y = y - x;
  printf("GGT(%d, %d): %d\n", x0, y0, x);
  exit(0);
}
```

Anzahl „Zeilen“ (Textzeilen): **15** – etwas naiv!

Anzahl C-Statements (nach Sprachdefinition / Grammatik): **24**

Komplexitätskennzahlen

Aussagen über „vermutete“ oder „empirisch belegte“ Zusammenhänge:

- „Je höher die Komplexität, desto schlechter die Pflegbarkeit“
- „Je höher die Komplexität, desto höher die Fehlerhäufigkeit“

Beispiel 2: Zyklomatische Komplexität nach **McCabe**

$$v(G) = e - n + 2 \cdot p \text{ mit}$$

v(G): Komplexität eines Programms, ermittelt als zyklomatische Zahl des durch die Kontrollstruktur des Programms definierten Graphen.

e: Anzahl der Kanten im Kontrollfluss-Graphen **G**

n: Anzahl der Knoten in **G**

p: Anzahl der Zusammenhangskomponenten von **G** (i.d.R. $p = 1$)

Liefert die Anzahl unabhängiger Pfade durch den Kontrollflussgraphen (z.B. Untere Schranke für die Zahl der Testfälle für eine Pfadüberdeckung!)

In obigem GGT-Programm:

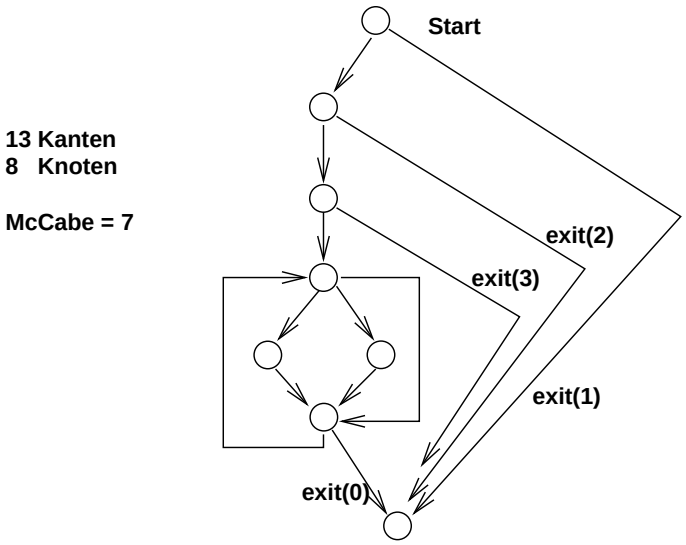


Abbildung 10.8: Kontrollfluss-Graph des GGT

Beispiel 3: Komplexitätsmaß nach Halstead

Basismessgrößen:

n_1:	Anzahl verschiedener Operatoren ($+, -, IF, WHILE, ->$, ...)
n_2:	Anzahl verschiedener Operanden (Variable, Objekte)
N_1:	Gesamtanzahl aller Operatoren
N_2:	Gesamtanzahl aller Operanden
$n = n_1 + n_2$:	Anzahl verschiedener Symbole
$N = N_1 + N_2$:	Anzahl aller Symbole

n wird auch als Vokabular und N als Länge des Programms bezeichnet. Als Programmvolumen definiert Halstead:

$$V = N \cdot log_2(n)$$

Die Komplexität eines Programms wird von Halstead beschrieben durch

$$D = \frac{n_1 \cdot N_2}{2 \cdot n_2}$$

In obigem GGT-Programm:

$$\begin{array}{rrl} N_1 = & 18 & \text{Operatoren insgesamt,} \\ n_1 = & 7 & \text{verschiedene Operatoren,} \\ N_2 = & 36 & \text{Operanden insgesamt,} \\ n_2 = & 8 & \text{verschiedene Operanden} \end{array}$$

Damit: $D = \frac{7 \cdot 36}{2 \cdot 8} \approx 16$

NB: Beide Größen sind sehr sprach- und programmstil-abhängig!

"Unnötige Variable" erhöhen Halstead, "kompakte" Programmierung senkt McCabe!

McCabe: **3** (vorher: 7) – Halstead: **33** (vorher: 16)

Anwendungen:

- Ein Modul hat McCabe = 58 und der Tester hat mit 25 Testfällen getestet!
- Version 1 eines Moduls hat McCabe = 58, Version 2 wurde unter großem Zeitdruck erstellt bei nur geringen funktionalen Änderungen – McCabe ist auf 187 abgestiegen!

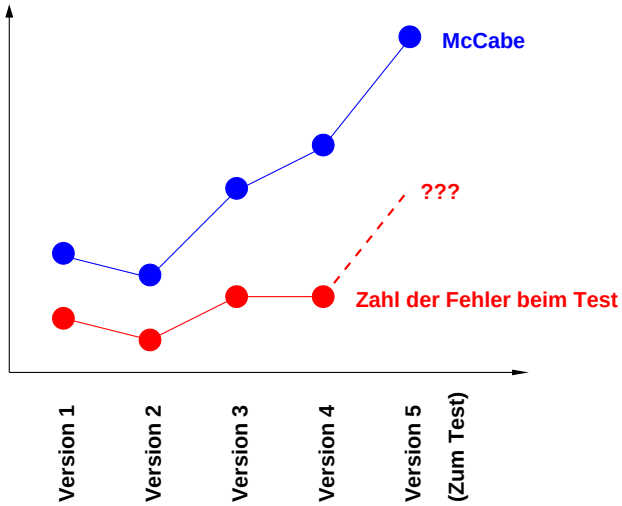


Abbildung 10.9: McCabe – Anwendung

Beispiel 4: Informationsflusskomplexität nach Henry/Kafura

- globaler Informationsfluss: Kommunikation über globale Datenstrukturen
- lokaler Informationsfluss über Prozeduren (Prozedur-Komplexität): Kommunikation über Parameter (evt. unterscheiden nach Art der Paramaterübergabe)
  - **fanIn:** Anzahl der via Parameter und via globale Daten in die Prozedur hineingehen
  - **fanOut:** Anzahl der via Parameter oder *return* oder via globale Daten aus der Prozedur herausgehen
  - **length:** Länge der Prozedur (incl. Kommentare)
  - **Kennzahl:** $length \cdot (fanIn \cdot fanOut)^2$
- Modul-Komplexität (Module, die Datenkapseln realisieren)
  read → nur lesend, *write* → nur schreibend, *read_write* → lesend und schreiben,

  Kennzahl:

  $$|write| \cdot |read| + |write| \cdot |read\_write| + |read| \cdot |read\_write| + |read\_write| \cdot (|read\_write| - 1)$$

Andere Produkt-Kennzahlen

- Anzahl Fehler (beim Test entdeckt und behoben) pro 1000 LOC
- Kommentierungsgrad (Anzahl Kommentarzeilen / Anzahl Code-Zeilen)
- Anzahl von Änderungen pro Modul (beim Übergang zur nächsten Version):
  Anzahl hinzugekommener + gelöschter + geänderter LOC
- Anzahl Testfälle / Modul
- McCabe'sche Zahl des Prozeduraufruf-Graphen
- und viele andere mehr

NB:

- Kennzahlen sollen helfen, erklären und nicht nur Phänomene beschreiben!
- Quantitative Zielvorgaben sind mit Vorsicht zu wählen, sie können u.U. leicht erfüllt werden, ohne dass sich tatsächlich etwas ändert!

  Bsp.: *Gute Programme haben eine geringe zyklomatische Komplexität, "McCabe* $<= 10$*"*
  Merke: Kennzahlen operieren auf rein syntaktischer Ebene und sagen nicht aus über die Semantik!

- Kennzahlen, die zur „Leistungsbewertung" von Mitarbeitern herangezogen werden, sind gefährlich, da sie u. U. die Leistungsbereitschaft unterminieren.

  Bsp.: Zahl der Fehler bei einer Code-Inspektion: „je höher, desto schlechter der Autor des geprüften Objekts"
  Code-Inspektionen sollen gerade Fehler finden, verlangen eine positive Einstellung zum Fehler

  Bsp.: Zahl der Fehler bei einer Code-Inspektion: „je höher, desto besser hat das Team garbeitet"
  Kein Problem – „wenn kein Fehler enthalten ist, so kommt eben einer rein"

10.3 Analytische Maßnahmen

10.3.1 Zielsetzung und Voraussetzungen

Ziele:

- die fertigen Produkte (Teilprodukte) hinsichtlich ihrer Qualität überprüfen
- Erkennung, Lokalisierung und Beseitigung von Fehlern
- Bewertung der Qualität

Voraussetzungen:

- Prüfgegenstände müssen prüfbar und so konstruiert sein, dass Aspekte leicht lokalisierbar und änderbar sind
- Systematik in der Vorgehensweise
- Reproduzierbarkeit
- „Wirtschaftlichkeit“: mit möglichst geringem Aufwand möglichst viele der Fehler aufdecken

Analytische Maßnahmen:

- Prüfungen zur Feststellung von Fehlern und Mängeln
  - formale Prüfungen (Syntax, Stil, . . .) und
  - inhaltliche Prüfungen (Semantik, Pragmatik)
- Nachweisprüfungen, also Messungen bezüglich vorgegebener quantitativer Kriterien und Vergleich mit Vorgaben

Wichtig: Prüfziele und Prüfkriterien **vorher** festlegen!

- **Explizite Prüfkriterien** sind Anforderungen, die auf irgendeine Art formuliert sind. Darunter fallen funktionale Anforderungen, Leistungsvorgaben, Modularisierungskriterien, Sicherheit gegen unerlaubte Ein-/Ausgabe, Einhaltung von vereinbarten Richtlinien, u. ä.
- **Implizite Prüfkriterien** können sein: Verständlichkeit von Aussagen, das Fehlen von logischen Inkonsistenzen wie Deadlocks, nicht-ausführbare Code-Segmente, Handhabungscharakteristiken von Programmen, u.ä.

10.3.2 Methoden und Verfahren

Begriffe: (z. B. nach IEEE-Std. 1028)

- **Management-Reviews**: Vergleich Projektstatus mit Projektplan
- **technische Reviews**: Prüfung und Bewertung eines Software-Elements (Entwicklungs-Ergebnis)
- **Software-Inspektionen**: Im Vordergrund steht Suche nach Fehlern in einem Software-Element
- **Walkthrough**: Gruppe geht zusammen mit Autor ein Software-Element Schritt für Schritt durch (z. B. anhand von Testfällen), sammelt Fehler, Änderungs- und Verbesserungsvorschläge
- **Audits**: Überprüfung der Einhaltung von Vorgaben, Standards und Richtlinien

---

Manchmal auch so:

- Inspektion:
  Eine statische Analysetechnik, die in der visuellen Untersuchung von Entwicklungsprodukten besteht, mit dem Ziel, Fehler, Verletzungen von Entwicklungsstandards oder sonstige Probleme aufzudecken. z. B. Code-Inspektion, Design-Inspektion.
- Review:
  Ein Prozess oder eine Sitzung, während dessen ein Arbeitsergebnis oder eine Gruppe von Arbeitsergebnissen Projektmitgliedern, Managern, Anwendern, Kunden oder anderen interessierten Parteien zum Zweck der **Kommentierung, Abstimmung oder Zustimmung** vorgelegt wird.

Software-Inspektion:

Prufung eines

- **„wohl-definierten und wohl-proportionierten" Objekts (↪ Konstruktion!)** bezüglich
- **klar definierter Prüfziele** (Korrektheit und Vollständigkeit bezgl. Vorgaben, Pflegbarkeit, u. dgl.) in einem
- **kleinen Team** mit
- **definierten Rollen** in
- **definierter Vorgehensweise**

Teammitglieder (Rollen):

- **Moderator** als Planer, Lenker, Koordinator, Prüfer
- **Autor** als Verantwortlicher für die Qualität des Prüfgegenstandes, um Unklarheiten zu beseitigen, um zu helfen, Mängel aufzudecken, ...
- **Leser** als derjenige, der das Objekt nach einem zuvor definierten „roten Faden" in der Team-Prüfung vorträgt, interpretiert,
- **ein weiterer Prüfer**, abhängig vom Prüfgegenstand und den Prüfzielen

Prinzipieller Ablauf:

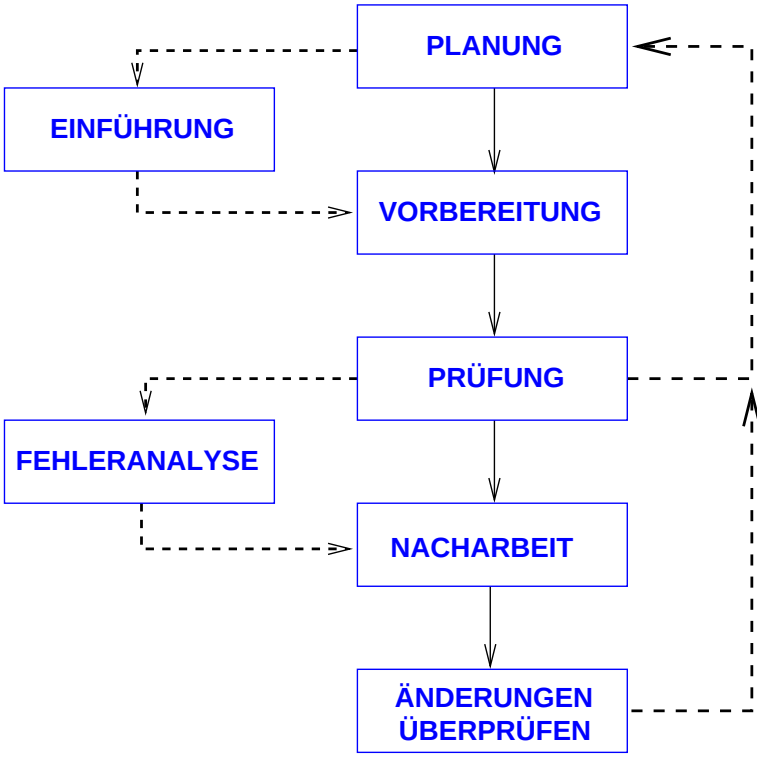


Abbildung 10.10: Ablauf einer Inspektion

Planzahlen für Programm-Inspektionen („Daumen mal pi“):

Aufwand für ca. 250 NLOC in MStd

	Moderator	Autor	Leser	4.Prüfer
Planung	0.5	1.0		
Einführung	0.5	0.5	0.5	0.5
Vorbereitung	2.0		2.0	2.0
Prüfung	**2.0**	**2.0**	**2.0**	**2.0**
Nacharbeit	0.5	0.5	0.5	0.5
Korrektur		X		
Überprüfung	0.5	0.5		

Für Dokument-Inspektionen (ca. 20 Seiten in MStd):

	Moderator	Autor	Leser	4.Prüfer
Planung	1.0	1.0		
Einführung	1.0	1.0	1.0	1.0
Vorbereitung	3.0		3.0	3.0
Prüfung	**2.0**	**2.0**	**2.0**	**2.0**
Nacharbeit	0.5	0.5	0.5	0.5
Korrektur		X		
Überprüfung	1.0	1.0		

One of the best ways of learning about inspections is actually doing it! (M. Fagan)

Vorteile:

- effektive Methode zur Fehler-/Mängelfindung
- Test: Fehlersymptome werden entdeckt, danach beginnt Fehlersuche
- Fehler oftmals in den Vorgaben, nicht im Prüfobjekt
- auch Schulungsmaßnahme (Know-How-Transfer)
- Förderung von Teamdenken und Kommunikation
- ...

Andere statische Verfahren

- **Programm-Verifikation:**

  Mathematischer Korrektheitsbeweis – verlangt: formale Spezifikation, formale Definition der Semantik der verwendeten Programmiersprache u. a. m.

- **Abbildungsmethode:**

  Abbildung von verbal beschriebenen Spezifikationen, aber auch von Programmen auf grafische / formale Strukturen zur Offenlegung der zu analysierenden Aspekte (Abstraktionsprinzip). Statt „alles auf einmal“ Konzentration auf fehlerträchtige Aspekte!

  Darstellungsmöglichkeiten dazu können sein:
  Datenflussgraphen, Data Dictionaries, Erreichbarkeitsgraphen, Aufrufgraphen, Petri-Netze, Flussdiagramme, Entscheidungstabellen, formal-mathematische Sprachen

- **Symbolische Ausführung:**

  Unter der symbolischen Ausführung von Programmen versteht man das „Durchrechnen“ der Quellprogrammen mit Symbolen statt mit konkreten Werten. Logische Programmbedingungen führen dabei zu Prädikaten, die ebenfalls über Symbolen gebildet werden.

- **Datenflussanalyse:**

  Die Datenflussanalyse eines Programms untersucht u.a. den zeitlichen Zustand der Programmdaten an ausgezeichneten Stellen (z. B. Schnittstellen) und bezieht sich weniger auf den Kontrollfluss und die Ablauflogik. Im Mittelpunkt steht also der „Lebenszyklus“ der Daten, ihre Veränderung während der Laufzeit.

Dynamische Prüfungen: Simulation und TEST

Kapitel 11

Test

11.1 Zielsetzung

- gezielte Fehlersuche im „fertigen“ Programm
  (Modul, Teil-System, Gesamt-System)

  (1) Testfälle (Testszenarien) entwickeln und Test durchführen (40%)
  (2) Fehler lokalisieren (20%)
  (3) Fehler beheben (neue Fehler?) (30%)
  (4) „Behobene“ Fehler (oder gar alles?) erneut testen (10%) – Regressionstest

- Überzeugen, dass das Programm korrekt arbeitet?
  Dijkstra: Mit Testen kann man nur die Anwesenheit, nicht die Abwesenheit von Fehlern zeigen!
- Entscheidungsproblem: Verkürzung der Testzeit / des Testaufwandes vs. Aufwand mit Kunden-Problemmeldungen

  Testen ist ein Prozess, ein Programm auszuführen, mit der Absicht, Fehler zu finden!
  ([?])

11.2 Test-Prinzipien

Was kann ich mir leisten, was muss ich mir leisten?

- Testen muss feststellen, ob ein Programm, das was es tun soll, nicht tut, und ob es das, was nicht tun soll, am Ende doch tut!
- Dazu müssen Testeingangsdaten, Anfangszustand, erwartete Ergebnisse und erwarteter Endzustand definiert werden!
- Tester und Autor des Programms müssen verschiedene Personen sein! (Testen — ein destruktiver Prozess)
- Testorganisation und Programmierorganisation müssen in der Linie getrennt sein!
- Ein Test muss geplant sein! (Aufwand)
- Ein Test muss (sollte) reproduzierbar sein!

Besonderheiten bei Software:

- ...
- weist keine "Stetigkeit" auf (kommt mit 1 und mit 10 das richtige Ergebnis, so nicht notwendig bei 5)

Beispiel: $f(x, y) = 9 * x^4 - y^4 + 2 * y^2$

Anm.: $f()$ weist keine Besonderheiten auf, ist insbesondere stetig!

Testdaten für das (C-) Programm "a.out":

```
1 1
10864 18817
20000 20000
```

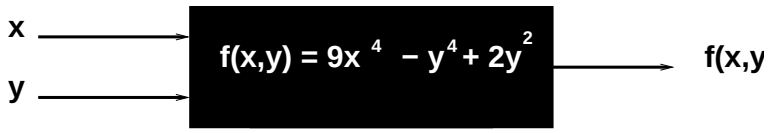


Abbildung 11.1: Black-Box

Ergebnisse:

1.000000000000e+01	2.000000000000e+00	1.280000034888e+18

Wie können die Soll-Ergebnisse bestimmt werden? ↪ diversitär!

Alternativ-Ergebnisse:

1.000000000000e+01	0.000000000000e+00	1.280000000800e+18

Vielleicht besser ganzzahlig gerechnet? ↪ **unsigned long int** in C

10	1	271517696

Wer weiß, ob das stimmt? ↪ Java-Klasse **BigDecimal** liefert:

10	1	1280000000800000000

11.3 Test-Arten

↪ Unterscheidung nach der Vorgehensweise bei der Testfallermittlung

Black-Box-Test

Bei diesem Test wird der Prüfling als „Schwarzer Kasten“ betrachtet. Der Prüfer ist nicht am internen Verhalten und an der inneren Struktur interessiert, sondern nur daran, Umstände zu finden, unter denen das Programm in seinem Gesamtverhalten von der Spezifikation abweicht. Die Testdaten werden nur aus der Spezifikation, nicht aber aus der internen Struktur abgeleitet.

White-Box-Test (Strukturtest)

Bei diesem Test werden die Testdaten aus der internen Struktur des Prüflings abgeleitet (aber unter Berücksichtigung der Spezifikation für den Soll-Ist-Vergleich). Hier liegt die Überlegung zugrunde, dass Programmfehler als Ursache fehlerhafte Strukturelemente (Ausdrücke, Statements, Bedingungen, . . .) haben.

Dies kann so geschehen, dass Testdaten mit dem Ziel definiert werden, so dass z. B. alle Programmpfade durchlaufen werden. Die Tatsache, dass ein Programmpfad fehlt, lässt sich allerdings aus der internen Struktur allein nicht ableiten.

Zur Wertung wird die Spezifikation benötigt!

11.4 Testfall-Entwurf

11.4.1 Generelles

Testfälle so wählen, dass die Wahrscheinlichkeit, Fehler zu finden, möglichst groß ist (Erfahrungswerte über „frühere Fehler“)

Notwendig dazu:

- Fundierte Kenntnis darüber, was das Programm tun soll und was nicht! (“Anwendungs-Know-How”)
- Gute Kenntnis der Basissysteme und ggf. der Schnittstellen zu anderen Systemen
- Systematik in der Vorgehensweise
- Transparente Aufbereitung (Testabdeckung)

Methoden des Testfallentwurfs:

- Black-Box-Test:
  - Bilden von Äquivalenzklassen
  - Grenzwertanalyse
  - Ursache-Wirkungsgraph
  - „Fehlererraten“ (*error guessing*)
    ↪ Erfahrung ist durch nichts zu ersetzen
- White-Box-Test: (bezogen auf Strukturelemente der jeweiligen Sprache(n))
  - Erfassen aller Anweisungen
  - Erfassen aller Entscheidungen
  - Erfassen aller Bedingungen
  - Erfassen aller Kombinationen von Bedingungen
  - . . . (weitere Strukturelemente)

11.4.2 Black-Box-Test – Ein (kleines) Beispiel

Gegeben sei folgende Spezifikation für ein kleines Programm:

> Das Programm erhält als Eingabe drei ganze Zahlen (jeweils mit *return* abzuschließen), die als Seitenlängen eines Dreiecks interpretiert werden. Das Programm ermittelt, ob diese Zahlen ein gleichseitiges, gleichschenkliges oder allgemeines Dreieck bestimmen!

Aufgabe:
Ermitteln von Testfällen durch Äquivalenzklassenbildung!

Abkürzungen für Aufstellung in Tabelle (Übersichtlichkeit):

- a die 1. Eingabe, b die 2. und c die 3.
- GZ: „ganzzahlig“,
- dr: $(a + b > c)$ & $(a + c > b)$ & $(b + c > a) \hookrightarrow$ (die Summe von zwei Seiten ist größer als die dritte)
- $MaxCard$: die maximal darstellbare pos. ganze Zahl
- $ZAHL$: korrekt gebildete Zahl (ganzzahlig oder reell)
- $a, b, c\ GZ$ & $a, b, c > 0$ ist zu lesen als: „a und b und c ganzzahlig und größer Null“
- $NOT(a, b, c\ GZ)$ ist zu lesen als: „Nicht (a und b und c ganzzahlig)“, also „a oder b oder c nicht-ganzzahlig)“

Äquivalenzklassen:

Klasse	Beschreibung	SOLL
1	a,b,c GZ & >0 & dr & (a#b#c) & $<=$ MaxCard	allgemein
2	a,b,c GZ & >0 & dr & (a=b=c) & $<=$ MaxCard	gleichseitig
3	a,b,c GZ & >0 & dr & NOT (a#b#c) & NOT (a=b=c) & $<=$ MaxCard	gleichschenklig
4	a,b,c GZ & >0 & NOT dr & $<=$ MaxCard	kein Dreieck
5	a,b,c GZ & >0 & dr & NOT $<=$ MaxCard	Bereichsüberschr.
