Einführung in die Programmierung (MA8003)

Teil 4.2: Einführung in die objektorientierte Programmierung

Dr. Laura Scarabosio

Technische Universität München
Fakultät Mathematik, Lehrstuhl für Numerische Mathematik M2

11.10.2019

Teil 4.2: Inhalt

Inhalt

Grundbegriffe objektorientierter Programmierung

Bei komplexen Problemen kann es sinnvoll sein objektorientierte Programmierung (OOP) einzusetzen um die Komplexität einfacher beherrschbar zu machen.

Grundideen:

- System wird durch das Zusammenspiel von **Objekten** beschrieben
- Objekten sind **Attribute** (Eigenschaften) und **Methoden** zugeordnet
- Jedes Objekt ist in der Lage mit anderen Objekten zu kommunizieren
- Das Konzept einer **Klasse** fasst Objekte aufgrund ähnlicher Eigenschaften zusammen.
- Ein Objekt wird im Programmcode als **Instanz** einer Klasse definiert.

Die wohl bekanntesten objektorientierten Sprachen sind C++ und Java. Allerdings stellt auch MATLAB einige Konzepte der OOP bereit.

Klassen, Attribute und Objekte

Um den Unterschied zwischen Klasse und Objekt zu verdeutlichen nehmen wir als Beispiel ein einfaches Autoquartett.

Klasse: Auto

Attribute:

- Preis (EUR)
- Hubraum (ccm)
- Leistung (PS)
- Von 0 auf 100 (s)
- Geschwindigkeit (km/h)
- Verbrauch (l)

Objekt der Klasse Auto: Porsche GT 3

Preis (EUR):	121.100
Hubraum (ccm):	3600
Leistung (PS):	381
Von 0 auf 100 (s):	4,4
Geschwindigkeit (km/h):	306
Verbrauch (l):	9,8

Das Autoquartett ist nun, abstrakt vom Standpunkt der Datenverarbeitung aus gesprochen, eine Menge von Objekten der Klasse Auto.

Klassen und Methoden

Das Konzept der Klasse beinhaltet mehr als eine Struktur zur Verwaltung zusammengehöriger Daten. Vielmehr dient es der **Abstraktion**:

Wir haben eine gewisse Abstraktion bereits kennen gelernt:

Beispiel

Wenn wir das Unterprogramm `sin(x)` aufrufen, dann erwarten wir, dass es uns den Sinus berechnet. Wie, das ist uns in der Regel egal.

Im Prinzip schafft die OOP lediglich eine weitere Abstraktionsebene:

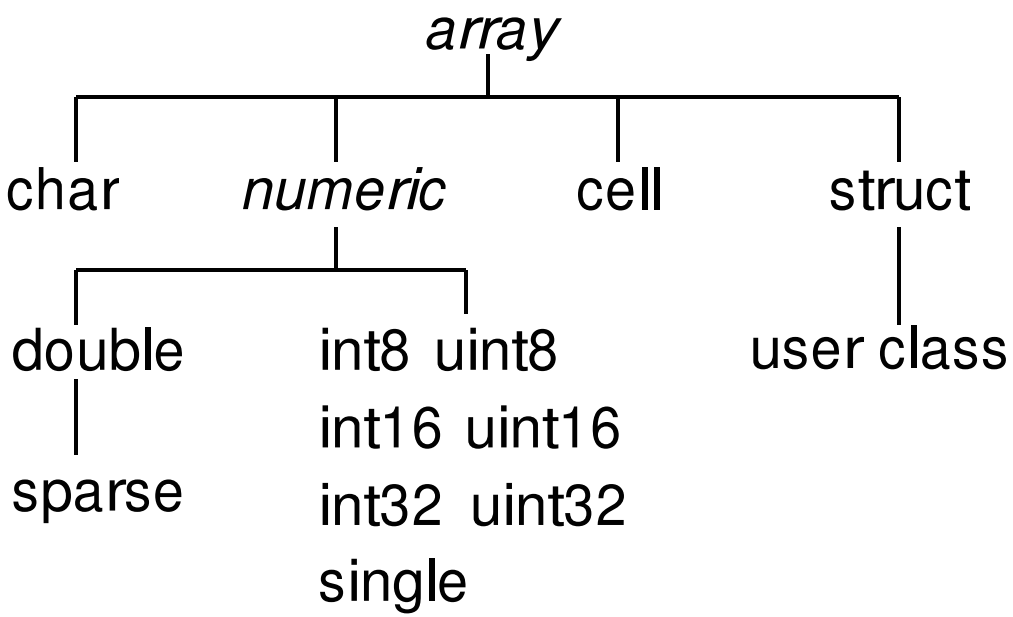
- den Benutzer interessiert es nicht, wie die Daten abgelegt werden (z.B. ob die Leistung intern in PS oder kW gespeichert wird)
- Objekte kommunizieren über definierte **Schnittstellen** in Form bereitgestellter **Methoden**.
- Programme können mit Objekten arbeiten, ohne über die exakte interne Realisierung Bescheid zu wissen!

