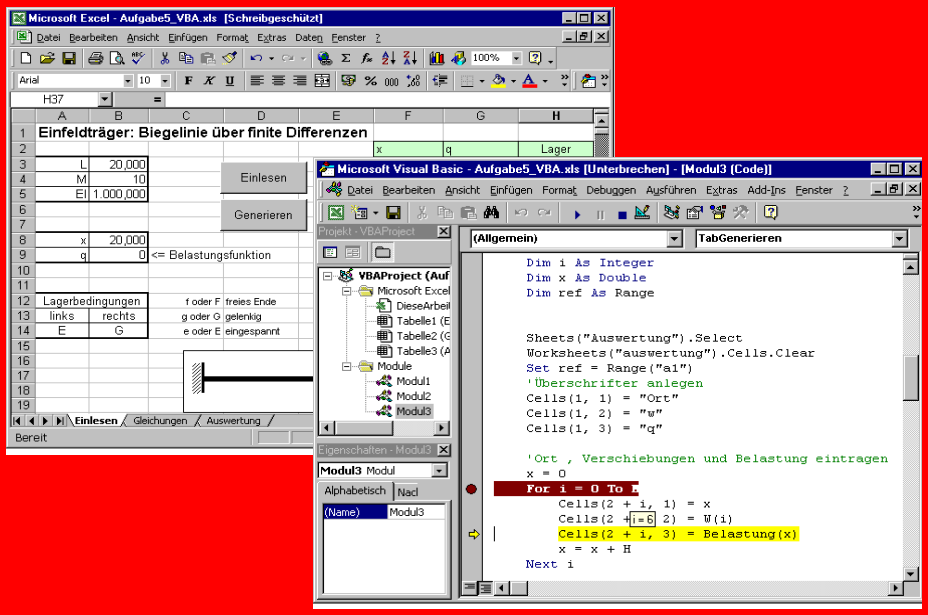
Karlsruher Institut für Technologie
Institut für Baustatik

Informationsverarbeitung im Bauwesen

Grundlagen der Programmierung mit VBA und Excel



BAUSTATIK

Vorbemerkungen

- Das vorliegende Manuskript ist der unveränderte Nachdruck des Manuskriptes Informationsverarbeitung im Bauwesen II von Herrn Kollegen Prof. Dr.-Ing. K.-U. Bletzinger, dem hiermit herzlich für die Bereitstellung gedankt sei.

- Die vorliegenden Arbeitsunterlagen sind lediglich als **Hilfsmittel** zur Lehrveranstaltung Informationsverarbeitung im Bauwesen am Karlsruher Institut für Technologie zu verstehen.

- Die Arbeitsunterlagen können über das Internet von der Homepage des Instituts bezogen werden.

- Ohne Genehmigung des Autors ist es nicht gestattet, dieses Heft ganz oder teilweise auf fotomechanischem Wege (Fotokopie, Mikrokopie) zu vervielfältigen.

- © Prof. Dr.–Ing. W. Wagner
Institut für Baustatik
Karlsruher Institut für Technologie
Postfach 6980
76128 Karlsruhe

Telefon: (0721) 608–2280
Telefax: (0721) 608–6015
E–mail: bs@.uni-karlsruhe.de
Internet: http://www.bs.uni-karlsruhe.de

Inhaltsverzeichnis

1 EINFÜHRUNG - BESONDERHEITEN VON VBA 3

1.1 EREIGNISORIENTIERTE PROGRAMMIERUNG 3
1.2 OBJEKTORIENTIERTE PROGRAMMIERUNG 3
1.3 DIE ONLINE HILFE 5
1.4 PROGRAMME ERSTELLEN 5

2 ERSTE SCHRITTE 6

2.1 BEISPIEL: AMPELSCHALTUNG 6
2.1.1 Makros: Ampellichter an- und ausschalten 6
2.1.2 Der generierte VBA-Code 9
2.1.3 weitere Prozeduren erstellen 10
2.2 DIE PROGRAMMIERUMGEBUNG 11
2.2.1 Programmeditor 11
2.2.2 Projektexplorer 12
2.2.3 Eigenschaftsfenster 12
2.2.4 Objektkatalog 12
2.2.5 Formularfenster 13
2.2.6 Hilfsmittel zum Testen von Programmen 14

3 VBA-SPRACHELEMENTE 19

3.1 MODULARE PROGRAMMIERUNG 19
3.2 PROZEDUREN UND FUNKTIONEN 19
3.2.1 Prozeduren 19
3.2.2 Funktionen 20
3.2.3 Parameter 20
3.2.4 Verfahren der Werteübergabe 21
3.2.5 Prozeduren und Funktionen abbrechen 22
3.2.6 Rekursion 22
3.2.7 VBA Funktionen 22
3.2.8 Excel-Funktionen 23
3.3 VARIABLEN, KONSTANTEN UND ARRAYS 24
3.3.1 Regeln für Namensgebung und Deklaration 24
3.3.2 Variable 24
3.3.3 Konstante 25
3.3.4 statische Arrays 25
3.3.5 dynamische Arrays 26
3.3.6 Datenfelder mit der Array-Funktion 27
3.3.7 Gültigkeitsbereich und -dauer 27
3.4 DATENTYPEN 29
3.4.1 Einfache Datentypen 29
3.4.2 implizite Typumwandlung 29
3.4.3 explizite Typumwandlung 30
3.4.4 selbstdefinierte Datentypen 31
3.4.5 Bestimmung des Variablentyps 32
3.5 OPERATOREN 34
3.5.1 Arithmetische Operatoren 34
3.5.2 Vergleichsoperatoren 34
3.5.3 Verkettungsoperatoren 35
3.5.4 logische Operatoren 35
3.6 STRUKTURELEMENTE 36
3.6.1 Verzweigung 36
3.6.2 Mehrfachauswahl 36
3.6.3 Schleifen 37
3.6.4 Behandlung von Benutzerfehlern 39
3.7 OBJEKTTECHNIKEN 40
3.7.1 Property-Prozeduren 40

3.7.2 Eigene Objektklassen 41
3.7.3 Das Collection-Objekt 43
3.8 DAS RANGE-OBJEKT - ZUGRIFF AUF ZELLEN 45
3.8.1 `Cells`-Eigenschaft 45
3.8.2 `CurrentRegion`- Eigenschaft 45
3.8.3 `Count`-Eigenschaft 46
3.8.4 `Column`- und `Row`-Eigenschaften 46
3.8.5 `Columns`- und `Rows`-Eigenschaften 47
3.8.6 `EntireColumn`- und `EntireRow`-Eigenschaften 47
3.8.7 `Offset`-Eigenschaft 48
3.8.8 `Resize`-Eigenschaft 48
3.8.9 `Value`-Eigenschaft 48
3.8.10 `Select`-Methode 48
3.8.11 `Clear` und `ClearContents`-Methoden 49
3.8.12 Beispiel: Matrix einlesen, speichern und zurückschreiben 49
3.9 ZEITGESTEUERTE AUFÜHRUNG VON ROUTINEN 50

4 PROGRAMMELEMENTE 52

4.1 PROGRAMMAUFBAU 52
4.2 PROZEDUREN UND FUNKTIONEN 52
4.3 VARIABLEN 53
4.4 KONSTANTEN 53
4.5 KONTROLLSTRUKTUREN 53
4.5.1 Die Folge (Sequenz) 53
4.5.2 Die Auswahl (Alternative) 54
4.5.3 Die Wiederholung (Iteration) 54
4.5.4 Zählergesteuerte Wiederholung 55
4.5.5 Bedingte Wiederholung 55

5 SOFTWARE ENGINEERING 56

5.1 ANFORDERUNGEN AN SOFTWARE 56
5.2 SOFTWARE-LEBENSZYKLUS 56
5.3 PRINZIPIEN DES SOFTWARE-ENGINEERING 57
5.4 METHODEN DES SOFTWARE-ENGINEERING 57
5.5 ENTSCHEIDUNGSTABELLE 58
5.6 PROGRAMMABLAUFPLAN NACH DIN 66001, PAP 59
5.7 STRUKTOGRAMM UND PSEUDOCODE 61

6 LITERATUR 65

Grundlagen der Programmierung mit VBA

Visual Basic for Applications mit Excel 2000

1 Einführung - Besonderheiten von VBA

Programmieren, also die Analyse eines Problems, seine Umsetzung in eine Abfolge einfacher Handlungen (Algorithmus) und die Darstellung in einer Programmiersprache, folgt verschiedenen Grundsätzen und Regeln. Diese Regeln können mit jeder Sprache erlernt werden. Im Rahmen dieser Veranstaltung haben wir uns für *Visual Basic for Applications*, kurz *VBA*, entschieden. VBA ist eine Untermenge von *Visual Basic* und speziell als *Makrosprache* für die Microsoft Office Produkte entwickelt worden. Seit der Version Office 97 ist VBA die gemeinsame Grundlage von *Excel, Word, Access* und *PowerPoint*. Damit ist der erste Grund für die Entscheidung offenbar: dieser Teil der Vorlesung *Informationsverarbeitung im Bauingenieurwesen* baut unmittelbar auf die im Wintersemester vermittelten Fähigkeiten im Umgang mit Excel auf. Die Kenntnis von Excel wird vorausgesetzt. Darüber hinaus ist VBA eine moderne und mächtige Programmiersprache, die über zwei wesentliche Eigenschaften moderner Programmiersprachen verfügt: sie ist *ereignisorientiert* und (zumindest ansatzweise) *objektorientiert*. Das mit VBA Erlernte kann direkt auf andere Sprachen übertragen werden. Eine weitere vorteilhafte Eigenschaft, ist die Entwicklungsumgebung mit automatischer Syntaxüberprüfung, "Quickinfo"-Eingabehilfe, ausführlicher On-Line-Hilfe, Debugger, Objektkatalog und Projektexplorer, die eine gründliche Einführung in moderne Software-Entwicklungsumgebungen (*Case-Tools)* vermittelt. Tatsächlich unterscheidet sich die VBA-Entwicklungsumgebung nicht von denjenigen für die professionelle Softwareentwicklung, wie z.B. *visual studio C++* von Microsoft, oder denen anderer Hersteller.

1.1 Ereignisorientierte Programmierung

Ereignisorentierte Programme "warten" nach dem Start auf Mitteilungen des Anwenders. Der Anwender steuert interaktiv, meist durch Klicken auf eine Schaltfläche, den Programmablauf. Insbesondere in der PC- und Mac-Welt ist das inzwischen der Standardfall. Im Gegensatz dazu kennt man den *Stapelbetrieb* sequentieller Programme: man gibt das Programm ein, startet und wartet auf das Ergebnis. Typischerweise werden Eingabedaten von einer Datei gelesen (*Input File*) und Ausgabedaten auf eine andere Datei geschrieben (*Output File*). Für die Bearbeitung großer Datenmengen, insbesondere auf Großrechnern, ist das nach wie vor der Normalfall. Das interaktive " 'rumklicken" wäre auch nicht zumutbar. Infolge der interaktiven Kommunikation mit dem Anwender verfügen ereignisorientierte Programme über verschiedene Fenster (*Dialoge* oder *Masken*), über die der Anwender den Ablauf steuert und Daten eingibt. Ereignisorientierte Programme besitzen auch eine andere Struktur als sequentielle Programme: anstelle einer Liste von Befehlen, die dem geplanten Ablauf entsprechend geordnet sind, hat man hier eine Sammlung von Prozeduren und Funktionen, die zumindest auf den ersten Blick in keinem direkten Zusammenhang stehen. In der Tat verfügen diese Programme nicht über ein geplantes, festes Ablaufschema, es wird mit jedem Programmlauf vom Anwender neu bestimmt. Die einzelnen Prozeduren müssen deshalb in unterschiedlichem Kontext lauffähig sein. Dies erfordert vom Programmentwickler eine klare Vorstellung von der Aufgabenstellung seines Programmes, den verschiedenen möglichen Abläufen und die daraus resultierenden Aufgabenbeschreibungen für die verschiedenen Prozeduren.

1.2 Objektorientierte Programmierung

Das Verständnis von Objekten und die Kenntnis der Objektmodelle ist eine Voraussetzung, um mit VBA erfolgreich programmieren zu können. Objekte sind logische Einheiten, die man programmieren und steuern kann. Der Zugriff auf jedes Datum, jede Funktion und jeden Prozeß erfolgt über ein Objekt.

Unter einem Objekt versteht man üblicherweise etwas gegenständliches, z.B. ein Haus. Es hat bestimmte *Eigenschaften* und es gibt bestimmte Dinge, die man damit machen kann, also *Methoden*, die darauf anwendbar sind.

Die meisten Eigenschaften sind veränderlich, z.B, Wert, Wohnfläche (Ausbau des Dachgeschosses), Farbe, Eindeckung, etc. Diese *Objekteigenschaften* charakterisieren das konkrete Objekt. Erst sie machen es möglich, sich eine konkrete Vorstellung von dem Objekt zu machen.

Auf ein Objekt sind bestimmte *Methoden* anwendbar. Das Haus kann man beispielsweise öffnen und schließen. Man kann es bewohnen, heizen, anstreichen, umbauen usw. Andere Methoden dagegen sind nicht anwendbar. Man kann z.B. nicht damit fahren. Dafür wäre das Objekt Auto geeignet.

Objektorientierte Programme sind nach demselben Muster organisiert. Objekte im Zusammenhang mit Excel sind dann z.B. eine Excel-Projekt-Mappe, ein Tabellenblatt, eine Zelle, eine Grafik oder ein Diagramm. Typische Eigenschaften einer Zelle sind der Schrifttyp, die Hintergrundfarbe, die Ausrichtung usw. Für eine Mappe sind "Öffnen" und "Schließen" charakteristische Methoden. Verallgemeinert sind Objekte eines Programms durch Daten (das sind die Eigenschaften) und Funktionen (auch in VBA *Methoden* genannt) bestimmt. Nur die zu einem Objekt gehörenden Funktionen können auf die Daten dieses Objektes zugreifen und diese ändern. Ein anderer Zugriff ist nicht möglich, d.h. die Daten eines Objektes sind vor unqualifizierter Manipulation geschützt (*Datenkapselung*).

Fast alle Elemente, die man in Excel findet, sind VBA-Objekte. Beipiele:

Objekt. Der Bereich "A1:B2" in einem Tabellenblatt. Dem Bereich kann ein Symbolischer Name, d.h. eine Variable, zugewiesen werden, um die weitere Programmierung zu vereinfachen:

```
Dim Auswahl As Range
Set Auswahl = Worksheet("Tabelle1").Range("A1:B2")
```

Die erste Zeile vereinbart das Objekt `Auswahl` vom Typ `Range`, das geeignet ist, einen Tabellenbereich darzustellen. Die zweite Zeile weist dem Objekt den Bereich `"A1:B2"` des Tabellenblattes mit dem Namen `Tabelle1` zu.

Methode. Der Bereich wird mit einer speziellen Anweisung, d.h. Methode, markiert, die nur in Verbindung mit dem Objekt anwendbar ist. `Auswahl` markieren mit der Methode `Select`:

```
Auswahl.Select
```

Eigenschaft. Die Farbe des Hintergrundes ist z.B. eine Eigenschaft. Farbe ändern mit:

```
Auswahl.Interior.ColorIndex = 8
```

Das Wort `Interior` bezeichnet das Innere,d.h. den Hintergrund, des Zellbereiches. Der Hintergrund selbst wiederum ist ein Objekt. `ColorIndex` ist der Name eines der möglichen Attribute, bzw. eine Eigenschaft des Objektes `Interior`. Der Wert der Eigenschaft ist 8, das entspricht der Farbe Zyan.

In der VBA-Online-Hilfe kann eine grafische Übersicht über die Objekte und ihre hierarchische Ordnung gefunden werden. Z.B. für das Application-Objekt.

Es ist ersichtlich, dass die Objekte hierarchisch geordnet sind. Das oberste Objekt ist die Anwendung (*application*). Sie stellt das Excel-Programm dar. Direkt darunter befinden sich z.B. die Objekte *Workbook* (das ist die Arbeitsmappe), *AddIns*, *AutoCorrect*, usw. Das Objekt *Workbook* wiederum beinhaltet die Objekte *Worksheet* (das ist ein Tabellenblatt), *Chart* (ein Diagrammblatt) und viele mehr.

