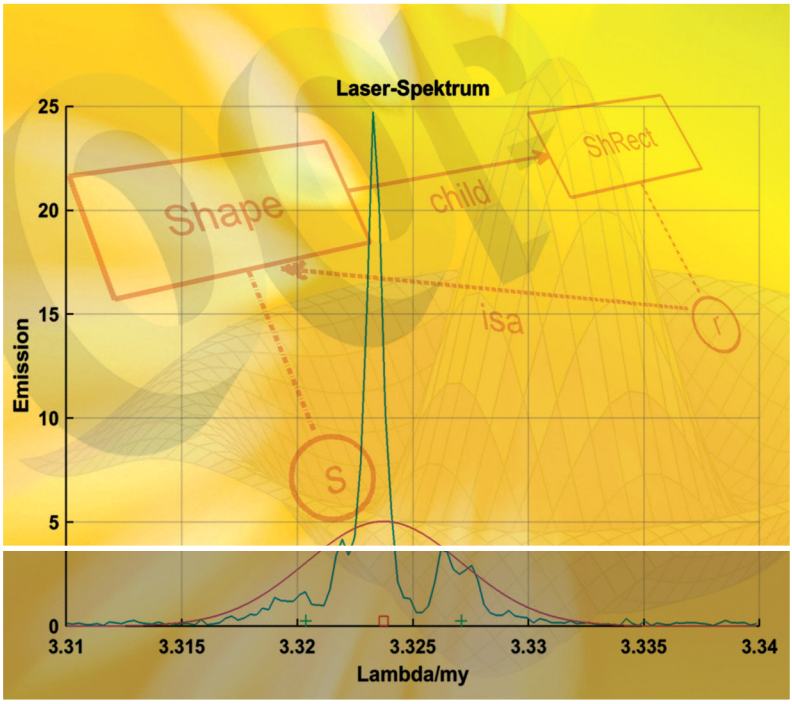
Ulrich Stein

Objektorientierte Programmierung mit MATLAB

Klassen, Vererbung, Polymorphie



HANSER

Stein

Objektorientierte Programmierung mit MATLAB

Ulrich Stein

Objektorientierte Programmierung mit MATLAB

Klassen, Vererbung, Polymorphie

Mit 57 Bildern, 31 Aufgaben und zahlreichen Listings

Fachbuchverlag Leipzig
im Carl Hanser Verlag

Prof. Dr. rer. nat. Ulrich Stein
Hochschule für Angewandte Wissenschaften Hamburg, Department Maschinenbau und Produktion

Bibliografische Information der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.d-nb.de abrufbar.

ISBN: 978-3-446-44298-6
E-Book-ISBN: 978-3-446-44536-9

Lektorat: Franziska Jacob, M.A.
Herstellung: Dipl.-Ing. (FH) Franziska Kaufmann
Coverconcept: Marc Müller-Bremer, www.rebranding.de, München
Coverrealisierung: Stephan Rönigk
Satz: Kösel Media GmbH, Krugzell
Druck und Bindung: Pustet, Regensburg
Printed in Germany

Vorwort

Zugegeben, es hat eine Weile gedauert, bis ich die Möglichkeiten der Objektorientierung in MATLAB® schätzen lernte. Im Jahr 2007 schrieb ich ein Lehrbuch, das sich mit dem „Programmieren mit MATLAB“ beschäftigt. Damals standen wir am Fachbereich Maschinenbau und Produktion der HAW Hamburg vor einem Problem: Unsere Studenten waren nicht recht zu motivieren, eine Programmiersprache zu erlernen. Wir „zwangen“ sie, sich längere Zeit mit Compiler, Linker und unverständlichen Fehlermeldungen herumzuquälen, um am Schluss als Ergebnis die Zahl 42 auf dem Bildschirm zu sehen. Ingenieure kommen inzwischen immer weniger mit Programmiersprachen wie C++ in Kontakt, ob in der Industrie oder im weiteren Verlauf des Studiums. Anders liegt die Situation beim Programm MATLAB. Dessen Funktionalität wird bei uns intensiv genutzt, beispielsweise zum Lösen von Differentialgleichungen oder zum Ansteuern von Robotern. Und besonders die grafischen Möglichkeiten von MATLAB fördern das Verständnis für viele technische Anwendungen. Deshalb habe ich damals, zusammen mit Kollegen, unser Bachelor-Modul „Angewandte Informatik“ auf die in MATLAB integrierte Programmiersprache umgestellt. Aus dem Skript zu dieser Vorlesung entstand das oben erwähnte Lehrbuch.

