Fachhochschule Köln
Cologne University of Applied Sciences

Angewandte Mathematik und Programmierung

Einführung in das Konzept der objektorientierten Anwendungen zu mathematischen Rechnens

WS 2012/13

Fomuso Ekellem

Inhalt

- **Rekursive Programmierung**
- **Beispiel_ Determinanten von mehr dimensionale Matrizen**
- **Fehlerbehandlung**
- **Fehler und Nichtfehler**
- **Fehler in C++**
- **Ausnahmen**
- **Try und Catch Blöcke**
- **UML**

Rekursive Programmierung

Grundprinzip rekursiver Prozeduren:

- **Die entstehenden rekursiven Aufrufe müssen immer einfacher werden.**
- **Und schließlich auf einen nicht rekursiven Fall führen, der die Aufrufkette abbricht.**

Beispiel:

- **Die Fakultät einer positiven ganzen Zahl n > 0 wird definiert als Produkt**

n! = fact(n)= n(n-1).....2.1

Iterative Berechnung der Fakultät:

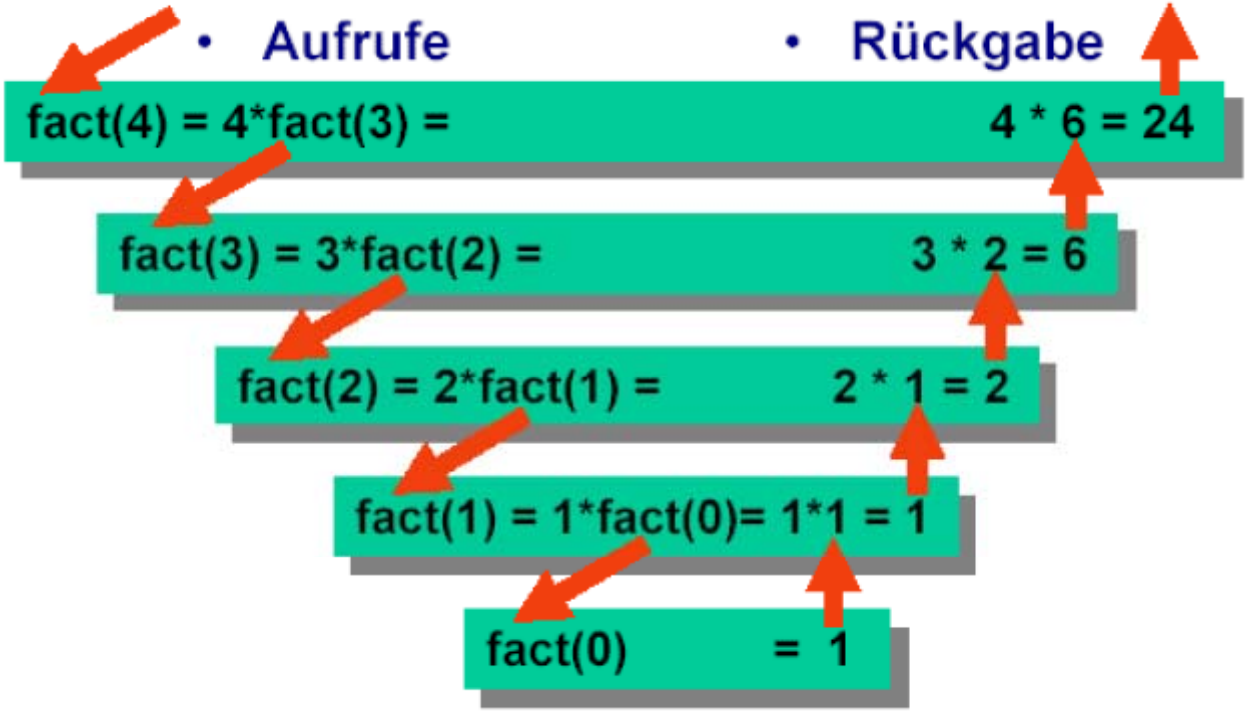
```
int  fact(int n)
{
  int prod = 1;
  while (n > 0)
    prod *= n--;

  return prod;
}
```