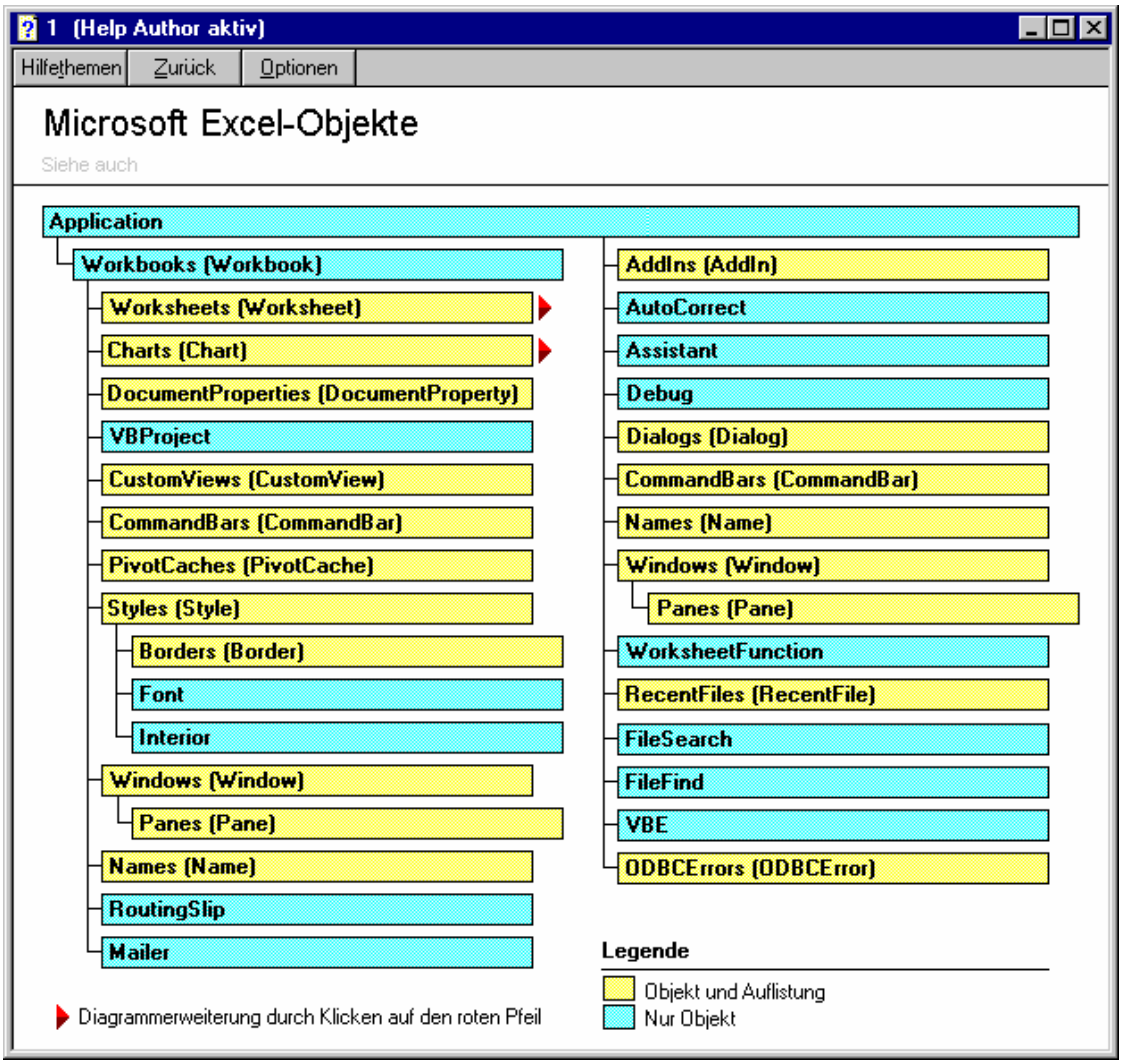
Objekte, die andere Objekte enthalten, werden auch *Container-Objekte* genannt. Um auf eines der unteren Objekte zugreifen zu können, müssen die Namen aller darüberliegenden Objekte genannt werden. Siehe Beispiel oben:

```
Auswahl.Interior.Colorindex = 8
```

Bzw. ohne Vereinfachung:

```
Worksheet("Tabelle1").Range("A1:B2").Interior.Colorindex = 8
```

Excel verfügt über viele verschiedene Container-Objekte. Eine Excel-Mappe beinhaltet z.B. Tabellenblätter, sie ist ein Container für die Blätter. Jedes einzelne Tabellenblatt (*Worksheet)* ist seinerseits ein Container-objekt, z.B. für die im Blatt enthaltenen Grafik- oder Diagrammobjekte. Andererseits ist jedes Tabellenblatt Teil der Menge aller Tabellenblätter einer Mappe. Dafür gibt es das Auflistungs-Objekt *Worksheets.* Auflistungs-Objekte werden immer mit der englischen Mehrzahl des Objektnamens bezeichnet. Für alle Objekte, die über ein Auflistungs-Objekt verfügen, werden inder Online-Übersicht beide Namen angegeben (das einzelne Objekt in Klammern).



1.3 Die Online Hilfe

Die Online Hilfe ist mehr als eine Hilfe in aussichtslosen Situationen. Sie ist ein wichtiges Werkzeug bei der Programmierung und ermöglicht erst die Übersicht und den Einsatz der vielen verschiedenen Objekte. Vor dem Programmieren sollte man sie auf jeden Fall installieren.

1.4 Programme erstellen

Durch Aufzeichnen. Excel besitzt einen sogenannten *Makro-Recorder*. Damit kann man alle Schritte protokollieren, die man in einer interaktiven Excel-Anwendung ausführt. Z.B., eine Zelle markieren, Werte und Formeln eintragen, grafische Objekte erzeugen und manipulieren, etc. Das Ergebnis ist ein VBA-Code, den man ohne weitere Änderungen ausführen kann oder als Ausgangspunkt für eigene Programmentwicklungen verwenden kann. Mit dem Recorder erstellt man Programme ohne Kenntnisse der Programmierung besitzen zu müssen! Aus diesem Grund werden VBA-Programma oft auch als *Makros* bezeichnet. *Mit* dem Recorder können auch die kompliziertesten Objektzugriffe ermittelt werden. Die Recherche in der Online-Hilfe oder Handbüchern entfällt.

Durch Programmieren. Andere wichtige Bestandteile eines richtigen Programmes können nicht durch Aufzeichnen ermittelt werden, sondern müssen "von Hand" programmiert werden. Z.B., Programmverzweigungen, Schleifen oder interaktive Eingabemasken. Aufgezeichnete Makros können von Hand hinsichtlich Speicherplatz und Ausführungsgeschwindigkeit optimiert werden, z.B. durch explizite Typvereinbarungen, die Vereinbarung von Objektvariablen (siehe Beispiel oben) oder vereinfachte Abläufe.

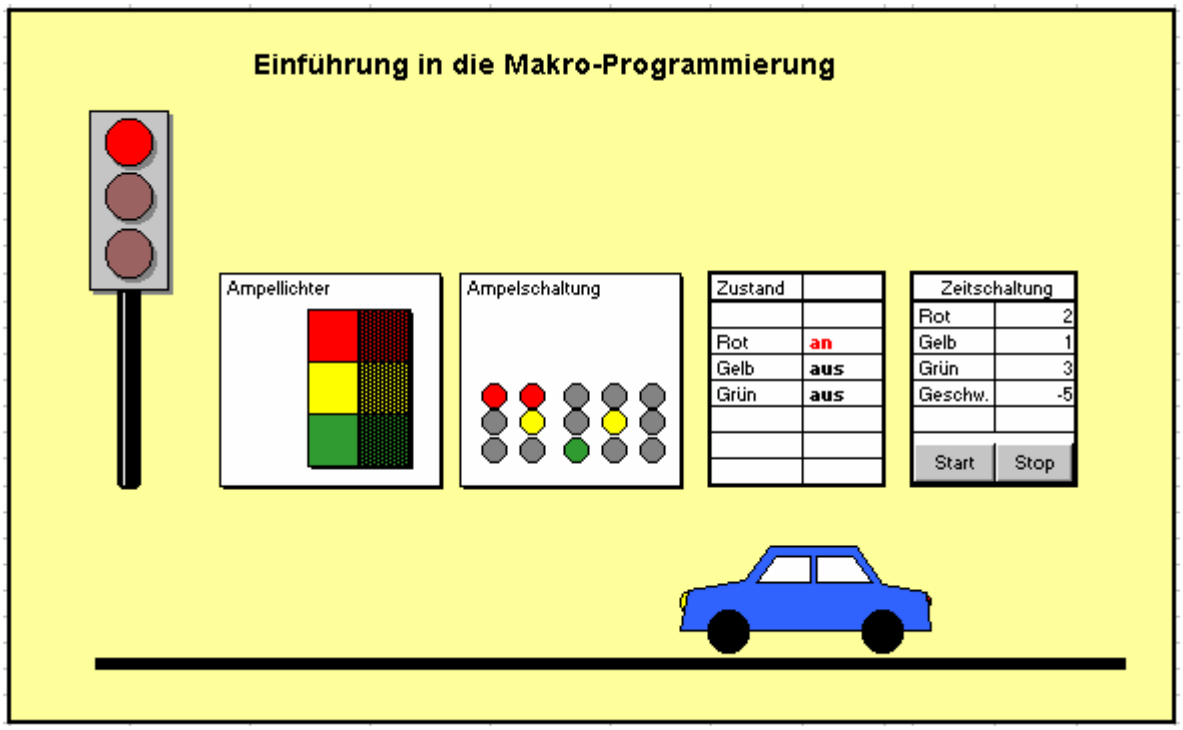
Die Kombination von Aufzeichnen, Programmieren und Online-Hilfe ergibt insgesamt eine sehr effektive Methode zur Erstellung von Programmen für alle denkbaren Anwendungen im Bauingenieurwesen. Das Vorgehen ist intuitiv erlernbar, am besten durch eigenes Üben.

2 Erste Schritte

Anhand eines Beispiels wollen wir die Verwendung des Makro-Recorders kennenlernen. Die aufgezeichneten Makros werden im weiteren zu größeren Programmen weiterentwickelt.

2.1 Beispiel: Ampelschaltung

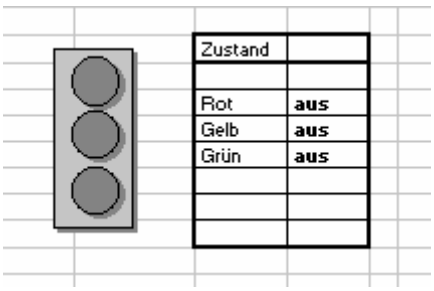
Als Fernziel wollen wir eine Ampelschaltung für ein kleines Auto entwickeln. Bei Rot und Gelb soll es stehen, nur bei Grün darf es fahren.



2.1.1 Makros: Ampellichter an- und ausschalten

Als erstes wollen wir die einzelnen Lichter der Ampel an- und ausschalten.

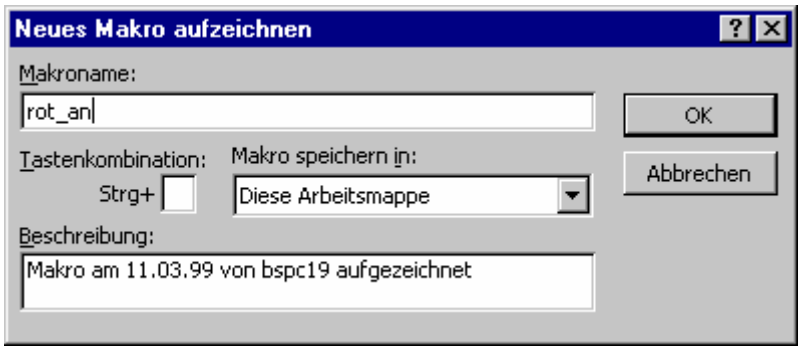
Zunächst wird Excel gestartet. Wir zeichnen eine Ampel aus drei Kreisen und einem Rechteck und erstellen eine Tabelle für den Zustand der Lichter. Auf alles weitere soll vorerst verzichtet werden.



Aufzeichnung beginnen. Das Einschalten des roten Lichtes und der Eintrag "Rot **an**" werden aufgezeichnet.

Makro-Recorder starten. `Extras -> Makro -> Aufzeichnen ...`

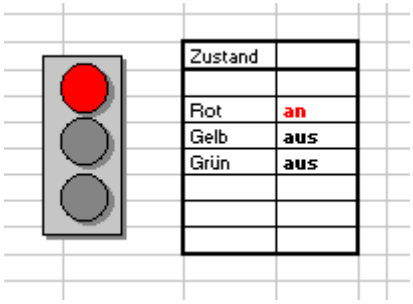
Folgende Maske erscheint, der Name des Makros wird angegeben, hier "rot_an":



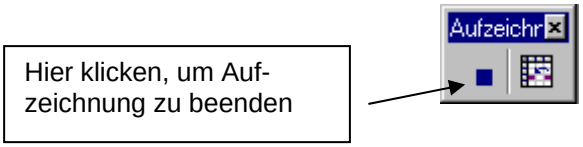
Nach der Bestätigung öffnet sich die Maske, um die Aufzeichnung beenden zu können:

Interaktiv Änderungen vornehmen.

Oberen Kreis auswählen, Füllung nach Rot ändern. Für den Zustand des roten Lichtes "an" eintragen, Schriftstärke **fett** und -farbe **rot** setzen.



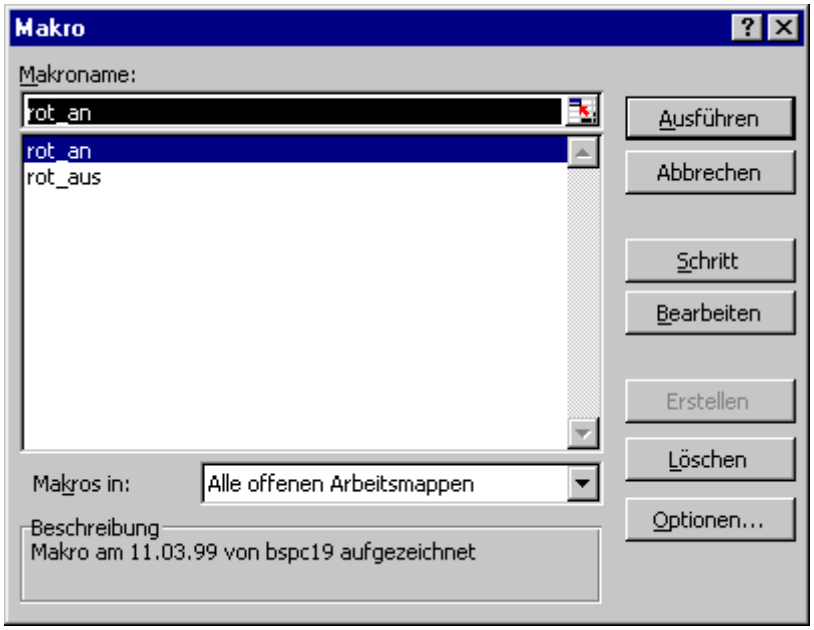
Makro beenden. Auf das blaue Rechteck der Aufzeichnungsmaske klicken.



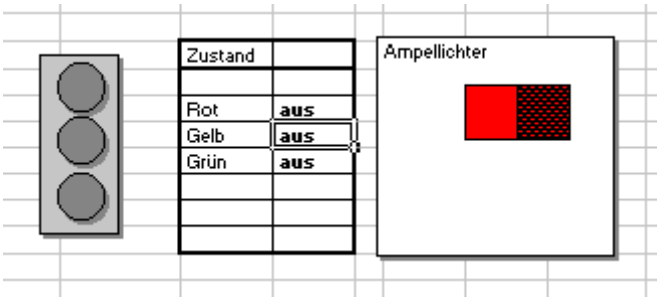
Aufzeichnung wiederholen. Makro "rot_aus". Das Farbe des roten Lichts auf "Grau 50%" setzen, Zustand **aus**.

Makros ausführen. Menü `Extras -> Makro -> Makros` oder Alt F8

Licht an- und ausschalten mit dem entsprechenden Makro.



Makro zuweisen. Die Ausführung über die Makroliste ist mühsam und wenig intuitiv. Makros können deshalb verschiedenen Schaltflächen oder Knöpfen auf der Steuerleiste zugewiesen werden. Wir wollen "rot_an" einem roten Quadrat und "rot_aus" einem grauen Quadrat zuweisen. Deshalb zunächst diese beiden Rechtecke zeichnen. Das rote Quadrat auswählen und über das Kontextmenü (rechte Maustaste) `Makro zuweisen ...` auswählen. Es öffnet sich eine Makroliste, ähnlich der oben dargestellten. "rot_an" auswählen und Eingabe bestätigen. Entsprechend mit dem grauen Quadrat für "rot_aus" verfahren. Nachdem noch ein Rahmen und Text hinzugefügt wurde, stellt sich das Tabellenblatt so ähnlich dar:



Die beiden Quadrate sind nun Schaltflächen, um das rote Licht an- und auszuschalten. Beim Überfahren mit dem Cursor ändert der sich zu einer Hand mit Zeigefinger. Klickt man eine Fläche an, ändert sich das Licht entsprechend.