An die objektorientierte Programmierung (OOP) dachte ich dabei nur am Rande. Zwar hatte MATLAB auch damals bereits eingeschränkte OOP-Funktionalität, die ich in meinem Buch beschrieb. Die Vorgehensweise war jedoch noch sehr umständlich. Der große Schritt kam im Jahr 2008 mit der Einführung der *classdef*-Datei, die ähnlich wie in anderen OOP-Sprachen die Struktur einer Klasse festlegt. Durch die Definition von Handle-Klassen erhielt MATLAB außerdem eine Art von Referenzen.

Trotzdem, eine Zeit lang blieb ich noch skeptisch, ob man mit MATLAB wirklich vernünftig objektorientiert programmieren kann. Im Vergleich zu anderen OOP-Sprachen fehlten doch ein paar Dinge, beispielsweise die Möglichkeit, mehrere Methoden mit demselben Namen, aber unterschiedlicher Signatur zu deklarieren. Dies ist in MATLAB nicht vorgesehen. Aber hierfür – und auch für andere fehlende Bereiche – kann man sich recht einfach einen vernünftigen Workaround bauen.

Je länger ich mich mit der OOP-Funktionalität in MATLAB beschäftigte, desto mehr hat sie mich überzeugt. Und ich hoffe, dass ich auch Sie in diesem Buch dafür begeistern kann.

Nun zu Ihnen: Was erwarte ich von meinem Leser?

Idealerweise sollten Sie bereits ein wenig programmiert haben, nicht notwendigerweise in MATLAB. Zwar stelle ich Ihnen in diesem Buch, in den Kapiteln 1 und 2, die Oberfläche und die wichtigsten Sprachelemente von MATLAB vor – soweit es zum Verständnis der folgenden Abschnitte notwendig ist. Das ist jedoch nur eine knappe Zusammenfassung, eher gedacht für Umsteiger von anderen Programmiersprachen wie C++. Einem absoluten

Anfänger würde ich als Einstieg eher mein Buch „Programmieren mit MATLAB“ empfehlen – ein Lehrbuch, in dem Sie auch viele Beispiele und Übungen finden.

Doch zurück zum jetzigen Buch: Kapitel 3 beschreibt den objektorientierten Ansatz von MATLAB. Hier werden die zentralen Begriffe eingeführt, wie Klassen, Eigenschaften, Methoden, Datenkapselung, Vererbung, Polymorphie, Handle-Klassen etc.

In Kapitel 4 finden Sie längere Anwendungen aus der Physik und dem Maschinenbau. Dies soll den Umgang mit dem Erlernten vertiefen und weitere Tipps für ein strukturiertes Vorgehen geben.

Zum Abschluss liefert Kapitel 5 eine Befehlsreferenz und den Vergleich mit anderen OOP-Sprachen.

Dieses Buch ist aber kein Referenz-Handbuch für MATLAB. Die MATLAB-Funktionen werden oft nur so weit vorgestellt, wie es für die aktuelle Aufgabenstellung nötig ist. Für eine vollständige Definition der Funktionen sei auf die MATLAB-Hilfe verwiesen.

Die Idee, ein zweites Buch zu MATLAB zu schreiben, entstand während einer Unterredung mit Frau Franziska Jacob, M. A., meiner Ansprechpartnerin beim Fachbuchverlag Leipzig im Carl Hanser Verlag. Vielen Dank für ihr Engagement, mit dem sie das Projekt im Verlag durchsetzte. Dank auch an Frau Dipl.-Ing. (FH) Franziska Kaufmann, die mir beim Layout zur Seite stand. Dank an alle Kollegen, die mich zu diesem Projekt ermutigten und mir hilfreiche Tipps gaben, speziell Prof. Dr. rer. nat. Ivo Nowak, Prof. Dr. rer. nat. Thorsten Struckmann und Prof. Dr.-Ing. Jürgen Dankert. Und einen besonderen Dank an Elfriede Neubauer, die mir bei der stilistischen Überarbeitung eine große Hilfe war.

Die im Buch beschriebenen und abgebildeten Abläufe beziehen sich auf die Bedienoberfläche der Version MATLAB 2015a. Andere MATLAB-Versionen präsentieren sich dem Anwender zum Teil mit einer leicht abgewandelten Oberfläche. Lassen Sie sich deshalb nicht verwirren. Die vorgestellten Programme wurden mit verschiedenen Versionen getestet. Erweiterungen und die Lösungen der Aufgaben finden Sie auf meiner Homepage

www.Stein-Ulrich.de/Matlab/

Ich wünsche den Lesern, dass Ihnen das Programmieren auch Spaß macht und dass Ihnen möglichst viel vom hier präsentierten Stoff bei Problemlösungen nützt. Und nicht verdrängen oder vergessen: Informatik kann auch Schaden anrichten. Deshalb sollte jeder, der programmiert, sich überlegen, ob er sein Tun verantworten kann und will.