Rekursive Berechnung der Fakultät:

```
int  fact(int n)
{
  if (n > 0)
    return n * fact(n-1);
  else
    return 1;
}
```

Rekursive Programmierung

- **Aufrufe und Rückgaben**



Rekursive Programmierung

- **Determinanten von Matrizen Rekursive angehen**

$$|A| = \sum (-1)^{i+j} \; a_{ij} \; M_{ij}$$

$$|A| = a_{11} M_{11} - a_{12} M_{12} + a_{13} M_{13} - \ldots + a_{1n} M_{1n}$$

$$\det(A) = |A| = \begin{vmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \cdots & a_{2n} \\ a_{31} & a_{32} & a_{33} & \cdots & a_{3n} \\ \vdots & \vdots & \vdots & a_{ij} & \vdots \\ a_{n1} & a_{n2} & a_{n3} & \cdots & a_{nn} \end{vmatrix}$$

$$M_{12} = \begin{vmatrix} a_{21} & a_{23} & a_{24} & \cdots & a_{2n} \\ a_{31} & a_{33} & a_{34} & \cdots & a_{3n} \\ a_{41} & a_{43} & a_{44} & \cdots & a_{4n} \\ \vdots & \vdots & \vdots & a_{ij,\, i!=1,\, j!=2} & \vdots \\ a_{n1} & a_{n3} & a_{n4} & \cdots & a_{nn} \end{vmatrix}$$

Zwischen Schritt: Hilfe für Determinanten

- **Malloc und free Vs new und Delete**

```
class  Test
{
public:
    Test()
    {
        cout  <<  "Test  :  ctor\r\n";
    }
    ~Test()
    {
        cout  <<  "Test  :  dtor\r\n";
    }
    void Hello()
    {
        cout  <<  "Test  :  Hello  World\r\n";
    }
};
```

```
int  _tmain(int  argc,  _TCHAR*  argv[])
{
  cout  <<  "1\r\n";
  Test*  t1  =  new  Test();
  t1->Hello();
  delete  t1;

  cout  <<  "2\r\n";
  Test*  t2  =  (Test*)  malloc(sizeof  Test);
  t2->Hello();
  free(t2);

  return  0;
}
```

- **Ausgabe**

```
1
Test : ctor
Test : Hello World
Test : dtor
2
Test : Hello World
```

Zwischen Schritt: Hilfe für Determinanten

- **Andere Möglichkeiten**

- `Test* t1 = new Test();` **oder** `Test* t2 = (Test*) malloc(sizeof Test);`

```
//declaring native type

int* i1 = new int;
delete i1;

int* i2 = (int*) malloc(sizeof(int));
free(i2);

//declaring native type array

char** c1 = new char*[10];
delete[] c1;

char** c2 = (char**) malloc(sizeof(char)*10);
free(c2);
```

Rekursive Programmierung

- **Determinanten**

$$|A| = a_{11}\begin{vmatrix} a_{22} & a_{23} & a_{24} & \cdots & a_{2n} \\ a_{32} & a_{33} & a_{34} & \cdots & a_{3n} \\ a_{42} & a_{43} & a_{44} & \cdots & a_{4n} \\ \vdots & \vdots & \vdots & \cdots & \vdots \\ a_{n2} & a_{n3} & a_{n4} & \cdots & a_{nn} \end{vmatrix} - a_{12}\begin{vmatrix} a_{21} & a_{23} & a_{24} & \cdots & a_{2n} \\ a_{31} & a_{33} & a_{34} & \cdots & a_{3n} \\ a_{41} & a_{43} & a_{44} & \cdots & a_{4n} \\ \vdots & \vdots & \vdots & \cdots & \vdots \\ a_{n1} & a_{n3} & a_{n4} & \cdots & a_{nn} \end{vmatrix} + a_{13}\begin{vmatrix} a_{21} & a_{22} & a_{24} & \cdots & a_{2n} \\ a_{31} & a_{32} & a_{34} & \cdots & a_{3n} \\ a_{41} & a_{42} & a_{44} & \cdots & a_{4n} \\ \vdots & \vdots & \vdots & \cdots & \vdots \\ a_{n1} & a_{n2} & a_{n4} & \cdots & a_{nn} \end{vmatrix}$$