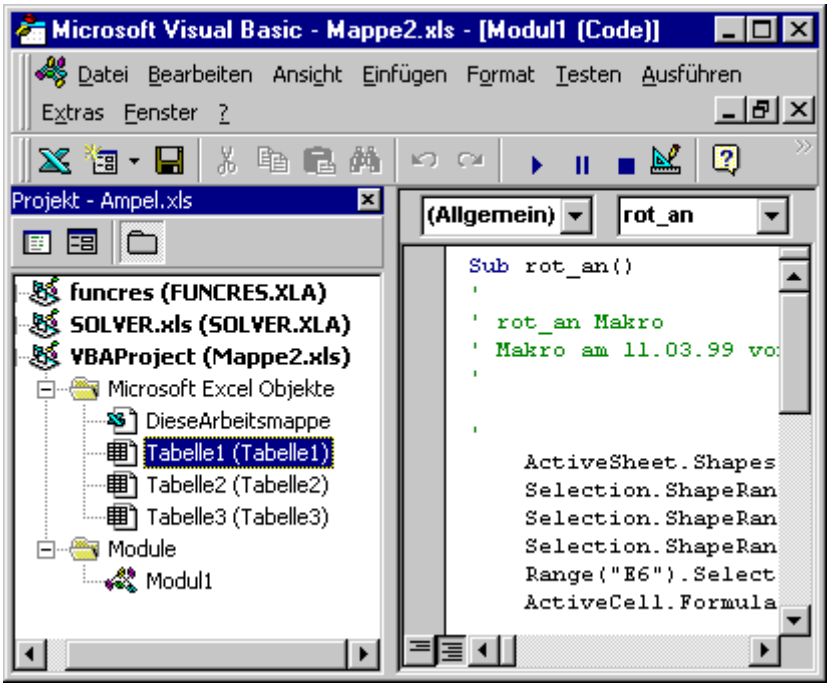
Übung. Erstellen Sie weitere Makros und Schaltflächen für das gelbe und das grüne Licht.

2.1.2 Der generierte VBA-Code

Beim Aufzeichnen wurde im verborgenen VBA-Code erzeugt. Im Visual Basic Editor (*VBE*) kann er angeschaut und weiter modifiziert werden. VBE ist ein eigenständiges Programm, das unabhängig von Excel läuft.

VBE starten. `Extras -> Makro -> Visual Basic Editor` oder Alt F11

Der Visual Basic Editor öffnet sich in einem neuen Fenster. Im linken Teil ist der *Project Explorer* sichtbar. Es ist das Excel-Projekt mit allen seinen Elementen dargestellt. Es sind dies die Mappe selbst, drei Tabellenblätter sowie unter *Modules* ein Programm-Modul "Modul1", in dem der generierte Code gespeichert wurde.



Code anzeigen. "Modul1" auswählen und `Ansicht -> Code` bzw. F7 oder Doppelklick auf "Modul1". Es öffnet sich im rechten Teil des VBE der Programmeditor mit dem Programm-Code. Im Programmeditor kann der Programm-Code auch bearbeitet werden. Das Makro "rot_an" müsste ungefähr folgendermaßen lauten:

```
Sub rot_an()

' rot_an Makro
' Makro am 11.03.99 von bspc19 aufgezeichnet
'
    ActiveSheet.Shapes("Oval 1").Select
    Selection.ShapeRange.Fill.ForeColor.SchemeColor = 10
    Selection.ShapeRange.Fill.Visible = msoTrue
    Selection.ShapeRange.Fill.Solid
    Range("E6").Select
    ActiveCell.FormulaR1C1 = "an"
    Range("E6").Select
    Selection.Font.ColorIndex = 3

End Sub
```

Offensichtlich besteht der Code ausschließlich aus einer Abfolge von Befehlen, die Objekte-Methoden anwenden (z.B. `Range("E6").Select`) oder Objekt-Eigenschaften modifizieren (z.B. `Selection.Font.ColorIndex = 3`). Ohne weiteres Zutun ist ein objektorientierter Code entstanden, der durch die Ereignisse "Schaltflächen anklicken" gesteuert wird.

Die Befehle bedeuten im einzelnen:

```
ActiveSheet.Shapes("Oval 1").Select                    Kreis "Ellipse 1" auswählen
Selection.ShapeRange.Fill.ForeColor.SchemeColor = 10   Füllfarbe rot
Selection.ShapeRange.Fill.Visible = msoTrue            Füllung sichtbar
Selection.ShapeRange.Fill.Solid                        Füllung monochrom
Range("E6").Select                                     Zelle E6 auswählen
ActiveCell.FormulaR1C1 = "an"                          Text "an" eintragen
Range("E6").Select                                     Zelle E6 auswählen
Selection.Font.ColorIndex = 3                          Textfarbe rot
```

2.1.3 weitere Prozeduren erstellen

Eine Ampelschaltung besteht aus definierten Lichtkombinationen. Dazu müssen gleichzeitig Lichter an- und ausgeschaltet werden:

Schaltung	rot	gelb	grün
Halt	an	aus	aus
AchtungFahrt	an	an	aus
Fahrt	aus	aus	an
AchtungHalt	aus	an	aus
AmpelAus	aus	aus	aus

Die einzelnen Ampelschaltungen können aus Kombinationen der "Lichter-Makros" zusammengesetzt werden. Dazu werden im Code-Fenster am Ende weitere Prozeduren angefügt, indem man den Text der Befehle über die Tastatur eingibt. Zum Beispiel für die Prozedur "Halt" (Die entsprechenden Makros für das gelbe und das grüne Licht wurden selbstverständlich vorher schon erzeugt.):

```
Sub Halt()
    ' Ampelschaltung "Halt"
    ' rot = an, gelb = aus, grün = aus
    rot_an
    gelb_aus
    grün_aus
End Sub
```

Ein Prozedur beginnt mit dem Schlüsselwort "Sub". Danach folgt der Name, z.B. "Halt", und dann die Liste der Übergabeparameter, eingeschlossen in Klammern. In unserem Falle werden keine Parameter übergeben, deshalb sind die Klammern leer. Die Prozedur wird mit "End Sub" beendet.

Der VBE-Editor verfügt über eine automatische Syntaxüberprüfung. D.h., schon bei der Eingabe wird der Code auf Fehler überprüft, wird die Syntax ergänzt oder erscheinen automatisch Hinweisfenster für die richtige Schreibweise. Besonders hilfreich ist, dass die Schreibweise bekannter Variablen und Schlüsselwörter nach der Eingabe angepasst wird. Wird z.B. "sub" mit Kleinbuchstaben eingegeben, wird automatisch das "s" in ein "S" umgewandelt. Zusätzlich erscheinen Schlüsselwörter blau. Ändern sich bei Eingabe Buchstabenschreibweise bzw. Farbe nicht, hat man sich vertippt.

Kommentar beginnt mit einem einfachen Hochkomma ' und erscheint grün. Syntaxfehler werden rot angezeigt.

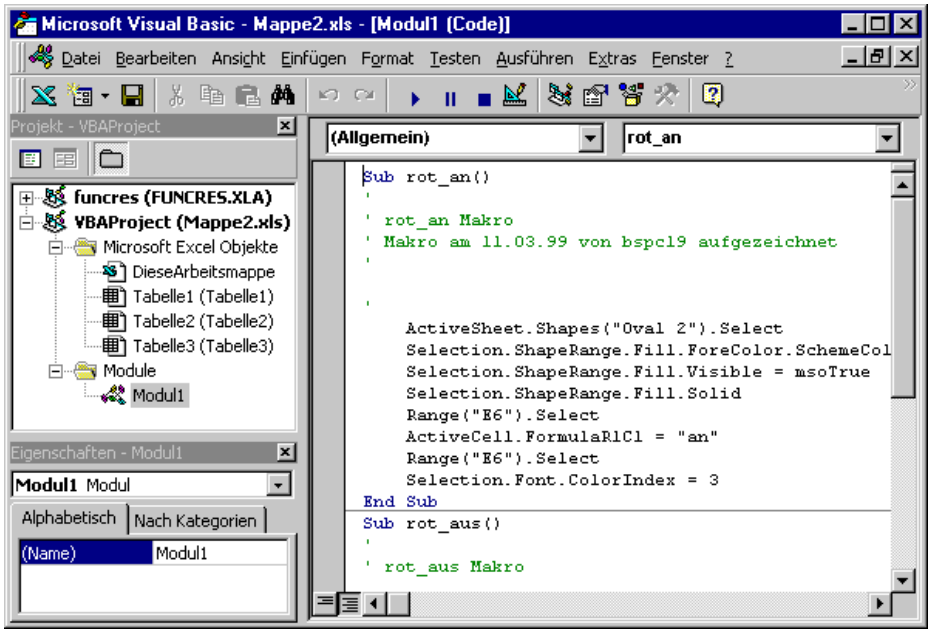
Insbesondere bei längeren Programmen ist es wichtig, sich einer strengen Disziplin zu unterwerfen und den Programmtext auch formal durch Einrückungen, Kommentare und Leerzeilen zu gliedern, um die Lesbarkeit zu verbessern. Das lohnt sich spätestens dann, wenn man logische Fehler suchen muß. Das wird sehr bald der Fall sein ... Aus diesem Grund wird der eigentliche Programmtext zwischen "Sub" und "End Sub" eingerückt.

Aufgabe. Erstellen Sie die Prozeduren "AchtungFahrt", "Fahrt", "AchtungHalt" und "AmpelAus". Erzeugen Sie Schaltflächen und weisen Sie diesen die neuen Prozeduren zu. Für die Gestaltung bietet es sich an, Grafiken zu erstellen, die die entsprechende Ampelschaltung darstellen.

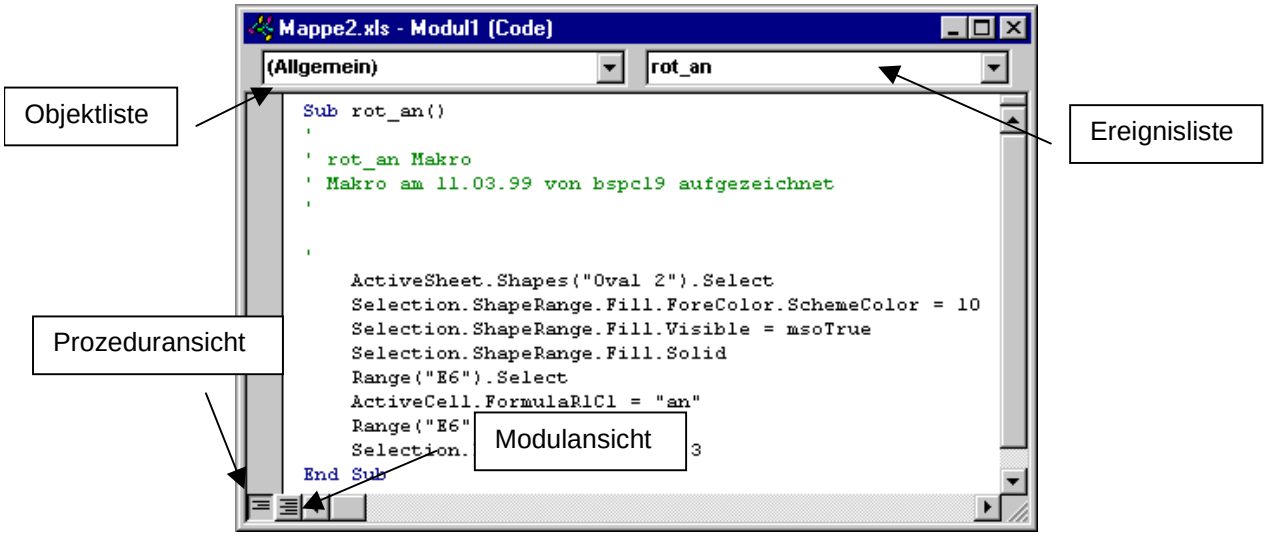
2.2 Die Programmierumgebung

Die Programmierumgebung *Visual Basic Editor (VBE)* wurde bereits im Ampel-Beispiel erwähnt. Sie ist eine einheitliche Umgebung für Excel, PowerPoint und Word. Bei der Bearbeitung von Makros dieser Office-Komponenten gibt es nahezu keine Unterschiede mehr. Eine Ausnahme ist Access. Die Programmiersprache VBA ist für alle gleich. Nach dem Aufruf von VBE präsentiert sich eine eigene Anwendung, in der die gesamte Entwicklung einschließlich des Formularentwurfs (Entwicklung eigener Eingabe-Masken) stattfindet. Neben dem Menü und der Symbolleiste gibt es drei unterschiedliche Fenster.

> VBE wird für jedes Programm einzeln aufgerufen. Jeder zusätzliche Aufruf erzeugt eine eigene Kopie und belegt zusätzlichen Speicherplatz. Mögliche Folgen: Systemüberlastung, verminderte Ausführungsgeschwindigkeit.
> Alle Fenster lassen sich entkoppeln und können unabhängig positioniert werden.
> Alle Fenster lassen sich aus- und einblenden.



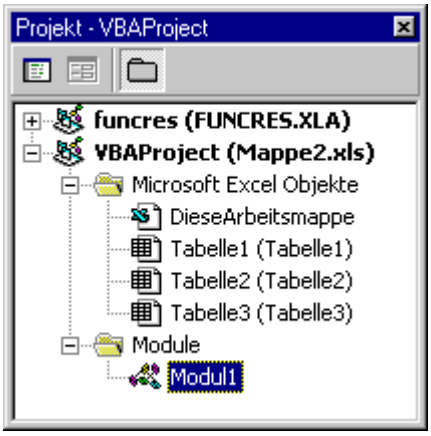
2.2.1 Programmeditor



> Es können mehrere Fenster geöffnet werden.
> Syntaxprüfung
> Objekt- und Ereignisliste

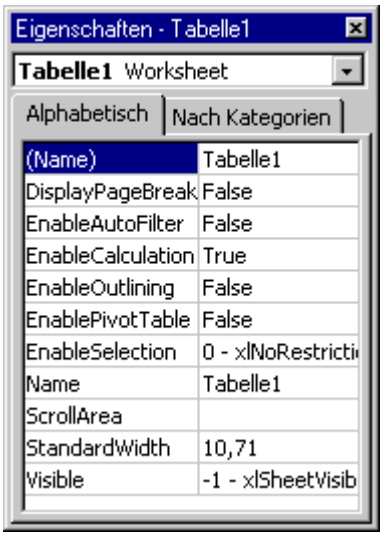
> Prozedur- und Gesamtansicht

2.2.2 Projektexplorer



> Aufruf: `Ansicht -> Projekt-Explorer`; Strg+R; Schaltfläche **Projekt-Explorer**
> Projekte werden in Form eines Verzeichnisbaumes angezeigt.
> Ein Projekt besteht verschiedenen Elementen, z.B.: Arbeitsmappe, Tabellenblätter, Module, Klassenmodule, Verweise, Formulare

2.2.3 Eigenschaftsfenster



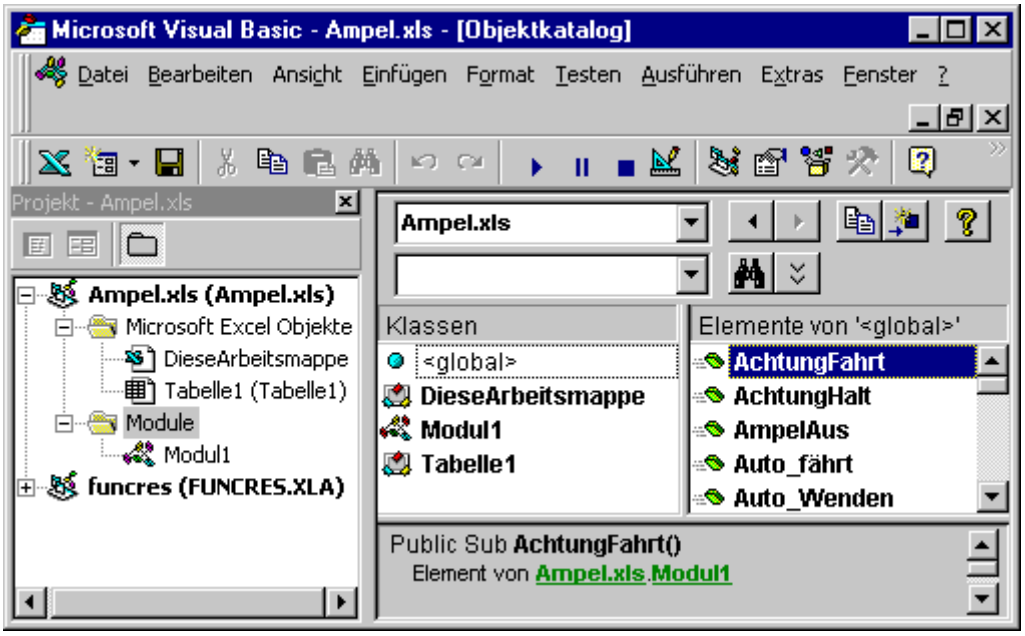
> Aufruf: `Ansicht -> Eigenschaftsfenster`; F4; Schaltfläche **Eigenschaftsfenster**
> Objekte besitzen Attribute oder Eigenschaften. Das Eigenschaftsfenster dient der Ansicht und Modifikation derselben.
> **Beispiel.** Eigenschaftsblatt des Arbeitsblattes "Tabelle1"
> Ordnung nach Kategorien oder alphabetisch
> Das Eigenschaftsfenster ist besonders bei der Entwicklung eigener Formulare und Masken von Bedeutung.