Inhalt

OOP in MATLAB: Funktionsumfang

- Überladen von Funktionen und Operatoren
- Kapselung von Daten und Methoden:
  - Objekteigenschaften nicht sichtbar von Kommandozeile
  - Objektvariablen nur änderbar über Klassenfunktionen
- Vererbung:
  - Vererbung von Variablen und Funktionen
  - einfache oder mehrfache Vererbung möglich
  - eine oder mehrere Generationen
- Aggregation (Zusammenfassen von Objekten)

Matlab-Datentypen und Klassenhierarchie



Unterschiede zu OOP in C++/Java

- Die neuen Klassen werden während der Laufzeit erzeugt (Matlab ist ein *Interpreter*, kein Compiler)
- Es gibt keine Destruktoren in Matlab, stattdessen muss man die `clear`-Funktion verwenden
- Vererbung wird durch Aufruf der `class`-Funktion mit Parametern realisiert
- Weitere Einschränkungen gegenüber C++/Java:
  - Es gibt keine abstrakten Klassen, virtuelle Vererbung, o.ä.
  - Es gibt keine Interfaces (Java)
  - Es gibt keinen "scoping"-Operator (`::` in C++)
  - Es gibt keine Templates (C++)

Entwerfen eigener Klassen

- Klassenverzeichnis erzeugen
  - Name: `@classname`
  - enthält alle zu dieser Klasse gehörenden m-Files (Methoden)
- Hilfsfunktionen und private Methoden
  - m-Files, die nicht direkt aufgerufen werden sollen kommen in das `private`-Verzeichnis: `@classname/private`
  - Wenn das Klassenobjekt übergeben wird, spricht man von "private methods", sonst von "helper functions"
- Objekte erzeugen
  - durch Aufruf des Konstruktors (gleicher Name wie die Klasse)
  - Übergeben von Initialisierungsparametern möglich
  - Beispiel: `p = polynom([1 0 -2 -5])`

Entwerfen eigener Klassen

- Aufruf einer zu einem Objekt gehörenden Methode
  - Syntax:

    `[out1,out2,...] = methodname(object,arg1,arg2,...)`
  - Der erste Parameter muss das Objekt enthalten, dann folgen die Argumente.
  - Beispiel: `d = evaluate(p,3);`
- Neues Klassenverzeichnis zum Matlab-Suchpfad hinzufügen
  - z.B.: die Klasse `polynom` befindet sich in

    `/my_classes/@polynom` (unix)
    `c:\my_classes\@polynom` (windows)
  - dann Hinzufügen zum Matlab-Suchpfad mit `addpath`:

```
>> addpath /my_classes        (unix)
>> addpath c:\my_classes      (windows)
```

  - Überprüfen mit

```
>> path
```

Methoden der Basis-Klasse

Ziel: Neue Klasse dem Standard-Verhalten von Matlab anpassen.