$$\cdots\cdots + a_{1n}\begin{vmatrix} a_{21} & a_{22} & a_{23} & \cdots & a_{2(n-1)} \\ a_{31} & a_{32} & a_{33} & \cdots & a_{3(n-1)} \\ a_{41} & a_{42} & a_{43} & \cdots & a_{4(n-1)} \\ \vdots & \vdots & \vdots & \cdots & \vdots \\ a_{n1} & a_{n2} & a_{n3} & \cdots & a_{n(n-1)} \end{vmatrix}$$

Determinanten:

```
double Determinant(double **a,int n)
{
   int i,j,j1,j2;
   double det = 0;
   double **m = NULL;

   if (n < 1) { /* Error */

   } else if (n == 1) { /* Shouldn't get used */
      det = a[0][0];
   } else if (n == 2) {
      det = a[0][0] * a[1][1] - a[1][0] * a[0][1];
   } else {
      det = 0;
      for (j1=0;j1<n;j1++) {
         m = malloc((n-1)*sizeof(double *));
         for (i=0;i<n-1;i++)
            m[i] = malloc((n-1)*sizeof(double));
         for (i=1;i<n;i++) {
            j2 = 0;
            for (j=0;j<n;j++) {
               if (j == j1)
                  continue;
               m[i-1][j2] = a[i][j];
               j2++;
            }
         }
         det += pow(-1.0,1.0+j1+1.0) * a[0][j1] * Determinant(m,n-1);
         for (i=0;i<n-1;i++)
            free(m[i]);
         free(m);
      }
   }
   return(det);
}
```

Fehlerbehandlung

- Wie wird in normalen Programmen, z.B. C-Programmen, mit Fehlern umgegangen, die bei der Ausführung einer Funktion erkannt werden?
- Es gibt klassisch unter anderen, die folgenden Möglichkeiten:
  1. Zurückgabe eines Funktionswert , der einen Fehler Signalisiert(z.B. -1)
  2. Setzen eines Statusbits, in der Hoffnung , dass der Anwender diesen Wert auch abfragt;
  3. Ausgabe einer Fehlermeldung und Programmabbruch mit exit().
- Alle 3 haben schwäche:
  - (1) ist anwendbar in Methoden/Funktionen, die gar keinen Rückgabewert zurückgeben. z.B in Konstruktoren oder Destruktoren
  - In (2) hofft man nur
  - (3) ist oft zu drastisch!
- Mit Exceptions lassen sich solche Situationen sehr flexibel bahandeln.

Fehler und Nichtfehler

- Der Begriff Fehler wird hier eingeschränkt auf Fehler, die den normalen Programmablauf "stören".
- Eine fehlerhafte Eingabe eines Benutzers ist in der Regel kein Fehler für ein Programm! Es sollte z.B. den Benutzer über die fehlerhafte Eingabe informieren und zu einer wiederholten Eingabe auffordern.
- Beim Traversieren einer Liste ist das Erreichen des Listenendes auch kein Fehler, sondern ein ganz normaler Zustand. Versucht ein Programm jedoch, ein Element hinter dem Listenende anzuschauen, ist es in der Regel ein Fehler.
- Ein typischer Fehler ist auch, auf ein Feldelement zuzugreifen, das nicht existiert.

- Ist ein Fehler eine wirkliche Ausnahme, das heißt, tritt er selten und kaum vorhersehbar auf, sind die obigen Fehlerbehandlungsarten nicht adäquat.

Fehler in C/C++

Häufige Fehler:

- **Syntaxfehler:** Fehler im formalen Aufbau und „Rechtschreibfehler“ führen zu Syntaxfehlern.