2.2.4 Objektkatalog

Der Objektkatalog dient der Anzeige aller verfügbaren Objekte, die in einem Projekt vorhanden sind, mit allen Eigenschaften, Methoden und programmierbaren Ereignissen. Der Objektkatalog hilft den Überblick zu bewahren und schnell die richtige Online-Hilfe zu finden.

Der Objektkatalog besitzt ein Listenfeld, das beim Öffnen den Eintrag "<Alle Bibliotheken>" anzeigt. Um die Objekte des eigenen Projektes anzuzeigen, muß aus der Liste entweder "VBAProject" oder

der Name des Excel-Projekts ausgewählt werden. Hier wurde das Ampel-Beispiel im Projekt "Ampel.xls" ausgewählt.



Im rechten Teil des Objektkataloges erscheint die Auflistung aller Eigenschaften und Methoden, die zum markierten Objekt des linken Teils gehören. Im Beispiel oben wurde "global" ausgewählt, d.h. sämtliche Eigenschaften, Methoden und Ereignisse der gewählten Kategorie, also "Ampel.xls" angezeigt. Man erkennt die Methoden "AchtungFahrt", "AchtungHalt", "AmpelAus", "Auto_fährt" und "Auto_Wenden". Im unteren Teil des Objektkatalog wird eine Kurzinformation zur ausgewählten Methode angezeigt mit weiteren Verweisen auf die Online-Hilfe.

> Aufruf: `Ansicht -> Objektkatalog`; F2; Schaltfläche **Objektkatalog**
> Übersicht über alle Objekte, deren Methoden, Eigenschaften und Ereignisse des gesamten Projekts; VBA-Objekte und selbst definierte
> Doppelklick auf Methodenname öffnet den Projekteditor
> Online-Hilfe für ausgewählte Methode etc. über das gelbe Fragezeichen des Objektkataloges
> Suchfunktion

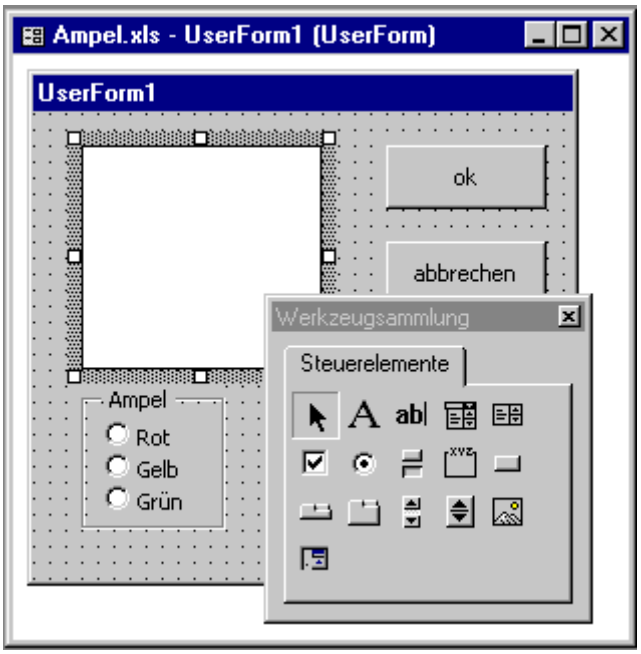
2.2.5 Formularfenster

Benutzerdefinierte Dialoge sind Fenster, die der interaktiven Programmsteuerung dienen. Sie verfügen über Schaltflächen, Optionen, Listen, etc. Eine einfache Form von Schaltflächen ohne Dialog wurde bereits im Ampel-Beispiel gezeigt. Dialoge sind eigene Objekte und Elemente des bearbeiteten Projektes.

Ein Dialog wird entweder über das Menü `Einfügen -> Userform` oder über die entsprechende Schaltfläche der Steuerleiste eingefügt. Das neue Dialogfeld ist leer. Die *Steuerelemente* müssen noch hinzugefügt und editiert werden.

Die Steuerelemente entnimmt man der *Werkzeugsammlung* (Toolbox). Aufruf der Toolbox: `Ansicht -> Werkzeugsammlung` oder über die Schaltfläche **Werkzeugsammlung**. Jedes Element der Werkzeugsammlung wird durch Drag & Drop in das neue Formular übernommen. Die Eigenschaften werden im Eigenschaftsfenster an die speziellen Bedürfnisse angepasst.

> Formulare werden durch ein eigenes Objekt *UserForm* repräsentiert
> Formulare werden durch Steuerelemente gestaltet
> Steuerelemente werden durch Drag&Drop eingefügt
> Steuerelemente werden mit dem Eigenschaftsfenster verändert
> Aufruf: `Einfügen -> Userform`; Schaltfläche **Userform einfügen**

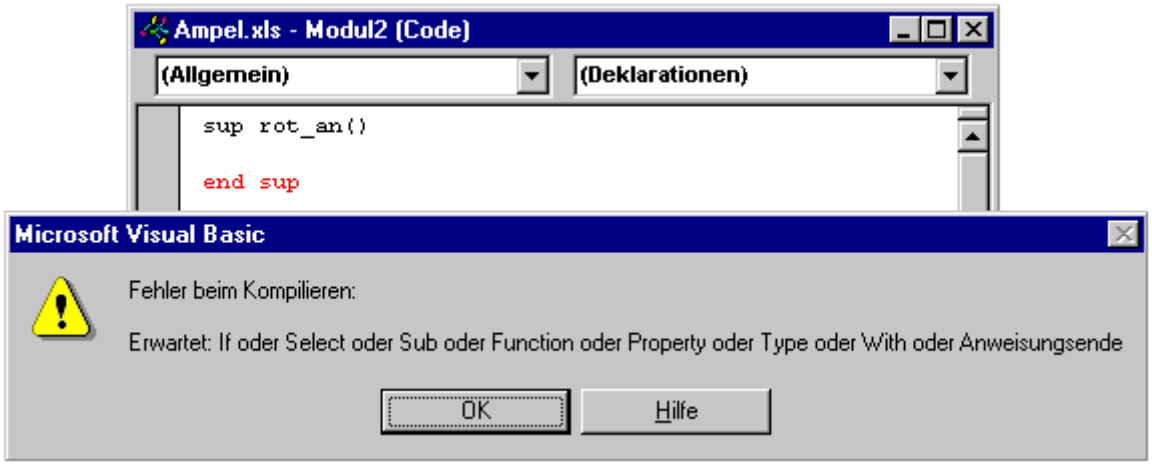


Das Formularfenster mit Werkzeugsammlung

2.2.6 Hilfsmittel zum Testen von Programmen

Fehlerfreie Programme gibt es nicht! Man unterscheidet *syntaktische*, *logische (semanische) Fehler* und *Benutzerfehler*.

Syntaktische Fehler. Sie entstehen, wenn die Syntax, d.h. die Sprachregeln, der Programmiersprache verletzt werden. Ein falsch geschriebenes Schlüsselwort ist z.B. ein Syntaxfehler (*Sup* statt *Sub*). Solche Fehler werden schon bei der Eingabe erkannt. In jedem Fall werden sie rot dargestellt. Bei der Standardinstallation wird sofort eine Warnung angegeben:



Den Vorgang der Syntaxüberprüfung nennt man *Kompilieren*. Ein VBA-Programm wird immer vor der Ausführung automatisch kompiliert. Man kann den Vorgang aber auch "trocken" auslösen, durch den Menüpunkt `Testen -> Kompilieren von VBA-Projekt`.

Logische (semantische) Fehler. Diese Fehler entstehen aufgrund von Denkfehlern der Programmierer. Diese Fehler können nicht vom Kompiler entdeckt werden. Logische Fehler müssen oft in mühsamer Detektivarbeit aufgespürt werden. Oft muss das Programm Zeile für Zeile nachvollzogen werden.

Benutzerfehler. Entstehen durch falsche Anwendung des Programmes. Dagegen kann man nur durch entsprechenden Programmcode Vorkehrungen treffen (z.B., kontrollierter Programmabbruch).

Debugger. Testhilfen für das Aufspüren logischer Fehler nennt man Debugger (amerik. *Bug* = Fehler). Der Vorgang der Fehlersuche heisst dementsprechend *Debugging*. VBA verfügt über verschiedene solcher Hilfen. Grundsätzlich dienen sie dazu, Eigenschaften und Zustände der Programmvariablen abzufragen und anzuzeigen, um somit in der Lage zu sein, den Fehler zu finden.

Programm mit dem Debugger testen. Meist hat man eine Vorstellung davon, wo im Programm sich der Fehler befinden könnte. Um den Zustand überprüfen zu können, muß das Programm an dieser Stelle angehalten werden.

Haltepunkt setzen. Ein Haltepunkt wird in einer Zeile gesetzt, in dem man den Cursor in dieser Zeile positioniert und entweder

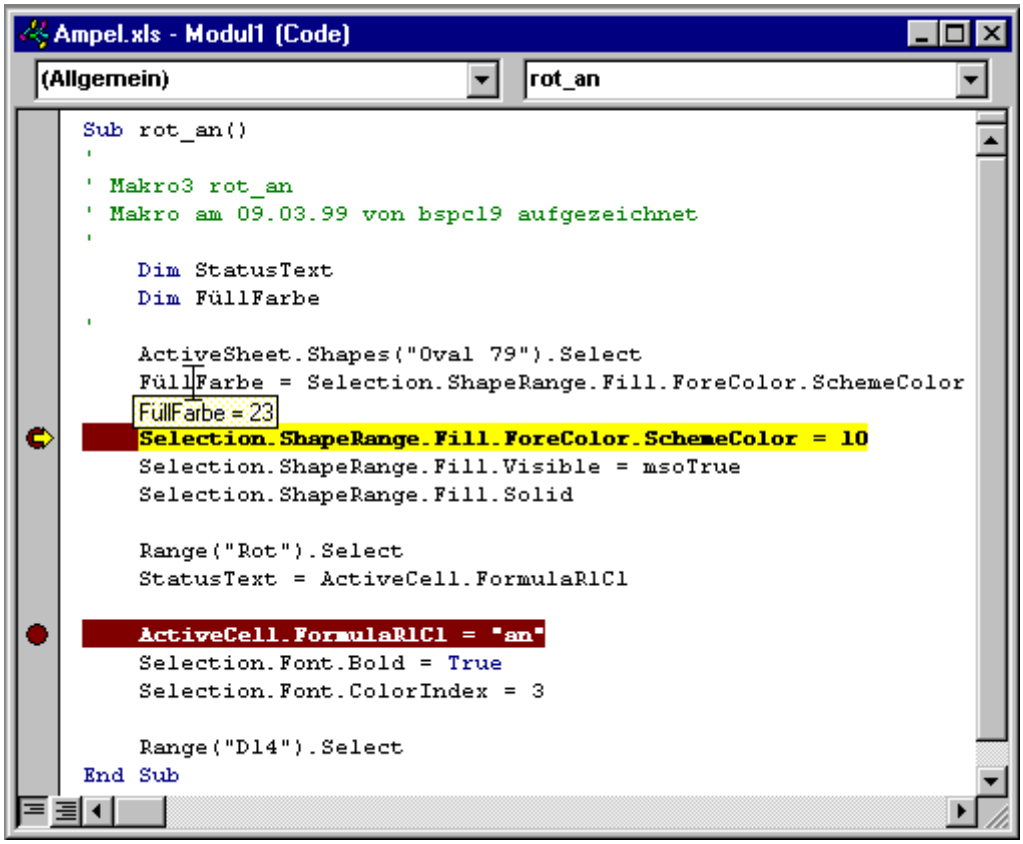
> `Testen -> Haltepunkt Ein/Aus` wählt
> F9 drückt oder
> den Rahmen des Code-Fensters vor der betreffenden Zeile anklickt. Ein weiterer Klick schaltet den Haltepunkt wieder aus.

Eine Prozedur wird in einer Zeile grundsätzlich **vor** der Ausführung dieser Zeile angehalten.

Prozedur probeweise starten. Egal auf welche Weise das Programm gestartet wird, solange ein Haltepunkt gesetzt ist, wird es an dieser Stelle anhalten und auf weitere Anweisungen warten. Eine einzelne Methode bzw. Prozedur kann für sich getestet werden. Man startet sie indem man

> `Ausführen -> Sub/UserForm Ausführen` wählt
> F5 drückt oder
> die Schaltfläche **Sub/UserForm ausführen** drückt.

Beispiel. Der Prozedur "rot_an" wurden noch ein paar Zeilen hinzugefügt, um das Debugging darstellen zu können. Es wurde die Variable *FüllFarbe* definiert, um an geeigneter Stelle die vorhandene Farbe zuzuweisen. In der folgenden Zeile wurde eine Haltepunkt gesetzt und die Prozedur gestartet. An einer anderen Stelle wird ebenfalls ein Haltepunkt gesetzt, um den Status Text des Lichtes (an/aus) abzufragen. Das Programm hält an der angegebenen Stelle und die betreffende Zeile wird gelb hinterlegt. Im Rahmen des Programm-Editors wird ein gelber Pfeil eingeblendet.



Variablen abfragen. Methode 1: mit dem Cursor über die betreffende Variable fahren. Es wird im QuickInfo-Feld der aktuelle Stand angezeigt. Im Beispiel hat die Füllfarbe die Nummer 23. Das entspricht 50%-Grau und stellt das ausgeschaltete Licht dar. Offensichtlich hält der Debugger das Pro-

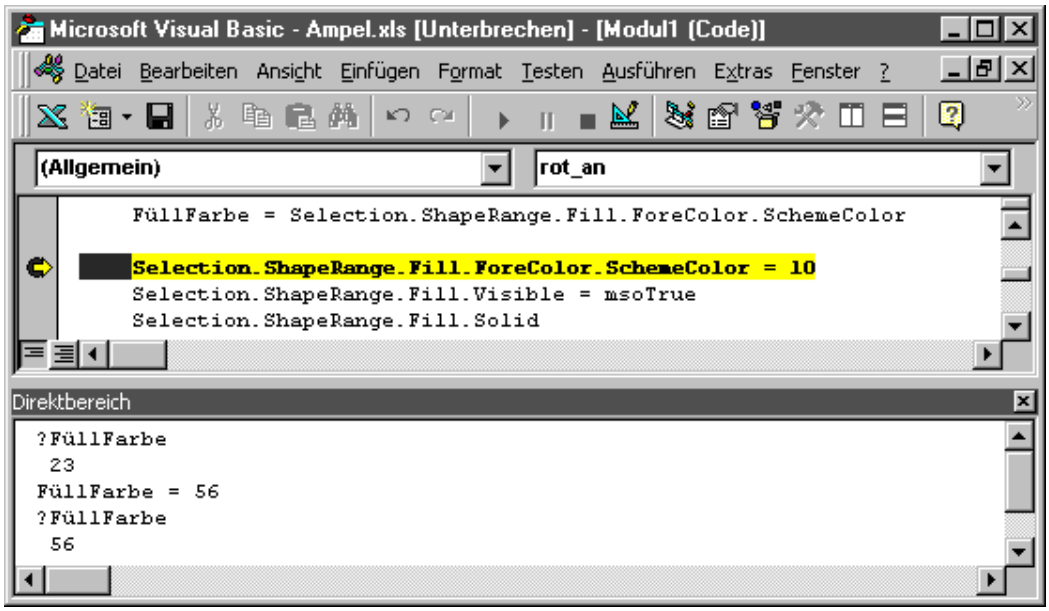
gramm in der Zeile vor der Ausführung an. Im nächsten Schritt wird die Farbe auf die Nummer 10 (rot) gesetzt.