Hamburg, im August 2015

Ulrich Stein

Inhalt

1 Einführung

Hier im ersten Kapitel erwartet Sie ein kleiner Schnupperkurs zu Objekten und zu MATLAB – als Einstimmung in das Thema und zur Abklärung einiger Begriffe. Eine systematische und eingehendere Behandlung folgt erst später in den weiteren Kapiteln des Buches.

1.1 Warum objektorientiert?

Objektorientierte Programmierung, abgekürzt **OOP**, gibt es schon recht lange. Bereits Ende der 1960er-Jahre entstand mit Simula-67 die erste bekannte objektorientierte Programmiersprache. Später kamen weitere Sprachen hinzu, die die objektorientierten Prinzipien ausbauten, Sprachen wie Smalltalk, Lisp und Ada. Den großen Durchbruch schaffte OOP aber erst in den 1990er-Jahren. Viele Programmiersprachen unterstützen inzwischen sowohl OOP als auch die prozedurale Programmierung.

Die am häufigsten verwendete OOP-Sprache ist aktuell C++, zusammen mit Java und C++-Ablegern wie C#. Entwickelt wurde C++ von Bjarne Stroustrup ab Ende der 1970er-Jahre, als Erweiterung der Sprache C, die Dennis Ritchie kurz zuvor präsentiert hatte. Stroustrup integrierte in C++ zentrale Mechanismen der Sprache Simula-67.

Mit dem Release von 2008 hat MATLAB seinen vorher etwas holprigen OOP-Ansatz überarbeitet und bietet nun ebenfalls interessante Möglichkeiten, objektorientiert zu programmieren.

Bei den vielen, unterschiedlichen OOP-Sprachen bleibt es nicht aus, dass die Konzepte und vor allem die verwendeten Bezeichner voneinander abweichen. In den deutschen Versionen der Sprachen kommt noch hinzu, dass für gleiche Begriffe zum Teil unterschiedliche Übersetzungen gewählt wurden. Ich werde mich hier in der Wortwahl meist an die deutsche Übersetzung von Stroustrups C++-Buch halten – weil einerseits die MATLAB-Hilfe nur die englischen Bezeichner kennt, und es mir andererseits wichtig erscheint, den OOP-Neuling nicht durch eine Vielzahl von abweichenden, deutschen Bezeichnungen zu verwirren.

Soweit die Vorgeschichte – aber was ist denn nun eigentlich „objektorientiert“?

Bjarne Stroustrup bringt in seinem hervorragenden Buch „Einführung in die Programmierung mit C++“ die folgenden Definitionen:

„Programmieren ist die Kunst, Lösungen für Probleme so zu formulieren, dass ein Computer diese Lösungen ausführen kann.“

Und: OOP ist „Programmierung mit Vererbung, Laufzeitpolymorphie und Kapselung“.

Alles klar? Und wie man das in MATLAB macht, wissen Sie auch bereits?

Dann müssen Sie dieses Buch gar nicht lesen. Denn um diese Begriffe geht es hier. Und darum, wie man mit Hilfe der OOP Software-Projekte strukturiert anlegt, größere Datenmengen zusammen mit den Berechnungen vernünftig verwaltet oder Informationen zur weiteren Verwendung in verketteten Listen ablegt.

Und noch eine Warnung vorweg: Dieses Buch ist nicht für den absoluten Anfänger gedacht. Ich gehe davon aus, dass Sie bereits ein wenig Kontakt mit MATLAB gehabt haben oder zumindest etwas Erfahrung im Programmieren mit einer beliebigen sonstigen Programmiersprache mitbringen.

Doch zurück zur Frage der Überschrift: Warum objektorientiert?

Welche Vorteile bringt OOP? Auf diese Frage kommt meist die Standardantwort: OOP macht es einfacher, einmal geschriebenen Programm-Code für ein anders geartetes Projekt weiterzuentwickeln und wiederzuverwenden.

Das klingt sehr technisch und ist sicher nur für einen Teil der Programmierer das Wichtigste. Mich fasziniert an der OOP eher die neue Sichtweise, manche sagen „das andere Paradigma“:

Hier geht es um Objekte, das Abbilden von realen Gegenständen im Computer. Objekte mit speziellen Eigenschaften und Zuständen. Um die Kontrolle über diese Eigenschaften. Und es geht um die Kommunikation der Objekte miteinander, wie wir es als Wechselwirkungen im Alltag erleben. Um die Entwicklung von Schnittstellen zu den Objekten. Und um die Trennung zwischen Schnittstelle und Implementierung, welche es ermöglicht, einzelne Programmbausteine für eine Weiterentwicklung abzuändern, ohne die Bereiche antasten zu müssen, die diese Funktionalität aufrufen.