6	a,b,c GZ & NOT >0	keine Seitenlänge
7	a,b,c NOT GZ & >0 & dr	unzul. Eing.
8	a,b,c NOT ZAHL	"unzul.Eing."

konkrete Testfälle:

Kl.	Nr.	Testeingangsdaten
1	1	3 , 4 , 5
	2	MaxCard-2,MaxCard-1,MaxCard
2	3	3 , 3 , 3
	4	MaxCard , MaxCard, MaxCard
3	5	3 , 3 , 4
	6	3 , 4 , 3
	7	4 , 3 , 3
	8	MaxCard-1, MaxCard-1, MaxCard
	9	MaxCard-1, MaxCard, MaxCard-1
	10	MaxCard, MaxCard-1, MaxCard-1
4	11	1 , 2 , 3
	12	3 , 1 , 2
	13	1 , 3 , 2
	14	1 , 2 , 4
	15	1 , 4 , 2
	16	4 , 1 , 2
	17	MaxCard , 1 , 1
5	18	MacCard, MaxCard+1, MaxCard
6	19	0 , 0 , 0
	20	-2 , 2 , 2
	21	2 , -2, -2
	22	-2 , -2 , -2
7	23	3.14 , 3.14 , 3.14
8	24	# , a , ?
	25	3 , & , _

Fazit:

- Testfallklassen ermitteln heißt „die Spezifikation interpretieren“
  (Seitenlängen sind positiv!)
- Testfallklassen ermitteln verlangt Verständnis der Anwendung (Dreiecksungleichung) und der Basissysteme wie Hardware, Betriebssystem oder Compiler (*MaxCard*)
- Testfallklassen ermitteln verlangt Transparenz in der Aufschreibung (Vollständigkeit?)
- Konkrete Testfälle können u.U. mehrere Testfallklassen ansprechen (Reduktion der Zahl der durchzuführenden Tests)

11.4.3 error guessing – Fehlererwartungs-Methode

(1) Dem Tester liegt eine Liste von möglichen Fehlern oder fehlerträchtigen Situationen vor, die bei der Erstellung der Software auftreten können (Erfahrungswerte).

(2) Der Tester versucht nachzuvollziehen, welche Überlegungen der Entwickler bei der Interpretation der Vorgaben angestellt hat und welche Situationen er möglicherweise übersehen, nicht berücksichtigt oder falsch interpretiert hat ($\hookrightarrow$ Liste von Testfällen)

Beispiel: Sortierprogramm

- Datei nicht vorhanden oder Zugriffsrechte nicht in Ordnung
- Datei leer
- Datei bereits sortiert
- Datei enthält nur gleiche Objekte
- Datei in anderer Ordnung sortiert

11.4.4 White-Box-Test

- Überlegung: Wenn Fehler enthalten sind, so haben sie ihre Ursache in fehlerhaften Statements im Programm.
- Die Herleitung der Testdaten erfolgt beim White-Box-Test aus der inneren Struktur des Testlings. Welche Strukturelemente herangezogen werden, hängt zum einen von der Programmiersprache, zum anderen auch von „schlechten Erfahrungen“ ab.
- Wertung: $\hookrightarrow$ Spezifikation
- Test-Vorgabe:
  - x% aller Anweisung sind mindestens einmal anzusprechen (x% **C0-Überdeckung**)
  - x% aller Pfade sind mindestens einmal anzusprechen (x% **C7-Überdeckung**)

Überdeckungsmaße (abhängig von Sprache):

S_a x% aller Anweisungen **(C0)**

S_b x% aller Zweige, insbesondere Entscheidungen mit den möglichen Ausgängen incl. Schleife keinmal bzw. mindestens einmal durchlaufen
(`IF-THEN-ELSE-FI, CASE, DO-WHILE, computed GOTO, ...`) **(C1)**

S_c wie S_b und zusätzlich bezogen auf die elementaren Bedingungen innerhalb von Entscheidungen (`IF (A AND (B OR C) ...`)

S_d wie S_b und zusätzlich bezogen auf die Kombination der Werte der elementaren Bedingungen innerhalb von Entscheidungen

s_e wie S_b und zusätzlich bezogen auf mehrfache Schleifendurchläufe

S_f wie S_b und zusätzlich alle datenabhängigen Kombinationen von Zweigen

S_g x% aller Pfade **(C7)**

White-Box-Test: Ein kleines Beispiel

Gegeben sei folgendes Flussdiagramm als Spezifikation:

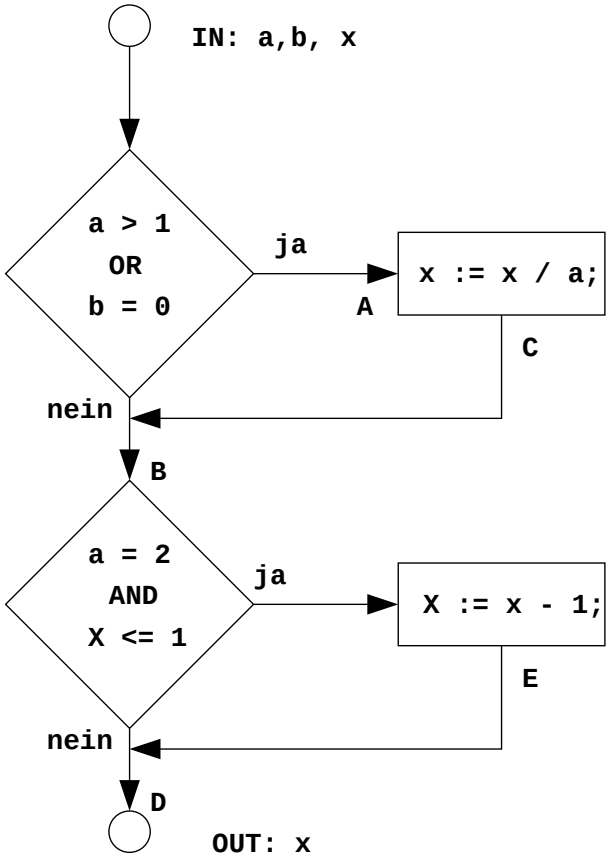


Abbildung 11.2: Beispiel zum White-Box-Test

```
(* Fehlerhafte Implementierung *)
IF (a > 1.0) AND (*statt OR*) (b = 0.0)
THEN
     x := x/a;
END;
IF (a = 2.0) OR (*statt AND*)
           (x > 1.0) (*statt x <= 1*)
THEN
     x := x - 1.0;
END;
```

White-Box-Test:

Art	a	b	x	Soll	Ist	"Weg"	Bedingungen
a	2	0	1	-0.5	-0.5		
b	2	0	1	-0.5	-0.5	ACE	
	1	1	1	1	1	ABD	
c	2	0	1	-0.5	-0.5	ACE	TTTF
	1	1	1	1	1	ABD	FFFF
	1	0	2	2	1	ABE	FTFT
d	2	0	1	-0.5	-0.5	ACE	TTTF
	1	1	1	1	1	ABD	FFFF
	1	0	2	2	1	ABE	FTFT
	2	2	2	0	1	ABE	TFTT
g	1	1	1	1	1	ABD	
	2	2	2	0	1	ABE	
	4	0	2	0.5	0.5	ACD	
	2	0	1	-0.5	-0.5	ACE	

11.4.5 Erstes Fazit

- Der Aufwand der Testfallermittlung beim White-Box-Test kann sehr hoch sein!
- Der Wert kann sehr zweifelhaft sein!
- Hier wird nur berücksichtigt, was vorhanden ist – fehlende Statements bleiben unberücksichtigt!

Folgerung:

1. Black-Box-Test hat Vorrang!
2. Sollte dabei eine nur geringe Programm-Überdeckung festgestellt werden, so kann der White-Box-Ansatz folgen!

11.5 Der Test-Prozess

- Test-Planung: Festlegung von
  - genereller Vorgehensweise
  - Testkonfiguration
  - Ressourcen
    - Personal (insb. Know-How)
    - Tools
    - Hardware
  - Einbettung in den Entwicklungsprozess (Zeitplan)
  - ...
- Spezifikation der Tests
  - Entwurf: Zerlegung in testbare Einheiten
    ↪ ggf. einen Teilprozess mit gleicher Struktur aufsetzen
  - Kriterien für "Bestanden / Nicht bestanden" festlegen
  - Bezugsdokumente definieren
- Testfallklassen / Szenarien ermitteln
- Test-Prozeduren festlegen
  - was ist in welcher Reihenfolge zu testen?
  - wie ist zu dokumentieren? (Protokoll)
- Bewertung / Regressionstests

Wichtige Voraussetzungen:

- **geregelter** Entwicklungsprozess
- Enge Kopplung zwischen Entwicklungs- und Testprozess
- Klares Konfigurationsmanagementsystem
- *Human Resources*: Hoher Anspruch an Tester (Testfallermittlung)
- Hilfsmittel zur Dokumentation (Beschreibung der Testszenarien, Erstellen der Testprotokolle, ...)
- ...

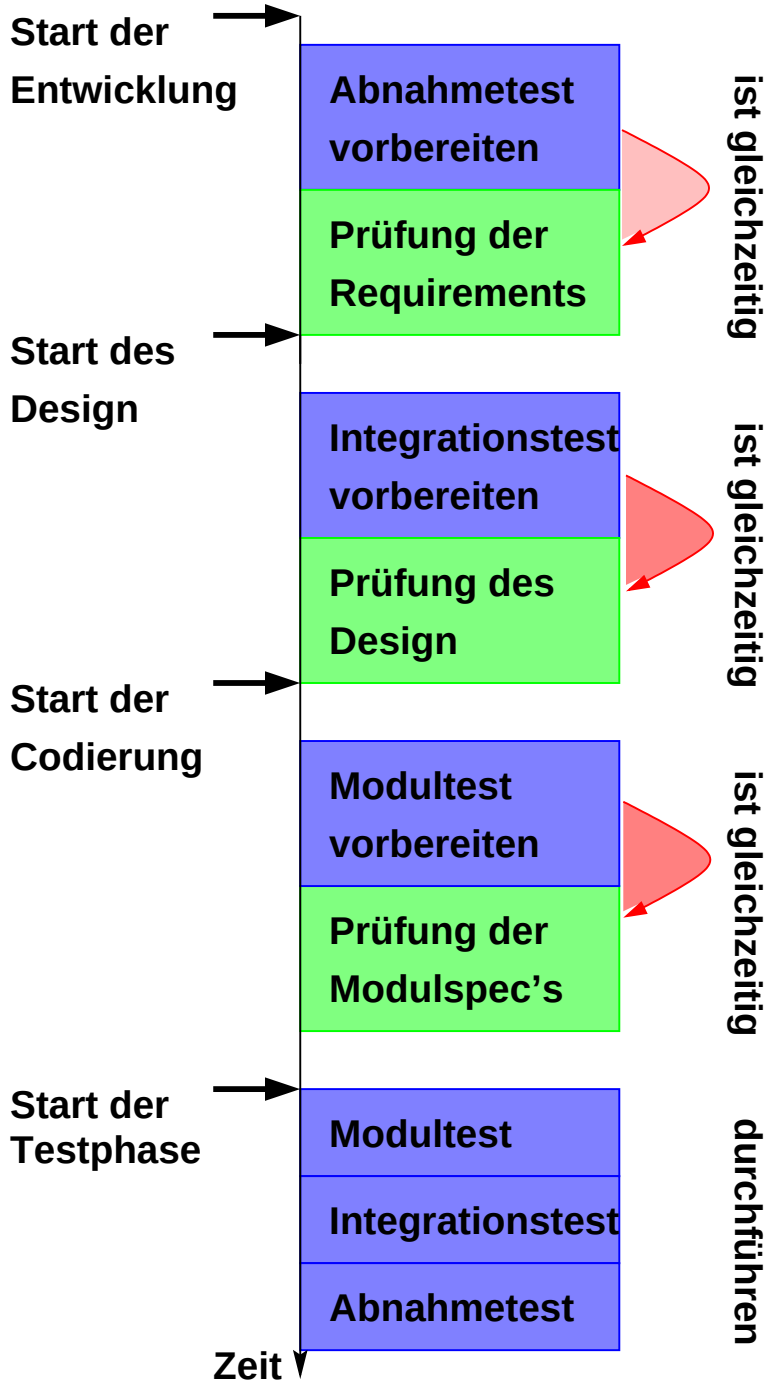


Abbildung 11.3: Der Test-Prozess

11.6 Klassifikationsbaum-Methode

11.6.1 Das Verfahren

Die grundsätzliche Idee der Klassifikationsbaum-Methode ist es, zuerst die Menge der möglichen Eingaben für das Testobjekt getrennt auf verschiedene Weisen unter jeweils einem geeig-

neten Gesichtspunkt zu zerlegen, um dann durch Kombination dieser Zerlegungen zu Testfällen zu kommen ([?])

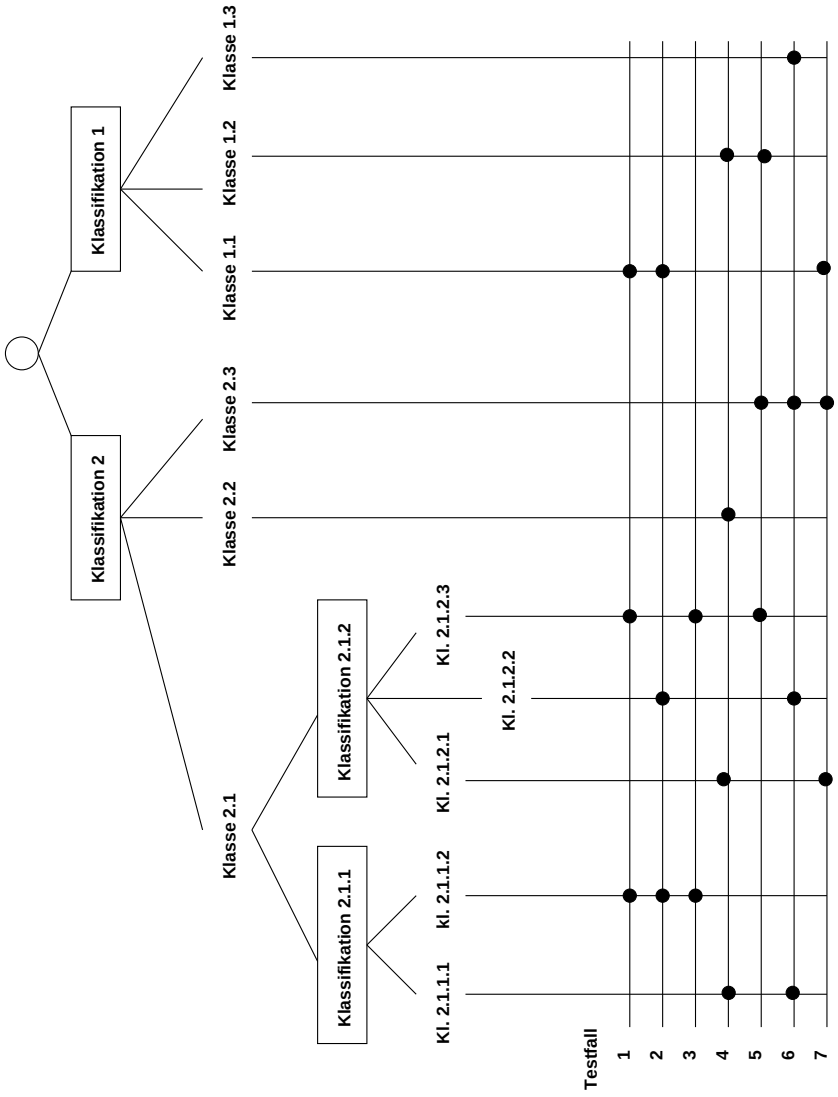


Abbildung 11.4: KLassifikationsbaum – Grundstruktur

- Welche Aspekte sind für den Test relevant?
- Wie können diese sukzessive sinnvoll zerlegt werden – von generellen hin zu elementaren Aspekten?
- Wie werden die elementaren Aspekte zu einem Testfall kombiniert?
- Wie können die Testfälle transparent dargestellt werden?

Ein einfaches Beispiel: Prämienberechnung

Spezifikation:

- $15 <= \textit{Dienst jahre} < 25$: 10% Prämie
- $25 <= \textit{Dienst jahre} < 40$: 30% Prämie
- $40 <= \textit{Dienst jahre}$: 100% Prämie
- mindestens 1 minderjähriges Kind, unabhängig vom Dienstalter: 3%
- "Gesundheitsprämie" (keinen Tag im aktuellen Jahr krank): 5%, unabhängig vom Dienstalter

Klassifikationsbaum:

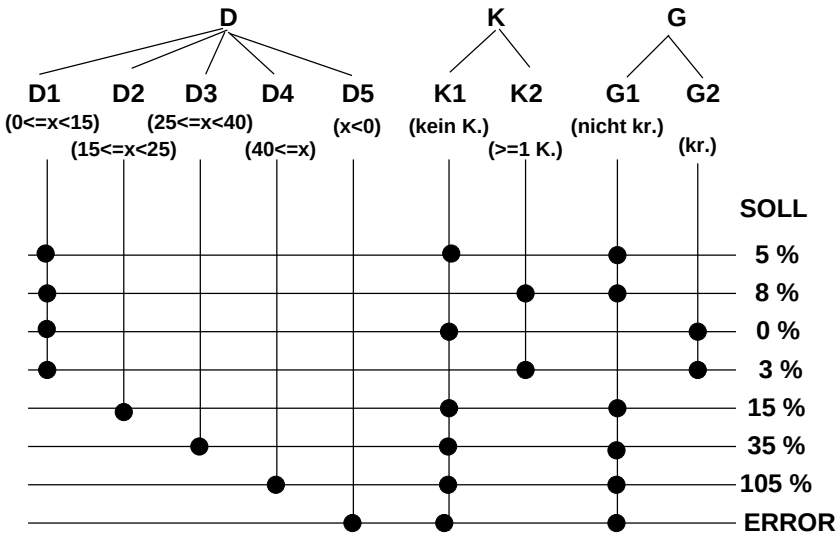


Abbildung 11.5: Klassifikationsbaum – Prämienberechnung

11.6.2 Beispiel Blinkersteuerung

nach [?]

Spezifikation der Blinkersteuerung (unvollständig!):

Die Blinkersteuerung sorgt dafür, dass die Fahrtrichtungsanzeiger eines PKWs entsprechend der Spezifikation funktionieren. Hierbei müssen verschiedene Formen des Blinkens wie Richtungsblinken, Warnblinken und Crash-Blinken berücksichtigt und entsprechend ihrer Priorisierung durchgeführt werden.

1. Richtungsblinken:

Dies erfolgt auf Anforderung vom EZS (Elektronisches Zündschloss), welches den Lenkstockschalter einliest.

Blinkfrequenz:	1.5 Hz
Tastverhältnis hell / dunkel:	52 zu 48%

Die Funktion ist nur aktiv, wenn der Zündschlüssel auf Stellung *eins*, *zwei* oder *drei* ist. Wird der Ausfall eines Fahrtrichtungsanzeigers erkannt, verdoppelt sich die Frequenz der Blinkerkontrollleuchten im Kombi-Instrument.

2. Warnblinken:

Erfolgt auf Anforderung vom OBF (Oberes Bedienfeld), welches den Warnblinkschalter einliest.

Blinkfrequenz	1.5 Hz
Tastverhältnis hell / dunkel	
Zündschlüssel auf Pos. 1, 2 oder 3	52 zu 48%
sonst	35 zu 65%

Die Kontrollleuchte im Warnblinkschalter spricht nur an, wenn das Bit “Warnblinklicht aktiv” vom Steuergerät SAM-H (Signalerfass- und Ansteuer-Modul – Heck) gesetzt ist.

3. Crash-Blinken:

Bei einem detektierten Crash werden die Warnblinker wie zuvor beschrieben angesteuert. Es ist zu gewährleisten, dass bei Beschleunigung unter 3g keine Auslösung erfolgt. Deaktiviert wird Crash-Blinken durch zweimaliges Betätigen des Warnblinkschalters (entscheidend: "Warnblinklicht ein" Bit hat Flankenwechsel von 1 nach 0).

4. Priorisierung:
Die dargestellte Priorisierung bezieht sich nur auf die Fahrtrichtungsanzeiger.

Funktion	Priorität
Crash-Blinken	1
Warnblinken	2
Richtungsblinken	3

Vorgehensweise zur Testfallermittlung:

- **Schritt 1: Definition des Eingabedatenraums**
- **Schritt 2: Klassifikation des Eingabedatenraums**
- **Schritt 3: Definition der Testfälle**

Schritt 1: Definition des Eingabedatenraums

Wurzel des Klassifikationsbaumes: Gesamter Eingabedatenraum des Testobjekts

- Bei der Blinkersteuerung gegeben durch die physikalischen Eingänge der beteiligten Steuergeräte, implizit in der Sepzifikation genannt
- Die Steuergeräte erhalten über ihre Schnittstellen Informationen über die Stellung des Lenkstockschalters, des Zündschlüssels und des Warnblinkschalters
- Dazu kommen Informationen, die der Fahrer nicht direkt einstellen kann, sondern die durch die Kfz-Elektronik (z.B. Sensoren) detektiert werden (Ausfall eines Fahrtrichtungsanzeigers, Beschleunigung bei Crash-Blinken, Position des „Warnblinklicht ein“-Bits)

Schritt 2: Klassifikation des Eingabedatenraums

1. Definition von Aspekten
2. vollständige Zerlegung des Eingaberaums in disjunkte Klassen unter jedem einzelnen Aspekt

Diese beiden Teilschritte werden zunächst auf die Wurzel des Klassifikationsbaumes angewandt, können jedoch auch anschließend iterativ jeweils für die einzelnen Klassen des Baumes durchgeführt werden.

Teilschritt 2.1: Definition von Aspekten

- Lenkstockschalter
- Position des Zündschlüssels
- Warnblinkschalter
- Ausfall Blinklicht
- Crash-Sensor

Die Unterscheidung nach den Bedienelementen bzw. den Sensoren, die auf das Testobjekt Einfluss haben, lässt ein systematisches Vorgehen bei der Erstellung eines Klassifikationsbaumes zu.

Um den Überblick über die Klassifikationen, deren Anzahl bei komplexen Funktionen sehr groß werden kann, nicht zu verlieren, können mehrere zu einer zu testenden Teilfunktion gehörende Klassifikationen zusammengefasst werden.

Zu Dokumentationszwecken kann eine weitere Klassifikation „Priorisierung“ eingeführt werden – dient lediglich dazu, anzuzeigen, ob nur **eine** Blinkanforderung im jeweiligen Testfall aktiv ist oder ob **mehrere** Anforderungen gemäß Priorisierungsvorschrift berücksichtigt werden müssen.