  Beispiele:

1)
```
a = b + c          // Semikolon vergessen
cout << a << endl;
```
2)
```
while(a != b);  // irrtümlich Semikolon hinter while-Anweisung
{ …….}     ;        // (oder for   oder if)
```
3)
```
do{……}
while(a <= b)    // Semikolon hinter do…while vergessen
```
4)
```
while(a = b)     // Zweisungsoperator “=” statt Gleichheitsoperator
                 // “==” verwendet
```
5)
```
int auto, moped;
Auto = moped / 7;        // Variable auto groß geschrieben, in der
                         // Deklaration aber klein
```
6)
```
int summe, i, a[100];
for(i = 0; i <= 100; i++)        // Indexbereich ueberschritten, letztes
summe = summe + a[i];     // Element ist a[99] !
```

Fehler in C/C++

```
7)   double x, y, *pd;
      x= 4.1;
      y = 7.18;
      *pd = y;  // formal korrekt aber Pointervariable pd ist noch
                // uninitialisiert (zeigt irgendwo hin)
       pd = x;    // Wert zu Pointer ist verboten, richtig: pd = &x
8)   class TIME{
  public:
    void read_time();
    void write_time();
    void add_time(TIME t1, TIME t2);
  private:
    int hh, mm, ss;
}          // hier wurde das abschließende Semikolon vergessen, richtig: };
```

Fehler in C/C++

- **Laufzeitfehler:** Ein syntaktisch korrektes Programm kann auch nach seinem Start während der Programmausführung mit einer Fehlermeldung abbrechen. Diese erst zur Laufzeit auftretenden Fehler heißen „Laufzeitfehler“.

  Beispiele:

  1) Division durch Null:

  ```
  ….
  cin >> n;
  q = z/n;              //  Abbruch , falls für n Null eingegeben wird.
  …..
  ```

  2) Wurzel aus negativer Zahl:

  ```
  …..
  …..
  c = sqrt(x-y);        //  Abbruch , falls aktuell x-y < 0.
  ```

- **…..**

Fehler in C/C++

- **Semantikfehler:** (Logische Fehler) Semantikfehler verstoßen wird weder gegen Rechtschreib- noch gegen Grammatikregeln einer Sprache .

  Beispiel:

  „ Das Rad ist viereckig“

- Wenn ein Programm ohne Fehlermeldungen abgearbeitet wird aber falsche Ergebnisse liefert, liegt ein logischer Fehler vor. Logische Fehler werden nur erkannt, wenn zu bestimmten Test-Eingaben die erwarteten Programm-Ergebnisse bekannt sind(z. B. durch Handrechnung oder Taschenrechner). Diese Fehler entstehen durch einen falschen Algorithmus und zwingen manchmal zu einer grundlegenden Umorganisation des Programms.

  Beispiel:

  - Statt Berechnung der Summe zweier Größen wird das Produkt gebildet. Ein solcher logischer Fehler kann natürlich auch auf einem Tippfehler beruhen.
  - Fehler in der Logik größerer Programme lässt sich durch ein klares Konzept des Programmaufbaus(Struktogramm) vorbeugen.

Ausnahmen

- Als **Ausnahmen** (exceptions) werden Ereignisse bezeichnet, die bei der Ausführung eines Programms auftreten und die nicht im normalen Kontrollfluß des Programms behandelt werden.
  - Es ist eine typische Ausnahme, wenn bei einer Gleitkommaoperation ein Überlauf stattfindet.
  - Eine andere Ausnahme tritt z.B. auf, wenn die Kommunikationsverbindung zwischen zwei Prozessen während eines Datenaustausches abbricht.
- **Ausnahmebehandlung** (exception handling) ist die Bearbeitung solcher Ereignisse **innerhalb** des Programms und die Unterstützung durch die Programmiersprache und das Laufzeitsystem.
- Ausnahmen sind somit eine spezielle Art von Fehlern und Ausnahmebehandlung eine spezielle Fehlerbehandlung. Die Abgrenzung zwischen "normalen" Fehlern und Ausnahmen ist unscharf und insbesondere anwendungsabhängig.