Durch diesen Ansatz wird es leichter, Ideen im Programm-Code zu repräsentieren, Abläufe zu entwickeln und den Code verständlicher zu machen. Viel leichter als in der herkömmlichen Programmierung, welche in erster Linie Daten verarbeitet – Daten, die nur eine Menge von Zahlen oder Texten sind, ohne weiteren Zusammenhang.

Auf den folgenden Seiten möchte ich versuchen, auch Sie für diese andere Sichtweise zu begeistern. Es wird allerdings etwas dauern. Sie sollten dieses Buch nicht einfach nur durchlesen. Sie müssen die Beispiele nachprogrammieren, um zum Erfolg zu kommen.

Um die syntaktische Nähe zur Programmiersprache C bzw. C++ hervorzuheben, finden sich in den Programmbeispielen dieses Buches oft zusätzliche Klammern und Semikolons, die in MATLAB nicht zwingend notwendig sind, den Programm-Code aber „C-ähnlicher“ machen.

Weitere Informationen zu MATLAB finden Sie im Internet auf den Seiten von „The MathWorks, Inc.“:

http://www.mathworks.com/

http://www.mathworks.de/

und speziell auf der Seite von Cleve Moler:

http://www.mathworks.com/moler/

1.2 Erstes Objekt: Auto

Objekte beschreiben (normalerweise) Dinge aus der realen Welt. Dinge, die wir durch Datenstrukturen im Computer repräsentieren werden. Diese Objekte haben eine gewisse „Intelligenz“. Sie besitzen Eigenschaften und haben Fähigkeiten.

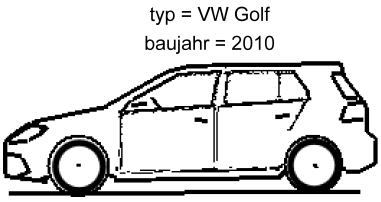


Bild 1.1 Objekt *VW Golf*

Betrachten wir ein reales Objekt, sagen wir ein Auto, beispielsweise einen VW Golf. Um dieses Objekt zu identifizieren, um es eindeutig wiederzufinden, geben wir ihm als Bezeichnung einen Namen. Nein, nicht „Schnuffel“ oder etwas Ähnliches – das wäre nicht eindeutig genug –, wir nehmen sein Kennzeichen, zum Beispiel „HH_BT_21“:

```
>> HH_BT_21 = Auto( 'VW_Golf', 2010 );
```

Unser Auto-Objekt *HH_BT_21* wurde durch den obigen Aufruf im Speicher des Rechners angelegt. Es hat gewisse Eigenschaften, die es mit anderen Autos teilt oder in denen es sich von ihnen unterscheidet. Da wäre der Fahrzeugtyp „VW Golf“. Diese Eigenschaft sollte sich im Laufe des Autolebens nicht ändern. Das gilt auch für das Baujahr, hier 2010.

Daneben gibt es weitere, konkrete Eigenschaften, die vom aktuellen Zustand des Autos abhängen. Beschrieben durch einen Satz von Variablen, deren Werte mit der Zeit variieren, zum Beispiel der Kilometerstand, die aktuelle Geschwindigkeit, die Tankfüllung.

Diese Daten, die **Eigenschaften** (auf Englisch *properties*), legen den **Zustand** des Autos fest:

```
properties
  typ = 'VW_Golf';
  baujahr = 2010;
  kilometer = 50000;          % km
  geschwindigkeit = 100;      % km/h
  benzin = 20;                % Liter im Tank
end
```

Die aktuellen Eigenschaften unseres Autos, zum Beispiel die Geschwindigkeit, kann man sich vom Auto-Objekt *HH_BT_21* mit Hilfe des Punktoperators (`HH_BT_21.`) holen und mit der MATLAB-Funktion *disp* (Display) ausgeben:

```
>> disp( HH_BT_21.geschwindigkeit );
   100
```

Nach dieser Abfrage erscheint auf dem Bildschirm die Zahl 100, also 100 km/h. Analog können Sie sich auch Überblick über die anderen Eigenschaften verschaffen. Im realen Auto macht dies ein Blick auf die Anzeige im Armaturenbrett.

Ein Auto, das sich selbst überlassen bleibt, ist, zumindest für den heutigen Straßenverkehr, nicht sinnvoll. Es muss mit anderen Objekten kooperieren, primär mit dem Fahrer, der zum Beispiel das Gaspedal tritt, um das Auto zu beschleunigen. Die Fähigkeit zu beschleunigen wirkt als Operation, die das Auto dann ausführt, wenn es eine spezielle „Nachricht" erhält – das Durchtreten des Gaspedals.