Methode	**Beschreibung**
Konstruktor der Klasse	Erzeugt ein Objekt der Klasse
`display`	Wird von Matlab aufgerufen, wenn der Inhalt eines Objekts angezeigt wird (z.B. bei Eingabe eines Ausdrucks ohne abschließenden Strichpunkt)
`set` und `get`	Setzen/Auslesung von Attributen der Klasse
`subsref` und `subsasgn`	Ermöglicht Verwendung von Indizes
`end`	Unterstützung der `end`-Syntax bei Indizes
`subsindex`	Wird aufgerufen, falls Objekte zur Indizierung verwendet werden; z.B. `array(A)` mit `A` Objekt
Konverter, z.B. `double` oder `char`	Konvertieren von Objekten in Matlab-Datentypen
Funktionen & Operatoren	Jeder Operator hat zugehörige Funktion, die überladen werden kann (z.B. `plus(a,b)`).

Beispiel 1: Die Polynom-Klasse

Designentscheidungen für die Klasse `polynom`:

- **Daten:**
  - lediglich einen Zeilenvektor `c` mit den Koeffizienten des Polynoms
- **Methoden:**
  - Der Konstruktor zum Erzeugen neuer Polynome
  - Zwei Konverter: nach `double` und nach `char`
  - Die `display`-Methode zum einfachen Anzeigen
  - Indizes-Funktionalität (`subsref`) (hier nicht behandelt)
  - überladene `+`, `*` Operatoren (weitere könnten leicht hinzugefügt werden)
  - überladene `polyval`- und `diff`- Funktionen (weitere könnten leicht hinzugefügt werden)

Das folgende Code-Beispiel ist entnommen aus der Matlab-Anleitung.

Konstruktor: @polynom/polynom.m

```
function p = polynom(a)
%POLYNOM Polynomial class constructor.
%   p = POLYNOM(v) creates a polynomial object
%   from the vector v, containing the coefficients
%   of descending powers of x.

if nargin == 0
   p.c = [];
   p = class(p,'polynom');
elseif isa(a,'polynom')
   p = a;
else
   p.c = a(:).';
   p = class(p,'polynom');
end
```

Der Konstruktor könnte gleichzeitig auch die Eingaben prüfen um die Integrität der Daten sicher zu stellen. Zum Beispiel könnte man hier verhindern, dass der Vektor a die Werte `NaN` oder `inf` enthält und eine entsprechende Fehlermeldung ausgeben.

double-Konverter: @polynom/double.m

```
function c = double(p)
% POLYNOM/DOUBLE  Convert polynom object to coefficient vector.
%   c = DOUBLE(p) converts a polynomial object to the vector c
%   containing the coefficients of descending powers of x.

c = p.c;
```

char-Konverter: @polynom/char.m

```
function s = char(p)
% POLYNOM/CHAR
% CHAR(p) is the string representation of p.c
if all(p.c == 0)
  s = '0';
else
  d = length(p.c) - 1;
  s = [];
  for a = p.c;
    if a ~= 0;
      if ~isempty(s)
        if a > 0
          s = [s ' + '];
        else
          s = [s ' - ']; a = -a;
        end
      end
      if a ~= 1 | d == 0
        s = [s num2str(a)];
        if d > 0
          s = [s '*'];
        end
      end
      %...
```

char-Konverter: @polynom/char.m

```
      %...
      if d >= 2
        s = [s 'x^' int2str(d)];
      elseif d == 1
        s = [s 'x'];
      end
    end
    d = d - 1;
  end
end
```

Beispiel:

```
>> p = polynom([1 0 -2 -5]);
>> char(p)
ans = x^3 - 2*x - 5
```

dispay-Methode: @polynom/display.m

```
function display(p)
% POLYNOM/DISPLAY Command window display of a polynom

disp(' ');
disp([inputname(1),' = '])
disp(' ');
disp(['   ' char(p)])
disp(' ');
```