Ausnahmen

- Für die Ausnahmebehandlung bietet C++ einen Mechanismus an, von der Stelle in einer Funktion, in der eine Ausnahme auftritt, die Kontrolle und Informationen an einen unbestimmten Aufrufer zu übergeben, der seine Bereitschaft erklärt hat, Ausnahmen eines bestimmten Typs zu behandeln.
- Die Ausnahmebehandlung wird im folgenden an einigen kleinen Beispielen eingeführt.
- Ein Ausnahmetyp wird durch eine ganz normale Klassendefinition festgelegt, der außer am Namen nicht angesehen werden kann, dass es ein Ausnahmetyp ist.

Ausnahmen melden

- Wenn eine Ausnahme erkannt wird, muss sie gemeldet werden: eine Ausnahme wird ausgeworfen.

Try und Catch Blöcke

- **Einfache try und Catch Beispiel**
  - Im Block direkt nach dem Schlüsselwort „try“ steht der Programmcode. Tritt jetzt innerhalb dieses Blocks eine Ausnahme auf, so werden die darin folgenden Catch-Blöcke durchsucht, ob der momentan aufgetretene Ausnahmefall darin vorkommt.
  - Wenn ja, so wird der entsprechende Catch-Block angesprungen. Die darin enthaltenen Befehle werden abgearbeitet. Anschließend wird mit dem nächsten Befehl nach dem letzten Catch-Block fortgesetzt.

```
try
{ // Programmcode, der untersucht wird
}
catch ( /* Fall 1 */ )
{ // Fehlerbehandlung zu Fall 1
}
catch ( /* Fall 2 */ )
{ // Fehlerbehandlung zu Fall 2
}
```

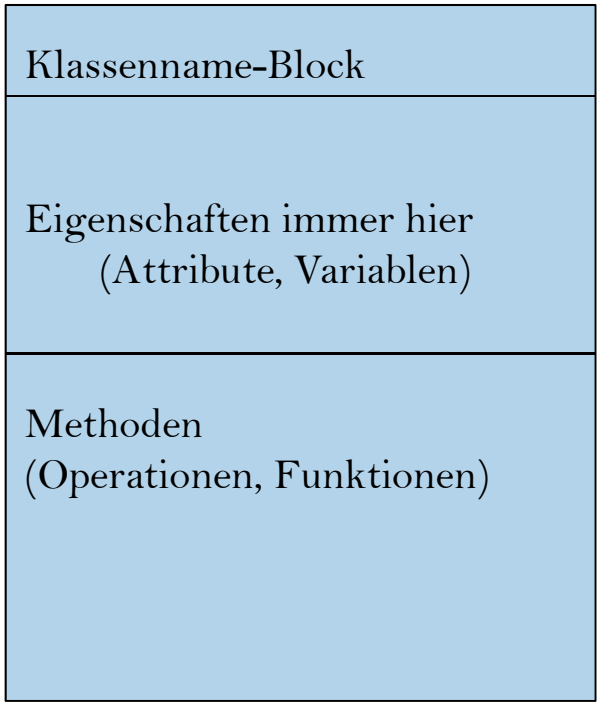
```
#include <iostream>
using namespace std;

int main()
{
 cout << "Start\n";

 try {
  cout << "Inside try block\n";
  throw 1;                    // throw an error
  cout << "This will not execute";
 }
 catch (int i) {              // catch an error
  cout << "Caught an exception -- value is: ";
  cout << i << "\n";
 }
 cout << "End";
 return 0;
}
```

Try und Catch Blöcke

- Wird der aufgetretene Fehlerfall in keinem Catch- Block abgefangen, so wird der Fehler an den übergeordneten Block weitergeleitet. Existiert kein übergeordneter Try-Catch-Block mehr oder wird dieser Fehler auch dort nicht behandelt, so löst das Laufzeitsystem einen Laufzeitfehler aus.
- Try-Catch-Blöcke dürfen beliebig geschachtelt werden. Im Ausnahmefall werden zunächst die Catch-Blöcke durchsucht, die zum aktuell durchlaufenen Try-Block gehören. Behandelt kein Catch-Block diese Ausnahme, so werden die Catch-Blöcke des übergeordneten Try-Catch-Blockes durchsucht und so fort.
- Ausnahmen können direkt vom System ausgelöst werden (Laufzeitfehler), aber auch der Programmierer selbst kann gezielt Ausnahmen erzögen; man spricht hier von einer Ausnahme werfen. Hierzu dient das Schlüsselwort *throw.*