In der Nomenklatur der objektorientierten Programmierung bezeichnet man eine solche Fähigkeit als **Methode** (auf Englisch *method*). Die Methode *beschleunigen* hat den Effekt, dass sich die Geschwindigkeit unseres Objekts um einen gewissen Wert erhöht, hier spezifiziert durch den formalen Parameter *wie_viel*:

```
methods
  function obj = beschleunigen( obj, wie_viel )
    neueGeschwindigkeit = obj.geschwindigkeit + wie_viel;
    obj.geschwindigkeit = neueGeschwindigkeit;
  end
end
```

Dieser Wert der Beschleunigung wird der Methode beim Aufruf als explizite Zahl übergeben. Machen wir das Auto-Objekt mit dem Namen *HH_BT_21* um den Wert 30 schneller, also um 30 km/h:

```
>> HH_BT_21 = beschleunigen( HH_BT_21, 30 );
```

MATLAB erlaubt für diesen Aufruf noch eine zweite Version, die wie bei der Abfrage der Eigenschaften den **Punktoperator** verwendet:

```
>> HH_BT_21 = HH_BT_21.beschleunigen( 30 );
```

In beiden Fällen sollte sich die Geschwindigkeit erhöht haben. Der Zustand des Autos hat sich verändert. Wichtig ist, dass Sie nach dem Aufruf der Methode das veränderte Objekt *obj* wieder dem Auto-Objekt *HH_BT_21* zuweisen, also `HH_BT_21 = ...`

Lassen wir uns den aktuellen Wert ausgeben:

```
>> disp( HH_BT_21.geschwindigkeit );
   130
```

Auf dem Bildschirm erscheint jetzt die Zahl 130, also 30 km/h schneller als vorher.

Da wir gerade bei den Methoden sind – eine Methode, die gleich zu Beginn eine Rolle spielte, habe ich Ihnen bisher verschwiegen. Mit dem Aufruf

```
>> HH_BT_21 = Auto( 'VW_Golf', 2010 );
```

wurde ein Auto-Objekt vom Typ VW_Golf und dem Baujahr 2010 erzeugt. Der Fähigkeit zur Erzeugung von Objekten liegt eine besondere Methode zu Grunde, der **Konstruktor**.

In unserem Fall könnte diese Konstruktor-Methode wie folgt implementiert sein:

```
methods
  function obj = Auto( typ, baujahr )
    obj.typ = typ;              % übergebene Daten
    obj.baujahr = baujahr;
    obj.kilometer = 0;          % 0 km, initialisiert
    obj.geschwindigkeit = 0;    % 0 km/h
    obj.benzin = 40;            % 40 Liter im Tank
  end
end
```

Der Konstruktor erzeugt das Auto-Objekt. Und beim Aufruf `Auto( 'VW_Golf', 2010 )` werden ihm die Werte für die Eigenschaften *typ* und *baujahr* explizit mitgeteilt. Mit Hilfe der Zuweisungen `obj.typ = typ;` bzw. `obj.baujahr = baujahr;` übergibt der Konstruktor diese Daten dann an das von ihm erzeugte Objekt *obj*.

Die restlichen Eigenschaften haben wir selbständig auf feste Werte gesetzt, also *kilometer* und *geschwindigkeit* auf null, und der Tank wurde uns netterweise mit 40 Liter *benzin* gefüllt. Eine solche Vorgabe der Eigenschaften mit expliziten Werten nennt man **Initialisierung**.

Fahren wir mit dem Auto eine Weile durch die Gegend. Irgendwann wird sich die Tankfüllung dem Ende zuneigen. Im Armaturenbrett sollte dann eine Warnleuchte blinken. Auch dieses **Ereignis** (auf Englisch *event*) lässt sich in MATLAB nachbauen. Der Fahrer muss natürlich auf das Signal achten, im übertragenen Sinn darauf hören, ein *listener* sein, wie die englische Bezeichnung lautet.

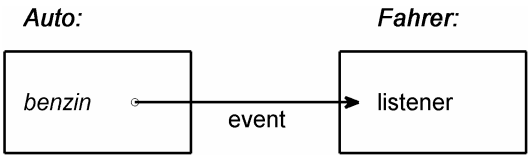


Bild 1.2 Ereignis (*event*) als Nachricht an den „*listener*“

Es gibt auf der Straße aber nicht nur den einen VW Golf. Erzeugen wir ein zweites Auto:

```
>> HH_EN_100 = Auto( 'Mazda_MX5', 2009 );
```

Wieder werden im Speicher des Computers die Daten für das weitere Auto-Objekt abgelegt, diesmal unter dem Namen *HH_EN_100*. Dieses Auto hat andere Eigenschaften, zum Beispiel einen anderen Typ als das vorherige mit dem Namen *HH_BT_21*. In der Grundstruktur sind sich beide Objekte jedoch ähnlich. Unter anderem sollte jedes Auto seine

Geschwindigkeit erhöhen, wenn das Gaspedal gedrückt wird und das Objekt auf diese Nachricht mit der Methode *beschleunigen* reagiert.