Klassifikationsbaum für die Blinkersteuerung ohne Klassen:

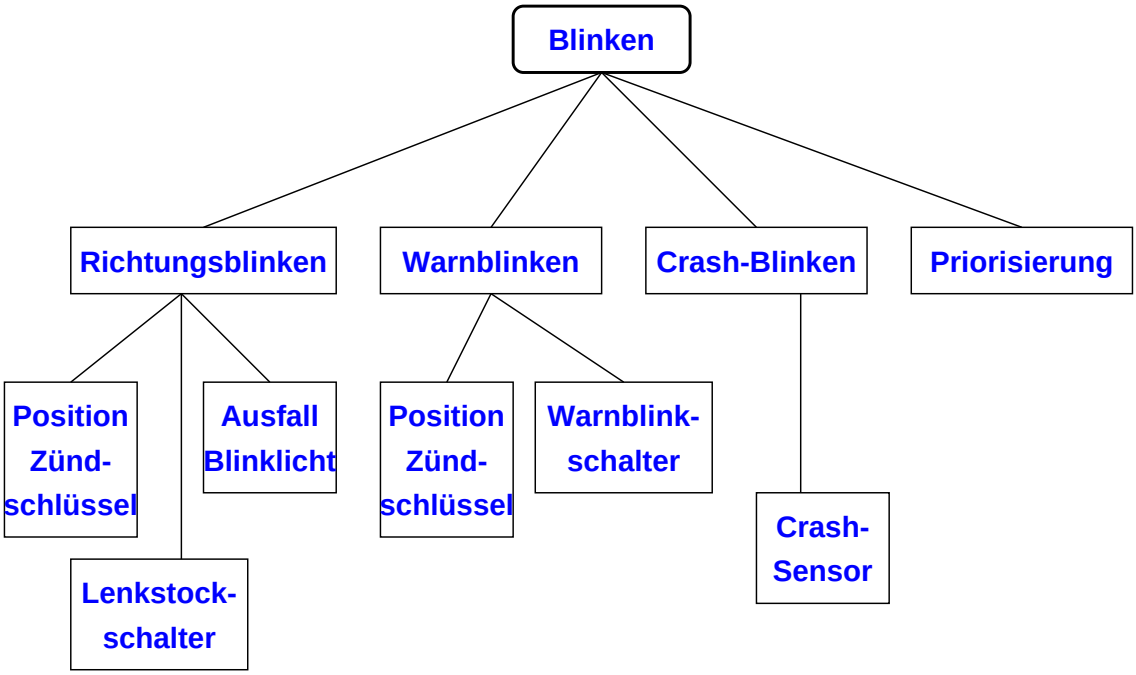


Abbildung 11.6: Blinkersteuerung ohne Klassen

Teilschritt 2.2: Vollständige Zerlegung in disjunkte Klassen

Lenkstockschalter:	- Position oben - Position mitte - Position unten
Zündschlüsselposition:	- Stellung 0 - Stellung 1 - Stellung 2 - Stellung 3
Warnblinkschalter:	- gedrückt - nicht gedrückt

Bei der Klassifikation „Crash-Sensor“ muss nicht unterscheiden werden, ob ein Crash detektiert wurde oder nicht, sondern ob ein Crash **gerade erkannt wird** oder ob er **bereits erkannt wurde**. Diese Zeitabhängigkeit ist entscheidend für das Deaktivieren der Blinklichter, da dies nur möglich ist, wenn der Crash bereits erkannt wurde.

Crash-Sensor:	- Crash gerade erkannt - Crash bereits erkannt - kein Crash
Ausfall Blinklicht:	- ja - nein
Priorisierung:	- keine Anforderung - eine Anforderung - mehr als eine Anforderung

Iteration:

Die Klasse „bereits erkannt“ der Klassifikation „Crash-Sensor“ wird durch die Klassifikation „Warnblinklicht ein“-Bit detaillierter beschrieben. Ist ein Crash bereits erkannt worden, so kann das Blinken durch zweimaliges Betätigen des Warnblinkschalters deaktiviert werden, was einem Flankenwechsel des „Warnblink ein“-Bits von **1** nach **0** entspricht.

Klassifikation des „Warnblink ein“-Bits:

- Zustand 0	- Flankenwechsel von 0 nach 1
- Zustand 1	- Flankenwechsel von 1 nach 0

Klassifikationsbaum für die Blinkersteuerung:

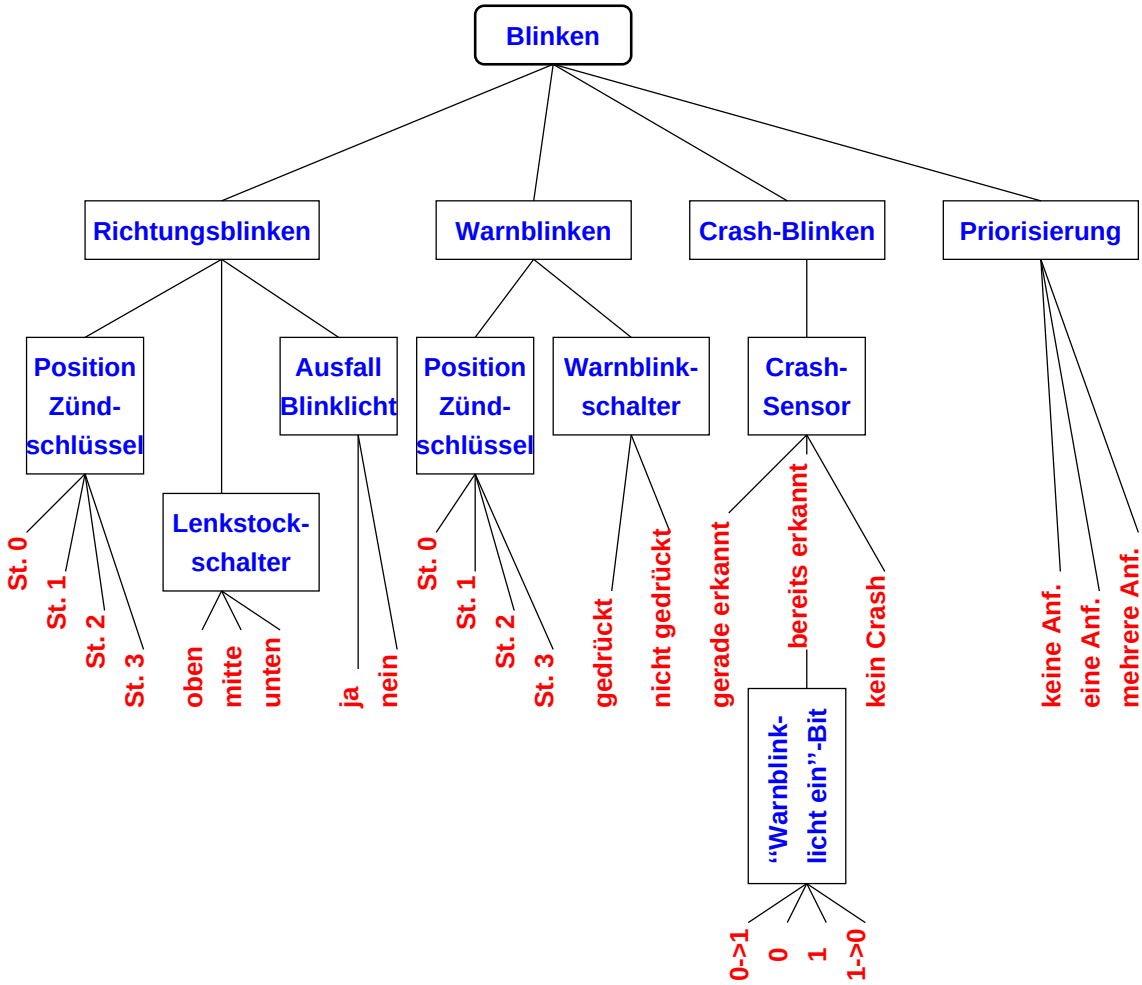


Abbildung 11.7: Blinkersteuerung

Schritt 3: Definition der Testfälle

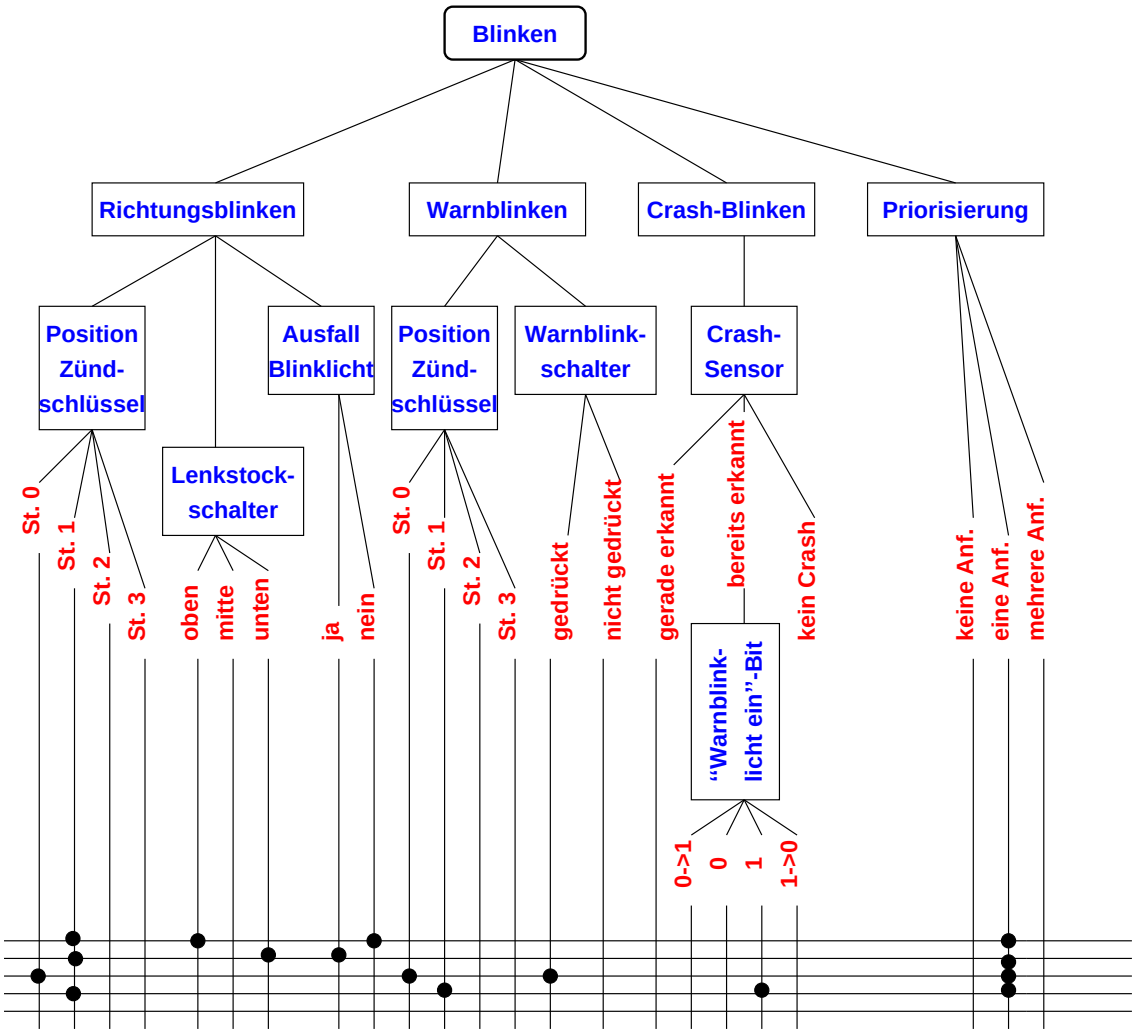


Abbildung 11.8: Blinkersteuerung – Testfälle

Anm.: Die Testeingangsdaten müssen i.a. durch entsprechende Skripte umgesetzt werden!

11.6.3 Abbildung von Echtzeitanforderungen

Eingangs- und Ausgangssignale wie auch das Kfz-Elektronik-System als Ganzes sind zeitbehaftet. Das Verhalten hängt oftmals nicht nur von zeitbehafteten Eingaben, sondern direkt von der Zeit ab:

- nach Ablauf einer bestimmten Zeit wechselt das System seinen Zustand).
- das System soll sich abschalten, wenn innerhalb einer bestimmten Zeitspanne kein Ereignis eintritt, das eine Änderung des Eingabevektors bewirkt

Allgemeiner Eingabevektor:

$$\vec{e}(t) = (e_1(t), \ldots, e_n(t), t)$$

mit den Eingabegrößen $e_1, \ldots, e_n$ und der Zeit t.

Um Testfälle mit der Klassifikationsbaum-Methode zu erstellen, müssen verschiedenste Zeitabhängigkeiten der Eingangs- und Ausgangssignale eingebracht werden; nur so besteht die Möglichkeit, Fehler nicht nur im rein funktionalen, sondern auch im zeitlichen Verhalten aufzudecken:

- Nachdem alle Fahrzeugtüren geschlossen wurden, soll die Innenbeleuchtung nach 20 sec abschalten
- Ein Schalter muss mindestens 2 sec gedrückt werden, um eine Reaktion hervorzurufen
- Zwei Eingangsgrößen müssen ihren Wert gleichzeitig ändern: Zwei sich im Normalbetrieb ausschließende Ereignisse, beispielsweise die Betätigung des Brems- und des Gaspedals sollen gleichzeitig eintreten; dadurch soll die Reaktion des Steuergeräts für diese Ausnahmesituation getestet werden können.

Ergo: Nicht nur die Wertebereiche bestimmter Größen müssen definiert werden, sondern auch deren zeitlicher Bezug!

Möglichkeit der Abbildung von Zeitbedingungen:

- Die Eingabegröße wird zunächst bzgl. der Werte klassifiziert, die für den Test relevant sind
- danach wird jeder sich daraus ergebende Wertebereich unter dem Gesichtspunkt des zeitlichen Verhaltens betrachtet

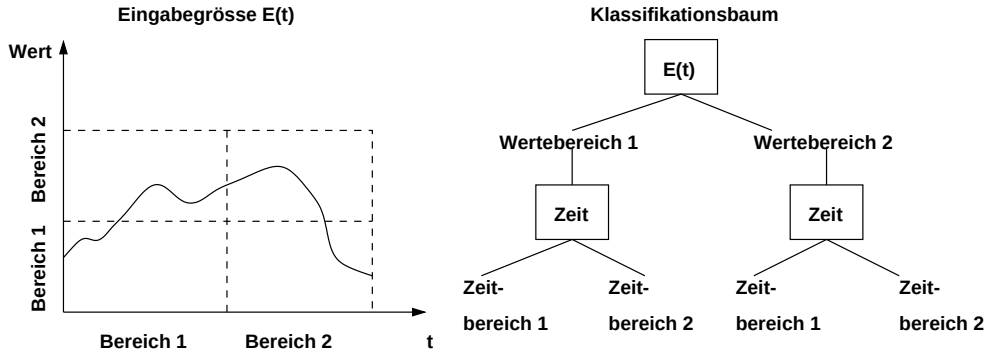


Abbildung 11.9: Abbildung von Zeitbedingungen

Beispiel: Innenbeleuchtung

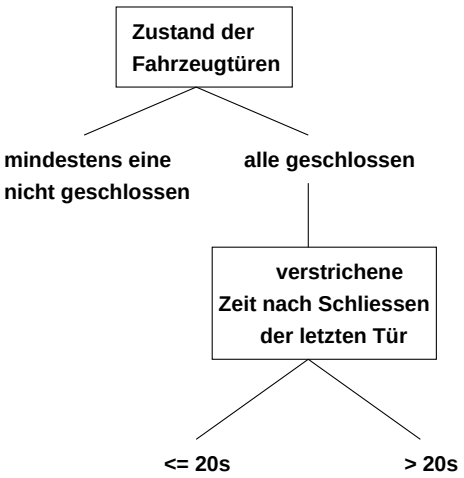


Abbildung 11.10: Zeitbedingungen – Beispiel

11.6.4 Komponenten eines Testsystems

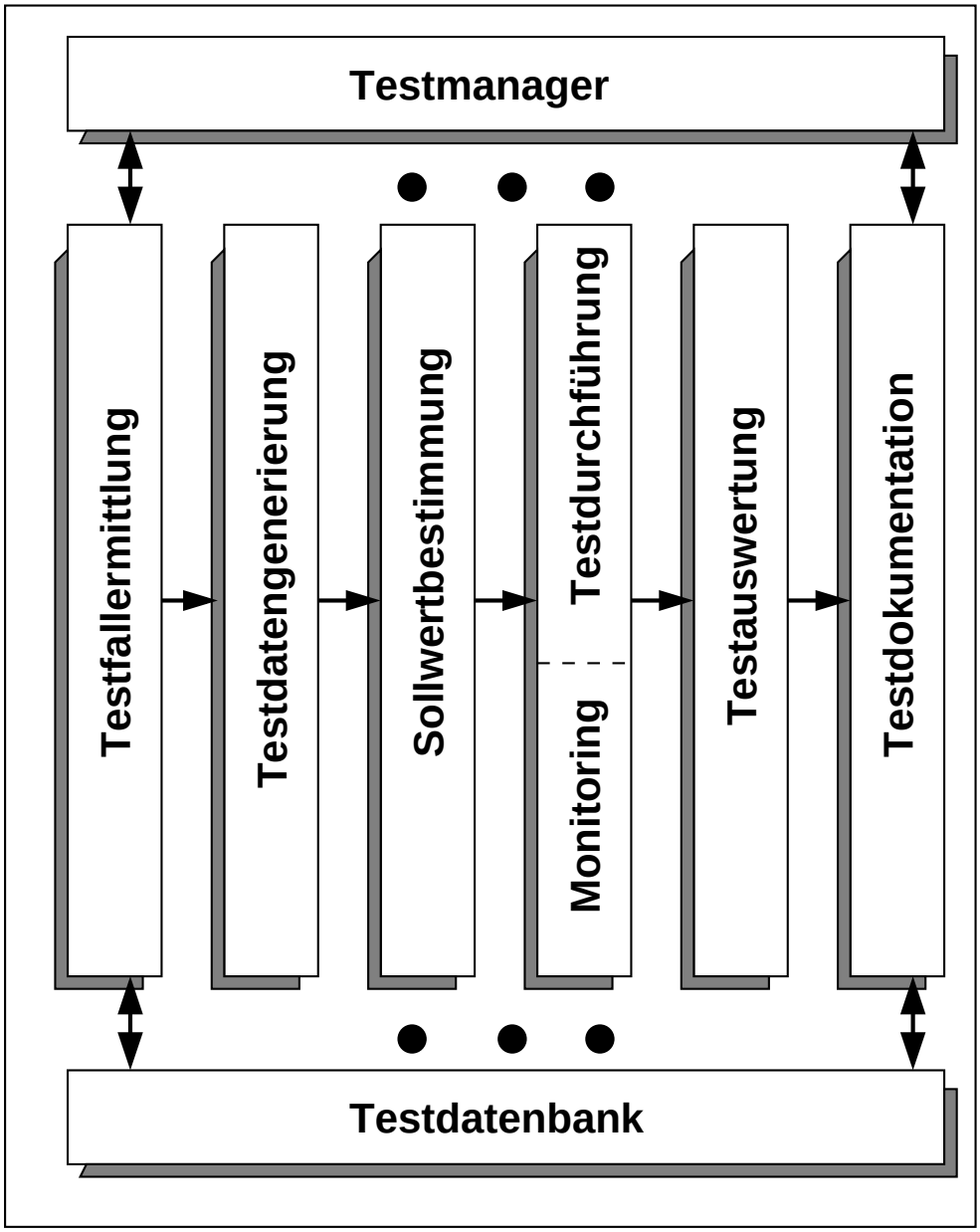


Abbildung 11.11: Testsystem – Komponenten

11.6.5 Testdatengenerierung

Für jeden im Klassifikationsbaum festgelegten Testfall sind Testdaten für die Eingangsgrößen der Kfz-Elektronik festzulegen, mit denen die vernetzten Steuergeräte im Zuge der Testdurchführung entsprechend stimuliert werden. In der Komponente "Testdatengenerierung" wird die Beziehung zwischen Testfallspezifikation, Schnittstellen der Kfz-Elektronik und den einzugeben-

den Testdaten hergestellt.

Die einzugebenden Testdaten werden sehr häufig über entsprechende Programme erzeugt, d.h. de facto sind Programme zu erstellen, mit denen Steuersequenzen erzeugt werden können.

Aufbau eines Testprogramms:

1. Header
2. Precondition
3. Program
4. Evaluation

Header

Hier werden Veraltungsinformationen vermerkt: Bezeichnung des Tests, Datum und Uhrzeit der Erstellung, Name des Ersteller, textuelle Beschreibung, ...

Precondition

Hier wird vermerkt, welche Bedingungen erfüllt sein müssen, damit der Test sinnvoll durchgeführt werden kann. Z. Bsp. kann geprüft werden, ob die notwendigen Steuergeräte, bestimmt durch Name, Software- / Hardware-Stand oder Diagnoseversion vorhanden sind.

Da jeder Test andere Signale beeinflussen kann, muss die Möglichkeit bestehen, nur die für den Test notwendigen Signale zu visualisieren bzw. zur Aufzeichnung vorzubereiten. Ebenso müssen Fehlerbilder vorbereitet werden können, damit sie während der eigentlichen Testdurchführung nur noch aktiviert oder deaktiviert werden müssen. Die Ausführung des "eigentlichen" Tests macht nur Sinn, wenn dieser Abschnitt erfolgreich ausgeführt wurde, also alle Vorbereitungen korrekt getroffen sind.

Program

Beschreibt die eigentliche Testdurchdurchführung. Hier erfolgt die Manipulation der Signale und die Steuerung des Testablaufs.

Evaluation

Nach der Testdurchführung erfolgt die *offline*-Auswertung. Dazu stehen die Fehlercodes aus den Fehlerspeichern der Steuergeräte und die Messdaten der während des Monitorings aufgezeichneten Signale zur Verfügung. Hier könnten auch entsprechende Kommentare in der zur Testdruchführung gehörenden Protokolldatei vermerkt werden.

Anmerkungen zur Sollwertbestimmung:

Ziel der Sollwertbestimmung ist es, in Abhängigkeit von der Testdatengenerierung festgelegten Testdaten die vom Testobjekt zu liefernden Ausgangswerte für die Ausgangsgrößen des Testobjekts zu bestimmen. Dies können erwartete Signalverläufe oder bestimmte charakteristische Merkmale, denen die aufgezeichneten Signale entsprechen sollen, sein.

Für Testfälle, bei denen keine konkreten Sollwerte angegeben werden können, müssen zumindest Plausibilitätsbedingungen formuliert werden.

Der Test der Kfz-Elektronik muss aber auch die Überprüfung der Selbstdiagnose der Steuergeräte zum Ziel haben. Dafür werden gezielt Fehler (Verkabelungsfehler, Bus-Fehler) simuliert, um die Fehlerspeicher der Steuergeräte gegen ihre Spezifikation zu prüfen.

11.7 Vom Modul- zum System-Test

Der Modultest ist i. d. R. ein Test der kleinsten, im Feinentwurf herausgearbeiteten Programmeinheit.

Aufgabe des Modultests ist ein Vergleich mit den funktionellen Anforderungen sowie den Schnittstellenspezifikationen, die den Effekt des Moduls definieren.

Testfallentwurf:

Grundlage sind die Spezifikation (Black-Box-Test) und das Programmlisting (White-Box-Test, Schreibtischtest, u.a.). Die Testdaten werden (Kritikalität!) durch eine Kombination der für beide Testklassen bekannten Methoden entworfen.