Das Direktfenster. Im Direktfenster können Debug-Befehle direkt eingegeben werden, z.B. Variablen abfragen und Variablen neue Werte zuweisen. Das ist dann sinnvoll, wenn man den Fehler erkannt hat, das Programm in korrigierter Form gleich weiter laufen zu lassen, um evtl. weitere Fehler zu entdecken. Direktfenster öffnen:

> `Ansicht -> Direktfenster` oder
> Strg+G

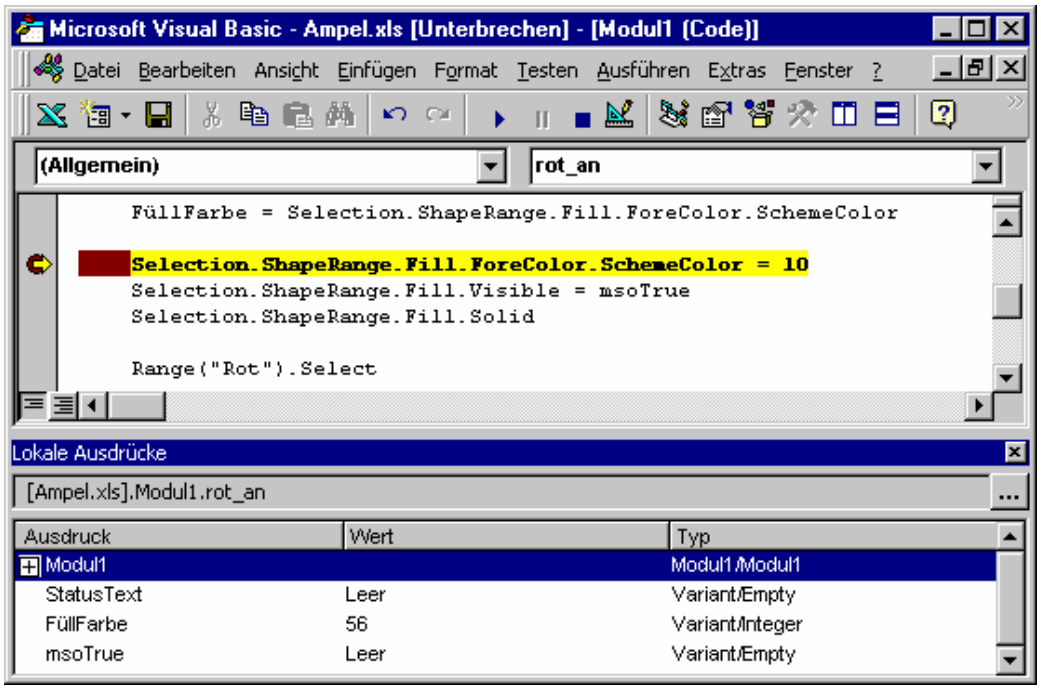
In der Regel wird es am unteren Ende der von VBE erscheinen.

Beispiel. Abfragen und Setzen der Variable `FüllFarbe` im Direktfenster.



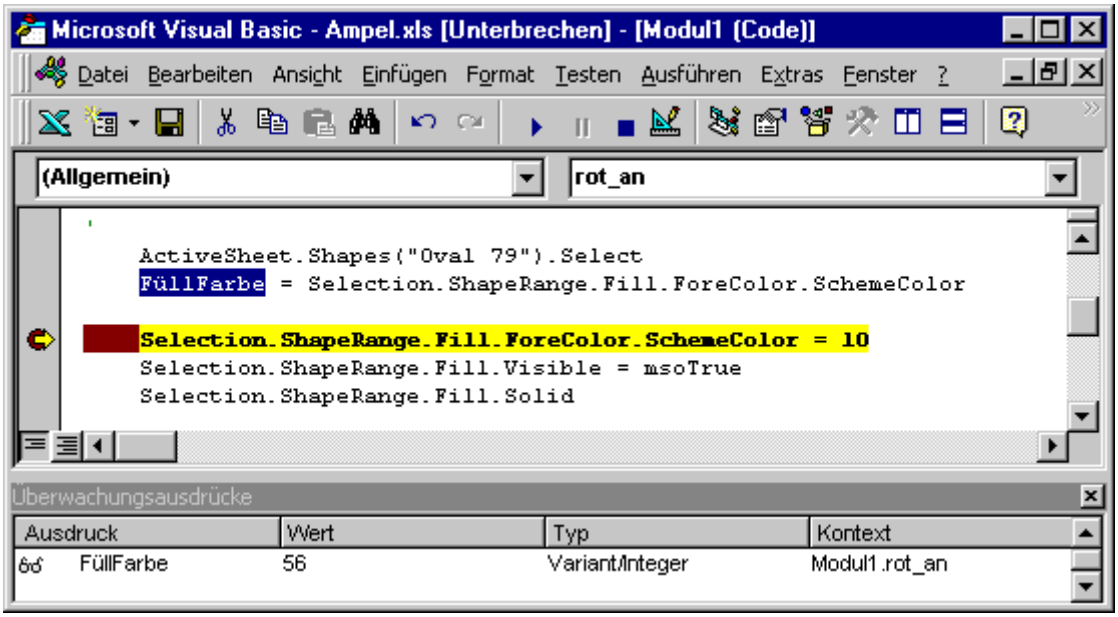
Das Lokalfenster. Das Lokalfenster zeigt für alle deklarierten Variablen der Prozedur Wert und Typ an. Der Typ von Variablen wird in einem späteren Kapitel besprochen. Das Lokalfenster ist für viele Variablen weniger geeignet. Aufruf:

> `Ansicht -> Lokalfenster`



Das Überwachungsfenster. Wie das Lokalfenster, nur für ausgesuchte Variablen.

> Aufruf: `Ansicht -> Überwachungsfenster`
> Variable hinzufügen: Cursor über Variable führen, rechte Maustaste `Überwachung hinzufügen`



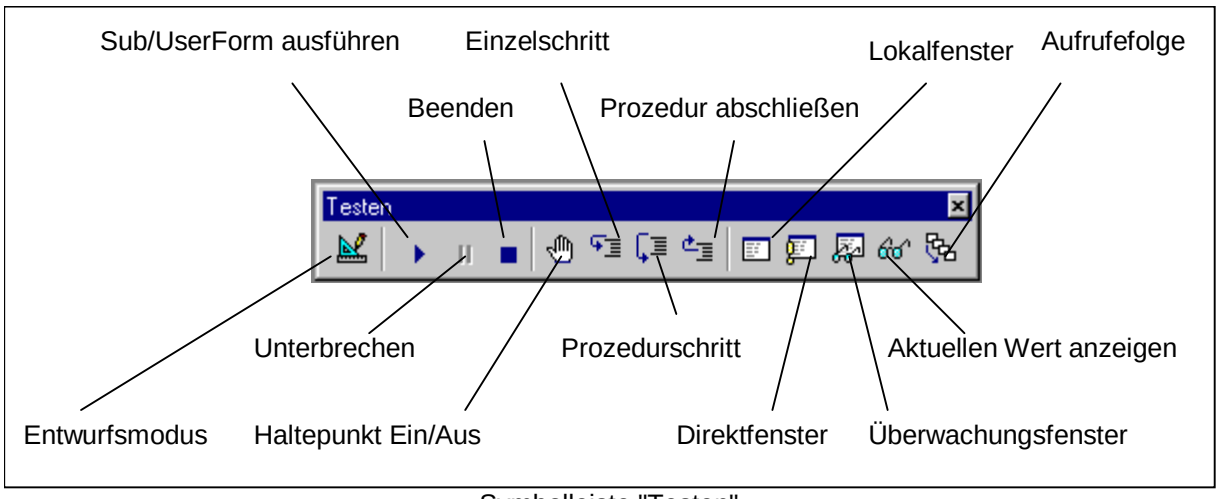
Programm fortsetzen. Am Haltepunkt wird das Programm fortgesetzt mit

> `Ausführen -> fortsetzen`
> F5 oder
> Auswählen der entsprechenden Schaltfläche

Einzelschritte. Das Programm kann Zeile für Zeile ausgeführt werden. Man unterscheidet:

> **Einzelschritt:** `Testen -> Einzelschritt` oder F8; das Programm geht zur nächsten Anweisung und verzweigt dabei auch in die aufgerufenen Prozeduren.
> **Pozedurschritt:** `Testen -> Prozedurschritt` oder Shift+F8; das Programm geht zur nächsten Befehlszeile der aktuellen Prozedur, es verzweigt **nicht** in Unterprozeduren.

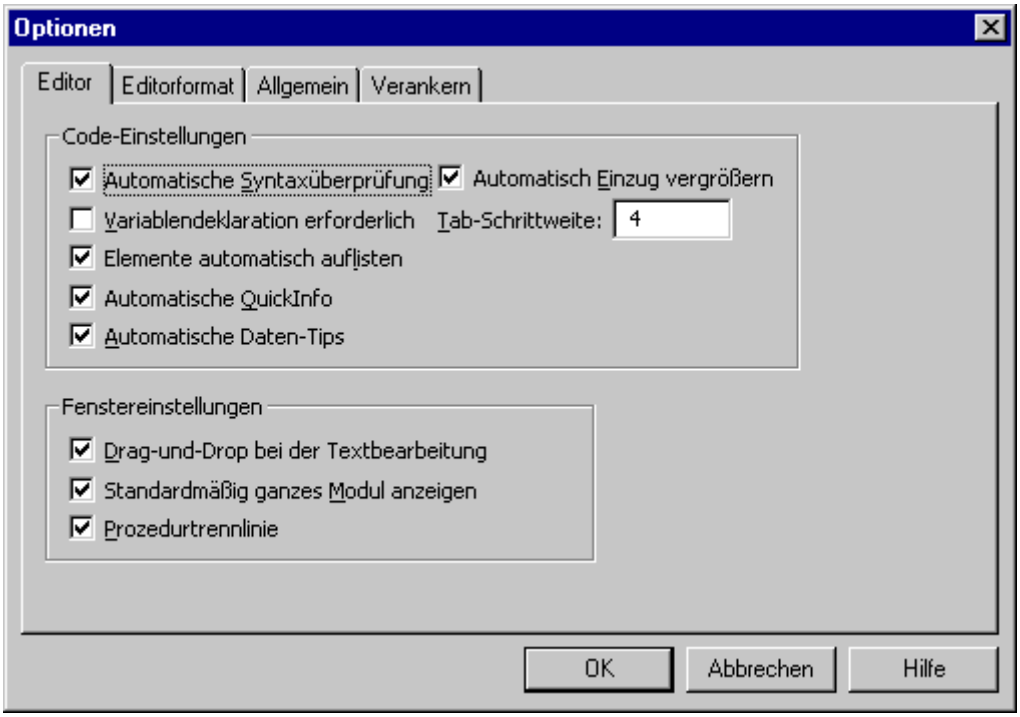
Symbolleiste "Testen". Die wichtigsten Debug-Befehle sind in der Symbolleiste "Testen" zusammengefasst. Aufruf: `Ansicht -> Symbolleisten -> Testen`



Symbolleiste "Testen"

Weitere Symbolleisten. Unterstützung beim
> Programmieren: Symbolleiste `Bearbeiten`
> Entwurf von Formularen: Symbolleiste `UserForm`

Optionen. Weitere VBE-Einstellungen können unter `Extras -> Optionen` eingestellt werden.



3 VBA-Sprachelemente

3.1 Modulare Programmierung

Im Ampel-Beispiel des zweiten Kapitels wurden bereits die Grundsätze der *Modularen Programmierung* angewendet, ohne sie explizit zu erwähnen. Wir haben bereits gesehen, dass die Methoden "Licht an/ Licht aus" für die Farben rot, gelb, grün - das sind bereits sechs kleine Prozeduren - für die verschiedenen Ampelschaltungen in unterschiedlichen Kombinationen eingesetzt werden. Ganz selbstverständlich läßt sich die Aufgabe der Ampelschaltung auf einfache Grundfunktionen, den *Modulen* eines modularen Programmes, zurückführen. Dieses Vorgehen lässt sich verallgemeinern. Obwohl es zunächst aufwendig ist, lohnt es sich, da kleine Programmbausteine übersichtlicher aufgebaut werden können als große, deshalb grundsätzlich weniger fehleranfällig sind und besser erweitert werden können. Unter dem Strich ist man schneller und besser.

- Aufgabe vor dem Programmieren analysieren und Ablauf strukturieren
- Teilaufgaben (*Module*) definieren, die wiederholt eingesetzt werden können
- übersichtlich programmieren (einrücken, Kommentare)

3.2 Prozeduren und Funktionen

In VBA gibt es zwei unterschiedliche Arten von Teilprogrammen: *Prozeduren* und *Funktionen*. Sie unterscheiden sich darin, ob sie einen Wert zurückgeben oder nicht. In C und C++ gibt es übrigens nur Funktionen. Eine Prozedur ist dort als Funktion definiert, die den Wert *void*, d.h. "nichts", zurückgibt.

3.2.1 Prozeduren

Eine Prozedur beginnt mit dem Schlüsselwort Sub. Bislang wurden in den Beispielen ausschließlich Prozeduren verwendet. Die syntaktische Verwandtschaft mit FORTRAN fällt auf.

Syntax	

`[Private|Public][Static] Sub` *Name* `(`*Parameterliste*`)`
`    [`*Anweisungen*`]`
`    [Exit Sub]`
`    [Anweisungen]`
`End Sub`	

Parameter	Verwendung
	Die Erklärung der Schlüsselwörter siehe hinten, z.B. bei Variable

Vergleich.

FORTRAN: `Subroutine(` *Parameterliste* `)`
`[`*Anweisungen*`]`
`Return`
`End`

C/C++: `void` *Name* `(` *Parameterliste* `){` *Anweisungen* `}`

Aufruf einer Prozedur. Beispiel.

```
Sub rot_schalten (AnAus)
    If AnAus = 1 Then
        rot_an              ' Aufruf der Prozedur
    Else
        rot_aus             ' Aufruf der Prozedur
    End If
End Sub

Sub Halt()
    gelb_aus
    grün_aus
    Call rot_schalten ( 1 )  'Aufruf der Prozedur, vgl. FORTRAN
```

```
    Rot_schalten 1            'Aufruf der Prozedur
End Sub
```

3.2.2 Funktionen

Funktionen geben einen Wert zurück. Der Wert wird einer Variablen mit dem Namen der Funktion zugewiesen.

Syntax

`[Public|Private][Static] Function` *FktName* `(`*Parameterliste*`) [As Typ]`
`    [`*Anweisungen*`]`
`    FktName = Ergebnisausdruck`
`End Function`

Vergleich.

FORTRAN: [Typ] Function *FktName* (*Parameterliste*)
[Anweisungen]
FktName = Ergebnisausdruck
Return
End

C: Typ function *FktName* (*Parameterliste*)
{ ...
return(Ergebniswert);
}

Beispiel.

```
Function StatusRot() As Integer
    ' gibt die aktuelle Farbe des Rotlichtes zurück
    ActiveSheet.Shapes("Oval 79").Select
    StatusRot = Selection.ShapeRange.Fill.ForeColor.SchemeColor
End Function

Sub TestRot()
    Dim Farbe As Integer
    Farbe = StatusRot      ' Funktionsaufruf rechts vom Komma
End Sub
```

Zusätzlich wurde hier der Datentyp `Integer` verwendet. Siehe dazu weiter unten.

3.2.3 Parameter

Einfache Parameter.

Syntax
`Sub` *Name* `(`*Para1* `[As Typ],` *Para2* `[As Typ], ...)`

Parameter werden durch Komma getrennt. Der Datentyp kann explizit vereinbart werden. Beim Aufruf muß auf die Identität der Datentypen geachtet werden. Die Anzahl der Parameter muß bei jedem Aufruf der Definition entsprechen.

Optionale Parameter.

Syntax
`Sub` *Name* `(`*Para1* `[As Typ], Optional` *Para2* `[As Typ], ...)`

Optionale Parameter können beim Aufruf weggelassen werden. Sie müssen am Ende der Parameterliste vereinbart werden. Folgende Aufrufe sind für das Beispiel möglich:

```
Name 100
Name 100, 50
```

Parameter-Array.

Syntax
`Sub` *Name* `(Para1 [As Typ], ParamArray` *arg()* `As Variant)`

Ein Parameterfeld variabler Länge. Ein Parameter-Array muss vom Typ `Variant` sein. Parameter-Arrays müssen am Ende der Parameterliste stehen. Beispiel:

```
Sub TestMax()
    MsgBox Max(5, 2, 1, -10)
End Sub

Function Max(ParamArray arg() As Variant) As Variant
    Dim val As Variant
    Max = arg(0)
    For Each val In arg
        If val > Max Then Max = val
    Next
End Function
```

Benannte Argumente.

Syntax
`Sub` *Name* `(`*Para1* `[As Typ], Optional` *Para2* `[As Typ], ...)`

Beim Aufruf können die Parameter in beliebiger Reihenfolge angegeben werden, wenn die bei der Definition verwendeten Parameternamen angegeben werden. Beispiel:

```
Name Para1:=100, Para2:=50
Name Para2:=50,  Para1:=100
```

Aufgabe. Verwenden Sie die Übergabe mit einem Parameter-Array und erstellen Sie ein Programm für die Bestimmung des arithmetischen Mittelwertes beliebig vieler Werte.

3.2.4 Verfahren der Werteübergabe

Werte können auf zwei Arten übergeben werden:

Call by Reference: die Speicheradresse des Parameters bleibt unverändert. Wird der Parameter im Unterprogramm geändert, wirkt sich das auch auf die übergeordneten Programme aus. Standardfall in VBA und FORTRAN. Häufige Ursache zunächst unerklärbarer Programmfehler.