Beispiel:

```
>> p = polynom([1 0 -2 -5]);
>> p
p =
    x^3 - 2*x - 5
```

+ Operator: @polynom/plus.m

```
function r = plus(p,q)
% POLYNOM/PLUS  Implement p + q for polynoms.

% nach Polynom konvertieren falls nötig
p = polynom(p);
q = polynom(q);

% Laenge der Vektoren anpassen
k = length(q.c) - length(p.c);
r = polynom([zeros(1,k) p.c] + [zeros(1,-k) q.c]);
```

* Operator: @polynom/mtimes.m

```
function r = mtimes(p,q)
% POLYNOM/MTIMES   Implement p * q for polynoms.

p = polynom(p);
q = polynom(q);
r = polynom(conv(p.c,q.c));
```

Beispiele:

```
>> p = polynom([1 0 -2 -5]);
>> q = p+1
q =
    x^3 - 2*x - 4
>> r = p*q
r =
    x^6 - 4*x^4 - 9*x^3 + 4*x^2 + 18*x + 20
```

polyval-Methode: @polynom/polyval.m

```
function y = polyval(p,x)
% POLYNOM/POLYVAL  POLYVAL(p,x) evaluates p at the points x.
y = 0;
for a = p.c
  y = y.*x + a;
end
```

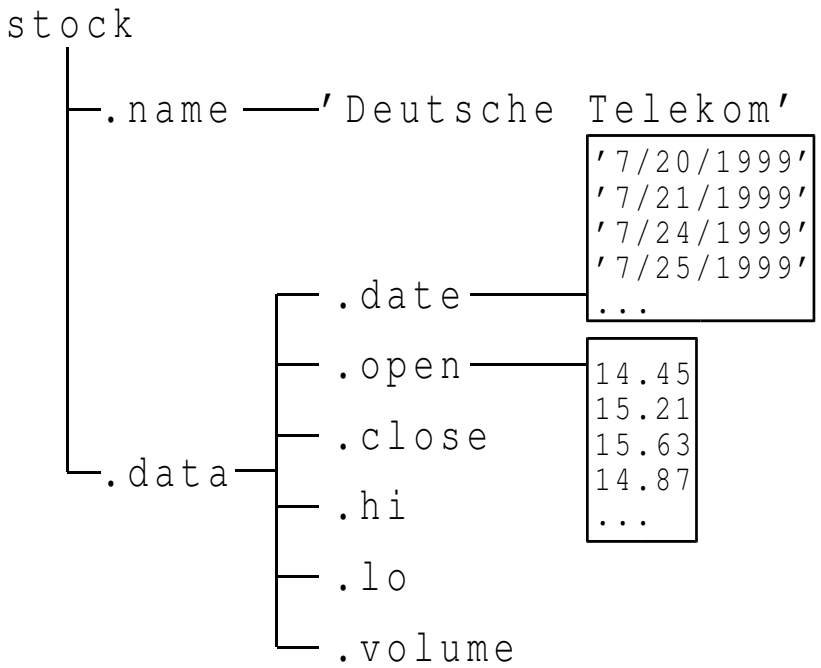
diff-Methode: @polynom/diff.m

```
function q = diff(p)
% POLYNOM/DIFF  DIFF(p) is the derivative of the polynom p.
c = p.c;
d = length(c) - 1;  % degree
q = polynom(p.c(1:d).*(d:-1:1));
```

Beispiel 2: Aktienkursanalyse

Designentscheidungen für die Klasse `stock`:

- **Daten:** Name der Aktie und Struktur mit Daten:



- **Methoden:**
  - Der Konstruktor zum Erzeugen der `stock`-Objekte
  - Methode `loaddata` zum Laden von Datensätzen
  - Überladene `display` und `plot`-Funktionen