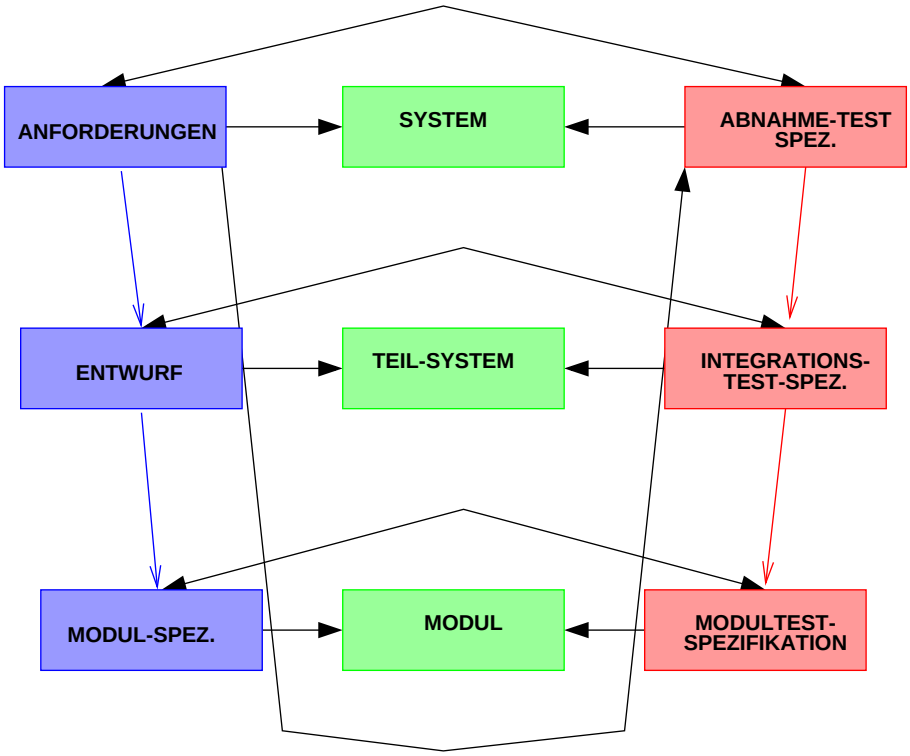


Abbildung 11.12: Integrationstest

Integrationstest:

- inkrementelles Testen (*bottom-up*, *top-down*, *sandwich*)

  Beim inkrementellen Testen werden die Module zunächst (oder auch nicht) separat getestet und dann mit stufenweisem Testen zum Gesamtsystem integriert (↪ Treibersoftware).

- nicht-inkrementelles Testen (*Big-Bang*-Integration / -Test)

  Beim nicht-inkrementellen Testen werden die Module einzeln ausgetestet und dann in einem Zug zum übergeordneten Modul (Programm) zusammengefügt und als Einheit erneut getestet.

Die Art der Modularisierung spielt beim Integrationstest eine Rolle:

- Inwieweit sind die Module entkoppelt, so dass sie ohne großen Aufwand einzeln getestet werden können?
- Inwieweit sind die Schnittstellen explizit, so dass der tatsächliche Zusammenhang ohne großen Aufwand ermittelt werden kann (wichtig auch zur Fehlersuche)?
- Inwieweit sind die zentrale Datenobjekte wie auch der Zugriff auf Betriebssystem-Ressourcen „abgekapselt“, so dass auch der Datenzugriff resp. die Interaktion mit dem Betriebssystem seperat getestet werden kann?

Aufgabe des Integrationstests ist i.w., Fehler in den Schnittstellen (explizite, implizite wie globale Datenstrukturen) zwischen den Modulen aufzuzeigen:

- Aufruf von Funktionen (Parameterübergabe)
- Verwendung der Datenobjekte

Testkriterien:

Ähnlich wie beim Modultest können auch auf der Ebene des Integrationstests Kriterien für die Testfallermittlung definiert werden. Diese können sich am intermodularen Kontrollfluss, am intermodularen Kommunikationsfluss (Datenfluss) oder an der spezifizierten Funktionalität orientieren. Beim Kontrollfluss können analog wieder Überdeckungsmaße, beim Datenfluss Äquivalenklassen oder Grenzwerte herangezogen werden.

11.8 High-Order-Test

- Funktionstest

  Ziel des Funktionstest ist, Unstimmigkeiten zwischen dem realisierten System und seiner Spezifikation (funktionale Anforderung, Leistungsbeschreibung) aufzudecken, also festzustellen, welche Systemfunktionen sich anders als spezifiziert verhalten. Der Funktionstest zielt also primär auf die Merkmale Korrektheit und Genauigkeit der Implementierung einzelner Funktionen.

- Systemtest

  Im Systemtest wird versucht, Unstimmigkeiten im Verhalten des **Gesamtsystems** mit den in der Leistungsbeschreibung oder im Lasten-/Pflichtenheft definierten allgemeinen Anforderungen, Zielsetzungen und Zielvorstellungen herauszufinden. Insbesondere sind hier auch die Benutzervorstellungen zu berücksichtigen.

  Einige Kriterien (vgl. dazu auch [?]):

  – **Volumentest:**

    Hier wird das Programm einem großen Datenvolumen ausgesetzt. Die (destruktive) Zielsetzung beim Volumentest ist, zu zeigen, dass das Programm das gemäß Anforderung maximale Datenvolumen nicht behandeln kann.

  – **Stresstest** (Prozessleitsysteme, e-commerce, . . .):

    Beim Stresstest wird das Programm für eine bestimmte Zeitspanne einer großen Last ausgesetzt.

    Wenn ein Prozessleitsystem maximal 100 Objekte führen können muss, sollte die Situation (mit simulierten Objekten) getestet werden, dass gleichzeitig 100 Objekte in den Erfassungsbereich eingeführt werden. Was passiert, wenn ein zusätzliches Objekt dazukommt? Was passiert, wenn alle gleichzeitig kritische Informationen liefern?

- **Sicherheitstest** (Sicherheit vor der Umgebung)

  Beim Sicherheitstest wird versucht, Testfälle zu entwerfen, die die Sicherheitsprüfungen des Programms in Frage stellen.

  Damit kann z. B. versucht werden, die Sicherheitsvorkehrungen eines Datenbanksystems zu unterlaufen (Aufspüren von sog. Sicherheitslöchern).

- Abnahmetest

  Bei dieser Art von Prüfung unterscheidet man in der Software-Entwicklung (insbesondere bei Auftragsentwicklung) häufig zwischen Werksabnahme (interne Abnahme ohne reale operationelle Umgebung) und Kundenabnahme (nach Installation auf realem operationellen Zielsystem, endgültige Bestätigung der **Vertragserfüllung**).

  Beide Formen werden in der Art der Vorgehensweise und insbesondere bei der Auswahl der Testfälle vom Kunden und / oder Benutzer des Systems geprägt. Zwar wird häufig die Prüfvorschrift noch vom Auftragnehmer erstellt, muss aber vom Auftraggeber “genehmigt” werden.

Vorgehensweise beim Abnahmetest:

- "Zerlegen" der **Anwendung** in überschaubare Einheiten
- herausarbeiten des jeweils interessierenden Zusammenhangs (kommunikativ, kausal, ...)
- orientiert an Geschäftsprozessen, operativen Zusammenhängen, Datenflüssen

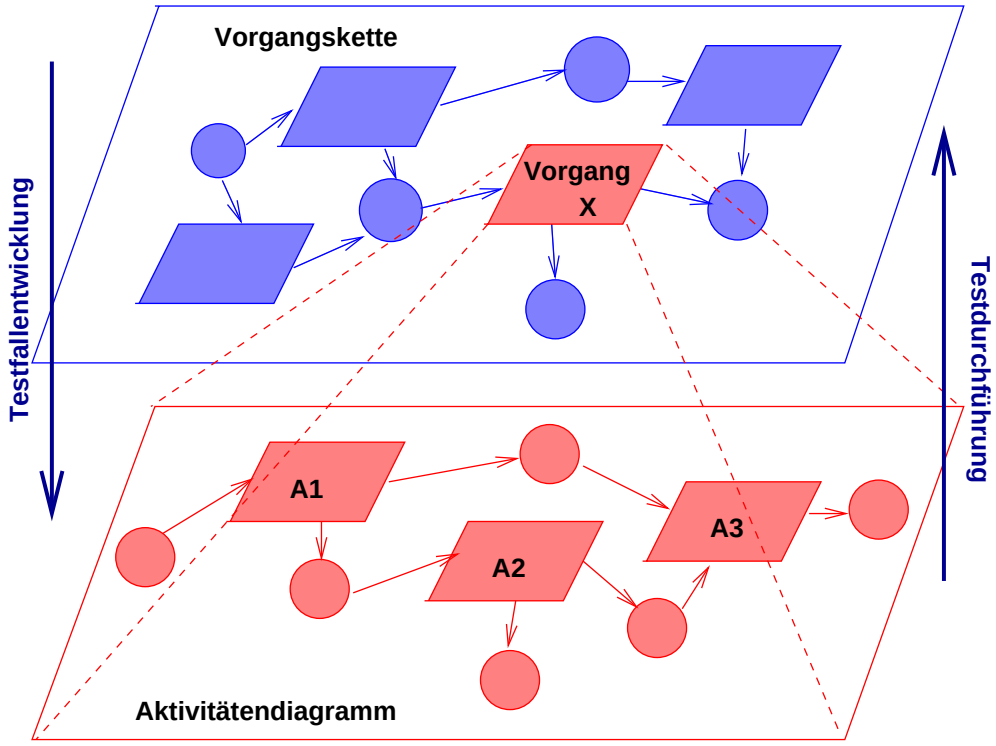


Abbildung 11.13: Abnahmetest

11.9 Testbeendigung

- Testen kostet Geld
- Fehler beim Kunden kosten Geld
- $\hookrightarrow$ ein nichttriviales Entscheidungsproblem

Kriterien für die Testbeendigung:

- Nach Ablauf der geplanten Testzeit
- Nachdem alle geplanten Testfälle (endlich) ohne Fehler durchgeführt wurden
- Nachdem eine vorgegebene Testüberdeckung (Aufrufgraphen, Vorgangsketten, Programmpfade, ...) erreicht wurde
- Nachdem ein bestimmter Prozentsatz "geschätzter" Fehler aufgedeckt wurde
  - Abschätzung der Gesamtzahl von Fehlern
  - Abschätzung, welcher Prozentsatz davon durch Testen überhaupt gefunden werden kann
  - Erfahrungswerte!!!
- Nachdem x% von gezielt "eingebrachten" Fehlern gefunden wurden

11.10 Test-Automatisierung

Möglichkeiten:

- Generierung von Test-Eingangsdaten
  (Formale Spezifikation liegt vor!)
- Testaufzeichnung (*capture*)
- Testdurchführung
- Testauswertung
- Testwiederholung (*replay*)

Einfache Struktur eines zu testenden Programm unter UNIX:

Abbildung 11.14: Filter-Programm

Das Dreiecksbeispiel (implementiert als C-Programm):

```
#include <stdio.h>

int main(){

  unsigned long int a,b,c;
  while ( scanf("%ld %ld %ld\n", &a, &b, &c) == 3) {
      if ( (a == b) && (b == c)){
            printf("gleichseitig\n"); continue;
      }
      if ( (a + b <= c) || (a + c <= b) || (b + c <= a)){
            printf("Kein Dreieck\n"); continue;
      }
      if ( (a == c) || (a == b) || (b == c)){
            printf("gleichschenklig\n"); continue;
      }
      printf("allgemeines Dreieck\n");
  }
  return 0;
}
```

Test-Sitzung aufzeichnen — Kommando *script*

Script started on Wed Mar 7 09:51:17 2001
hypatia$ **dreieck**
1 2 3
Kein Dreieck
hypatia$ **dreieck**
2 2 2
Gleichseitiges Dreieck
hypatia$ **dreieck**
3 3 5
Gleichschenkliges Dreieck
hypatia$ **dreieck**
1 2 -
Drei Zahlen hatte ich gesagt - Sorry
hypatia$ **exit**

Script done on Wed Mar 7 09:52:08 2001

Einfache Lösung

(Verhalten des Testlings muss exakt bekannt sein):

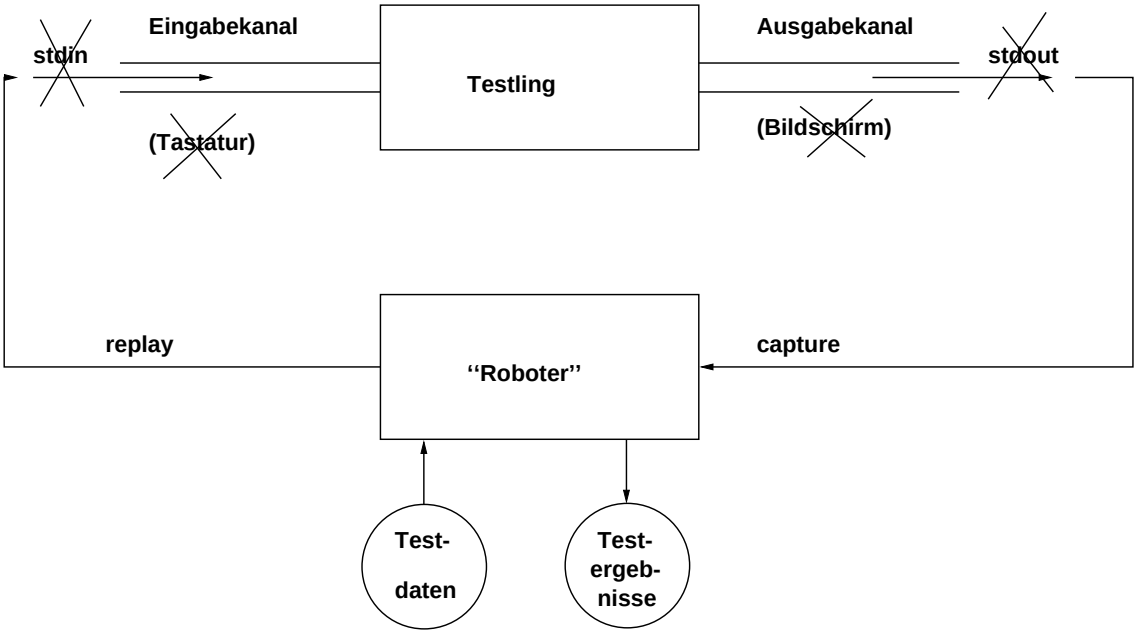


Abbildung 11.15: Test-Roboter

Implementierung

```
# include <stdio.h>
# include <unistd.h>
# include <sys/types.h>
# include <sys/wait.h>
# include <fcntl.h>

int main(int argc, char ** argv) {
   char senddata[81], getdata[81];
   FILE * infile, * outfile;
   int nr, status, count, pid, s_p[2], g_p[2];
   if (argc <= 1) exit(1);
   outfile = fopen("results","w");
   infile = fopen(argv[1],"r");
   nr = 0;
   while( fgets(senddata,81,infile) != NULL ) {
      nr++;
      fprintf(outfile,"============================\n");
      fprintf(outfile,"%d:\n", nr);
      fprintf(outfile,"testdata: %s\n", senddata);
      fflush(outfile);
      if( (pipe(s_p) < 0) || (pipe(g_p) < 0)) exit(1);
      switch (pid = fork()) {
        case -1: exit(2);
        case 0:  close(s_p[1]); close(g_p[0]); close(0);
                 dup(s_p[0]); close(1); dup(g_p[1]);
                 execl("./dreieck", "dreieck", NULL);
        default: close(s_p[0]); close(g_p[1]);
                 write(s_p[1],senddata,81);
                 count=read(g_p[0], getdata, 81);
                 getdata[count]='\0';
                 fprintf(outfile,"out: %s\n", getdata);
                 close(s_p[1]); close(g_p[0]);
                 wait(&status);
                 fflush(outfile);
       }
   }
   exit(0);
}
```

Aufruf: **robot test.dat**

Testdaten (test.dat):

```
1 2 3
6 6 6
2
4 5 6
5 5 8
-1 2 3
0 0 0
```

Ergebnisse: (results)

```
=====================================
1:
testdata: 1 2 3
out: Kein Dreieck
=====================================
2:
testdata: 6 6 6
out: Gleichseitiges Dreieck
=====================================
3:
testdata: 2
out: Drei Zahlen!!!
=====================================
4:
testdata: 4 5 6
out: Allgemeines Dreieck
=====================================
5:
testdata: 5 5 8
out: Gleichschenkliges Dreieck
=====================================
6:
testdata: -1 2 3
out: Positive Zahlen bitte!
=====================================
7:
testdata: 0 0 0
out: Positive Zahlen bitte!
```

Probleme:

- Architektur des Testlings ("Anzapfen" von I/O-Schnittstellen)
- Betriebssystemabhängigkeiten
- Abhangigkeiten von anderen Komponenten (z.B. Window-System, GUI)
- Testling muss unter Roboter gleiches Verhalten zeigen wie ohne (Signal- / Interrupt-Verhalten)
- Ggf. zeitliche Simulation des Benutzerverhaltens
- und weitere mehr!

Mehr dazu z. B. in:
B. Waldbauer: Reproduzierbare Aktionen unter Unix. Dissertation Universität Ulm 1995

Kapitel 12

Randbedingungen in der SW-Entwicklung

12.1 Juristische Aspekte

Produkthaftung für Software - Haftungsentlastung durch Qualitätssicherung, Frank A. Koch ([?]):

> ...
> *Hierbei zeigt sich, dass eine Haftungsentlastung des Anbieters vielfach am Fehlen einer geschlossenen Qualitätssicherungskonzeption scheitert, spätestens aber an der unzureichenden und deshalb nicht beweisfähigen Dokumentation zu dieser Qualitätssicherung.*
> ...
> *Der Hinweis, dass sich Fehler von Software niemals vollständig ausschließen lassen, führt zu keiner Haftungsentlastung (weder vertraglich noch aus deliktischer bzw. gesetzlicher Produkthaftung) und kann sogar umgekehrt aufgrund des erhöhten Gefährdungspotentials zu einer Intensivierung der anbieterseitigen Pflicht zur Produktionsabsicherung führen.*

Abb. 12.1 (S. 392) zeigt in einer Übersicht nocheinmal (siehe auch **??**, S. **??**) mögliche Haftungsbereiche.

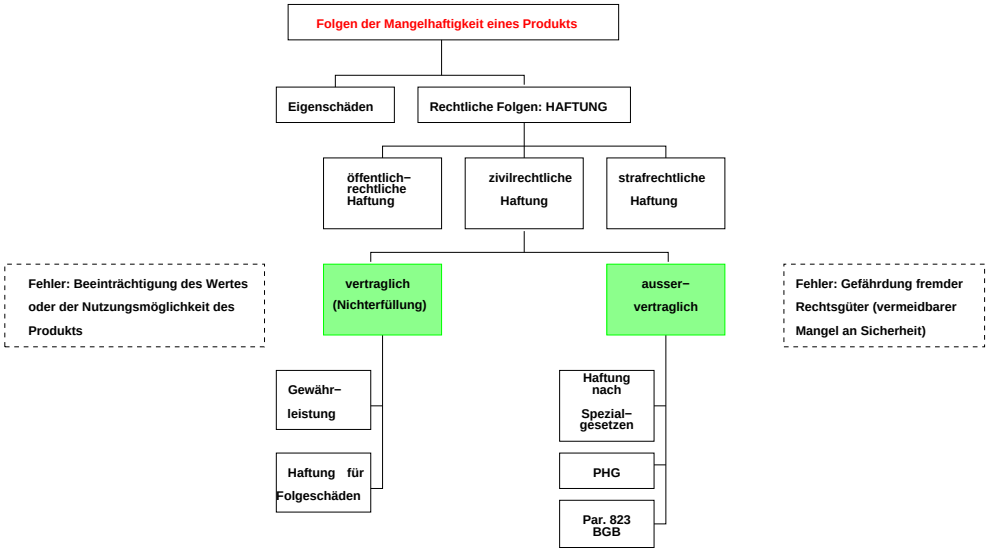


Abbildung 12.1: Haftungsbereiche

Ausservertragliche Haftung basiert i.w. auf dem **BGB** (Bürgerliches Gesetzbuch) sowie hier auf dem **PHG** (Produkthaftungsgesetz). Diese Art der Haftung kann durch vertragliche Regelungen oder **AGB**s (Allgemeine Geschäftsbedingungen) nicht ausgeschlossen, sondern allenfalls eingeschränkt werden. Daher haben Aussagen bei Software-Produkten wie „jegliche Haftung wird ausgeschlossen" keinerlei Bedeutung.

Vertragliche Haftung basiert i.w. auf den typischen Vertragstypen, die im BGB definiert sind (siehe z.B. M. Bartsch: Softwareüberlassung - was ist das? Computer&Recht 7, 1992).

Vertragstypen bei Softwarebeschaffung:

Vertragstyp	Definition	Ansprüche bei „Leistungsstörung"
Kaufvertrag (§433 ff. BGB)	Uberlassung einer Sache ohne zeitliche Begrenzung gegen Entgelt (z.B. Standardsoftware) – Achtung: Urheberrecht, copyright	Wandlung (Rückgängigmachen des Kaufs) oder Minderung des Kaufpreises
Mietvertrag (§535 ff. BGB)	Zeitliche Überlassung einer Sache gegen Entgelt	Minderung des Mietpreises oder Kündigung des Mietverhältnisses

Quelle: Diebold Management Report 10/90

Vertragstypen bei Softwareerstellung:

Vertragstyp	Definition	Ansprüche bei „Leistungsstörung"
Werkvertrag (§631 ff. BGB)	Erbringung einer Leistung oder Tätigkeit, bei dem ein Erfolg geschuldet wird (Individualsoftware)	Nachbesserung, Schadensersatz, Einbehaltung / Reduzierung des vereinbarten Preises
Dienstvertrag (§611 ff. BGB)	Tätigkeit gegen Entgelt, bei der kein Erfolg geschuldet wird (Beratung)	Schadensersatz bei grober Fahrlässigkeit oder bei Vorsatz

Quelle: Diebold Management Report 10/90

In Frage kommt ebenfalls noch der Werklieferungsvertrag, §651 ff. BGB.

Wichtig: Beachtung des **Urheberrechtsschutzes** und die ausservertraglichen Anforderungen aus dem Produkthaftungsgesetz.

Bei Standardsoftware (z.B. PC-Produkte) liegt liegt i.a. immer ein Kaufvertrag zugrunde mit „bestimmungsgemäßer“ Erlaubnis des Kopierens von Diskette (gekauftes Datenträgermedium) auf Platte und wiederholten Kopierens in den Hauptspeicher zur Ausführung des Programmes (Sicherungskopien).

Die Begriffe **Fehler** und **Mangel** werden in der Rechtsprechung weitgehend synonym verwendet. Ausgangspunkt ist §823 Abs. 1 BGB – daraus abgeleitet wird „die allgemeine Pflicht für jedermann, der eine Gefahrenquelle schafft, die notwendigen Vorkehrungen zum Schutze Dritter zu schaffen (Verkehrssicherungspflicht $\hookrightarrow$ Schadensersatz). Bei Produkthaftung wird statt Verletzung der Verkehrssicherungspflicht von Produktmangel oder Produktfehler gesprochen.

Die Gefahrabwendungs- oder Sicherungspflicht ist auf das objektiv (?) Erforderliche oder objektiv (?) Zumutbare beschränkt.

Die vertragliche Haftung resultiert aus der Art des Vertrages; sie bezieht sich auf die Fehlerfreiheit der vertraglich geschuldeten Leistung; ein Verschulden ist nicht notwendig erforderlich. Bei Fehlen zugesicherter Eigenschaften ergibt sich u.U. auch eine Haftung für Folgeschäden, verursacht durch schuldhafte (vorsätzliche oder fahrlässige) Verletzung vertraglicher (Neben-) Pflichten („Haftung aus positiver Vertragsverletzung“).

Softwareschutz:

- Patentrecht: nur ausnahmsweise, Software ist integraler Bestandteil der patentfähigen Gesamterfindung (z.B. ABS)
- Wettbewerbsrecht (UWG) (Unlauterer Wettbewerb) – wegen Urheberrechtsgesetz nur noch von geringer Bedeutung
- Warenzeichen (WZG) – Warenzeichen direkt im Produkt, illegales Kopieren des Warenzeichens ist strafbar
- Urheberrecht
  Dem Urheber werden spezifische Verwertungshandlungen als Rechte vorbehalten. Für Software sind von Bedeutung:
  - Vervielfältigung
  - Umarbeitung
  - Verbreitung

Haftungsbeschränkung:

Vorgaben des AGBG (Allgemeine Geschäftsbedingungen):

- §11 Nr. 7 AGBG:
  Die Haftung für Vorsatz und grobe Fahrlässigkeit kann nicht beschränkt werden.