Call by Value: bei der Übergabe wird eine Kopie des Parameters erstellt. Eine Änderung wirkt sich nicht nach oben aus. Standardfall in C/C++. Unbedarfte Anwendung kann bei rekursiven Programmen zu Speicherüberlauf führen.

Beispiel. Es werden zwei Prozeduren verwendet, um zwei übergebene Parameter auszutauschen. Der Unterschied besteht in der Art der Übergabe, `ByRef` (Call by Reference) bzw. `By Val` (Call by Value).

```
Sub Main()
    Dim x As Integer, y As Integer
    x = 3:  y = 5
    TauscheR x, y        ' Die Werte von x und y ändern sich
    TauscheV x, y        ' Die Werte von x und y ändern sich nicht
End Sub

Sub TauscheR(ByRef a As Integer, ByRef b As Integer)
    Dim c As Integer
    c = a
    a = b
    b = c
End Sub

Sub TauscheV(ByVal a As Integer, ByVal b As Integer)
    Dim c As Integer
    c = a
    a = b
    b = c
End Sub
```

3.2.5 Prozeduren und Funktionen abbrechen

Prozeduren und Funktionen können während der Abarbeitung beendet werden.

Syntax
`Exit Sub` `Exit Function`

Beispiel. Subroutine nach 100 Schleifendurchgängen beeenden

```
Sub Count()
    Dim i As Integer
    i = 0
    Do
        i = i + 1
        If i > 100 Then
            MsgBox i
            Exit Sub
        End If
    Loop
End Sub
```

3.2.6 Rekursion

Wenn Programme sich selbst aufrufen, spricht man von Rekursion. Dies wird i.d.R. bei Such- oder Sortieralgorithmen verwendet. Rekursion macht nur mit "Call By Value" Sinn, mit "Call By Reference" funktioniert sie nicht. Mit dem klassischen FORTRAN war Rekursion unmöglich.

Beispiel. Berechnung der Fakultät einer Zahl: `n! = 1*2*3*...*(n-1)*n`

```
Sub fakultät()
    MsgBox "Iterativ:  " & FakI(6)
    MsgBox "Rekursiv:  " & FakR(6)
End Sub
```

Iterative Berechnung.

```
Function FakI (ByRef n As Long) As Long
    Dim erg As Long, i As Long
    Erg = 1
    For i = 1 To n
        Erg = erg*i
    Next i
    FakI = erg
End Function
```

Rekursive Berechnung.

```
Function FakR (ByVal n As Long) As Long
    If n = 1 Then
        FakR = 1
    Else
        FakR = n * FakR(n-1)
    End If
End Function
```

Den Ablauf einer rekursiven Programmierung versteht man am besten, wenn man ihn sich im Debugger anschaut.

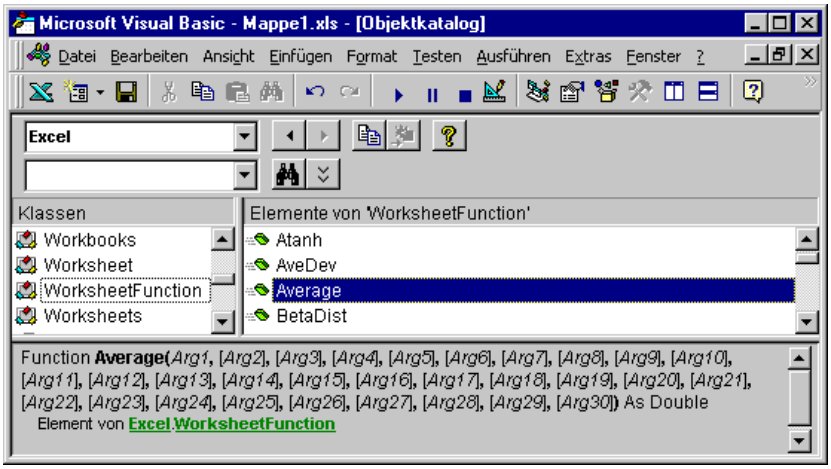
Aufgabe. Programmieren Sie beide Möglichkeiten zur Berechnung der Fakultät einer Zahl. Verfolgen Sie den Ablauf im Debugger. Ergründen Sie bei dieser Gelegenheit das VBA-Objekt *MsgBox*.

3.2.7 VBA Funktionen

Es gibt eine große Menge von fertigen Funktionen in VBA. Den besten Überblick erhält man in der Online-Sprachreferenz `Hilfe -> Visual Basic Sprachreferenz`

3.2.8 Excel-Funktionen

VBA-Funktionen sind nicht mit den Tabellen-Funktionen von Excel identisch. Sind Alternativen vorhanden, sollte man der VBA-Version den Vorzug geben. Jedoch muß man gelegentlich dennoch auf Tabellenfunktionen zurückgreifen. Das Problem bei der Sache ist, dass man den englischen Namen der Tabellenfunktion wissen muss. Man ermittelt ihn am besten durch die Aufzeichnung eines Makros. Z.B., die Tabellenfunktion MITTELWERT(Zahl1;Zahl2;...) zur Bestimmung des arithmetischen Mittelwertes, heißt auf englisch AVERAGE. Angewandt wird die Tabellenblattfunktion am besten als Methode des Objektes *WorksheetFunction*. Der Objektkatalog gibt für die Kategorie Excel einen kompletten Überblick.



Beispiel. Berechnung des arithmetischen Mittelwertes.

```
Sub MW()
    Dim Miw As Variant

    Miw = WorksheetFunction.Average(1, 2, 3, 4, 5, 6)
    MsgBox "Mittelwert: " & Miw

End Sub
```

3.3 Variablen, Konstanten und Arrays

3.3.1 Regeln für Namensgebung und Deklaration

Programme verarbeiten Daten. Daten sind im Hauptspeicher des Computers gespeichert. Zugriff auf die Daten erlangt über die Speicheradresse. Um nicht mit Adressen arbeiten zu müssen, verwendet man statt dessen Namen. Variablen und Konstante sind die Kombination von benannten Speicheradressen und gewissen Zugriffsregeln, die im Einzelfall vom Benutzer definiert und zugewiesen werden. Einerseits gibt es dabei verbindliche VBA-Vorschriften, andererseits empfiehlt es sich, zusätzliche Regeln bei der Namensgebung einzuhalten, um das Programm übersichtlicher und weniger fehleranfällig zu machen.

VBA-Regeln. Verbindlich.

- Das erste Zeichen muss ein Buchstabe sein.
- Folgende Sonderzeichen dürfen nicht verwendet werden: **.!@&$#**
- reservierte Schlüsselwörter dürfen nicht verwendet werden
- Namen müssen innerhalb ihres Gültigkeitsbereiches eindeutig sein
- Die Groß- und Kleinschreibung ist unerheblich, bleibt aber erhalten
- Ein Name darf nicht länger als 255 Zeichen sein.

Zusätzliche Regeln. Freiwillig.

- Namen sollen etwas über ihren Inhalt aussagen. Dabei die Möglichkeit langer Namen nutzen. Die bessere Lesbarkeit ist wichtiger als der Eingabeaufwand.
- Namen sollen mit Präfixes versehen werden. Namen für bestimmte Bereiche, z.B. Dialoge, Tabellen usw., sollten mit einem kurzen aber aussagekräftigen Präfix versehen werden.
  Beispiel: `TblSpalten, TblZeilen` zur Bezeichnung der Anzahl von Tabellenspalten- und Zeilen
- Namen sollen deklariert werden. Zur Vermeidung unbemerkter Doppelbelegungen und Falschinterpretationen durch Schreibfehler. Mit `Option Explicit` im Deklarationsbereich eines Moduls erzwingt man die Deklaration.
- Namen sollen typisiert werden, z.B. `Integer, Double, String, usw.`
- Namen sollen nicht durch Typkennzeichen typisiert werden. Diese Möglichkeit stammt noch aus alten Basic-Zeiten und hat keine Zukunft mehr.

3.3.2 Variable

Variablen sind änderbare Daten. Sie können aus einfachen Datentypen bestehn, aus selbstdefinierten Datentypen oder aus Arrays.

Syntax	
`[Static][Public|Private] Dim` *`Name`* `[As` *`Typ`*`]` `[Static][Public|Private] Dim` *`Name`*`(`*`Indizes`*`) [As` *`Typ`*`]`	
Parameter	Verwendung
`Static`	Nur in Prozedur. Variable behält Wert während der ganzen Laufzeit.
`Public`	Nur auf Modulebene anstelle von `Dim`. Variable im ganzen Projekt bekannt.
`Private`	Auf Modulebene anstelle von `Dim`. Variable nur im Modul bekannt. Default.
`Name`	Name der Variablen
`Typ`	Datentyp
`Indizes`	Indizes von Arrays

Bemerkungen.

- Wenn kein Datentyp vereinbart wird, ist Variant die Voreinstellung
- Option Explicit erzwingt die Deklarationspflicht
- Initialisierung:

Numerische Variablen	0
Variable Zeichenfolgen	Leer
Zeichenfolgen fester Länge	0
Variable vom Typ `Variant`	`Empty`
Objekt Variablen	`Nothing`

Beispiele.

```
Dim Anzahl As Long
Dim Matrix(2,3) As Double
```

3.3.3 Konstante

Konstanten sind nicht änderbare Daten.

Syntax	
`[Public|Private] Const` *`Name`* `[As` *`Typ`*`] =` *`Ausdruck`*	
Parameter	Verwendung
`Public`	Nur auf Modulebene. Konstante ist im ganzen Projekt bekannt.
`Private`	Nur auf Modulebene. Konstante ist nur im Modul bekannt. Default.
`Name`	Name der Konstante
`Typ`	Datentyp
`Ausdruck`	Wert der Konstanten

Beispiele.

```
Const RoteFarbe As Integer = 10
Const MwSt As Single = 0.17
```

Bemerkungen.

> Wird kein Datentyp vereinbart, wird derjenige des Ausdrucks übernommen

> Oft lohnt sich ein eigenes Modul für Konstanten. Vgl. Header-File in C.

3.3.4 statische Arrays

Arrays sind mehrdimensionale Matrizen (oder Tabellen) gleichartiger Datentypen. Die Dimensionen werden bei der Deklaration festgelegt und können während des Programmlaufes nicht modifiziert werden. Arrays eignen sich für Matrizenoperationen der numerischen Mathematik. Arrays können nicht als Konstante vereinbart werden. Mit den Funktionen `LBound` und `UBound` können die unteren und oberen Grenzen der einzelnen Matrixdimensionen abgefragt werden.

Syntax	

`Dim` *`Name`* `(`*`Index1`*`) [As` *`Typ`*`]`
`Dim` *`Name`* `(`*`Index1, Index2, ...`*` ) [As` *`Typ`*`]`
`Dim` *`Name`* `(`*`Untergrenze`* `To` *`Obergrenze`*`) [As` *`Typ`*`]`
`Erase` *`Name`*	

Parameter	Verwendung
`Name`	Namen des Arrays
`IndexN.`	Obere Indexgrenze für N-te Dimension; positive ganze Zahl. Die untere Grenze wird standardmäßig zu 0 vereinbart (vgl. C). Mit `Option Base 1` im Deklarationsbereich des Moduls kann die untere Grenze standardmäßig zu 1 (vgl. FORTRAN).
`Unter-, Obergrenze`	Explizite Angabe des unteren und oberen Index Wird keine Untergrenze a
`Erase`	Feld wird neu initialisiert, bleibt im Speicher erhalten
`Public/Static`	Wie bei Variablen

Beispiel.

```
Dim MatrixA(2,2) As Integer
Dim MatrixB(1 To 2, 1 To 2) As Integer
Dim i As Integer, j As Integer, n As Integer
n = 0

For i = 0 To 2
    For j = 0 To 2
        n = n + 1
        MatrixA(i,j) = n
        If i > 0 And j > 0 Then MatrixB(i,j) = n
```

```
    Next j
Next i
```

Beispiel. Anwendung von LBound und UBound.

```
Sub PrintMatrix(Matrix)
    ' Anzeigen von Matrix "Matrix" mit der Message Box
    Dim i As Integer, j As Integer
    Dim msg As String
    Dim CR As String: CR = Chr(13) & Chr(10)
    For i = LBound(Matrix, 1) To UBound(Matrix, 1)      ' 1. Index
        For j = LBound(Matrix, 2) To UBound(Matrix, 2)  ' 2. Index
            msg = msg & Matrix(i, j) & " "
        Next j
        msg = msg & CR
    Next i
    MsgBox msg
End Sub
```

Aufgabe. Erstellen Sie ein Programm, um die Transponierte einer Matrix **A(n,m)** zu bestimmen. Benutzen Sie LBound und UBound, um die Indexgrenzen zu ermittlen.

3.3.5 dynamische Arrays

Die Dimension dynamischer Arrays wird im Laufe des Programmes bestimmt und ggf. modifiziert. Elegante Möglichkeit zur Bearbeitung von Daten dynamischen Umfangs.

Syntax	

`Dim` *Name* `()`
`ReDim [Preserve]` *Name* `(`*Index1, Index2, ...*`)`
`Erase` *Name*	

Parameter	Verwendung
`Dim`	Anfangsdefiniton ohne Größenangabe!
`ReDim`	Modifikation der Dimension. Achtung: wird das Array verkleinert, gehen die "überstehenden" Daten verloren. Standardmäßig wird das Feld neu initialisiert. Mit `Preserve` bleiben die Daten erhalten.
`Erase`	Feld wird aus dem Speicher gelöscht.
Name	Name des Arrays
IndexN	Dimensionsvereinbarungen wie bei statischen Arrays
`Public/Static`	Wie bei Variablen

Beispiel. `Redim` mit und ohne `Preserve`. Bei mehrdimensionalen Arrays kann `Preserve` nur auf die letzte Dimension angewendet werden.

```
Sub MatReDim()
    Dim MatrixA() As Integer, MatrixB() As Integer  ' Deklaration
    Dim i As Integer, j As Integer, n As Integer
    ReDim MatrixA(1, 1): ReDim MatrixB(1, 1)        ' 1. Anpassung
    n = 0
    For i = 0 To 1
        For j = 0 To 1
            n = n + 1: MatrixA(i, j) = n: MatrixB(i, j) = n
        Next j
    Next i
    PrintMatrix (MatrixA): PrintMatrix (MatrixB)

    ReDim MatrixA(2, 2)              ' 2. Anpassung
    ReDim Preserve MatrixB(1, 2)    ' ReDim mit Preserve nur
                                    ' für letzten Index möglich!
    For i = 0 To 2
        n = n + 1: MatrixA(2, i) = n
        n = n + 1: MatrixA(i, 2) = n
        If i < 2 Then MatrixB(i, 2) = n
```

```
    Next i
    PrintMatrix (MatrixA): PrintMatrix (MatrixB)
End Sub
```

3.3.6 Datenfelder mit der Array-Funktion

Die Array-Funktion bildet aus den Übergabeparametern ein Array vom Typ `Variant`. Für kleinere Felder mit unterschiedlichen Datentypen geeignet.

Beispiel.

```
Sub ArrayFkt()
    Dim Mischung As Variant
    Mischung = Array(2, "Äpfel", "und", 3, "Birnen")
    For i = 0 To UBound(Mischung)
        msg = msg & Mischung(i) & " "
    Next i
    MsgBox msg & " " & Chr(13) & Chr(10) & _
                 TypeName(Mischung(0)) & " " & _TypeName(Mischung(1))
End Sub
```

3.3.7 Gültigkeitsbereich und -dauer

Variablen und Konstanten sind nur in ihrem Gültigkeitsbereich bekannt. D.h., in verschiedenen Gültigkeitsbereichen verweisen Variablen gleichen Namens dennoch auf unterschiedliche Speicherinhalte. Variablen können gültig sein:

- **>** nur in der Prozedur
- **>** im Modul
- **>** im ganzen Projekt

Prozedur. Variable wird in der Prozedur vereinbart und dort verwendet.

```
Sub Beispiel()
    Dim BspVar1 As Single
    ...
End Sub
```

Modul. Modulvariablen werden **vor** der ersten Prozedur deklariert. `Private` ist optional.

```
Dim MdlVar1 As Integer                          ' Modulvariablen und -
Private MdlVar2 As Long                         ' konstanten werden vor der
Const MdlKonstante1 As Double = 3.141           ' ersten Prozedur bzw.
Private Const MdlKonstante2 As Long = 2500      ' Funktion vereinbart.

Sub Beispiel()
    Dim BspVar1 As Single                       ' Prozedurvariable
    ...
End Sub
```

Projekt. Projektvariablen werden **vor** der ersten Prozedur deklariert. `Dim` wird durch `Public` ersetzt.

```
Public MdlVar1 As Integer                       ' Projektvariablen und -
Public Const MdlKonstante1 As Double = 3.141    ' konstanten werden vor der
Public Const MdlKonstante2 As Long = 2500       ' ersten Prozedur bzw.
                                                ' Funktion vereinbart.

Sub Beispiel()
    Dim BspVar1 As Single
    ...
End Sub
```

Flüchtige und Nichtflüchtige Variablen.