Konstruktor: @stock/stock.m

```
function s = stock(name)
% STOCK class construction
%    s = STOCK(name) creates a stock object of with
%    the specified name.
s.data = struct([]);
if nargin == 0
  s.name = 'none';
else
  s.name = name;
end
s = class(s,'stock');
```

loaddata-Methode: @stock/loaddata.m

```
function s = loaddata(a, filename)
% LOADDATA load stock data from specified filename
%    LOADDATA(stock, filename)
%    Example:
%      mystock = stock('SellIt Inc.');
%      loaddata(mystock, 'sellit.txt');
%    Required data file format:
%      <date>    <open> <hi>  <lo> <close> <volume>
%      02/14/2000  65.4 66.95 65.25 66.95   1914379
%      02/15/2000  66.2 67.63 66.1  66.29   2164369
%      ...
  a.data = struct('date','','open',0,'close',0,'hi',0,'lo',0);
  [a.data.date, a.data.open, a.data.hi, ...
    a.data.lo, a.data.close, a.data.volume] = ...
    textread(filename, '%s %n %n %n %n %n', ...
             'commentstyle', 'matlab');
  s = a;
```

Überladene display-Methode: @stock/display.m

```
function display(stock)
% STOCK/DISPLAY Command window display of a stock
if ~isempty(inputname(1))
  disp(' '); disp([inputname(1),' = ']); disp(' ');
end
disp(['  ' stock.name ])
disp(['      last quote from ' cell2mat(stock.data.date(end)) ]);
open = stock.data.open(end);
close = stock.data.close(end);
close_old = stock.data.close(end-1);
disp(['                open: ' num2str(open) ]);
disp(['               close: ' num2str(close) ]);
disp(['                high: ' num2str(stock.data.hi(end)) ]);
disp(['                 low: ' num2str(stock.data.lo(end)) ]);
disp(['  delta prev day (%): ' num2str((close-close_old)/...
                                    close_old*100,2) '%']);
disp(' ');
```