- §11 Nr. 11 AGBG:
  Die Haftung für zugesicherte Eigenschaften kann nicht beschränkt werden.
- §11 Nr. 8 AGBG:
  Die Haftung aus Verzug oder Unmöglichkeit darf auch unterhalb grober Fahrlässigkeit nicht ausgeschlossen werden; eine nennenswerte Haftung muss bestehen bleiben.
- Produkthaftung:
  Die Haftung aus dem Produkthaftungsgesetz (§14 PHG) kann nicht beschränkt werden. Die Haftung aus §823 BGB kann jedenfalls für Körperverletzungen nicht beschränkt werden

Produkthaftung:

Der Begriff „Produkthaftung“ enthält das Wort „Haftung“. Haftung bedeutet, dass eine Person einer anderen für etwas einstehen muss. Die eine Person nennt man Schädiger, Haftender, Haftpflichtiger, Haftungsschuldner oder schlicht Beklagter. Bei der Produkthaftung kann man ihn auch „Produkthaftpflicht-Schuldner“ oder „Produkthaftpflichtiger“ nennen. Mit diesem weiten Begriff anstelle des engeren „Produzenten“ soll verdeutlicht werden, dass die Haftung neben dem Produzenten auch den Händler, Importeur oder Montagebetrieb treffen kann. Die andere Person heisst Anspruchsteller, Geschädigter oder Kläger. Aus dem ersten Teil des Wortes Produkthaftung lässt sich herleiten, dass es um das Einstehenmüssen für mangelhafte „Produkte“ geht. (nach [?])

- Wer haftet? (siehe auch Abb. **??**, S. **??**)
  - jeder, der an der Herstellung des Produktes beteiligt ist,
  - jeder, der damit handelt,
  - jeder, der sonst Leistungen daran ausführt

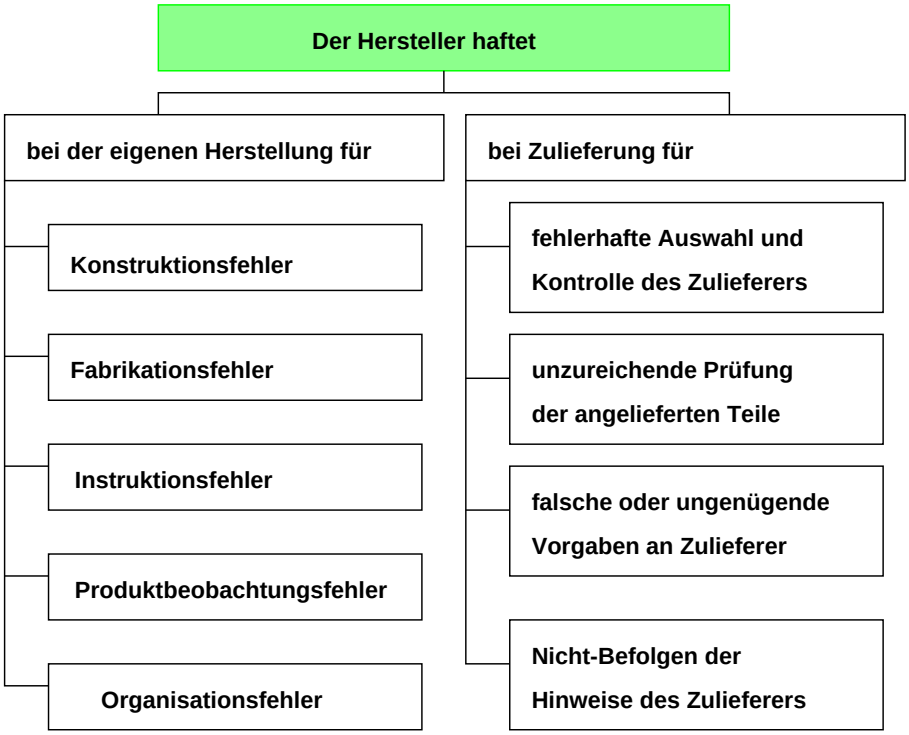


Abbildung 12.2: Haftung des Produktherstellers

- Produktbegriff ist sehr weit gefasst, schließt Software ein
- Welche Schäden werden ersetzt?
  - Personenschäden
  - Sachschäden (außerhalb der mangelhaften Sache)
  - Vermögensschäden nur unter bestimmten Voraussetzungen
- unmittelbare Kausalität muss gegeben sein

- Das Produkt muss so konstruiert werden, dass es dem **Stand der Technik** entspricht. DIN-Normen . . . bilden dabei nur die Untergrenze der Sicherheitsanforderungen.
- Ebensowenig wie die Einhaltung technischer Standards wie DIN-Normen etc. schützt den Hersteller eine behördliche Prüfung, . . .

Urheberrecht:
(Quelle: A. Wiebe: Rechtsschutz für Software in den 90er Jahren. in Betriebs-Berater (BB) 10.6.93, Heft 16, S. 1094 ff)

- gültig seit 6.93
- Schutzfähigkeit
  Software muss das Produkt „eigener geistiger Schöpfung“ sein, darf nicht “banal” sein ↪
  gilt praktisch für jedes professionelle Programm
  Software ist in allen Entwicklungsstufen geschützt (Quell-/Objektcode, Dokumentation)
  Nicht geschützt sind (anders in den USA) zugrundeliegende Ideen, Algorithmen u.dgl.
- Schutzumfang
  Urheber hat Verwertungsrecht (Vervielfältigung, Bearbeitung, Verbreitung, Vermietung)
  Das Kopieren eines Programms von Diskette auf Platte ist eine Vervielfältigung, ebenso das Laden von Platte in Hauptspeicher!!!
  Jede dauerhafte, vorübergehende, einmalige Vervielfältigung bedarf der Zustimmung des Urhebers
  Ausnahme: Erstellen von Sicherungskopien
- Dekompilierung (Gewinnung des Quellcodes aus dem Objektcode) ist nur dann zulässig, wenn dies zur Herstellung der Interoperabilität mit anderen Programmen notwendig ist. Übergibt der Urheber allerdings enstprechende Schnittstellenbeschreibungen, so ist die Dekompilierung untersagt!
- Eine Verletzung des Urheberrechts kann strafrechtlich verfolgt werden (Geldstrafe oder Freiheitsstrafe bis zu 3 Jahren)
- Urheber / Urheberrechtsinhaber ist zunächst diejenige Person, die die Software entwickelt hat. Arbeitnehmer, zu deren Aufgaben die Softwareentwicklung gehört (Programmierer), überlassen, soweit vertraglich nicht anders geregelt, das Recht dem Arbeitgeber.

12.2 Prozessbewertung und Prozessverbesserung: Normen und Standards

- **Norm:** (lat.); Richtschnur, Regel, Vorschrift o.a. Muster;
  Im Bereich von Technik, Handel, Industrie und Handwerk versteht man unter Normung *die planmäßige Aufstellung von Normen für die Ausführung von Werkstücken, Ge- und Verbrauchsgütern, die auf Normblättern festgelegt werden; diese beschreiben einen nach Form, Größenklasse, Maßgenauigkeit, Baustoff und Fertigung einheitlichen Gütegrad vor*
- **Standard:**
  ein Maßstab, ein Normalmaß oder ein Muster, legt im Handel die Durchschnittsbeschaffenheit von Waren resp. anerkannte Qualitätstypen fest

Ziele von Normen und Standards:

- Einfachere Verständigung ($\hookrightarrow$ Protokolle zur Rechnerkommunikation)
- Höhere Flexibilität (Portabilität $\hookrightarrow$ POSIX)
- Kostenreduktion (z.B. höhere Personenunabhängigkeit, leichtere Einarbeitung)

Normen kommen nicht vom Gesetzgeber, es sind „freiwillige“ Vereinbarungen der Industrie; sie beschreiben unter Umständen den **Stand der Technik**!

Institutionen im Bereich der Informationsverarbeitung:

ANSI	American National Standards Institute
BSI	Bundesamt für Sicherheit in der Informationstechnik
CCITT	Comite Consultatif International Telegraphique et Telephonique – Unterorganisation der UN, alle Postverwaltungen, viele Hersteller – International gültige Empfehlungen für Datenübertragung (im Telefonnetz: **V.nn**; in anderen Datennetzen: **X.nn**), werden von anderen nationalen Normungsstellen (DIN) übernommen
DIN	Deutsches Institut für Normung e.V.
ECMA	European Computer Manufacturers Association (Mitglied von ISO u. CCITT)
IEEE	Institute of Electrical and Electronic Engineers – Zusammenschluß von Firmen und Universitäten; Standards für neue Produkte und Märkte
ISO	International Standardization Organization – Dachorganisation nationaler Standardisierungsinstitutionen
OSF	Open Software Foundation (Herstellervereinigung)
POSI	(Conference for) Promoting Open Systems Interconnection
X/Open	Gremium zur Integration von Normen und Standards zu einer einheitlichen Entwicklungsumgebung

Einige Standards / Normen:

- **ANSI / IEEE - Standards:**
  - 729-1983: Standard Glossary of Software Engineering Terminology
  - 730-1984: Standard for Software Quality Assurance Plans
  - 828-1984: Standard for Software Configuration Management Plans
  - 829-1983: Standard for Software Test Documentation
  - 830-1984: Guide to Software Requirements Specifications
  - 983-1986: Guide for Software Quality Assurance Planning
  - 1002-1987: Standard Taxonomy for Software Engineering Standards
  - 1008-1987: Standards for Software Unit Testing
  - 1012-1986: Standard for Software Verification and Validation Plans
  - 1016-1987: Recommended Practice for Software Design Descriptions
  - 1042-1987: Guide for Software Configuration Management
  - 1058.1-1987: Standard for Software Project Management Plans
  - 1063-1987: Standard for Software User Documentation
- **Zum Thema "Qualitätsmanagement"**:
  - **ISO 9000:2000** Quality Management Systems – Fundamentals and vocabulary
  - **ISO 9001:2000** Quality Management Systems – Requirements
  - **ISO 9004:2000** Quality Management Systems – Guidelines for performance improvement
  - **ISO 12207** Information Technology – Software Life Cycle Processes
  - **ISO 15504** Standard for Software Process Assessment and Improvement (SPICE)
  - **ISO 19011** Quality Management Systems – Guidelines on Quality and Environmental Auditing

  ISO 9001:2000 beschreibt generelle Anforderungen an **QM-Systeme**) für die Erstellung von Produkten. „Produkt" umfasst alle Produktkategorien, d.h. Hardware, verfahrenstechnische Produkte, Software, Dienstleistungen und Kombinationen daraus.

  (mehr im nächsten Abschnitt)
- Im Automobilbereich: ISO/TS 16949:2002

  Die Automobilindustrie hat sich mit dieser prozessorientierten Norm weltweit auf eine einheitliche QM-Norm geeinigt; diese Norm bezieht die ISO 9001:2000, die QS9000 und die VDA6.1 ein. Sie fordert insbesondere
  - die Integration der unternehmerischen Prozesse
  - den Teamgedanken in der Produktentwicklung
  - einen systematischen Ansatz für Zielvereinbarungen
  - Prozesskennzahlen und einen darauf basierenden kontinuierlichen Verbesserungsprozess (KVP)
  - die systematische Ermittlung der Kundenzufriedenheit, sowie Benchmarking
  - die konsequente Verfolgung und das Beseitigen der Grundursachen von Fehlern in Produktion und Abwicklung

  - eine stärkere Ausrichtung auf die internen und externen Kunden

- im Bereich der Deutschen Bahn AG: DIN EN 50128 (Software für Eisenbahnsteuerungs- und Überwachungssysteme)
- Zum Thema „Funktionale Sicherheit“: IEC 61508 (Teile 1-7) (*Functional safety of electrical / electronic / programmable electronic safety-related systems*
- und weitere mehr (Luft- und Raumfahrt, Atomkraftwerke, . . .)

12.3 ISO 9001:2000

Die ISO-Norm 9001:2000-12 mit dem Titel „Qualitätsmanagementsysteme – Anforderungen“ löst die „alte“ Norm ISO 9001:1994-08 ab (die Veränderungen sind in [?] detailliert beschrieben); sie beschreibt generelle Anforderungen an Qualitätsmanagement-Systeme (**QM-Systeme**) für die Erstellung von Produkten. Die in der Norm verwendete Bennenung „Produkt“ umfasst alle Produktkategorien, d.h. Hardware, verfahrenstechnische Produkte, Software, Dienstleistungen und Kombinationen daraus.

Ein sehr anschauliche Darstellung dieser Norm findet sich in [?], an der sich auch die folgenden Ausführungen orientieren.

Im Gegensatz zur „alten“ Norm ist die Neufassung sehr stark prozess- und kundenorientiert. Sie umfasst generelle Anforderungen an

- Führungsprozesse (Geschäftsprozesse der Unternehmensentwicklung, Unternehmensplanung, Ressourcenbereitstellung, Mitarbeiterentwicklung und Erfolgsmessung)
- Hauptprozesse (Geschäftsprozesse, die direkt zur Herstellung von Produkten oder Dienstleistungen dienen, die die Produktplanung und die Prozessplanung umfassen)
- Serviceprozesse (Hilfsprozesse, die obige Prozesse möglich und / oder sicher machen, die diese unterstützen oder auch beschleunigen)

Abb. 12.3, S.404 nach [?] gibt eine Vorstellung der Intuition resp. der Zielsetzung der Norm.

Nach Vorwort, Darlegung des Anwendungsbereichs, Verweise auf andere Normen und Begriffsbildungen beginnt die Beschreibung der Anforderungen mit Abschnitt 4. Im folgenden sollen kurz die Inhalte der Abschnitte 4 – 8 skizziert werden.

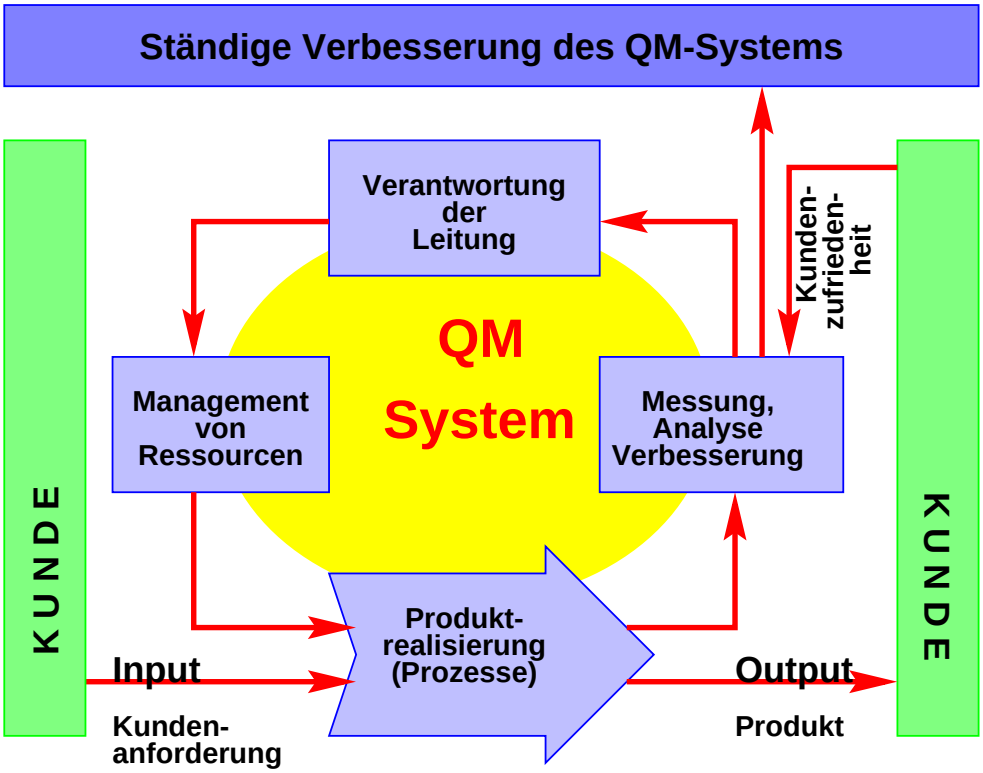


Abbildung 12.3: ISO 9001 – Zielsetzung

ISO 9001:2000 – Aufbau und Inhalt:

4. Qualitätsmanagementsystem

   4.1 Allgemeine Anforderungen
   Die Organisation muss die Wirksamkeit des QM-Systems ständig verbessern, die für das QM-System erforderlichen Prozesse und ihre Anwendung in der gesamten Organisation erkennen, die Abfolge und Wechselwirkungen dieser Prozesse festlegen.

   4.2 Dokumentationsanforderungen

      4.2.1 Allgemeines
      Die Dokumentation muss folgende Dokumente enthalten: Qualitätspolitik und Qualitätsziele, QM-Handbuch, dokumentierte Verfahren (wie von der Norm gefordert), Dokumente, die die Organisation benötigt.

      4.2.2 QM-Handbuch
      enthält insbesondere die Verfahrensanweisungen

      4.2.3 Lenkung von Dokumenten
      Wer erstellt, prüft, genehmigt Dokumente? Wie wird mit ungültigen Dokumenten verfahren?

      4.2.4 Lenkung von Qualitätsaufzeichnungen
      Welche Aufzeichnungen werden warum, wie lange und wie / wo aufbewahrt?

5. Verantwortung der Leitung
   Die Leitung muss ihrer Verpflichtung bezgl. der Entwicklung und Verwirklichung des QM-Systems

und der ständigen Verbesserung der Wirksamkeit des QM-Systems nachweisen. Zu diesem Zweck muss sie u.a. der Organisation die Bedeutung der Erfüllung der Kundenanforderungen sowie der behördlichen und gesetzlichen Anforderungen vermitteln.

Sie muss sicherstellen, dass für die betroffenen Funktionsbereiche und Ebenen innerhalb der Organisation Qualitätsziele festgelegt werden. Die Qualitätsziele müssen solche einschließen, die für die Erfüllung der Anforderungen an Produkte erforderlich sind. Die Qualitätsziele müssen messbar sein.

Die oberste Leitung muss sicherstellen, dass die Kundenanforderungen ermittelt (z.B. Marktanalyse) und mit dem Ziel der Erhöhung der Kundenzufriedenheit erfüllt werden.

Die Unterabschnitte sind:

- 5.1 Verpflichtung der Leitung
- 5.2 Kundenorientierung
- 5.3 Qualitätspolitik
- 5.4 Planung
  - 5.4.1 Qualitätsziele
    Diese müssen klar, verständlich, erreichbar und vor allem messbar sein!
  - 5.4.2 Planung des QM-Systems
    Prozesse, Ressourcen, unterstützende Dienstleistungen
- 5.5 Verantwortlichkeiten, Befugnis und Kommunikation
  - 5.5.1 Verantwortung und Befugnis
    Wer entscheidet was?
  - 5.5.2 Beauftragter der Leitung
    Kompetenz und Unabhängigkeit
  - 5.5.3 Interne Kommunikation
    Informations-, Berichtswege
- 5.6 Managementbewertung
  - 5.6.1 Allgemeines
    Die Managementbewertung muss außer Eignung und Wirksamkeit des QM-Systems auch dessen Angemessenheit betrachten. Sie muss die Bewertung von Möglichkeiten für Verbesserungen und den Änderungsbedarf für das QM-System inklusive der Qualitätspolitik und der Qualitätsziele enthalten.
  - 5.6.2 Eingaben für die Bewertung
    Ergebnisse der Leistungsmessung
  - 5.6.3 Ergebnisse der Bewertung

6. Management der Ressourcen
   - 6.1 Bereitstellung von Ressourcen
   - 6.2 Personal
     - 6.2.1 Allgemeines
     - 6.2.2 Fähigkeit, Bewusstsein und Schulung
       Ermittlung des Schulungsbedarfs bzw. von Maßnahmen, um die notwendigen Fähigkeiten der Personals zu erreichen, das die Produktqualität beeinflussenden Tätigkeiten ausübt.
   - 6.3 Infrastruktur
     u.a. Arbeitsplatzgestaltung

6.4 Arbeitsumgebung
u.a. Gesundheits- und Arbeitsschutz

7. Produktrealisierung

7.1 Planung der Produktrealisierung

7.2 Kundenbezogene Prozesse

7.2.1 Ermittlung der Anforderungen in Bezug auf das Produkt
Die vom Kunden festgelegten Anforderungen einschließlich der Anforderungen hinsichtlich Lieferung und Tätigkeiten nach der Lieferung – vom Kunden nicht angegebene Anforderungen, die jedoch für den festgelegten oder beabsichtigten Gebrauch, soweit bekannt, notwendig sind – gesetzliche oder behördliche Anforderungen – alle weiteren von der Organisation festgelegten Anforderungen

7.2.2 Bewertung der Anforderungen in Bezug auf das Produkt
Wer prüft, ob die Anforderungen tatsächlich erfüllt werden können?
Die Organiisation muss u.a. sicherstellen, dass bei Änderungen in den Produktanforderungen die zutreffenden Dokumente ebenfalls geändert werden und dass dem Personal die geänderten Anforderungen bewusst gemacht werden.

7.2.3 Kommunikation mit den Kunden
Die Organisaton muss wirksame Regelungen für die Komunikation mit den Kunden festlegen und verwirklichen; dies betrifft:
- *Produktinformationen,*
- *Anfragen, Verträge, Auftragsbearbeitung einschließlich Änderungen*
- *Rückmeldungen von Kunden einschließlich Kundenbeschwerden*

7.3 Entwicklung

7.3.1 Entwicklungsplanung
Die Organisation muss u.a. festlegen:
- *die Entwicklungsphasen*
- *für jede Entwicklungsphase die angemessene Bewertung, Verifizierung und Validierung*
- *Verantwortlichkeiten und Befugnisse*

7.3.2 Entwicklungseingaben
Was wird zur Entwicklung benötigt? Dazu gehören u.a.:
- *Funktions- und Leistungsanforderungen*
- *wo zutreffend auch Informationen, die aus früheren ähnlichen Entwicklungen abgeleitet wurden*
- *andere für die Entwicklung wesentliche Anforderungen*

7.3.3 Entwicklungsergebnisse
Was muss im Ergebnis der Entwicklung alles vorliegen? Dazu zählen auch Informationen für die Beschaffung, Produktion und Dienstleistungserbringung

7.3.4 Entwicklungsbewertung
Prüfung des Projektstandes in den einzelnen Entwicklungsphasen unter Einbeziehung aller betroffener Bereiche samt deren Dokumentation

7.3.5 Entwicklungsverifizierung
Prüfpläne zur Bewertung von Prototypen, Aufzeichnung der Prüfergebnisse, ...

7.3.6 Entwicklungsvalidierung
Ziel ist die Sicherstellung, dass das Produkt in der Lage ist, die Anforderungen für den festgelegten oder den beabsichtigten Gebrauch, soweit bekannt, zu erfüllen. Die Entwicklungsvalidierung muss nach geplanten Regelungen durchgeführt werden. Wenn möglich, muss sie vor der Auslieferung oder Einführung des Produkts abgeschlossen sein.

7.3.7 Lenkung von Entwicklungsveränderungen
Dokumentation aller Änderungen (wann, warum) – die Bewertung muss die Beurteilung der Auswirkungen der Änderungen auf Bestandteile der Produkte und auf gelieferte Produkte einschließen – Änderungen in Dokumenten müssen gekennzeichnet werden.

7.4 Beschaffung

7.4.1 Beschaffungsprozess
Es muss Kriterien für die Auswahl, Beurteilung / Neu-Beurteilung von Lieferanten existieren. Art und Umfang der auf den Lieferanten und das beschafte Produkt angewandten Überwachung müssen von dem Einfluss des beschafften Produkts auf die nachfolgende Produktrealisierung oder auf das Endprodukt abhängen.