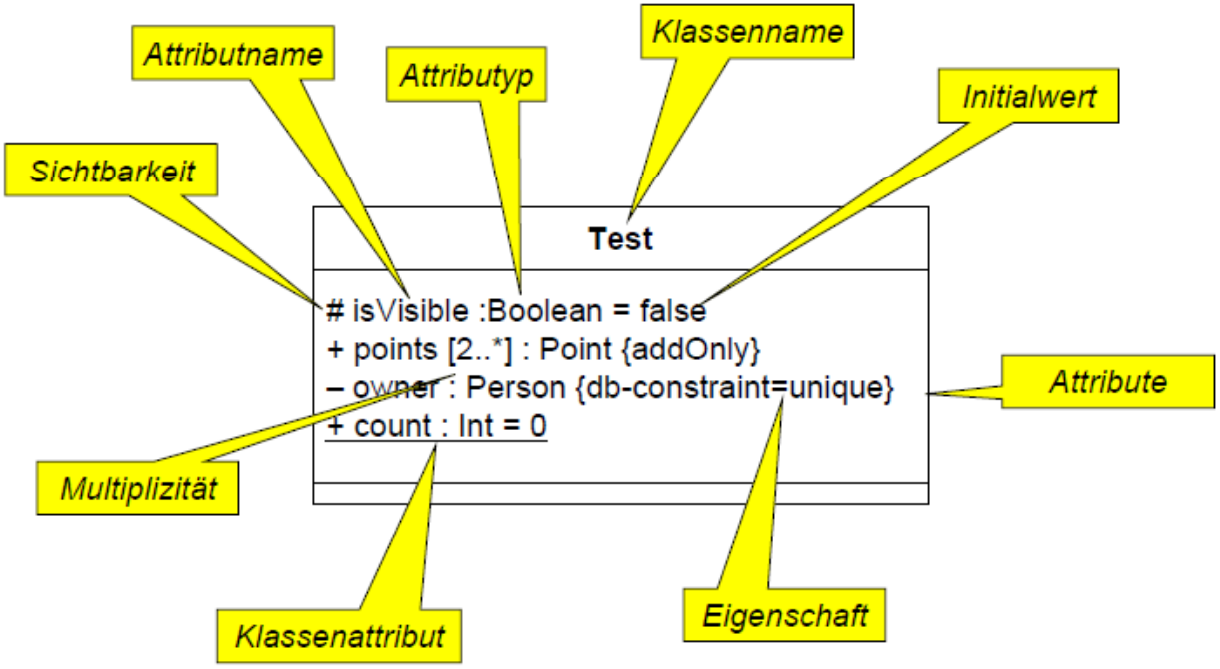
UML (Klassendiagramme)

- **Klassendiagram: Immer in drei Abteilungen für Klassenname, Variablen(Daten) und Methoden**