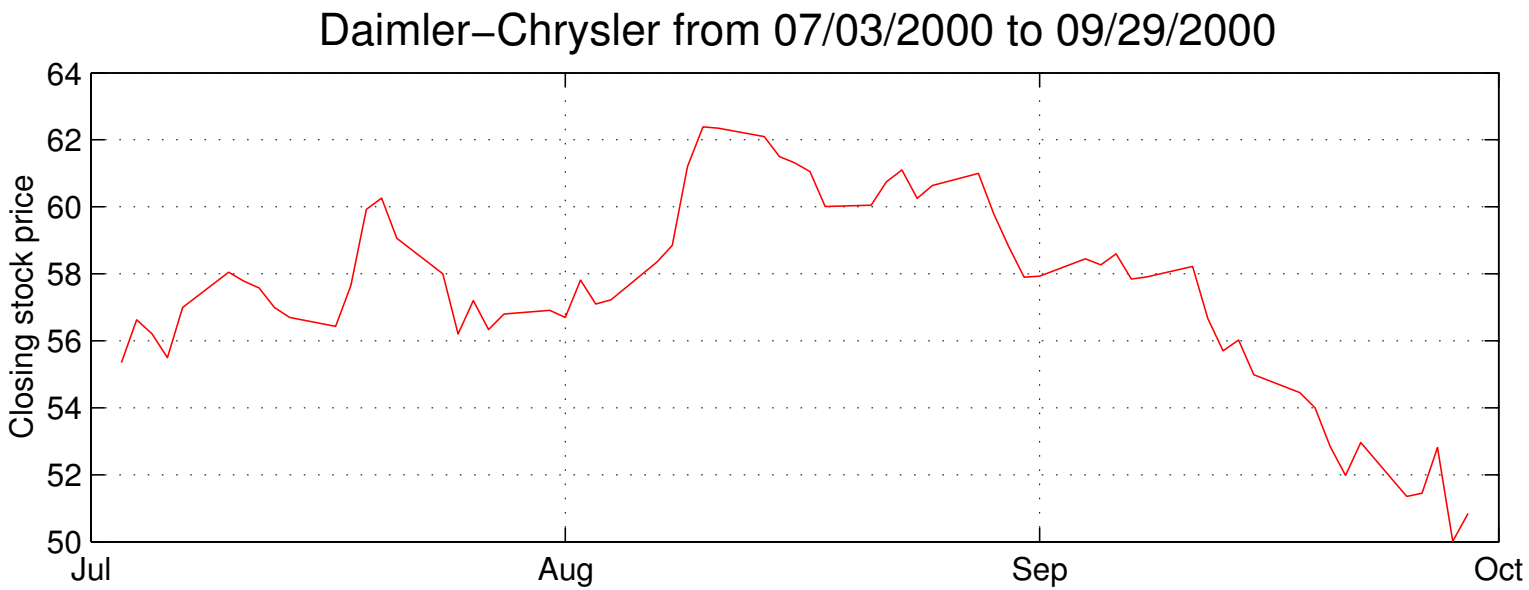
Überladene plot-Methode: @stock/plot.m

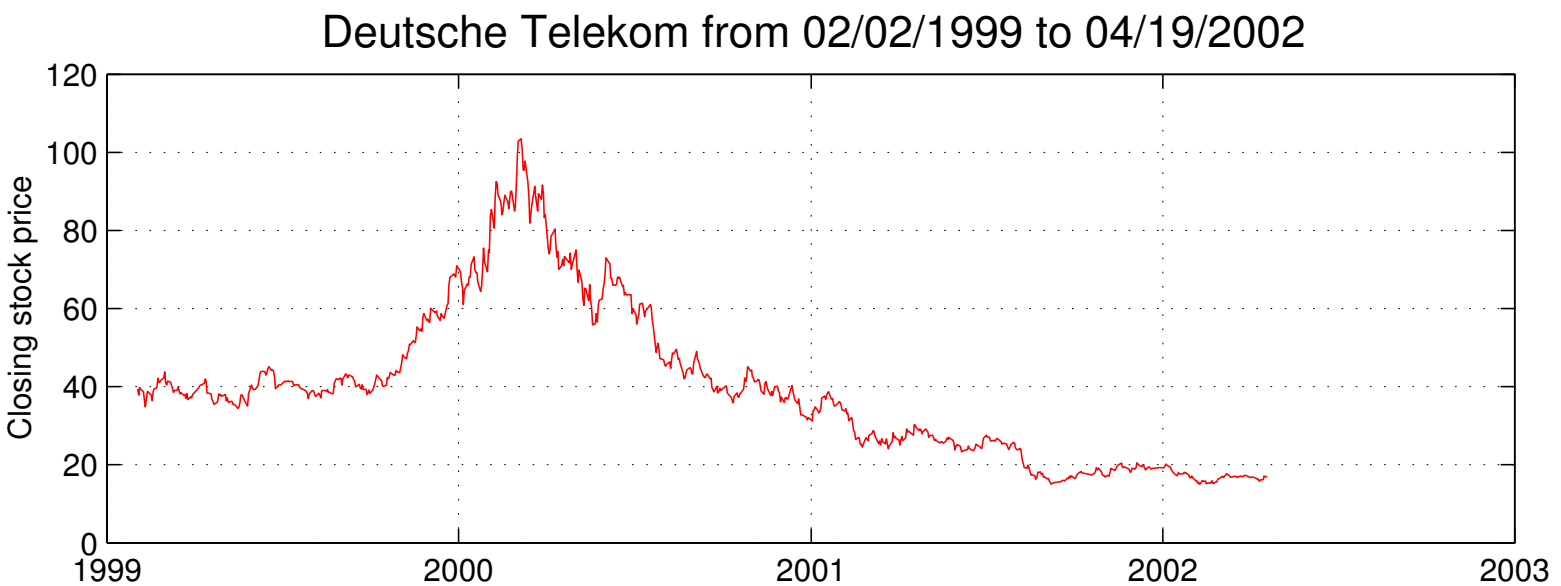
```
function plot(s, startdate, enddate)
% PLOT Draw a line graph of a stock
%    PLOT(stock, startdate, enddate) shows
%    the stock for the specified date range.
% (startdate and enddate are optional)
  if nargin < 2
    start_index = 1;
  else
    start_index = min(find(datenum(s.data.date)>=datenum(startdate)));
  end
  if nargin < 3
    end_index = length(s.data.date);
  else
    end_index = max(find(...
      datenum(s.data.date)<=datenum(enddate)));
  end
  plot(datenum(s.data.date(start_index:end_index)),...
               s.data.close(start_index:end_index),'r-');
  title([s.name ' from ' ...
         cell2mat(s.data.date(start_index)) ' to ' ...
         cell2mat(s.data.date(end_index))],'FontSize',14);
  datetick('x');
  ylabel('Closing stock price');
  grid;
```