7.4.2 Beschaffungsangaben
Gibt es Bestellformulare? Ist jede Bestellung eindeutig beschrieben? Sind die Spezifikation klar und präzise? Beschaffungsangaben müssen, soweit angemessen, Anforderungen an das QM-System des Leiferanten enthalten.

7.4.3 Verifizierung von beschafften Produkten
Wie wird die Wareneingansprüfung durchgeführt? Gibt es Prüfvorschriften? Gibt es Prüfprotokolle oder Zertifikate? Die Organisation muss die erforderlichen Prüfungen oder sonstigen Tätigkeiten festlegen und verwirklichen, durch die sichergestellt wird, dass das beschaffte Produkt die festgelegten Beschaffungsanforderungen erfüllt.

7.5 Produktion und Dienstleistungserbringung
Die Planung der Produktrealisierung muss mit den Anforderungen der anderen Prozesse des QM-Systems harmonieren.

7.5.1 Lenkung der Produktion und Dienstleistungserbringung
Das Ergebnis der Planung der Produktrealisierungsprozess (Ausrüstung, Ressourcen, Anweisungen, . . .) muss in einer für die Betriebsweise der Organisation geeigneten Form vorliegen.

7.5.2 Validierung der Prozesse zur Produktion und zur Dienstleistungserbringung
Die Organisation muss sämtliche derartigen Prozesse validieren (überprüfen hinsichtlich ihrer Eignung).

7.5.3 Kennzeichnung und Rückverfolgbarkeit
Bei Software i.w. Konfigurationsmanagement. Die Organisation muss den Produktstatus in Bezug auf die Überwachungs- und Messanforderungen kennzeichnen

7.5.4 Eigentum des Kunden
Die Organisation muss sorgfältig mit Eigentum des Kunden umgehen. Sie muss Eigentum des Kunden kennzeichnen.

7.5.5 Produkterhaltung
Die Organisation muss sicherstellen, dass alle Waren qualitätserhaltend behandelt werden (z.B. Freilager?).

7.6 Lenkung von Überwachungs- und Messmitteln
Soweit zur Sicherstellung gültiger Ergebnisse erforderlich sind diese zu Kalibrieren und zu Verfizieren anhand von Messnormalen, bei Bedarf zu justieren oder nachzujustieren, der Kalibrierstatus ist zu kennzeichen, . . .

8. Messung, Analyse und Verbesserung

8.1 Allgemeines
Die Organisation muss die Überwachungs-, Mess-, Analyse- und Verbesserungsprozesse planen und verwirklichen, die erforderlich sind, um

a) *die Konformität des Produktes darzulegen,*
b) *die Konformität des QM-Systems sicherzustellen und*
c) *die Wirksamkeit des QM-Systems ständig zu verbessern.*

Dies muss die Festlegung von zutreffenden Methoden einschließlich statistischer Methoden und das Ausmaß ihrer Anwendung enthalten.

8.2 Überwachung und Messung

8.2.1 Kundenzufriedenheit

Die Organisation muss Informationen über die Wahrnehmung der Kunden in der Frage, ob die Organisation deren Anforderungen erfüllt hat, als eines der Maße für die Leistungsfähigkeit des QM-Systems überwachen. Die Methoden zur Erlangung und zum Gebrauch dieser Informationen müssen festgelegt werden.

8.2.2 Internes Audit

Um die Wirksamkeit des QM-Systems zu überprüfen, müssen regelmäßig interne Audits durchgeführt werden. Die Auditkriterien, der Auditumfang, die Audithäufigkeit und die Auditmethoden müssen festgelegt werden. Die Auswahl der Auditoren und die Durchführung der Audits müssen die Objektivität und Unparteilichkeit des Auditprozesses sicherstellen.

8.2.3 Überwachung und Messung von Prozessen

Die Organisation muss geeignete Methoden zur Überwachung und, falls zutreffend, Messung der Prozesse des QM-Systems anwenden. Diese Methoden müssen darlegen, dass die Prozesse in der Lage sind, die geplanten Ergebnisse zu erzielen. Andernfalls müssen Korrekturmaßnahme ergriffen werden.

8.2.4 Überwachung und Messung des Produkts

Die Organisation muss in geeigneten Phasen des Produktrealisierungsprozesses die Merkmale des Produkts überwachen und messen, um die Erfüllung der Anforderungen an das Produkt zu verifizieren. Es muss ein Nachweis über die Konformität des Produkts mit den Annahmekriterien geführt werden.

8.3 Lenkung fehlerhafter Produkte

Die Organisation muss darlegen, wie die Behandlung von Fehlern durchgeführt wird (Nacharbeit, Sonderfreigabe, Verschrottung oder andere Maßnahmen, um den ursprünglich beabsichtigen Gebrauch oder die Anwendung auszuschließen, ...). Wenn ein fehlerhaftes Produkt nach der Auslieferung oder im Gebrauch entdeckt wird, muss die Organisation Maßnahmen ergreifen, die den Folgen oder möglichen Folgen des Fehlers angemessen sind.

8.4 Datenanalyse

Die Organisation muss geeignete Daten ermitteln, erfassen und analysieren, um die Eignung und Wirksamkeit des QM-Systems darzulegen und um zu beurteilen, wo ständige Verbesserungen der Wirksamkeit des QM-Systems vorgenommen werden können. Dies muss Daten einschließen, die durch Überwachung und Messung und aus anderen relevanten Quellen gewonnen werden.

Die Datenanalyse muss Angaben liefern über

a) *Kundenzufriedenheit,*
b) *Erfüllung der Anforderungen an das Produkt,*
c) *Prozess- und Produktmerkmale und deren Trends einschließlich Möglichkeiten für Vorbeugungsmaßnahmen und*
d) *Lieferanten*

8.5.1 Ständige Verbeserung

Die Organisation muss die Wirksamkeit des QM-Systems durch Einsatz der Qualitätspolitik, Qualitätsziele, Auditergebnisse, Datenanalyse, Korrektur- und Vorbeugungsmaßnahmen sowie Managementbewertung ständig verbessern.

8.5.2 Korrekturmaßnahmen

Die Organisation muss geeignete Verfahren zur systematischen Fehlerbehandlung nachweisen. Dies schließt Ursachenanalysen und Verfahren zur Ermittlung eines evt. notwendigen Handlungsbedarfs ebenso wie zur Überprüfung der Wirksamkeit der eingeleiteten Maßnahmen mit ein.

8.5.3 Vorbeugemaßnahmen
Die Organisation muss aufzeigen, wie der Entstehung von Fehlern vorgebeugt wird (z.B. durch Risikoanalysen oder FMEA). Die Wirksamkeit der hierzu eingeleiteten Maßnahmen muss ständig bewertet werden.

Aus der ISO 9001 wie auch aus der gängigen Rechtsprechung leiten sich organisatorische Anforderungen an Organisationen ab:

- *Die Rechtssprechung des Bundesgerichtshofes zur Organisationsverantwortung bewertet das Erfüllen allgemeiner Sorgfaltspflichten nach den Erkenntnissen der betrieblichen Organisationslehre als Grundlage der Unternehmensorganisation. Regelungsbedürftigkeit und notwendige, unterschiedliche Regelungstiefe umschreiben die in den Unternehmen zu erfüllenden Aufgaben. ([?]).*
- *Aufbau-, Ablauforganisation und Stellenbeschreibungen sind sich wechselseitig ergänzende Teile der Unternehmensorganisation. Sie bilden die dokumentierte Grundlage für die rechtlich so bedeutsame 'tatsächliche innerbetriebliche Verantwortung', die nach dem Strafrecht wichtigster Maßstab für das Bewerten von Handeln oder Unterlassen ist. ([?])*
- *Qualitätsorganisation:*
  Die rechtlichen Anforderungen aus der Produkthaftung sind üblicherweise nur zu erfüllen durch eine unternehmensumfassende bereichsübergreifende Qualitätsorganisation. Sie hat alle Vorgaben, Voraussetzungen, Maßnahmen und Abläufe zu erfassen, in oder bei denen sicherheits-/qualitätsrelevante Funktionen betroffen, ausgeführt oder darauf Einflüsse ausgeübt werden. ([?])

12.4 CMM(I) & Co.

12.4.1 Hintergrund

Situation heute und morgen:

- von lokalen Märkten zu globalen Märkten (**hohe Marktdynamik**
- neue Unternehmensbewertung (*shareholder value* vs. *stakeholder value*)
- weltweite Konkurrenz über weltweite Standards (ISO-Normen, Bilanzierungsgrundsätze, ...)
- steigende Veränderungsgeschwindigkeit von Wissen und Technik
- erhöhte Transparenz durch steigende Vernetzung
- verschärfte juristische Randbedingungen: → Haftung, KonTraG, ...) ...

Haftung bei Software?

Produkthaftung für Software - Haftungsentlastung durch Qualitätssicherung, Frank A. Koch:

- ...
  Hierbei zeigt sich, dass eine Haftungsentlastung des Anbieters vielfach am Fehlen einer geschlossenen Qualitätssicherungskonzeption scheitert, spätestens aber an der unzureichenden und deshalb nicht beweisfähigen Dokumentation zu dieser Qualitätssicherung.
- ...
  *Der Hinweis, dass sich Fehler von Software niemals vollständig ausschließen lassen, führt zu keiner Haftungsentlastung — weder **vertraglich** noch aus **deliktischer bzw. gesetzlicher** Produkthaftung — und kann sogar umgekehrt aufgrund des erhöhten Gefährdungspotentials zu einer Intensivierung der anbieterseitigen Pflicht zur Produktionsabsicherung führen.*

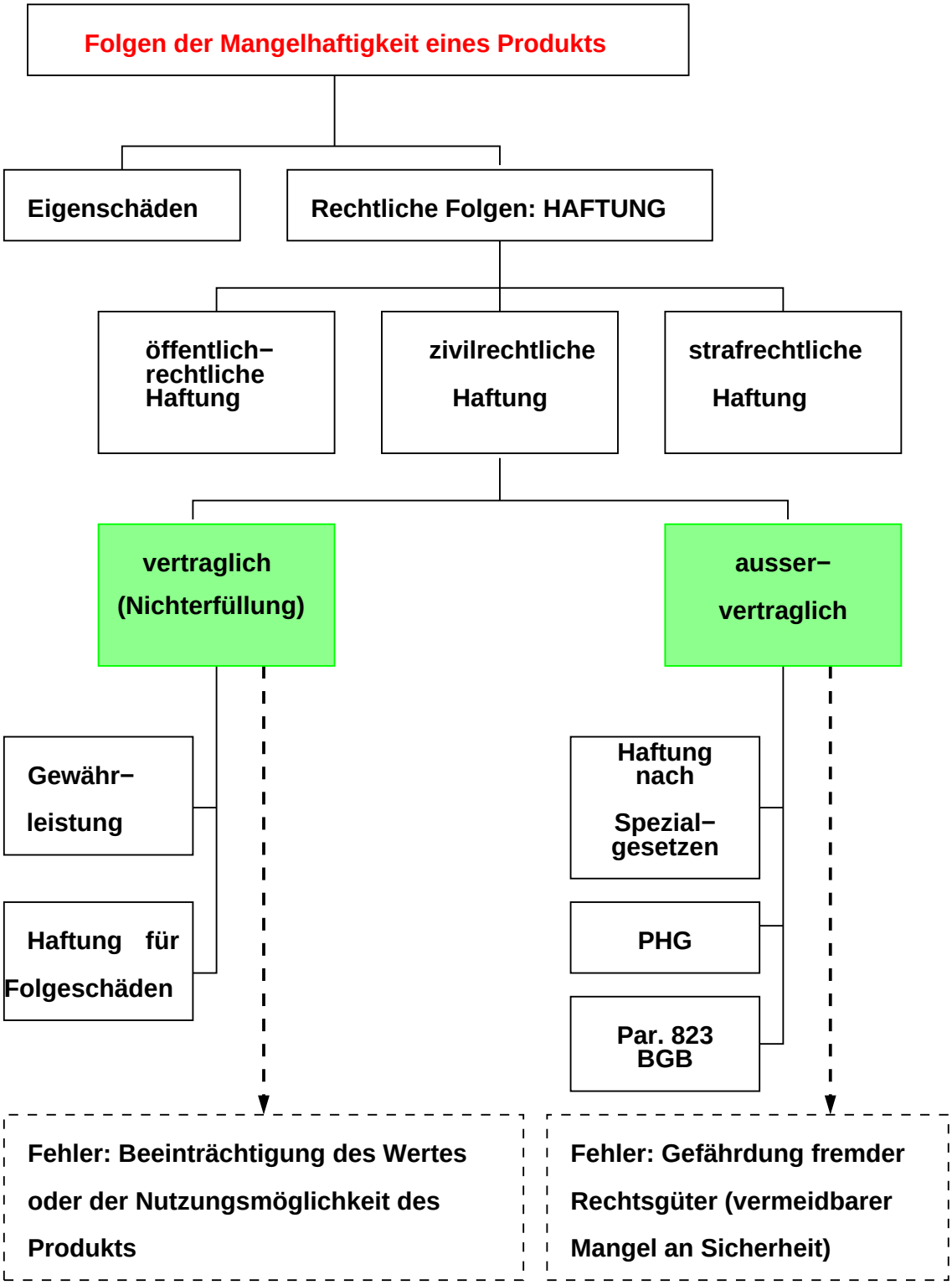


Abbildung 12.4: Folgen der Mangelhaftigkeit eines Produktes

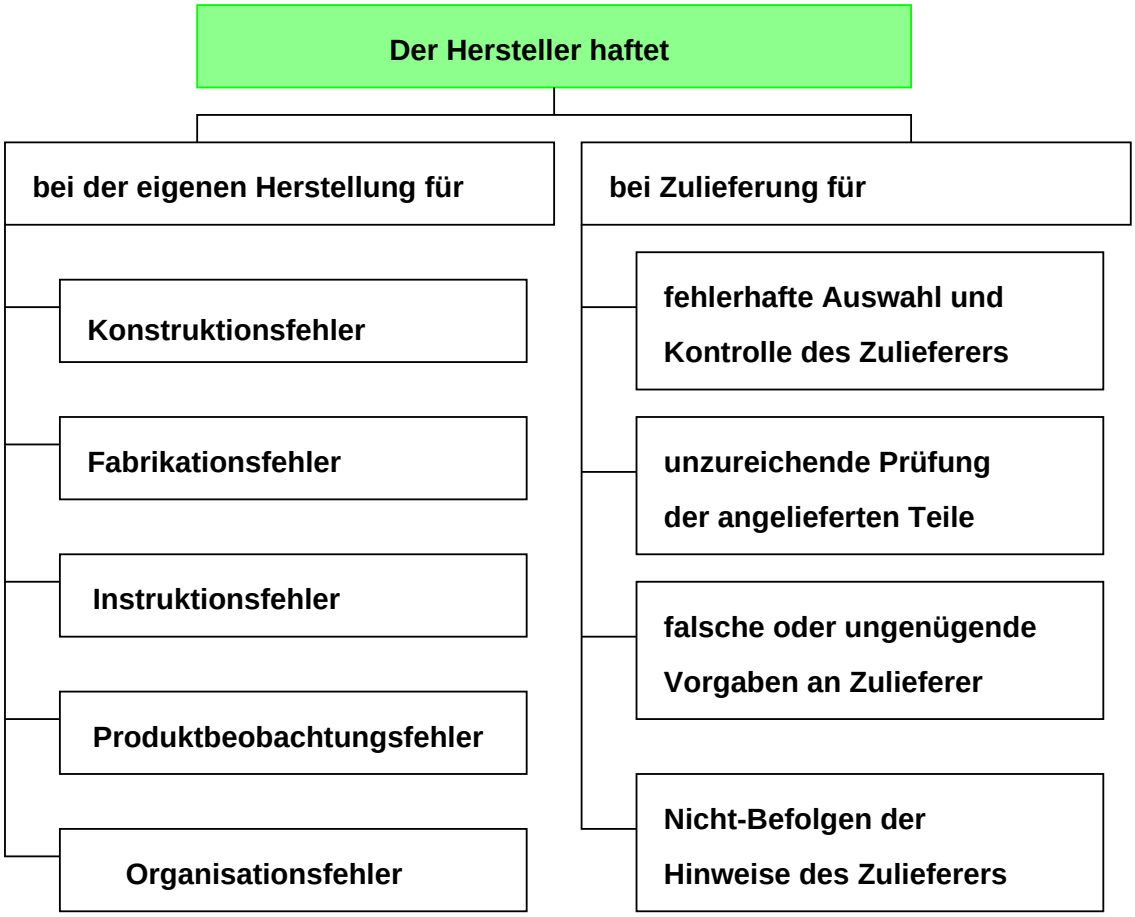


Abbildung 12.5: Haftung nach PHG

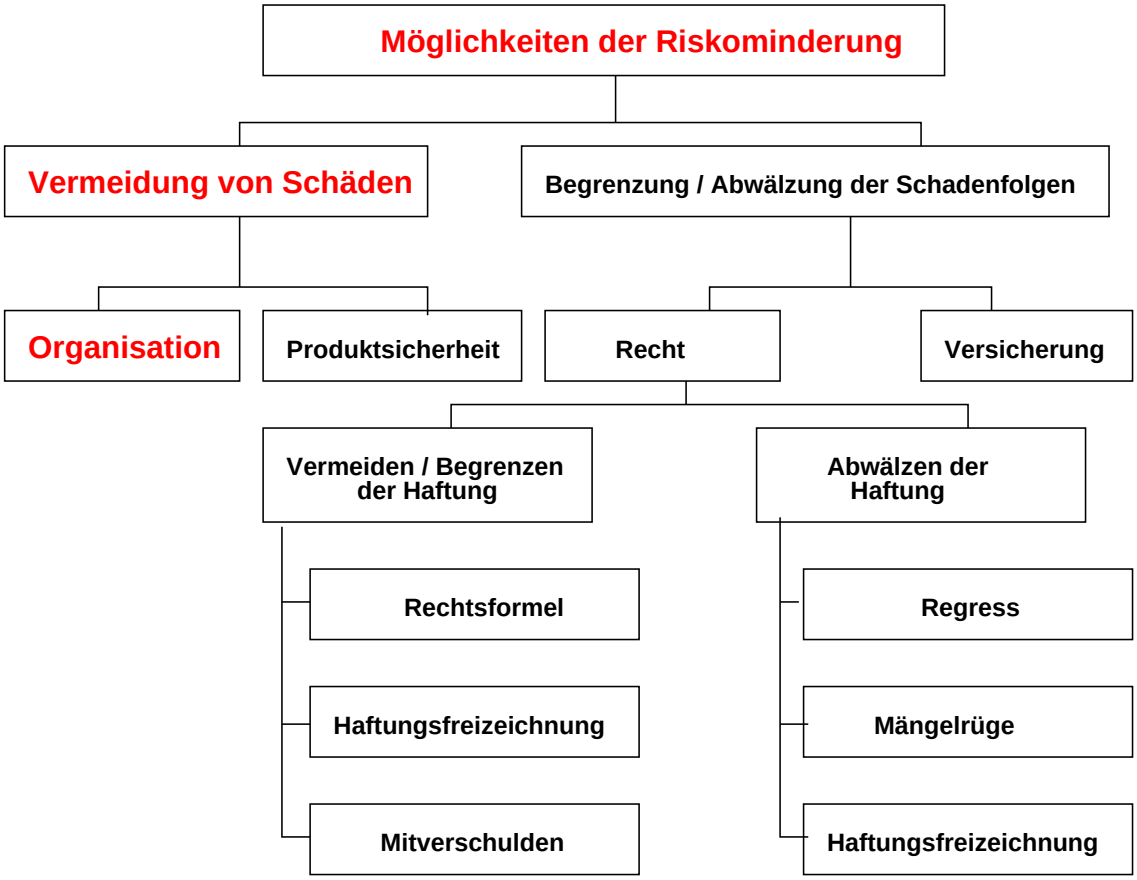


Abbildung 12.6: Risikominderung

12.4.2 KonTraG

Vom Gesetzgeber: Vertragsrecht (BGB) – Gesellschaftsrecht (AktG, HGB, . . . – Spezialgesetze (PHG, . . .) – Strafrecht

relativ **neu: KonTraG** – gesetzlich verordnetes Risikomanagement?

Das Bundesjustizministerium veröffentlichte Ende 1996 die Entwürfe zum **Gesetz zur Kontrolle und Transparenz im Unternehmensbereich** (KonTraG) und ein **Kapitalaufnahmeerleichterungsgesetz** (KapAEG); diese traten im April 1998 in Kraft!

Auf der Basis von §91 II AktG sind Vorstände deutscher Aktiengesellschaften explizit zur Einrichtung eines Risiko-Management-Systems verpflichtet!

Geltungsbereich:

- Alle Kapitalgesellschaften
- Börsennotierte Aktiengesellschaften

- Gesellschaften, die einen Aufsichtsrat haben
- Nach breiter Auffassung auch für GmbHs, aber auch für andere Rechtsformen

Im Falle einer "Unternehmenskrise" hat der Vorstand auf Basis von §93 II AktG zu beweisen, dass er sich objektiv und subjektiv pflichtgemäß verhalten hat, d. h., er muss nachweisen, dass er Maßnahmen zur Risikofrüherkennung und Risikoabwehr getroffen hat!

Die IT gilt als wichtiger Risiko-Bereich:

- Organisatorische Risiken
  - Schutz von Daten und Programmen vor unzulässigen Zugriffen
  - Anwendungsprogramme werden nicht prozessbezogen eingesetzt
- Denial-of-Service
- Anwendungs- und prozessbezogene Risiken
  - veraltete, nicht integrierte Softwarelösungen
  - wichtig: strategische Neu-Konzipierungen
- Kosten- und leistungsbezogene Risiken
  - IT-Kostenrechnung?
- Projektbezogene Risiken
  - Kosten- und / oder Terminüberschreitungen
  - kein professionelles Projektmanagement
  - ...
- Infrastrukturelle Risiken
  - kein Sicherungskonzept, kein Notfallplan, kein Wiederanlaufplan
  - baulich-technisch Standards werden nicht erfüllt: Schutz vor Zutritt, Feuer, Energieausfall

Notwendig, um den neuen Herausforderungen gerecht zu werden, sind:

- Kenntnis der Stärken und Schwächen, also **des Profils**: Wo lohnen sich Verbesserungen?
- Vermittlung des Profils nach außen, zum Kunden hin
- **Standortbestimmung**: wie stehen wir im Vergleich zu unseren **Mitbewerbern**?
- **Standortbestimmung**: wie stehen wir relativ zu einer **Zielvorstellung** (Ideal-Unternehmen)?