Prozedurvariablen verlieren den zugewiesenen Wert nachdem die Prozedur abgearbeitet wurde. Die Variable ist flüchtig. Wird die Prozedur ein weiteres Mal aufgerufen, ist nicht sicher, welchen Wert sie hat. Es hängt von der Sprache und vom Kompiler ab, ob sie neu initialisiert ist oder irgendwelche zu-

fällige Daten enthält. Soll der Wert erhalten bleiben, muß die Variable als *nichtflüchtig* deklariert werden.

Syntax	
`Static` *`Name`* `[As` *`Typ`*`]`	
Parameter	Verwendung
`Static`	Schlüsselwort für nichtflüchtige Variablen
`Name`	Name der Variablen
`Typ`	Datentyp

Beispiel.

```
Function Zähler(Number As Integer) As Long
    Static Zwischenstand As Long
    Dim i As Integer
    For i = 1 To Number
        Zwischenstand = Zwischenstand + 1
    Next i
    Zähler = Zwischenstand
End Function

Sub Beispiel()
    MsgBox Zähler(5)
    MsgBox Zähler(3)
End Sub
```

3.4 Datentypen

Die Datentypen unterscheiden sich nach den darstellbaren Werten und dem darstellbaren Bereich. Der Datentyp `Variant` ist eine Besonderheit von VBA. Er kann anstelle aller anderen eingesetzt werden. Dabei wird relativ viel Speicherplatz und Rechenzeit benötigt, weshalb man bei größeren Programmen die spezifischen Datentypen bevorzugen sollte. Vorsicht: `Variant` ist der voreingestellte Datentyp, wenn Variablen nicht explizit deklariert werden.

3.4.1 Einfache Datentypen

Datentyp	Speicher-platz [Bytes]	Wertebereich	Beschreibung
Byte	1	0 bis 255	Ein Byte
Boolean	2	True oder False	Wahrheitswert
Integer	2	-32768 bis 32767	Ganzzahl
Long	4	-2.147.483.648 bis 2.147.483.647	Lange Ganzzahl
Single	4	Negativ: -3,402823E38 bis -1,401298E-45 positiv: 1,401298E-45 bis 3,402823E38	Einfach genaue Gleitkommazahl
Double	8	Negativ: -1,79769313486232E308 bis -4,94065645841247E-324 positiv: -4,94065645841247E-324 bis 1,79769313486232E308	Doppelt genaue Gleitkommazahl
String	Fest: 1 Variabel: 10 + 1 pro Zeichen	0 bis ca. 2 Milliarden Zeichen	Zeichenketten
Variant	Num: 16 String: 22 + 1 pro Zeichen	Numerische Werte vgl. Double. Zeichenketten wie String mit var. Länge	Beliebiger Datentyp
Object	4		Zeiger auf beliebiges Objekt
Currency	8	-922.337.203.685.477,5808 bis 922.337.203.685.477,587	Währung
Date	8	01.01.100 bis 31.12.9999 0:00:00 bis 23:59:59	Datum und Zeit **Eingabe im engl. Format ! Monat/Tag/Jahr**
Decimal	14	Festkommazahl mit 0 bis 28 festlegbaren Nachkommazahlen. Damit können Rundungsfehler nahezu ausgeschlossen werden. Nicht direkt zuweisbar.	

3.4.2 implizite Typumwandlung

Wird das Ergebnis der Verknüpfung von Variablen unterschiedlicher Typen einer dritten Variablen zugewiesen, können folgende Kombinationen auftreten:

Die Datentypen sind nicht kompatibel. z.B `String` und `Integer`.

> Laufzeitfehler. Das Programm ist nicht ausführbar.

Die Datentypen sind kompatibel. z.B `Integer` und `Double`.

> Der Typ der Ergebnisvariable ist zunächst nicht deklariert: dem Ergebnis wird automatisch der Typ des "größeren" Datentyps zugewiesen. Größer heißt größer im Sinne von beanspruchtem Speicherplatz. `Long` ist größer als `Integer`, `Single` und `Long` sind gleich groß.

> Der Typ der Ergebnisvariable ist bereits definiert und ist groß genug: das Ergebnis wird auf den zur Verfügung stehenden Speicherplatz abgebildet und VBA rechnet damit weiter.

> Der Typ der Ergebnisvariable ist bereits definiert und ist **nicht** groß genug: das Ergebnis wird

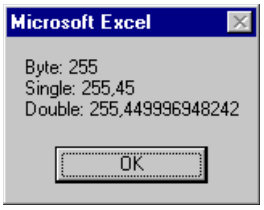
entweder "verstümmelt" oder das Programm bricht mit einem Fehler ab. Eine Verstümmelung entsteht durch Abschneiden der zu langen Bit-Folgen beim Speicherübertrag. Das kann zu einer völlig falschen Interpretation führen und ist nicht mit einem Rundungsfehler zu verwechseln.

> Die Verknüpfung unterschiedlicher Datentypen wird am besten vermieden oder nur mit größter Vorsicht vorgenommen.

> Sind alle Variablen als `Variant` definiert, bestimmt der Operator den Typ des Ergebnis.

Beispiele.

```
Sub Zuweisen()
    Dim BytErg As Byte, DblErg As Double
    Const IntVar As Integer = 132
    Const SglVar As Single = 123.45
    Dim CrgReturn As String: CrgReturn = Chr(13) & Chr(10)

    Erg = IntVar + SglVar
    DblErg = IntVar + SglVar
    BytErg = IntVar + SglVar

    MsgBox TypeName(BytErg) & ": " & CStr(BytErg) & CrgReturn & _
          TypeName(Erg) & ": " & CStr(Erg) & CrgReturn & _
           TypeName(DblErg) & ": " & CStr(DblErg)

    BytErg = BytErg + 1   ' Speicherüberlauf! BytErg kann nicht
                          ' größer als 255 werden. Laufzeitfehler.
End Sub
```



Aufgabe.

Welchen Datentyp und welchen Wert ergibt die Operation `1 & 1` ?

Gegeben sind die beiden Variablen a und b vom Typ `Variant`:

```
Dim a As Variant
Dim b As Variant
a = 1
b = 1
```

Welchen Datentyp und welchen Wert ergeben die Operationen `a + b` und `a & b` ?

3.4.3 explizite Typumwandlung

Explizite Typumwandlungen werden mit speziellen Funktionen vorgenommen. Nachfolgend eine Übersicht. Näheres siehe Online-Hilfe.

Syntax. Entsprechend für alle Funktionen.	
`CStr (`*`Arg`*`)`	
Parameter	Verwendung
Arg	Umzuwandelnder Ausdruck, Zeichenfolge oder numerisch.

Funktion	**Umwandlung in**	**Bemerkung**
`Cbool`	Boolean	Eine gültige Zeichenfolge oder ein gültiger numerischer Ausdruck.
`Cbyte`	Byte	0 bis 255.
`Ccur`	Currency	-922.337.203.685.477,5808 bis 922.337.203.685.477,5807.
`CDate`	Date	Ein beliebiger gültiger Datumsausdruck.
`CDbl`	Double	-1,79769313486232E308 bis -4,94065645841247E-324 für negative Werte; 4,94065645841247E-324 bis 1,79769313486232E308 für positive Werte.
`CDec`	Decimal	+/-79.228.162.514.264.337.593.543.950.335 für skalierte Ganzzahlen, d.h. Zahlen ohne Dezimalstellen.

		Für Zahlen mit 28 Dezimalstellen gilt der Bereich +/-7,9228162514264337593543950335. Die kleinste mögliche Zahl ungleich Null ist 0,0000000000000000000000000001.
`CInt`	Integer	-32.768 bis 32.767; Nachkommastellen werden gerundet.
`CLng`	Long	-2.147.483.648 bis 2.147.483.647; Nachkommastellen werden gerundet.
`CSng`	Single	-3,402823E38 bis -1,401298E-45 für negative Werte; 1,401298E-45 bis 3,402823E38 für positive Werte.
`Cvar`	Variant	Numerische Werte im Bereich des Typs Double. Nichtnumerische Werte im Bereich des Typs String.
`CStr`	String	Rückgabe für CStr hängt vom Argument *Arg* ab.

3.4.4 selbstdefinierte Datentypen

Selbstdefinierte Datentypen sind auch als *Record* (vgl. FORTRAN 90) oder *Structure* (vgl. C) bekannt. Ein Record ist die Zusammenfassung mehrerer unterschiedlicher Datentypen unter einem Namen, im Gegensatz zum Array.

Syntax

```
[Public|Private]Type Name
    Name1 As Typ
    Name2 As Typ
    ...
End Type
```

Parameter	Verwendung
Name	Name des Records
NameN	Namen der einzelnen Variablen des Records
Typ	Datentyp der einzelnen Variablen

- **>** Der Typ eines Records muß grundsätzlich auf Modulebene, vor der ersten Prozedur definiert werden.
- **>** Zur Verwendung von Records müssen Variablen seines Typs deklariert werden.
- **>** Variablen eines Records werden wie Eigenschaften eines Objektes angesprochen: *Name.Var*
- **>** Records können auch Elemente eines Arrays sein.
- **>** Records können untereinander als Ganzes verglichen werden.

Beispiel. Einfache Mitarbeiter-Datenbank.

```
Type TypeMitArb
    LfdNr As Long
    Name As String
    GebDat As Date
    Gehalt As Currency
End Type

Sub Mitarbeiter()
    ' einfache Mitarbeiter-Datenbank
    Dim MitArb(10) As TypeMitArb

    Dim msg As String

    MitArb(0).LfdNr = 10001        ' Stamm-Nummer
    MitArb(0).Name = "Mustermann"  ' Name
    MitArb(0).GebDat = "1/15/2000" ' Achtung: Monat/Tag/Jahr
    MitArb(0).Gehalt = 6200#       ' Gehalt


    msg = CStr(MitArb(0).LfdNr) & MitArb(0).Name & _
          CStr(MitArb(0).GebDat) & CStr(MitArb(0).Gehalt)
    MsgBox msg
End Sub
```

3.4.5 Bestimmung des Variablentyps

Variant-Variablen können alle Typen zugewiesen werden. Der jeweils tatsächlich zugewiesene kann bestimmt werden. Dafür gibt es eine Reihe von VBA-Funktionen. Siehe auch Online-Hilfe.

Syntax		
`TypeName(`*`VarName`*`)`		
Gibt eine Zeichenfolge zurück, die Informationen über eine Variable enthält.		
Parameter	Verwendung	
`VarName`	Name der zu testenden Variable	
	TypeName kann folgende Zeichenfolgen zurückgeben:	
	Zurückgegebene Zeichenfolge	Variable
	Objekttyp	Ein Objekt mit dem Typ Objekttyp
	Byte	Byte-Wert
	Integer	Ganzzahl (Integer)
	Long	Ganzzahl (Long)
	Single	Fließkommazahl einfacher Genauigkeit
	Double	Fließkommazahl doppelter Genauigkeit
	Currency	Währungsbetrag (Currency)
	Decimal	Dezimalwert
	Date	Datumswert (Date)
	String	Zeichenfolge (String)
	Boolean	Boolescher Wert
	Error	Fehlerwert
	Empty	Nicht initialisiert
	Null	Keine gültigen Daten
	Object	Ein Objekt
	Unknown	Ein Objekt mit unbekanntem Typ
	Nothing	Objektvariable, die auf kein Objekt verweist

Beispiel.

```
Dim NullVar, Type1, StrVar As String, IntVar As Integer, CurVar As Currency
Dim FeldVar (1 To 5) As Integer
NullVar = Null                      ' Null zuweisen.
Typ1 = TypeName(StrVar)             ' Liefert "String".
Typ1 = TypeName(IntVar)             ' Liefert "Integer".
Typ1 = TypeName(CurVar)             ' Liefert "Currency".
Typ1 = TypeName(NullVar)            ' Liefert "Null".
Typ1 = TypeName(FeldVar)            ' Liefert "Integer()"
```

Syntax			
`VarType(`*`VarName`*`)`			
Gibt einen Wert vom Typ Integer zurück, der den Untertyp einer Variablen anzeigt.			
Parameter	Verwendung		
`VarName`	Name der zu testenden Variable vom Typ `Variant`		
	Rückgabewerte		
	vbEmpty	0	Empty (nicht initialisiert)
	vbNull	1	Null (keine gültigen Daten)
	vbInteger	2	Ganzzahl (Integer)
	vbLong	3	Ganzzahl (Long)
	vbSingle	4	Fließkommazahl einfacher Genauigkeit
	vbDouble	5	Fließkommazahl doppelter Genauigkeit
	vbCurrency	6	Währungsbetrag (Currency)
	vbDate	7	Datumswert (Date)
	vbString	8	Zeichenfolge (String)
	vbObject	9	Objekt
	vbError	10	Fehlerwert
	vbBoolean	11	Boolescher Wert (Boolean)

	vbVariant	12	Variant (nur bei Datenfeldern mit Variant-Werten)
	vbDataObject	13	Ein Datenzugriffsobjekt
	vbDecimal	14	Dezimalwert
	vbByte	17	Byte-Wert
	vbArray	8192	Datenfeld (Array)

Beispiel.

```
Dim IntVar, StrVar, DatumVar, Test1
                                        ' Variablen initialisieren.
IntVar = 459: StrVar = "Informationsverarbeitung im Bauimgenieurwesen"
DatumVar = #03.10.91#

Test1 = VarType(IntVar) ' Liefert 2.
Test1 = VarType(DatumVar)     ' Liefert 7.
Test1 = VarType(StrVar) ' Liefert 8.
```

Syntax	
`IsNumeric(`*`Ausdruck`*`)`	
Gibt einen Wert vom Typ `Boolean` zurück, der angibt, ob ein Ausdruck als Zahl ausgewertet werden kann.	
Parameter	Verwendung
Audruck	Wert vom Typ `Variant` und enthält einen numerischen Ausdruck oder einen Zeichenfolgenausdruck.

Beispiel.

```
Dim Var1, Test1
Var1 = "53"                         ' Wert zuweisen.
Test1 = IsNumeric(Var1)             ' Liefert True.

Var1 = "459.95"                     ' Wert zuweisen.
Test1 = IsNumeric(Var1)             ' Liefert True.

Var1 = "45 Hilfe"                   ' Wert zuweisen.
Test1 = IsNumeric(Var1)             ' Liefert False.
```

3.5 Operatoren

3.5.1 Arithmetische Operatoren

Arithmetische Operatoren	
+	Addieren
-	Subtrahieren
*	Multiplizieren
/	Dividieren, mit Fließkomma-Zahl als Ergebnis
\	Dividieren, mit ganzer Zahl als Ergebnis
^	Potenzieren
Mod	Rest einer ganzzahligen Division

> Es gilt Punkt- vor Strichrechnung, also ^, *, /, \, Mod vor +, -
> Die Auswertung gleichrangiger Operatoren erfolgt von links nach rechts
> Es gelten die üblichen Regeln der Klammersetzung.

Beispiele.

```
Erg = 10 + 20 - 2 * 3    ' = 24
Erg = 10 + 20 / 5        ' = 14
Erg = 10 * (2 + 4) / 25  ' = 2,4
Erg = 10 * (2 + 4) / 25  ' = 2
Erg = 10 * 7 \ 3         ' = 23
Erg = 10 * (7 \ 3)       ' = 20
Erg = 10 * 5 ^ 2 + 5     ' = 255
```

3.5.2 Vergleichsoperatoren

Vergleichsoperatoren	
<	kleiner
<=	kleiner gleich
=	gleich
>	größer
>=	größer gleich
<>	ungleich
Is	Prüft die Gleichheit von Referenzen auf Objekte. Dient vornehmlich dem Vergleich mit `Nothing`.
Like	Vergleich von Zeichenketten

> Nur Ausdrücke gleicher Datenarten können verglichen werden
> Das Ergebnis des Vergleichs ist ein Wahrheitswert
> Der Vergleich von Ausdrüchen mit einfacher und doppelter Genauigkeit führt zur Rundung auf einfache Genauigkeit.

Beispiele. Like.

```
Dim Suchwert As String
Dim Erg As Boolean

Suchwert = "Meier, Jahrgang 1990"

Erg = Suchwert Like "M??er,*"          ' True
Erg = Suchwert Like "* 1991"           ' False
Erg = Suchwert Like "*, Jahrgang 199#" ' True
Erg = Suchwert Like "M[ae][yi]er, *"   ' True
Erg = Suchwert Like "Ma[yi]er, *"      ' False
```

3.5.3 Verkettungsoperatoren

Verkettungsoperatoren	
&	Verkettung zweier Zeichenketten
+	Wie &; besser nicht verwenden, Verwechslung mit arithmetischem +

> Nicht numerische Daten werden in Datentyp `Variant` umgewandelt
> Vor dem & muß immer ein Leerzeichen gesetzt werden.