Beispiel für Verwendung der stock-Klasse:

```
>> s1 = stock('Daimler-Chrysler');
>> s2 = stock('Deutsche Telekom');
>> s1 = loaddata(s1,'data/daimler-chrysler.txt');
>> s2 = loaddata(s2,'data/telekom.txt');
>> s2
ans =
      Deutsche Telekom
          last quote from 04/19/2002
                    open: 16.86
                   close: 17.05
                    high: 17.13
                     low: 16.76
      delta prev day (%): 0.83%
>> subplot(2,1,1);
>> plot(s1,'07/01/2000','09/30/2000');
>> subplot(2,1,2);
>> plot(s2);
```

Beispiel für Verwendung der `stock`-Klasse:





Vererbung

Die Objektorientierung erlaubt ein hohes Maß an Wiederverwendbarkeit und Wartbarkeit von Code. Einfaches Beispiel:

```
classdef Rectangle < handle
  properties (SetAccess = private)
    length;
    width;
  end
  methods
    function obj = Rectangle(len, wid)
      obj.length = len;
      obj.width = wid;
    end
    function a = diagonal(obj)
      a = sqrt(obj.length^2 + obj.width^2);
    end
    %... many additional functions
  end
end
```

(Achtung: wir verwenden hier eine kompaktere Art der Klassendefinition!)

Vererbung

Für Quadrate ist die Implementierung von `diagonal` effizienter machbar. Wir erben also von `rectangle` und überladen diese Methode:

```
classdef square < rectangle
  methods
    function obj=square(width)
      obj = obj@rectangle(width, width);
    end
    function a = diagonal(obj)
      a = 1.414213562373095*obj.length;
    end
  end
end
```

```
>> r = rectangle(2,2); s = square(2);
>> r.diagonal - s.diagonal

ans =

   4.4409e-16
```

(in diesem Fall sparen wir uns die Wurzel und brauchen dafür lediglich eine Methode neu zu implementieren anstelle der ganzen Klasse.)

Vererbung

Abschließende Kommentare zur Vererbung:

- Wir erben die Implementierung und Schnittstellen der Basisklasse.
- Die abgeleitete Klasse ergänzt die Funktionalität der Basisklasse oder schränkt diese ein.
- Falls die Funktionalität der Basisklasse ergänzt wird, profitieren wir davon auch in der abgeleiteten Klasse.
- Richtig eingesetzt erlaubt die Vererbung ein besseres Verständnis des Codes für den Autor und Außenstehende und damit eine verbesserte Wartbarkeit.
- Einfachere Codes sind zwar oft die besseren Codes, aber zwanghafter Einsatz von Vererbung macht Codes nicht unbedingt einfacher.

Sehr grobe Faustregel für sinnvolle Vererbung:
prüfen Sie immer, ob eine *ist ein*-Beziehung vorliegt...

- Gute Vererbung: `stummfilm < film` (ein Stummfilm *ist ein* Film)
- Schlechte Vererbung: `film < videokassette` (... *enthält einen* Film)

Ende Teil 4.2

Fragen?