- **Bewertung von Veränderungen**: geht es in die richtige Richtung und stellt die Verbesserung tatsächlich eine Verbesserung dar

  Mark Twain: *Als wir das Ziel aus den Augen verloren, verdoppelten wir unsere Anstrengungen*

Hilfreich:

- **allgemein anerkannte Prozessmodelle**
  - einstufig (Messlatte)
  - mehrstufig: verbesserungsorientiert
- (quantitative) Bewertungsschemata
- Bewertungsverfahren

Begriffe

- **Appraisal**: Begutachtung, Beurteilung, Bewertung, Einschätzung, Untersuchung, Würdigung
- **Assessment**: Umlage, Einschätzung, Bewertung, Beurteilung, Bemessung, Abschätzung
- **self assessment**: Selbsteinschätzung
- **supplier assessment**: Lieferantenbeurteilung
- **performance assessment**: Leistungsbewertung

Bewertung verlangt Bewertungsmerkmale (was), Bewertungskriterien (woran), Bewertungsskalen (wie) sowie eine Sollvorstellung (warum) ↪ **Soll-Modell**

Anforderungen wie z. B.:

- **Validität**
- **Verlässlichkeit**
- **Normierung**
- **Objektivität**

12.4.3 Übersicht

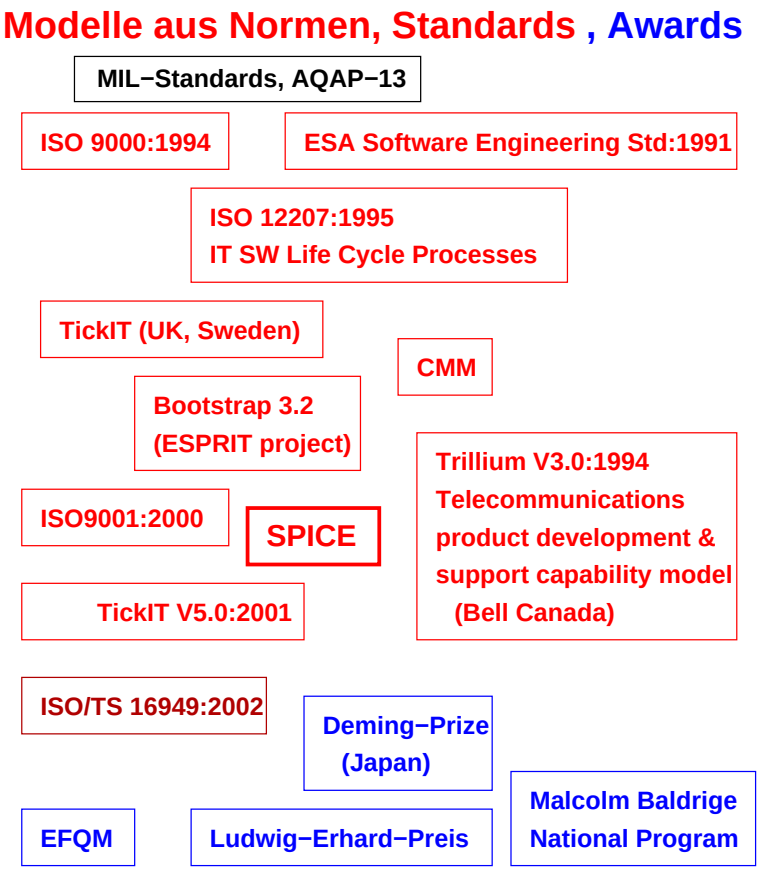


Abbildung 12.7: CMM & Co.

Anm.: Diese "Kästchen" sind nur bedingt miteinander vergleichbar!

12.4.4 MBNQA – Baldrige Award

Malcolm Baldrige National Quality Award

Quelle: **www.quality.nist.gov** 

Aus dem Vorwort:

*The U.S. Department of Commerce is responsible for the Baldrige National Quality Program and the Award. The National Institute of Standards and Technology (**NIST**), an Agency of the Department's Technology Administration, manages the Baldrige Program. NIST promotes U.S. economic growth by working with industry to develop and deliver the high-quality measurement tools, data, and services necessary for the nation's technology infrastructure.*

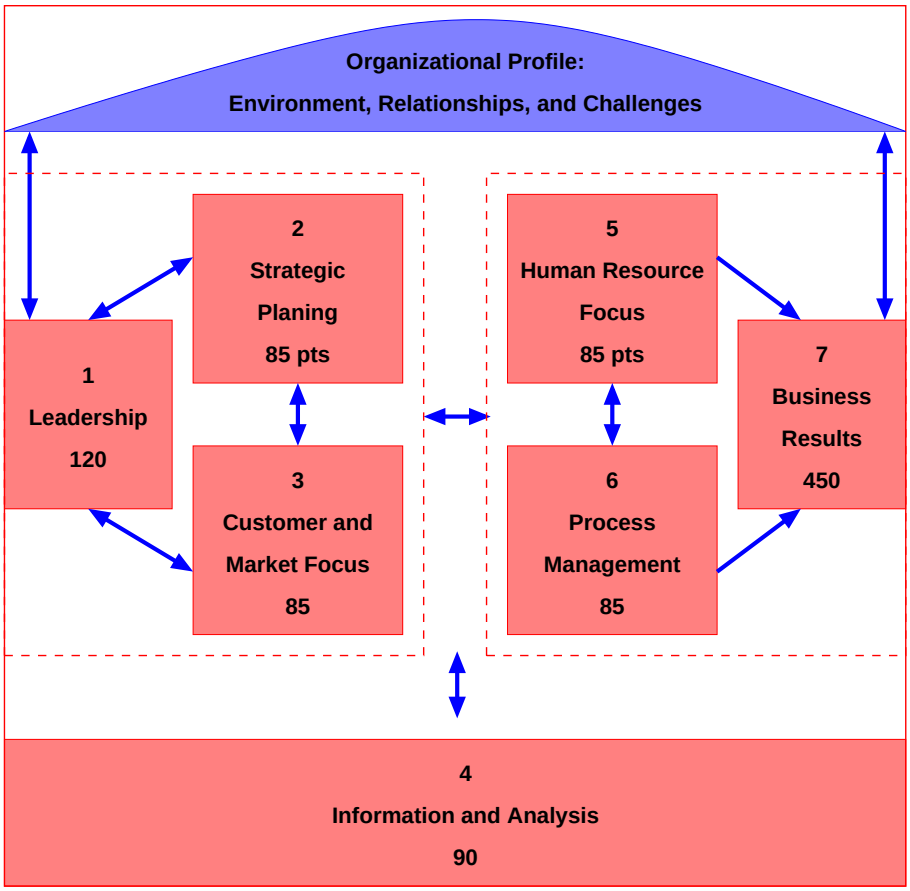


Abbildung 12.8: Baldrige Criteria for Performance Excellence Framework

...

*The American Society for Quality (**ASQ**) assists in administering the Award program under contract to NIST.*

- **"Organizational Profile"**
  - bestimmt den Kontext in dem das Unternehmen operiert
  - bestehend aus seiner Umwelt, seinen Beziehungen in dieser Umwelt (betriebswirtschaftlich, volkswirtschaftlich, gesellschaftlich) sowie seinen strategischen Zielen
  - bildet die Leitlinie für die Beurteilung des (Management) Systems
- Das **System** wird in 7 Kategorien eingeteilt, die die **Organisation**, die **Prozesse** und deren **Ergebnisse** (sehr weit gefasst: Kundenzufriedenheit, finanziell, gesellschaftlich, Mitarbeiterzufriedenheit, ...) betreffen

Bewertung:

- für alle Kategorien sind zu beantwortende Fragen verfügbar

- die Bewertung der Antworten erfolgt “unscharf”:

Score	Approach-Deployment
0%	No systematic approach is evident; information is anecdotal (not systematic)
10% to 20%	The beginning of a systematic approach to the basic requirements of the Item is evident. Major gaps exist in …
…	…

mehr zum **Self Assessment**: *e-baldrige*, ein web-basiertes self-assessment tool

www.quality.nist.gov/eBaldrige/Step_one.htm

12.4.5 EFQM

European Foundation for Quality Management
www.efqm.org

- 1987 gegr. als Antwort auf MBNQA und Deming Award (Japan)
- entwickelt Anfang der 90er Grundlagen für den ab 92 vergebenen European Quality Award (EQA)
- Nationaler Preis: **Ludwig Erhard Preis**
- Nutzung von Erfahrungen aus USA und Japan
- Zielsetzung:
  - Verbreitung von TQM in Europa
  - Stellung der europäischen Industrie auf dem Weltmarkt festigen / ausbauen
- periodische Selbstbewertung (Stärken, Verbesserungsmöglichkeiten)
- “am Besten messen” (**Excellence Model**)

Abbildung 12.9: Grundprinzip des EFQM-Awards

Basis: die drei fundamentalen Säulen von TQM, nämlich die gleichzeitige Beachtung von Menschen, Prozessen und Ergebnissen

- Durch Einbindung aller **Mitarbeiter**
- in einen **kontinuierlichen Verbesserungsprozess**
- **bessere Ergebnisse** erzielen

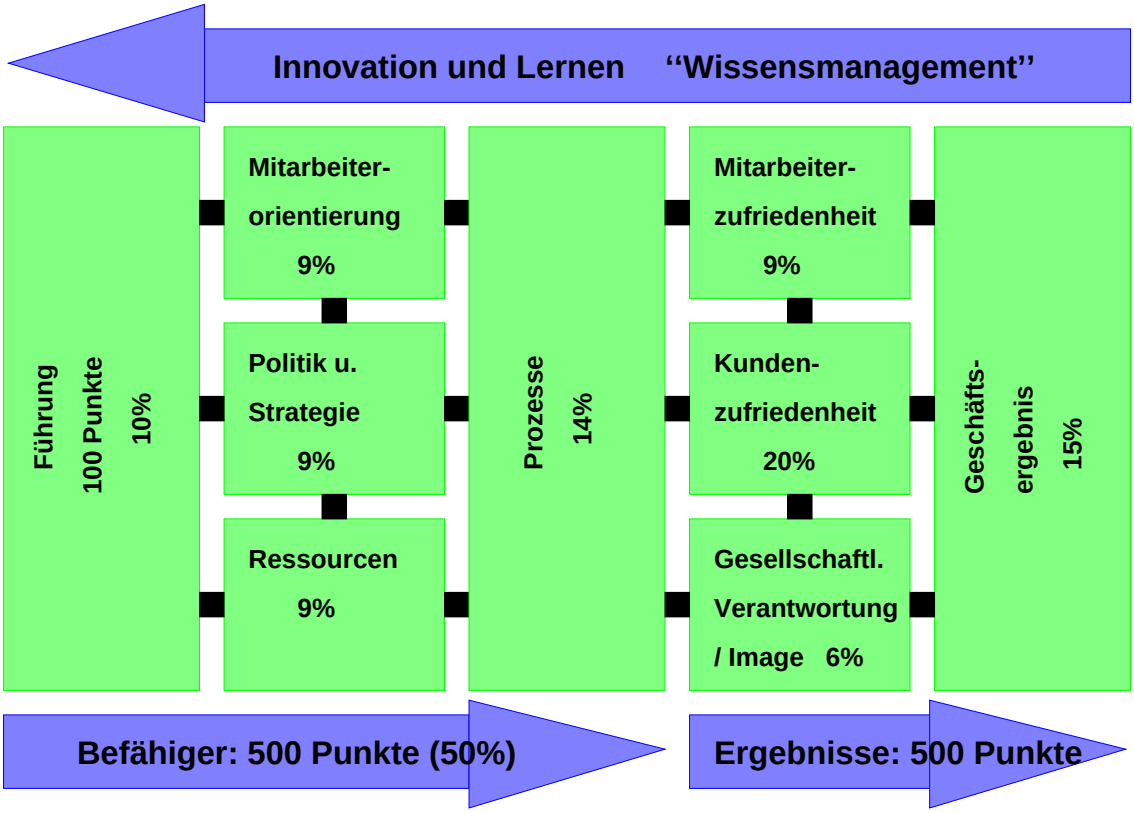


Abbildung 12.10: EFQM Excellence Model – neueste Version: 1999

(1) **Führung**
bezieht sich auf das Verhalten aller Führungskräfte, um das Unternehmen zu umfassender Qualität zu führen

Aus der Selbstbewertung sollte hervorgehen:

- wie Führungskräfte ihr Engagement für eine TQM-Kultur sichtbar unter Beweis stellen
- wie Führungskräfte den Verbesserungsprozess und die Mitwirkung daran fördern, indem sie geeignete Ressourcen zur Verfügung stellen und Unterstützung geben
- wie Führungskräfte sich um Kunden und Lieferanten sowie andere externe Organisationen bemühen
- wie Führungskräfte Anstrengungen und Erfolge ihrer Mitarbeiter anerkennen und würdigen

(2) **Politik & Strategie**
bezieht sich auf den "Daseinszweck", das Wertesystem, das Leitbild und die strategische Ausrichtung sowie die Art und Weise, wie diese Aspekte verwirklicht werden

Aus der Selbstbewertung sollte hervorgehen:

- wie Politik und Strategie entwickelt werden
- wie Politik und Strategie auf relevanten und umfassenden Informationen beruhen
- wie Politik und Strategie bekanntgemacht und eingeführt werden
- wie Politik und Strategie regelmäßig aktualisiert und verbessert werden

(3) **Mitarbeiterorientierung**
bezieht sich auf den Umgang des Unternehmens mit seinen Mitarbeitern

Aus der Selbstbewertung sollte hervorgehen:

- wie Mitarbeiterressourcen geplant und verbessert werden
- wie die Fähigkeiten der Mitarbeiter aufrechterhalten und weiterentwickelt werden
- wie Ziele mit Mitarbeitern vereinbart und die Leistungen kontinuierlich überprüft werden
- wie Mitarbeiter beteiligt, zu selbständigem Handeln autorisiert und wie ihre Leistungen anerkannt werden
- wie ein effektiver Dialog zwischen Mitarbeitern und Organisation erreicht wird
- wie für die Mitarbeiter gesorgt wird

(4) **Ressourcen**
bezieht sich den wirksamen Einsatz der Ressourcen zur Unterstützung der Unternehmenspolitik und -strategie

Aus der Selbstbewertung sollte hervorgehen:

- wie die Organisation ihre finanziellen Ressourcen handhabt
- wie die Organisation ihre Informations-Ressourcen handhabt
- wie die Organisation ihre Beziehungen zu Lieferanten handhabt
- wie die Organisation Material bewirtschaftet
- wie die Organisation Gebäude, Einrichtungen und anderes Anlagevermögen handhabt
- wie die Organisation Technologie und geistiges Eigentum handhabt

(5) **Prozesse**
bezieht sich darauf, wie Prozesse identifiziert, überprüft und ggf. geändert werden, um eine ständige Verbesserung der Geschäftstätigkeit zu gewährleisten

Aus der Selbstbewertung sollte hervorgehen:

- wie die für den Geschäftserfolg wichtigen Prozesse identifiziert werden
- wie Prozesse systematisch geführt werden
- wie Prozesse überprüft und Verbesserungsziele gesetzt werden
- wie Prozesse durch Innovation und Kreativität verbessert werden
- wie Prozesse geändert werden und der Nutzen der Änderung bewertet wird

(6) **Kundenzufriedenheit**

Aus der Selbstbewertung sollte hervorgehen:

- wie die Beurteilung der Produkte, Dienstleistungen und Kundenbeziehungen aus Sicht der Kunden erfolgt
- wie zusätzliche Messgrößen herangezogen werden, um die Zufriedenheit der Kunden mit der Organisation festzustellen

(7) **Mitarbeiterzufriedenheit**

Aus der Selbstbewertung sollte hervorgehen:

- wie die Mitarbeiter ihre Organisation beurteilen
- wie zusätzliche Messgrößen herangezogen werden, um die Zufriedenheit der Mitarbeiter mit der Organisation festzustellen

(8) **Gesellschaftliche Verantwortung / Image**
bezieht sich auf Fragen, was das Unternehmen im Hinblick auf die Erfüllung der Bedürfnisse und Erwartungen der Öffentlichkeit insgesamt leistet; dazu gehören die Bewertung der Einstellung des Unternehmens zu Lebensqualität, Umwelt und Erhaltung der globalen Ressourcen sowie aller unternehmensinterner Maßnahmen in diesem Zusammenhang

Aus der Selbstbewertung sollte hervorgehen:

- die Beurteilung des Unternehmens durch die Gesellschaft
- zusätzliche Messgrößen mit Bezug auf die Zufriedenheit

(9) **Geschäftsergebnisse**
bezieht sich auf Fragen, was das Unternehmen im Hinblick auf seine geplanten Unternehmensziele und die Erfüllung der Bedürfnisse und Erwartungen aller finanziell am Unternehmen Beteiligten sowie bei der Verwirklichung seiner geplanten Geschäftsziele leistet

Aus der Selbstbewertung sollte hervorgehen:

- mit welchen finanziellen Messgrößen die Leistung der Organisation gemessen wird
- mit welchen sonstigen Messgrößen die Leistung der Organisation gemessen wird

12.4.6 CMMI

CMMI® **C**apability **M**aturity **M**odel **I**ntegration

- Software Engineering Institute der Carnegie Mellon University
  www.sei.cmu.edu/cmmi/models/models.html
- Bisherige Ergebnisse: **www.sei.cmu.edu/cmmi/results.html**
- Ausgangspunkt: **W. Humphrey: Managing the Software Process. Addison Wesley, 1989 (!)**
- Ein Prozessmodell zur Beurteilung und Verbesserung der Qualität („Reife“) von Produkt-Entwicklungsprozessen in Organisationen
- Kann verwendet werden, um
  - um die Stärken und Schwächen einer Produktentwicklung „objektiv“ zu analysieren,
  - um Verbesserungsmaßnahmen zu bestimmen und diese in eine „sinnvolle“ Reihenfolge zu bringen.
- Primär: ein Mittel, die Produktentwicklung zu verbessern
- Sekundär: eine offizielle Überprüfung / Feststellung eines Reifegrades (***Appraisal***) – eine in der Industrie de-facto anerkannte Auszeichnung

CMMI

- die neue Version des Software Capability Maturity Model
- ersetzt verschiedene Qualitäts-Modelle für unterschiedliche Entwicklungs-Disziplinen (z.B. Software-Entwicklung, System-Entwicklung)
- integriert diese in einem neuen, modularen Modell
- ermöglicht damit
  - die Integration weiterer Entwicklungs-Disziplinen (z.B. Hardware-Entwicklung)
  - die Anwendung des Qualitäts-Modells in übergreifenden Disziplinen (z.B. Entwicklung von Chips mit Software).

Etwas Historie (nach Wikipedia)

1986 auf Initiative des US-Verteidigungsministeriums (DoD) beginnt das Software Engineering Institute (SEI) an der Carnegie Mellon University, Pittsburgh, mit der Entwicklung eines Systems zur Bewertung der Reife von Softwareprozessen

1991 Capability Maturity Model 1.0

1992 überarbeitet zur Version 1.1

1997 CMM 2.0 wird kurz vor der Verabschiedung vom DoD zurückgezogen: das CMMI-Projekt wird gestartet

2000 CMMI – damals unter dem Namen **Capability Maturity Model Integrated** – erscheint als Pilotversion 1.0

2002 CMMI wird unter dem neuen Namen **Capability Maturity Model Integration** (kurz CMMI) freigegeben

Einordnung des Modells

- CMMI definiert Anforderungen (das **Was**) an eine „gute“ Produktentwicklung, keine konkreten Schritte (kein **Wie**)
- Unterstützung von **kontinuierlicher** Prozessverbesserung, indem Anforderungen an eine bzw. Kriterien einer professionellen Produkt-Entwicklungs-Organisation definiert werden
- Definition des Entwicklungsprozesses selbst ist Aufgabe der Organisation und ist eine wichtige Teilaufgabe der Prozessverbesserung
- Da CMMI keinen konkreten Entwicklungsprozess definiert, kann es auf sehr unterschiedliche Organisationen und Organisationsgrößen angewandt werden – **Beispiel:** Forderung “bei der Projektplanung muss eine Zustimmung der Projektbeteiligten zum Projektplan eingeholt werden” kann sehr unterschiedlich konkret in einer Organisation umgesetzt werden
- Ergo: Es gibt nicht **die eine** richtige Umsetzung von CMMI.

Nota bene:

- CMMI adressiert nicht nur die Entwicklungsprojekte an sich, sondern auch projektbezogenen Aufgaben der Organisation (Bereitstellung von Ressourcen, Durchführung von Schulungsmaßnahmen, . . .)
- CMMI legt sehr viel Wert auf den gelebten, nicht nur dokumentierten Prozess (**keine *Schrankware***)
- CMMI definiert eine Reihe von **Prozessgebieten**: Projektplanung, Anforderungsentwicklung, organisationsweite Prozessdefinition, . . .
- Ein **Prozessgebiet** spezifiziert Anforderungen an eine professionelle Produkt-Entwicklung in einem bestimmten Gebiet durch ein Bündel verwandter **Praktiken**, die, gemeinsam ausgeführt, eine Reihe von Zielen erfüllen, die für eine deutliche Verbesserung auf diesem Gebiet wichtig sind
- Für die Prozessgebiete, Ziele und Praktiken gibt CMMI jeweils zusätzliche erklärende Informationen:
  - Prozessgebiete werden erläutert, die in Verbindung stehenden Prozessgebiete werden genannt
  - Jede Praktik wird durch Erklärungstexte, durch typische Arbeitsergebnisse und durch typische Arbeitsschritte weiter erläutert
  - Diese Hinweise sollen bei der Umsetzung helfen, sind keine Prüfgrundlage in einem Appraisal

Beispiel:

- Prozessgebiet „Projektplanung“
- Ziele: „Schätzungen aufstellen“, „Einen Projektplan entwickeln“ und „Verpflichtung auf den Plan herbeiführen“
- Praktiken zum Ziel „Schätzungen aufstellen“: „Umfang des Projekts schätzen“, „Attribute der Arbeitsergebnisse und Aufgaben schätzen“, „Projektlebenzyklus definieren“ und „Schätzungen von Aufwand und Kosten aufstellen“

Einteilung der Prozessgebiete in Kategorien:

- Projektmanagement (typischerweise projektbzogen)
- Entwicklung (typischerweise projektbzogen)
- Unterstützung (sowohl projekt- wie organisationsbezogen)
- Prozessmanagement (organisationsweite Aufgabe)

CMMI fokusiert auf

- Praktiken, die spezifisch für ein Prozessgebiet sind
- die Institutionalisierung der Prozesse
  - die Prozesse sollen in der Organisation **selbstverständlich** und als Teil der täglichen Arbeit **gelebt werden**
  - auch hierfür werden Praktiken definiert: sog. generische Praktiken (für alles Prozessgebiete gleich)
  - deren Umsetzung ist Aufgabe der Organisation

Fähigkeitsgrade (CMMI Continuous Representation)

- Fähigkeitsgrad (*capability level*): Grad der Institutionalisierung eines einzelnen Prozessgebiets
  - **0 Incomplete** – Ausgangszustand, keine Anforderungen
  - **1 Performed** – die spezifischen Ziele des Prozessgebiets werden erreicht
  - **2 Managed / Repeatable** – der Prozess wird „gemanagt“
  - **3 Defined** – der Prozess wird auf Basis eines **angepassten Standard-Prozesses** „gemanagt“ und verbessert
  - **4 Quantitatively Managed** – der Prozess steht unter quantitativer / statistischer Prozesskontrolle
  - **5 Optimizing** – der Prozess wird mit den Daten aus der quantitativen / statistischen Prozesskontrolle ständig verbessert

Reifegrade (CMMI Staged Representation)

- Reifegrad (*maturity level*): umfasst eine Menge von Prozessgebieten, die zu einem bestimmten Fähigkeitsgrad umgesetzt sein müssen – jeder Reifegrad ist ein Entwicklungsniveau in der Prozessverbesserung der Organisation

  1 **Initial** – keine Anforderungen, hat jede Organisation automatisch

  2 **Managed / Repeatable** – Projekte werden unter einem definierten Projektmanagement durchgeführt und ähnliche Projekte könnte erfolgreich wiederholt werden

  3 **Defined** – die Projekte werden nach einem angepassten Standard-Prozess durchgeführt, und es gibt eine kontinuierliche Prozessverbesserung

  4 **Quantitatively Managed** – es wird eine quantitative / statistische Projektkontrolle durchgeführt

  5 **Optimizing** – die Projekte werden mit den Daten aus der quantitativen / statistischen Projektkontrolle verbessert

Die einzelnen Stufen, vereinfacht, nach D.Malzahn (www.INTER-man.org): QS-SW, Seminar an der TAE

Level 1: Helden und Künstler

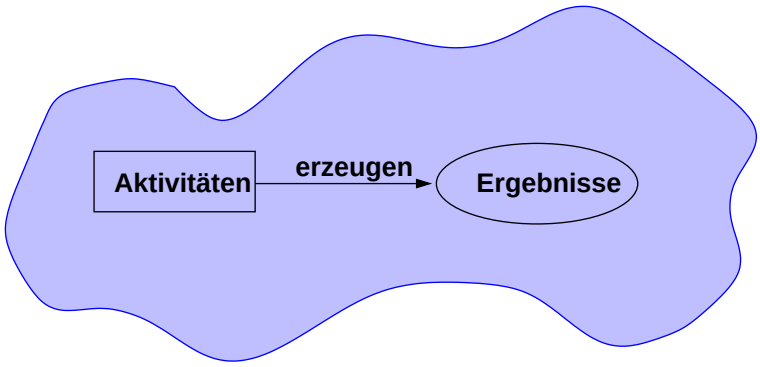


Abbildung 12.11: CMM - Level 1

Level 2: DENKE, BEVOR und NACHDEM Du etwas gemacht hast, um sicherzustellen, dass Du es richtig gemacht hast

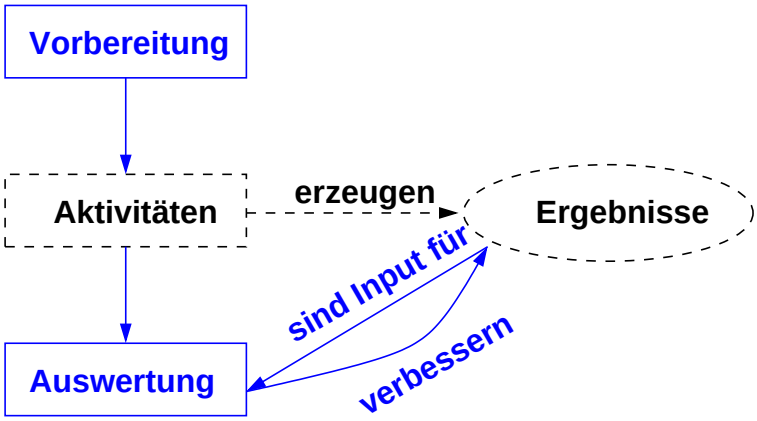


Abbildung 12.12: CMM - Level 2

Schlüsselprozesse:

- Requirements-Management
- Project Tracking and Review
- Quality Assurance
- Project Planning
- Subcontract(or) Management
- Configuration Management

Beispiel: **Level 2 – Repeatable**

- Die SW-Entwicklung wird für jedes Projekt geplant!