Solch gleichartige Objekte fasst man in der objektorientierten Programmierung in **Klassen** zusammen (auf Englisch *class*). Beide Fahrzeuge, *HH_BT_21* und *HH_EN_100*, sind Exemplare (auf Englisch *instances*) derselben Klasse *Auto*. Wenn auch der Zustand der jeweiligen Autos anders ist, ihre Eigenschaften unterschiedliche Werte haben, so sind sie doch Objekte vom selben Datentyp, der Klasse *Auto*. Das heißt, sie verfügen über die gleichen Methoden und haben die gleichen Variablen zur Beschreibung der Eigenschaften, wie zum Beispiel *geschwindigkeit* oder *kilometer*.

Die Definition der Klasse *Auto* lautet in MATLAB:

Listing 1.1 class *Auto*

```
% Definition der Klasse Auto im M-File Auto.m
classdef Auto
  % Eigenschaften:
  properties
    typ = '';
    baujahr = 0;
    kilometer = 0;
    geschwindigkeit = 0;
    benzin = 0;
  end   % properties
  % Methoden:
  methods
    % Konstruktor
    function obj = Auto( typ, baujahr )
      obj.typ = typ;
      obj.baujahr = baujahr;
      obj.kilometer = 0;         % 0 km
      obj.geschwindigkeit = 0;  % 0 km/h
      obj.benzin = 40;           % 40 Liter im Tank
    end
    % Methode beschleunigen
    function obj = beschleunigen( obj, wie_viel )
      neueGeschwindigkeit = obj.geschwindigkeit + wie_viel;
      obj.geschwindigkeit = neueGeschwindigkeit;
    end
  end   % methods
end
```

Wenn Sie diesen Programm-Code im aktuellen Verzeichnis von MATLAB in der Datei „Auto.m“ abspeichern, können Sie im Command-Window von MATLAB die Klasse testen:

```
>> HH_BT_21 = Auto( 'VW_Golf', 2010 );
>> HH_BT_21 = HH_BT_21.beschleunigen( 40 );
>> disp( HH_BT_21.geschwindigkeit );
    40
>> HH_EN_100 = Auto( 'Mazda_MX5', 2009 );
>> disp( HH_EN_100.typ )
    Mazda_MX5
```

Jetzt aber erst mal Stopp. Dies sollte nur ein Appetithappen sein, zur Einführung in OOP. Und den Lesern, die wenig Erfahrung mit MATLAB und objektorientierten Sprachen haben, wird der Kopf schon rauchen. Aber keine Angst, auch wenn einiges noch unverständlich war – das Ganze wird Ihnen in den folgenden Kapiteln in aller Ausführlichkeit erklärt.

1.3 MATLAB

MATLAB ist ein weites Feld und bietet eine Unzahl von Möglichkeiten. In diesem Buch gehe ich davon aus, dass Sie bereits ein wenig Kontakt mit MATLAB gehabt haben. Zur Auffrischung Ihres Wissens dient dieser Abschnitt, der kurz die wichtigsten Grundlagen zusammenstellt.

Nach dem Starten von MATLAB erscheint der MATLAB-Desktop, der in mehrere Fenster aufgeteilt ist. Im rechten Fenster, dem **Command Window**, können Sie hinter dem Prompt, „>>“, einzelne MATLAB-Befehle eintippen. Im linken Fenster mit dem Titel „Current Folder“ haben Sie Zugriff auf alle Dateien im aktuellen Verzeichnis. Darunter, zu dem Reiter „Workspace“, sehen Sie alle Variablen, die Sie bisher angelegt haben. Die „Command History“ erreichen Sie über die Taste „Cursor nach oben“. In dieser Liste können Sie vorher ausgeführte Befehle auswählen, um sie zu wiederholen.