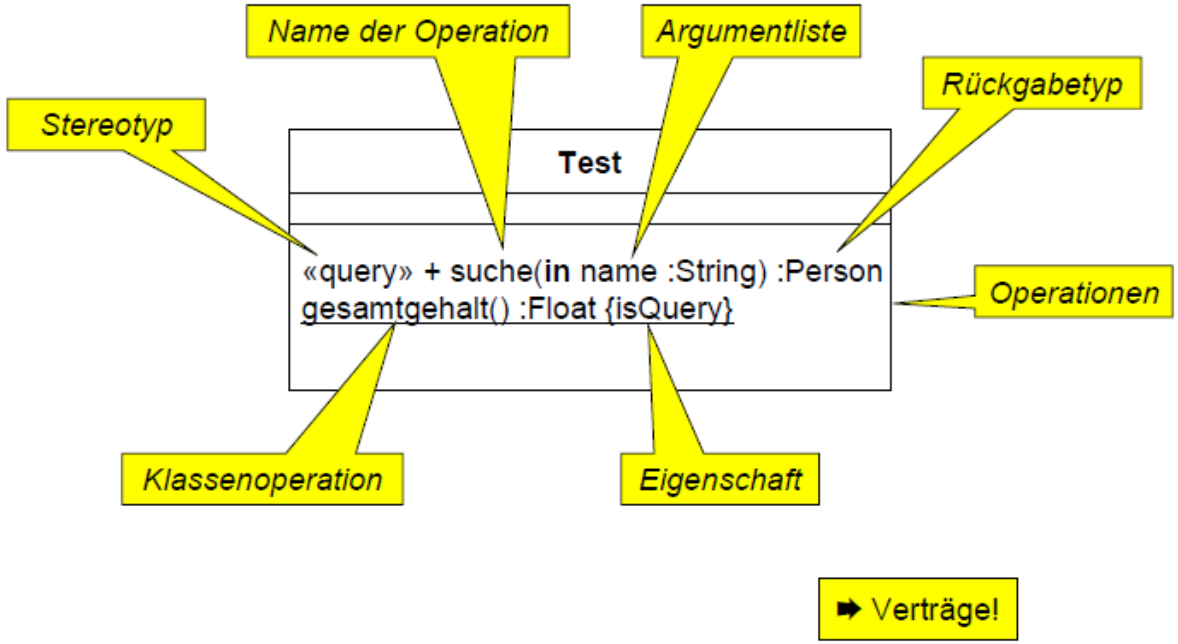
UML (Klassendiagramme)

- **Klassendiagram mit Attribute(Daten)**



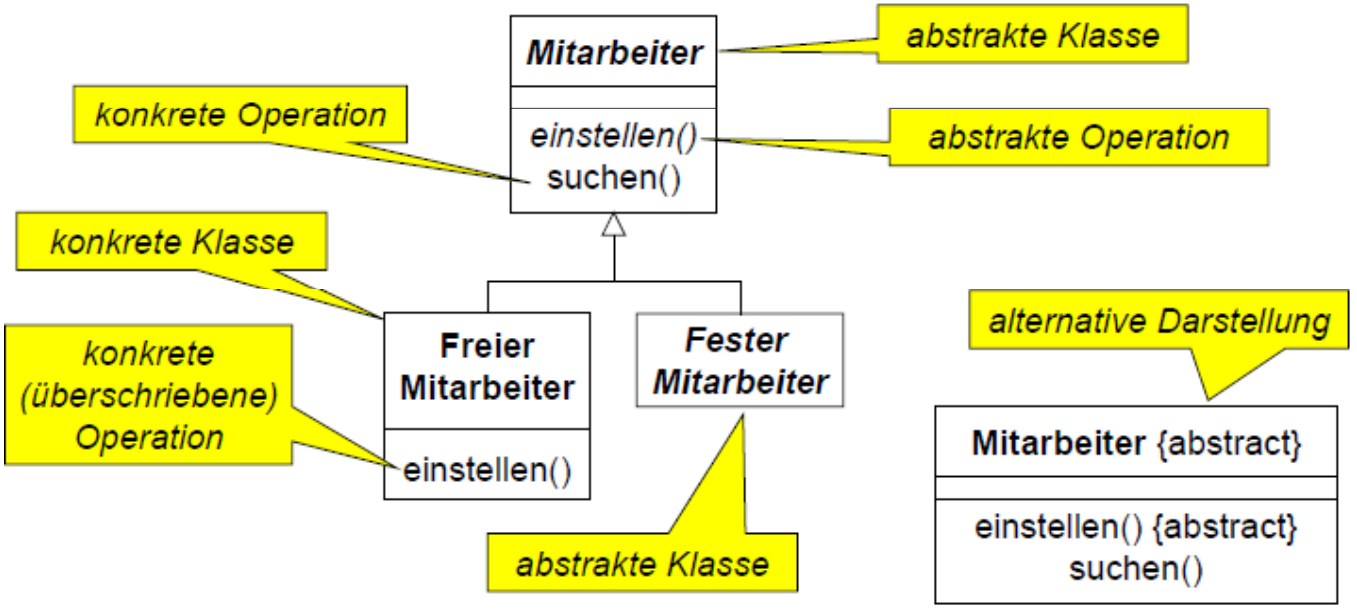
UML (Klassendiagramme)