- Schätzungen für Kosten- und Zeitaufwand werden erstellt und dokumentiert!
- Fehler und Abweichungen werden erfasst und beseitigt!
- ...
- Projekterfolg ist allerdings von fähigen Mitarbeitern abhängig!
- Kein einheitlicher Prozess über mehrere Projekte hinweg!

Level 3: VERWENDE, was Du gelernt hast

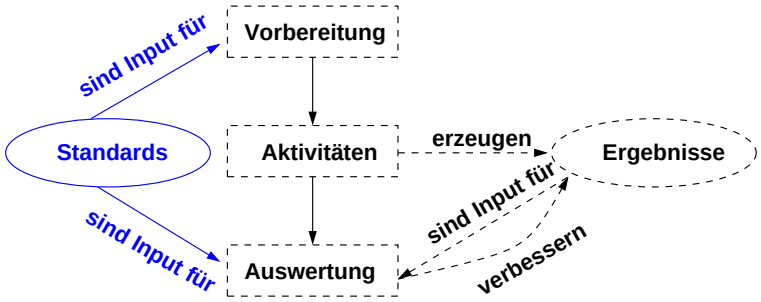


Abbildung 12.13: CMM - Level 3

Schlüsselprozesse:

- Organization Process Focus
- Training Program
- Software Product Engineering
- Peer Reviews
- Organization Process Def,
- Integrated Software Mgmt.
- Intergroup Coordination

Level 4: erzeuge selbsterfüllende Prophezeiungen

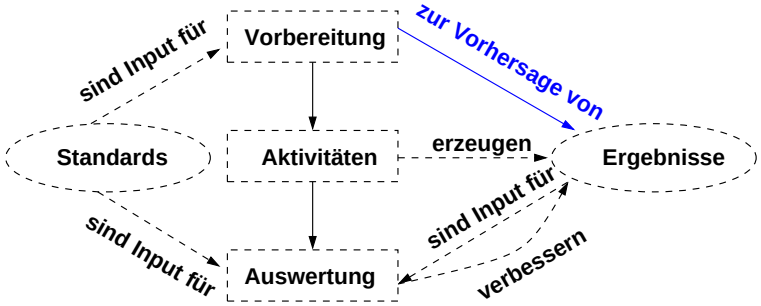


Abbildung 12.14: CMM - Level 4

Schlüsselprozesse:

- Software Quality Management
- Quantitative Process Management

Level 5: entwickle ein Bewusstsein der permanenten Verbesserung

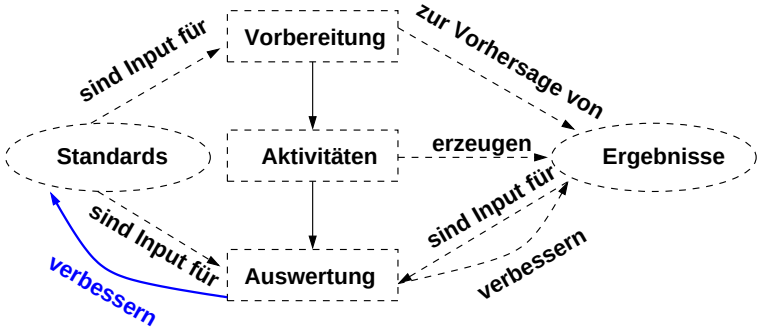


Abbildung 12.15: CMM - Level 5

Schlüsselprozesse:

- Defect Prevention
- Change Management for Technology
- Change Management for Processes

Beispiele für Schlüsselprozesse, deren Ziele und Aktivitäten

- Requirements Management
  - Ziele
    - Die Anforderungen werden so verwaltet, dass sie als Arbeitsgrundlage für das Software Engineering und das Projekt-Management dienen
    - Die Anforderungen werden konsistent in Pläne, Aktivitäten und Ergebnisse abgebildet
  - Aktivitäten
    - Anforderungen werden geprüft bevor sie zu Projektvorgaben werden
    - Änderungen an Anforderungen werden geprüft und in die Projektvorgaben übernommen
- Software Project Planning
  - Ziele
    - Projektschätzungen werden dokumentiert und fließen in die Projekt-Planung und in das Controlling ein
    - Aktivitäten, Forderungen und Ziele: geplant, dokumentiert
    - Personalressourcen vorhanden, akzeptieren Planung
  - Aktivitäten
    - Planungen werden nach einem dokumentierten Prozess überprüft, bevor sie freigegeben werden
    - Vorgehensmodell und Lebenszyklus werden definiert
    - Reviews und Teilergebnisse werden für das Controlling dokumentiert

    - Zwischenprodukte, Termine und Personalressourcen werden nach einem dokumentierten Prozess geplant
    - Bezüglich Kosten, Ressourcen, Terminen und Methoden erfolgt eine dokumentierte Risikoabschätzung
    - Alle Planungsaktivitäten werden dokumentiert

Bewertungsverfahren: **Assessment** oder **Appraisal**?

- Überprüfung einer Organisation hinsichtlich der Umsetzung der Anforderungen eines Prozessmodells (CMMI, ISO15504)
- SEI verwendet **Appraisal** als Oberbegriff, während **Assessment** sich nur auf solche *Appraisals* bezieht, die zur eigenen Prozessverbesserung dienen.
- **Evaluation**: ein *Appraisal*, das durch einen (möglichen) Auftraggeber zur Auswahl oder zur Überwachung seiner Lieferanten durchgeführt wird
- Bei CMM, dem Vorläufer des CMMI, gab es für *Assessments* und *Evaluations* verschiedene Methoden, bei CMMI wurden diese zusammengefasst und es ist jetzt nur noch von *Appraisals* die Rede
- Bei ISO 15504 keine Unterscheidung, als Oberbegriff wird hier *Assessment* verwendet

SEI definiert drei Typen von *Appraisals*:

- **Class A Appraisal**

  hoher Umfang, ca. 1x im Jahr, mindestens drei Nachweise je Aussage, hat Rating (*capability level* und/oder *maturity level*) zum Ziel
- **Class B Appraisal**

  mittler Umfang, ca. 1-2x im Jahr, mindestens zwei Nachweise je Aussage, hat Bestimmung von Stärken und Schwächen für die Prozessverbesserung zum Ziel
- **Class C Appraisal**

  kleiner Umfang, öfters durchgeführt, ein Nachweis je Aussage, hat kontinuierliche Beobachtung/Bewertung der Prozessverbesserung zum Ziel

Für jeden hat SEI eine Vorgehensweise veröffentlicht:

SCAMPI: **S**tandard **C**MMI **A**ppraisal **M**ethod for **P**rocess **I**mprovement

Abgrenzung zu anderen Normen:

- DIN EN ISO 9001 deckt die gesamte Organisation ab und geht somit mehr in die Breite
- CMMI ist speziell für den Produkt-Entwicklungsprozess, geht mehr in die Tiefe und gibt konkrete Prozessgebiete und Praktiken vor

- beide haben denselben Grundgedanken, Anforderungen von CMMI lassen sich auf die Anforderungen von der DIN EN ISO 9001 abbilden
  siehe: http://www.sei.cmu.edu/cmmi/adoption/iso-mapping.html
  andere Abbildungen:
  siehe: http://www.sei.cmu.edu/cmmi/adoption/comparisons.html
- CMMI setzt die Anforderungen der Norm ISO 15504 (SPICE) an ein Prozessmodell um!
- Das Appraisal-Verfahren SCAMPI setzt die Anforderungen der Norm ISO 15504 an ein Bewertungsverfahren um.
- ISO 12207 (Modell für *SOFTWARE LIFE CYCLE PROCESSES*) und ISO 15288 (Modell für *System Life Cycle Processes*) gehen **nicht** über die Definition der Titel der Praktiken von CMMI hinaus
- sie geben keine umfangreichen Erklärungen wie CMMI
- es gibt auch keine Integration der beiden Normen
- inhaltlich fordern sie im wesentlichen das Gleiche wie CMMI
- CMMI bezieht sich vor allem die Entwicklung von Produkten oder auf Wartungsprojekte zu existierenden Produkten
- Ein Prozessmodell für den Betrieb von Anwendungen stellt **ITIL** dar

12.4.7 IT Infrastructure Library

- im Auftrage der britischen Regierung entwickelter Leitfaden (BS 15000)
- heute der weltweite De-facto-Standard im Bereich **Service Management**
- beinhaltet eine umfassende und öffentlich verfügbare fachliche Dokumentation zur Planung, Erbringung und Unterstützung von IT-Serviceleistungen
  siehe **www.itil.org**
- bietet die Grundlage zur Verbesserung von Einsatz und Wirkung einer operationell eingesetzten IT-Infrastruktur
- beschreibt die Architektur zur Etablierung und zum Betrieb von IT Service Management

12.4.8 SPICE

Software **P**rocess **I**mprovement and **C**apability d**E**termination (ISO 15504)

Homepage: **http://www.sqi.gu.edu.au/spice/**

- 1998 als Technischer Report verabschiedet
- stellt ein Modell für das Assessment von Softwareprozessen
- Kernpunkte: Verbesserung von Prozessen (**Improvement**) und Bestimmung des Prozessreifegrads (**Capability Determination**)

- im Mittelpunkt steht das Self-Assessment

Prozess-Assessments

- werden anhand eines zweidimensionalen Referenz- und Assessment-Modells durchgeführt
- die Prozess-Dimension dient zur Kennzeichnung der Vollständigkeit von Prozessen
- die Reifegrad-Dimension dient der Bestimmung ihrer Leistungsfähigkeit
-

Prozess-Dimension:

jeder Prozess wird einer von fünf Kategorien zugeordnet

- Kunden-Lieferanten-Prozesse
- Entwicklungsprozesse
- Unterstützende Prozesse
- Managementprozesse
- Organisationsprozesse

Alle werden wegen ihrer hohen Komplexität weiter unterteilt zu insgesamt 27 Prozessen – diese werden jeweils durch grundlegende auszuführende Aktivitäten und deren Ergebnisse beschrieben (siehe ISO 12207 - Prozesse im Software-Lebenszyklus)

Reifegrad-Dimension:

- besteht aus 6 Reifegradstufen:
  unvollständig, durchgeführt, gesteuert, definiert, vorhersagbar, optimiert
- treffen Aussagen über die Leistungsfähigkeit der in der Prozess-Dimension beschriebenen Prozesse
- Den einzelnen Stufen sind die Aktivitäten zugeordnet, die dazu führen, dass die Ergebnisse systematisch erarbeitet und am Ende des Prozesses in der definierten Qualität vorliegen
- Den Reifegradstufen sind 9 Prozessattribute zugeordnet:
  - Diese werden jeweils durch die ihnen zugeordneten grundlegenden Managementaktivitäten beschrieben
  - dienen der Beurteilung der Prozesse

Reifegradbestimmung:

- Der Reifegrad wird für jeden Prozess einzeln bestimmt

- Es wird nicht nur die Existenz einer Prozessaktivität beurteilt, sondern auch die adäquate Durchführung der Aktivität bewertet
- Bewertungsskala:

  nicht erfüllt, teilweise erfüllt, weitgehend erfüllt, vollständig erfüllt
- Bei der Bewertung muss **objektiv** nachgewiesen werden, dass die Anforderungen auf der entsprechenden Stufe erfüllt werden (z.B. anhand von Arbeitsprodukten, welche als Ergebnisse aus den Prozessen hervorgehen

Auswertung des Prozess-Assessments:

- Die Prozess- und die Reifegraddimension zusammengeführt
- Die untersuchten Prozesse werden ihren Bewertungen in Form der Erfüllungsgrade der 9 Prozessattribute gegenübergestellt
- Der Fähigkeitsgrad wird für jeden Prozess einzeln bestimmt
- In der Gesamtheit aller untersuchten Prozesse ergibt sich daraus ein Stärken-Schwächen-Profil
  - daraus ergeben sich Verbesserungspotentiale
  - Beschreibungen des jeweils nächsthöheren Reifegrades zeigen Möglichkeiten zur Verbesserung der Prozesse auf

Literaturverzeichnis

[1] Balzert, H.: *Lehrbuch der Software-Technik (1/2).* Spektrum Akademischer Verlag, Heidelberg, Berlin, Oxford, 2000

[2] Bartsch, M.: *Software und das Jahr 2000.* Nomos Verlagsgesellschaft, Baden-Baden 1998

[3] Bauer, C.; a.a.: *Produkthaftung – Herausforderung an Manager und Ingenieure.* Springer 1994

[4] Beck, Kent: *Extreme Programming Explained: Embrace Change.* Addison-Wesley 1999.

[5] Beck, K,; Fowler, M.: *Planning Extreme Programming.* Addison Wesley 2000

[6] Beizer, B.: *Software System Testing and Quality Assurance.* Van Nostrand Reinhold Company, New York, 1984

[7] Beizer, B.: *Black-Box-Testing – Techniques for Functional Testing of Software and Systems.* John Wiley & Sons 1995

[8] Binder, R.: *Testing Object-Oriented Systems: Models, Patterns, and Tools.* Addison Wesley 1999

[9] Brooks, Frederick P.: *The Mythical Man-Month – Essays on Software Engineering.* Addison-Wesley 1995

[10] Burkhardt, R.: *UML – Unified Modeling Language.* Addison-Wesley 1997

[11] chromatic: *An Introduction to Extreme Programming.* O'Reilly 2001

[12] Cockburn, A.: *Agile Software Development.* Addison Wesley 2003

[13] Councill, W.T.; Heinemann, G.T.: *Component-Based Software Engineering.* Addison-Wesley, 2001

[14] I. Crnkovic, U. Asklund, A.P. Dahlquist: *Implementing and Integrating Product Data Management and Software Configuration Management.* Artech House Publisher 2003 (ISBN: 1-58053-498-8)

[15] Dröschel W.; Wiemers M. (Hrsg.): *Das V-Modell 97.* Oldenbourg 2000

[16] Dumke, R.: *Softwareentwicklung nach Maß.* Vieweg Verlag 1992

[17] Farny, D.: *Versicherungsbetriebslehre.* Verlag Versicherungswirtschaft. 3. Auflage, Karlsruhe 2000 (ISBN 3884878581)

[18] Farny, D.: *Handwörterbuch der Versicherung.* Karlsruhe 1988

[19] Fuchs, M.: *Neutrale Standardsoftware-Auswahl durch geschäftsprozess-orientierten Leistungsvergleich von Unternehmens- und Standardsoftware-Modellen.* Dissertation an der Universität Ulm 1995

[20] *GPS SoftwareAtlas – Generalplan Industrieunternehmen.* Produkt der GPS mbH, Hörvelsinger Weg 54, D-89081 Ulm

[21] *GPS-Testbericht: Standardsoftware für den Mittelstand.* GPS mbH, Hörvelsinger Weg 54, D-89081 Ulm

[22] Grimm, K.: *Systematisches Testen von Software – Eine neue Methode und eine effektive Teststrategie.* Oldenbourg, 1995

[23] Hagen, W.: Experiences for SW Quality Assurance in IBM Switzerland. in: C. Frühauf (Ed.): Software Quality for People – Proceedings of the Fourth European Conference on Software Quality, October 17-20, 1994, Basel; vdf Hochschulverlag AG an der ETH Zürich

[24] Hansen, H. R.: *Wirtschaftsinformatik I.* UTB 802, G. Fischer, Stuttgart, 1995

[25] Heilmann, H.: *Workflow Management – Integration von Organisation und Informationsverarbeitung.* in: HMD 176/1994, S. 8-21, Forkel-Verlag

[26] Herold, H.; Mayer, M.: *SCCS und RCS - Versionsverwaltung unter UNIX.* Addison-Wesley 1995

[27] Humphrey, W.S.: *Managing the Software Process.* Addison Wesley 1989

[28] NN: *IEEE Standard Glossary of Software Engineering Terminology.* IEEE-Std. 729-1983

[29] NN: *IEEE Standard for Software Configuration Management Plans.* IEEE Std. 828-1983

[30] ISO/IEC 9126: *Information technology – software product evaluation – quality characteristics and guidelines for their use.* 1991 - auch: DIN 66272: Informationstechnik; bewerten von Softwareprodukten; Qualitätsmerkmale und Leitfaden zu ihrer Verwendung (deutsche Übersetzung von ISO/IEC 9126

[31] DIN/ISO 12119: Informationstechnik; Software-Erzeugnisse; Qualitätsanforderungen und Prüfbestimmungen (deutsche Übersetzung von ISO/IEC 12 119 (1994), ersetzt DIN 66285), 1995

[32] *DIN EN IS0 9001:2000 – Die Änderungen.* Beuth-Verlag, 2000

[33] [ISO 10007:1995] NN: *Quality Management: Guidelines for Configuration Management*

[34] Koch, F.A,: *Produkthaftung für Software - Haftungsentlastung durch Qualitätssicherung.* in: Handbuch der modernen Datenverarbeitung Heft 163, Forkel Verlag 1992

[35] A. Leibl: *VCS Reloaded – Versionsverwaltung mit Subversion 1.0.* iX 8/2004

[36] Marick, B.: *Craft of Software Testing (Subsystem Testing, Including Object-Based and Object-Oriented Testing.* Prentice Hall 1997

[37] Mellor, S.J.; Balcer, M.J.: *Executable UML: A Foundation for Model-Driven Architecture.* Addison Wesley, 2002

[38] Mertins, K.; Jochem, R.: *Qualitätsorientierte Gestaltung von Geschäftsprozessen.* Beuth Verlag 1997

[39] Myers, G.J.: *Methodisches Testen von Programmen.* Oldenbourg 82

[40] Neubauer, B.; Ritte, T.; Stoinski, F.: *CORBA Komponenten.* Springer, 2004

[41] Oesterreich, B.: *Objektorientierte Softwareentwicklung: Analyse und Design mit der Unified Modeling Language.* Oldenbourg 1998

[42] Pagel, B.-U.; Six, H.-W.: *Software Engineering Band 1: Die Phasen der Software-Entwicklung.* Addison-Wesley, 1994

[43] Probst, G.; Raub, S.; Romhardt, K.: *Wissen Managen – Wie Unternehmen ihre wertvollste Ressource optimal nutzen.* Gabler Verlag 1998

[44] Rausch, A.; Broy, M.: *Das V-Modell XT - Grundlagen, Erfahrungen und Werkzeuge.* dpunkt.verlag, 2005

[45] Reisig, W.: *Systementwurf mit Netzen.* Springer Verlag 1985

[46] Reisig, W.: *Petrinetze.* Springer Verlag 1986

[47] Riedemann, E.H.: *Testmethoden für sequentielle und nebenläufige Software-Systeme.* B.G. Teubner Stuttgart 1997

[48] Rochkind, M.J.: *The source code control system.* IEEE ToSE, 1 (4) 1975

[49] Rupp, Ch.; Hahn, J.; Queins, S.; Jeckle, M.; Zengler, B.: *UML 2 glasklar.* Hanser 2005

[50] Rosenbaum, M.; Wagner, F.: *Versicherungsbetriebslehre.* Gabler 2000

[51] Österle, H.; Winter, R.: *Business Engineering.* Springer 2003

[52] Österreich, B.: *Objektorientierte Softwareentwicklung.* Oldenbourg 2004

[53] Scheer, A.-W.: *Wirtschaftsinformatik. Informationssysteme im Industriebetrieb.* Springer-Verlag, Berlin, 1990

[54] Schlageter, G.; Stucky, W.: *Datenbanksysteme: Konzepte und Modelle.* Teubner Verlag 1983

[55] Schmid, H.: *Entwicklung und Realisierung einer Teststrategie mit zugehöriger Testdatenbank für ein Kfz-Elektronik-Testsystem.* Diplomarbeit im Studiengang Informatik an der Abt. Angewandte Informationsverarbeitung der Universität Ulm, 1998

[56] H. Schwichtenberg, M. Weidner: *Geschichtsschreiber – Versionsverwaltungssysteme im Vergleich (Perforce, Visual Sourcesafe, Subversion).* iX 9/2004

[57] Shooman, M.L.: *Software Engineering.* McGraw-Hill 1993

[58] Siegel, S.; Muller, R.J.: *Object Oriented Software Testing – A Hierarchical Approach.* John Wiley & Sons 1996

[59] Sommerville, I.: *Software Engineering.* Addison Wesley 1992

[60] Speck, M.: *Geschäftsprozessorientierte Datenmodellierung.* Logos Verlag 2001

[61] Stahl, Th.; Völter, M.: *Modellgetriebene Software-Entwicklung.* dpunkt.verlag 2005

[62] Dr. Starke – Managementsysteme: *ISO9000:2000 als leichte Kost.* Beuth-Verlag, 2000

[63] Tichy, W.F.: *RCS – A System for Version Control.* Software-Practice & Experience, 15 (7) 1985

[64] *Anwendungsarchitektur der deutschen Versicherungswirtschaft.* VAA Edition 1999 (www.dgv-online.de/vaa/)

[65] Vossen, G.; Witt, K.-U.: *Das SQL/DS-Handbuch.* Addison-Wesley, 1988

[66] Wallmüller, E.: *Software-Qualitätsmanagement in der Praxis.* Hanser 2001

[67] Warmer, J.; Kleppe, A.: *The Object Constraint Language: Getting Your Models Ready For MDA*. Addison Wesley, 2003, 2. Auflage

[68] Wirth, N.: *Gedanken zur Software-Explosion.* Informatik-Spektrum, 17,1,Springer Verlag 1994

[69] Zehnder, C.A.: *Informationssysteme und Datenbanken.* Teubner 1989

Anhang

Abbildungsverzeichnis

Index