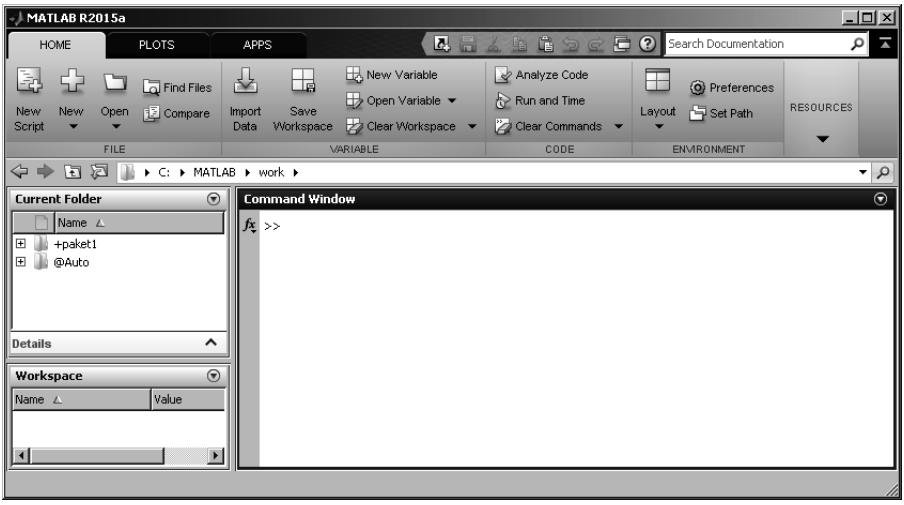


Bild 1.3 MATLAB-Desktop

Ihr Arbeitsverzeichnis wird oben neben der Icon-Leiste angezeigt. Weiter rechts liegen die Elemente zum Wechseln des Verzeichnisses. Sie können den MATLAB-Desktop beliebig anpassen. Der Menü-Befehl „Desktop + Desktop Layout + Default“ stellt Ihnen bei Bedarf die Originalkonfiguration wieder her. Links unten, über den Schaltknopf „Start“, haben Sie Zugriff auf andere Module und die **Toolboxen**™, die für Ihr System verfügbar sind.

Das Command-Window dient zum Ausführen von Befehlen bzw. selbst geschriebenen Funktionen, deren Definition im aktuellen Verzeichnis als M-File vorliegen sollte – oder Sie haben zusätzliche Suchpfade definiert. Das Kommando, das Sie ausführen wollen, geben Sie hinter dem Prompt „>>“ (der Eingabeaufforderung) ein, zum Beispiel:

```
>> x = 1+1
```

Nach dem Drücken der Eingabetaste (Return/Enter-Taste) antwortet MATLAB:

```
x = 2
>>
```

Durch diesen Befehl wird im Workspace eine Variable mit dem Namen *x* angelegt und ihr der Wert 2 zugewiesen.

In MATLAB müssen Sie bei einer Variablendefinition den **Datentyp** nicht notwendigerweise explizit angeben. Wenn Sie ihn nicht weiter spezifizieren, wählt MATLAB für Zahlen automatisch den Typ *double* (reelle Zahl in doppelter Genauigkeit) und für Texte den Typ *char*.

Weitere Datentypen sind zum Beispiel *int32* für ganze Zahlen (32 Bit), *uint8* für kleine ganze Zahlen ohne Vorzeichen (unsigned, 8 Bit) oder *logical* für die Wahrheitswerte.

Der folgende Befehl legt eine Variable *y* vom Typ *int32* an, mit dem Wert 5:

```
>> y = int32( 5 );
```

Den Datentyp einer Variablen können Sie mit dem Befehl *whos* auslesen:

```
>> whos y
  Name      Size     Bytes  Class
  y         1x1          4  int32 array
```

y ist in unserem Fall vom Typ *int32*, genauer gesagt ein 1x1-Array vom Typ *int32*. Die Standardtypen von MATLAB sind nicht einzelne Zahlen, sondern Matrizen, also zweidimensionale **Felder** (Arrays) mit einer gewissen Anzahl von Zeilen und Spalten.

Ein Feld mit mehr als einer Zelle erzeugen Sie durch die Angabe einer Wertliste, die zwischen eckigen Klammern [...] steht. Die einzelnen Zeilen des Feldes werden durch jeweils ein Semikolon `[ Zeile1; Zeile2; ... ]` getrennt. Die Daten innerhalb einer Zeile (die Spalten) werden durch Kommas oder Leerzeichen getrennt.

Die folgende Anweisung legt einen Spaltenvektor an (3 Zeilen, 1 Spalte):

```
>> a = [ 3; 2; 1 ];
```

Den entsprechenden Zeilenvektor (1 Zeile und 3 Spalten) erhält man durch:

```
>> b = [ 3, 2, 1 ];
```

Die Definition einer zweidimensionalen Matrix enthält sowohl Spalten- als auch Zeilentrenner, beispielsweise das Feld *c* mit zwei Zeilen und zwei Spalten:

```
>> c = [ 1 2; 3 4 ]
c =  1     2
     3     4
```

Text wird von MATLAB als Character-Array (Typ *char*) mit einer Zeile und *n* Spalten behandelt.