- **Klassendiagram mit methoden(operationen)**



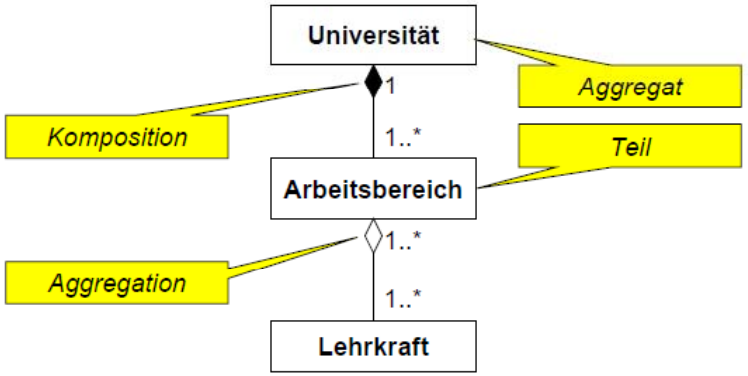
UML (Klassendiagramme)

- **Vererbung**



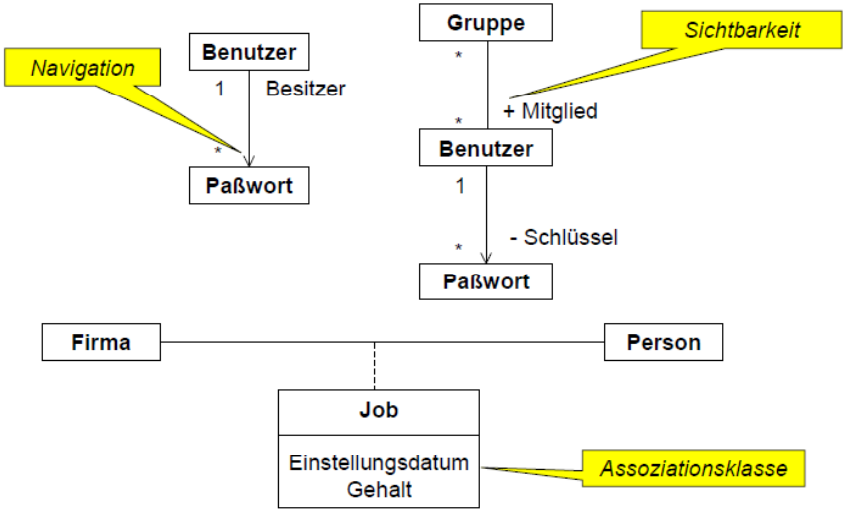
UML (Klassendiagramme)

- Komposition (echte Aggregation): Komponenten sind vom Aggregat existenzabhängig z.B. Arbeitsbereich ist Teil eine Universität.
- Aggregation (einfache Aggregation):Aggregat und Komponenten sind nicht existenzabhängig
- Häufig lässt sich nur schwierig festlegen, ob eine Aggregation oder Assoziation vorliegt.



UML (Klassendiagramme)

- Assoziationen : drückt das Verhältnis von zwei völlig selbständigen Objekten aus, die auf der gleichen Abstraktionsebene stehen und eigentlich nichts miteinander zu tun haben, aber unter bestimmten Gesichtspunkten in eine lose Kennt-Beziehung ('KNOWS') gebracht werden können.
- Bei einer m : n-Beziehung wird dazu eine neue Klasse gebildet, deren Exemplare (Linkobjekt) die aktuelle Beziehung realisieren.



UML (Klassendiagramme)

Kardinalität

- **Dabei werden drei Grundtypen unterschieden:**
  - 1 : 1
  - 1 : n
  - m : n
- **Andere Notationen (UML) sind:**

  1 : genau eine
  0,1 : konditionell, keine oder eine
  * : multiple, keine Einschränkung
  0..* : multiple
  1..* : mindestens eine