Sie können MATLAB wie einen Taschenrechner bedienen, mit den gewohnten mathematischen Symbolen und den runden Klammern:

```
>> y = (20+4*4) / 9
y = 4
```

In MATLAB sind Rechenoperationen, „+“, „–“, „*“, „/“ und die Potenzierung „^“ auch für Matrizen definiert. Dies erfolgt analog den Regeln für Matrix-Operationen.

Möchte man anstelle der Matrix-Multiplikation nur die Komponenten der Matrizen miteinander multiplizieren, erreicht man dies, indem man vor den Multiplikationsoperator * einen Punkt setzt. Für die Matrix `e = c.*c` ist die Komponente e_{21} beispielsweise als das Quadrat der Komponente c_{21} definiert: $e_{21} = c_{21} * c_{21} = 3 * 3 = 9$:

```
>> e = c .* c
e =  1     4
     9    16
```

MATLAB kennt folgende **Zahlenformate**:

- ganze Dezimalzahlen mit oder ohne Vorzeichen: `1`, `–2`, `+30`, `–400`,
- Dezimalbrüche, mit einem Dezimalpunkt (nicht mit einem Komma!), mit oder ohne Vorzeichen (vor oder nach dem Punkt ist keine Zahl notwendig): `1.5`, `–2.0`, `+.25`, `–425.`,
- dezimale Gleitkommazahlen, wobei Mantisse und Exponent mittels des Buchstabens *e* (entspricht einem „10-hoch“) verknüpft sind: `1.0e+3` bedeutet $1.0 * 10^{+3} = 1000$.

Imaginäre Zahlen schreibt man, wie üblich, mit einem (ohne Zwischenraum) angehängten *i*, das auch durch ein *j* ersetzt werden kann. Eine komplexe Zahl entsteht als Summe aus Real- und Imaginärteil:

```
>> c = 3 + 4i
c = 3.0000 + 4.0000i
```

Alternativ ist auch die Schreibweise mit dem Multiplikationspunkt erlaubt:

```
>> c = 3 + 4 * i
```

Texte (Strings, Character-Arrays) werden zwischen zwei einfache Hoch-Kommata gesetzt:

```
>> t = 'Das ist ein Text'
t = Das ist ein Text
```

Daneben gibt es Konstanten, die man über ihren Namen anspricht, zum Beispiel die Zahl *pi* = 3.14 … (eine Liste der Konstanten finden Sie in Kapitel 5).

MATLAB stellt eine Vielzahl von **Funktionen** zur Verfügung, die in den eigenen Programm-Code eingebaut werden können. Diese Funktionen sind in Bibliotheken organisiert, wie zum Beispiel mathematische Bibliotheken mit den Funktionen *sin*, *log* etc., Input/Output-Bibliotheken mit Funktionen zur Bildschirmausgabe etc., Grafikbibliotheken zum Zeichnen von 2- oder 3-dimensionalen Objekten.

Hilfe zu einer Funktion können Sie über *help* abrufen, zum Beispiel:

```
>> help sin
SIN  Sine.
   SIN(X) is the sine of the elements of X.
```

Eine etwas umfangreichere Dokumentation erhalten Sie durch die *doc*-Funktion:

```
>> doc sin
```

Um alle definierten Variablen aus dem Speicher von MATLAB zu löschen, gibt es die Funktion *clear all*. Mit dem Befehl *clc* löschen Sie die Anzeigen im Command-Window und erhalten so ein aufgeräumtes Fenster.

1.4 Aufbau des Buches

Im nächsten Kapitel geht es endlich los mit dem Programmieren. Damit Sie bei der Vielfalt des Stoffes den Überblick nicht verlieren, verrate ich Ihnen jetzt schon ein wenig von dem, was Sie auf den folgenden Seiten erwartet.

Kapitel 1 haben Sie ja bereits hinter sich – falls Sie das Buch von Anfang an gelesen haben und nicht nur zufällig beim Blättern auf diese Seite gestoßen sind. Das erste Kapitel ist für den Einstieg gedacht, für einen ersten Kontakt mit MATLAB und dem Bereich der Objektorientierung. Dabei wurden einige Themen nur sehr oberflächlich behandelt. Systematisch geht es erst in den folgenden Kapiteln los.

Kapitel 2 dient als Überblick über die Elemente der Programmiersprache von MATLAB. Aufgabe dieses Buches soll es nicht sein, Sie in das allgemeine Programmieren mit MATLAB einzuführen. Dann wäre der Umfang des Buches explodiert. Es wird vorausgesetzt, dass Sie sich schon ein wenig mit MATLAB auskennen oder Programmiererfahrung in einer anderen Sprache haben, wie C++ oder Java. Für eine intensivere Beschäftigung mit dem Thema Programmieren verweise ich Sie auf mein Lehrbuch „Programmieren mit MATLAB“. Dort wird das Ganze viel ausführlicher erklärt und eingeübt.