3.5.4 logische Operatoren

Logische Operatoren	
AND	Logisches Und
OR	Logisches Oder
NOT	Logisches Nicht
XOR	Exklusives (entweder) Oder
EQV	Logische Äquivalenz
IMP	Logische Implikation

Beispiele.

```
If Alter >= 12 And Alter <= 18        Then MsgBox "jugendlich"
If Not (Alter >= 12 And Alter <= 18)  Then MsgBox "nicht jugendlich"
If Alter < 12 Or Alter > 18           Then MsgBox "nicht jugendlich"
If Alter >= 12 Xor Alter <= 18        Then MsgBox "nicht jugendlich"
If Alter > 11 Imp Alter > 18          Then MsgBox "nicht jugendlich"
If Alter > 11 Eqv Alter > 18          Then MsgBox "nicht jugendlich"
```

3.6 Strukturelemente

3.6.1 Verzweigung

Einfache Verzweigung. Die Anweisung muß in einer Zeile stehen.

Syntax	
`If` *`Bedingung`* `Then` *`Befehl`* `[Else` *`Befehl`*`]`	
Parameter	Verwendung
`Bedingung`	Ausdruck, der `True` oder `False` ergibt
`Befehl`	Einzelner Befehl

Mehrfachverzweigungen.

Syntax	

`If` *`Bedingung`* `Then`
`    `*`Befehle`*
`ElseIf` *`Bedingung`* `Then`
`    `*`Befehle`*
`[Else`
`    `*`Befehle`*`]`
`End If`	

Parameter	Verwendung
`Bedingung`	Ausdruck, der `True` oder `False` ergibt
`Befehl`	Einzelner Befehl
`Befehle`	Mehrere Befehle

> Ermöglicht eine Verzweigung aufgrund einer Bedingung
> Die Bedingung muß so gestellt sein, dass deren Auswertung ein `True` oder `False` ergibt.

3.6.2 Mehrfachauswahl

Syntax	

`Select Case` *`Ausdruck`*
`    Case` *`Werteliste1`*
`        `*`Befehle`*
`    Case` *`Werteliste2`*
`        `*`Befehle`*
`    [Case Else`
`        `*`Befehle`*
`End Select`	

Parameter	Verwendung
`Ausdruck`	Ein Auswertbarer Ausdruck
`WertelisteN`	Liste mit Vergleichswerten
`Befehle`	Ein oder mehrere Befehle

> Wertet den Ausdruck in der Reihenfolge der `Case`-Zweige aus und führt den ersten zutreffenden aus.
> Trifft kein Fall zu und ist kein `Case Else` vorgesehen, gibt es eine Fehlermeldung.

Wertelisten. In Wertelisten können mehrere Ausdrücke stehen. Sie werden durch Komma getrennt. Sie werden so behandelt, als wären sie mit einem logischen Oder (`Or`) verknüpft. In Wertelisten können auch Konstanten, Variablen und Fnktionen stehen. Vergleichsoperatoren muß ein `Is` vorangestellt werden.

Beispiel.

```
Function Bewertung(Note As Single) As String
    Select Case Note
        Case Is < 1
            Bewertung = "Unzulässig"
        Case 1, 1.3
            Bewertung = "Sehr gut"
        Case Is <= 2.3
            Bewertung = "Gut"
        Case Is <= 3.3
            Bewertung = "Befriedigend"
        Case Is <= 4
            Bewertung = "Bestanden"
        Case 5
            Bewertung = "Durchgefallen"
        Case Else
            Bewertung = "Unzulässig"
    End Select
End Function
```

3.6.3 Schleifen

Do ... Loop

Kopfgesteuert.

Syntax	
`Do While|Until` *`Bedingung`* *`Befehle`* `Loop`	
Parameter	Verwendung
`While`	Laufbedingung
`Until`	Abbruchbedingung
`Bedingung`	Ausdruck der `True` oder `False` ergibt
`Befehle`	Ein oder mehrere Befehle

> Test erfolgt bevor die Anweisungen im Schleifenkörper ausgeführt werden.
> Trifft die Bedingung nie zu, wird die Schleife nicht ausgeführt.

Fußgesteuert.

Syntax
`Do`   *`Befehle`* `Loop While|Until` *`Bedingung`*

> Wird grundsätzlich einmal durchlaufen, auch wenn die Bedingung nicht zutrifft.

For ... Next

Syntax	
`For` *`Zähler = Startwert`* `To` *`Endwert`* `[Step` *`Schrittweite`*`]` *`Befehle`* `Next` *`Zähler`*	
Parameter	Verwendung
`Zähler`	Numerische Zählvariable, typischerweise `Integer` Der Zähler kann in der Schleife verändert werden. Sollte allerdings vermieden werden.
`Startwert`	Anfangswert der Zählung
`Endwert`	Endwert der Zählung; die Schleife wird beendet, wenn der Endwert erreicht oder überschritten wird
`Shrittweite`	Intervall der Zählung

Befehler	Ein oder mehrere Befehle

Beispiel. Eine 4x4-Matrix auf das Tabellenblatt rechts und unterhalb der aktiven Zelle zaubern. Elemente sind spaltenweise numeriert.

```
Sub MatrixOnTable()
    Dim i As Integer, j As Integer, n As Integer
    n = 0
    For j = 0 To 3
        For i = 0 To 3
            n = n + 1
            ActiveCell.Offset(i, j).Value = n
        Next i
    Next j
End Sub
```

For Each ... Next

Im Unterschied zur "For ... Next"-Schleife wird kein Zähler hoch- oder hinuntergezählt, sondern eine Auflistung von Datenfeldern oder Objekten abgearbeitet. Die Reihenfolge orientiert sich an der Speicherplatzfolge.

Syntax	
For Each *Element* In *Gruppe* *Befehle* Next	
Parameter	Verwendung
Element	Variable, die ein einzelnes Element der Auflistung enthält
Gruppe	Auflistung von Daten oder Objekten

Beispiel. Eine 4x4-Matrix auf das Tabellenblatt rechts und unterhalb der aktiven Zelle zaubern. Matrix zeilenweise numerieren. D.h, intern werden Zellbereich zeilenweise abgearbeitet.

```
Sub MatrixOnTable()
    Dim NextCell As Range, ThisRegion As Range
    n = 0
    Set ThisRegion = ActiveCell.Resize(4, 4)
    For Each NextCell In ThisRegion
        n = n + 1
        NextCell.Value = n
    Next
End Sub
```

3.6.3.1 Schleifen abbrechen

Eine Schleife kann während der Ausführung beendet werden.

Syntax	
Exit Do Exit For	
Parameter	Verwendung

Beispiel. Eine Endlosschleife nach 100 Durchläufen beenden.

```
Sub Count()
    Dim i As Integer
    i = 0
    Do
        i = i + 1
        If i > 100 Then Exit Do
    Loop
    MsgBox i
End Sub
```

3.6.4 Behandlung von Benutzerfehlern

Mit `On Error` steht ein Mechanismus zur Verfügung, der die Ausführung einer Prozedur überwacht und im Falle eines Fehlers an einer anderen Stelle des Codes eine gewisse Anweisung ausführt. Das Verfahren beruht auf der Anwendung von Sprüngen innerhalb des Programmcodes. Die Verwendung von Sprüngen ist ein inzwischen veralteter Programmierstil und sollte außer bei Fehlerbehandlungen nicht verwendet werden.

Syntax	
`On Error Goto` *`FehlerLabel`* `On Error Goto 0` `Resume` *`Fortsetzung`*	
Parameter	Verwendung
`FehlerLabel`	Sprungadresse innerhalb der Prozedur, nach der die Anweisungen des Fehlerfalles angegeben sind
`0`	Rückgabe der Fehlerkontrolle an VBA ("Zurücksetzen").
`Resume`	Angaben der Sprungadresse zum Fortsetzen des Programmes

Beispiel. Division durch 0 verhindern.

```
Sub Division()
    Dim inp As Variant, out As Variant

Eingabe:
    inp = InputBox("Bitte einen Wert eingeben")

    On Error Goto DivFehler
    out = 1 / inp
    On Error Goto 0

    MsgBox "1/" & inp & "=" & out
    Exit Sub

DivFehler:
    MsgBox "Fehler: Division durch 0!"
    Resume Eingabe
End Sub
```

3.7 Objekttechniken

Die wesentliche Eigenschaft eines Objektes ist die Einheit von Daten und Methoden. Nur mit den Methoden, die der allgemeinen Benutzung freigegeben sind, kann man die Daten manipulieren. Man spricht von *Datenkapselung*. Auf diese Weise kann der Datenzugriff insbesondere bei großen Projekten geregelt werden und weitgehend fehlerfrei gestaltet werden. Die Definition eines Objektes erfolgt in einer sogenannten *Objektklasse*. Eine Klasse ist definiert durch die Daten und Methoden, sowie der jeweiligen Zugriffsrechte. Meist beinhaltet eine Objektklasse innere, man sagt *private* oder *versteckte* Daten und Prozeduren, auf die der Anwender keinen Zugriff hat, die aber für die Ausführung der Methoden wichtig sind. VBA sieht verschiedene Möglichkeiten vor, eigene Objekte zu definieren oder objektorientierte Arbeitsweisen zu entwickeln. Ein weiterer Vorteil einer solchen Arbeitsweise ergibt sich aus dem damit möglichen dynamischen Speicherzugriff bei der Definition von *Objektsammlungen*.

3.7.1 Property-Prozeduren

Property-Prozeduren erlauben eine einfache Form der Datenkapselung. Variablen eines Modules sind anderen Modulen unsichtbar, wenn sie als `Private` vereinbart wurden. Man unterscheidet:

Syntax	

`Property Let ( `*`Parameterliste`*` )`
`Property Set ( `*`Parameterliste`*` )`
`Property Get ( `*`Parameterliste`*` ) As Typ`	

Parameter	Verwendung
`Let, Set`	Wertezuweisung, `Let` für Variablen, `Set` für Objektverweise
`Get`	Abfrage der Eigenschaftswerte

- **>** Die Anzahl der Argumente in der `Get`-Prozedur muß um eins kleiner sein als in der `Let`- bzw. `Set`-Prozedur
- **>** Die Datentypen der Parameterliste für `Get` müssen, abgesehen vom letzten Argument, der `Let`- bzw. `Set`-Prozeduren entsprechen
- **>** Der Typ des Rückgabewertes der `Get`-Prozedur muß dem Typ des letzten Parameters der `Let`- bzw. `Set`-Prozeduren entsprechen

Beispiel. Eine ungewöhnliche Ermittlung des Mittelwertes.

```
Option Explicit
Private Summe As Double             ' versteckte Modulvariablen
Private Anzahl As Integer

Property Let MW(Var As Double)      ' Property Let: "Links vom Gleichheits-
                                    '                zeichen"
    If Var = 0 Then                 ' Initialisieren
        Summe = 0
        Anzahl = 0
    Else
        Summe = Summe + Var         ' Summieren
        Anzahl = Anzahl + 1
    End If
End Property

Property Get MW() As Double         ' Property Get: "Rechts vom Gleich-
                                    '                heitszeichen"
    MW = Summe / Anzahl             ' Auswerten
End Property

Sub TestMW()                        ' Anwendung
    Dim Mittel As Double
    Dim i As Integer
    MW = 0                          ' Initialisieren
```

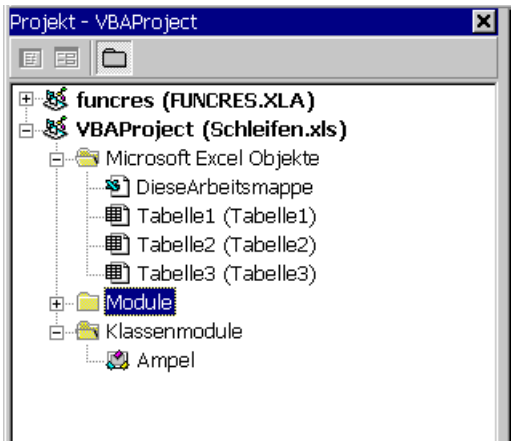
```
    For i = 1 To 10 Step 3
        MW = i                          ' Summieren
    Next
    Mittel = MW                         ' Auswerten
    MsgBox "Mittelwert = " & Mittel
End Sub
```

3.7.2 Eigene Objektklassen

Eine Objektklasse wird in einem eigenen Objektmodul verwaltet. Vor der eigentlichen Codierung muß deshalb zuerst ein neues Objektmodul erzeugt werden.

Vorgehen: im VBE-Editor Menü `Einfügen -> Klassenmodul`

Es wird ein neues Klassenmodul erzeugt. Name ändern mit `Ansicht -> Eigenschaftsfenster`.



Klassenmodul "Ampel" im Projekt-Explorer

Beispiel. Die Klasse "Eigenschaft". Eigene Klasse unter Verwendung von Property-Prozeduren.

Der Inhalt des Klassenmoduls:

```
Option Explicit
Private Buffer As Integer           ' Speicher: Interne gekapselte Variable

Public Property Let Prop(Inp As Integer)  ' Speicher belegen
    Buffer = Inp
End Property

Public Property Get Prop() As Integer     ' Speicher abfragen
    Prop = Buffer
End Property

Public Function Modify()                  ' Speicher modifizieren und
    Buffer = Buffer + 20                  ' abfragen
    Modify = Buffer
End Function
```

Anwenden der Klasse "Eigenschaft":

```
Sub TryProp()
    Dim TP As New Eigenschaft              ' Neue Instanz der Klasse
    Dim Inp As Integer
                                           ' Speicher belegen
    TP.Prop = CInt(InputBox("Bitte eine ganze Zahl eingeben"))
```

```
    Do
        Inp = TP.Modify                    ' Speicher modifizieren
        MsgBox "Speicher: " & CStr(Inp)
        If TP.Prop > 50 Then Exit Sub      ' Speicher abfragen
    Loop
End Sub
```

Beispiel. Definition einer Objektklasse *Ampel*. Zunächst wird ein Klassenmodul *Ampel* erzeugt. Der Name des Klassenmoduls definiert den Namen des Objekts. Sodann folgender Code:

```
Option Explicit

Public Name As String           ' Attribut "Name", öffentlich

Public xPos As Single           ' Attribute x- und y-Koordinate
Public yPos As Single

Public Rot As Object            ' Attribut "grafisches Objekt Rotlicht"
Public Gelb As Object           '                              "Gelblicht"
Public Grün As Object           '                              "Grünlicht"
Public Kasten As Object         '                            "Ampelkasten"

                                      ' Objekt Methode "Ampel Erzeugen"
Public Sub Create(x As Single, y As Single, label As String)

    xPos = x
    yPos = y
    Name = label

    ActiveSheet.Shapes.AddShape(msoShapeRectangle, xPos, yPos, 36, 96). _
        Select
    Set Kasten = Selection
    Selection.ShapeRange.Fill.ForeColor.SchemeColor = 22
    Selection.ShapeRange.Fill.Visible = msoTrue
    Selection.ShapeRange.Fill.Solid

    ActiveSheet.Shapes.AddShape(msoShapeOval, xPos + 6, yPos + 6, 24, _
                                24).Select
    Selection.ShapeRange.Fill.ForeColor.SchemeColor = 23
    Set Rot = Selection

    Selection.ShapeRange.Duplicate.Select
    Set Gelb = Selection
    Selection.ShapeRange.IncrementLeft -12
    Selection.ShapeRange.IncrementTop 18

    Selection.ShapeRange.Duplicate.Select
    Set Grün = Selection
    Selection.ShapeRange.IncrementLeft -12
    Selection.ShapeRange.IncrementTop 18

End Sub

Public Sub RotAn()               ' Objekt Methode "Rotlicht an"
    Rot.ShapeRange.Fill.ForeColor.SchemeColor = 10
End Sub

Public Sub RotAus()              ' Objekt Methode "Rotlicht aus"
    Rot.ShapeRange.Fill.ForeColor.SchemeColor = 23
End Sub
```

Die Funktionen für das An- und Ausschalten der Gelb- und Grünlicher sind entsprechend.

Aufgabe. Erstellen Sie analog zu den früher dargestellten Bespielen weitere Objektmethoden und -attribute für die Ampel, z.B. für die Schaltungen.

Beispiel. Anwendung der Objektklasse *Ampel.* Das Objekt `Collection` wird nachfolgend erklärt.

```
Option Explicit

Dim Ampeln As New Collection        ' Definition einer Sammlung aller Ampeln
                                    ' als Modulvariable


Sub NeueAmpeln()                    ' Prozedur für die Erstellung von Ampeln

    Dim AmpelEins As New Ampel      ' neue Objektinstanz vom Typ "Ampel"
    Dim AmpelZwei As New Ampel      ' neue Objektinstanz vom Typ "Ampel"

             ' Erzeuge erste Ampel, Ausführen der Objekt-Methode "Create"
    AmpelEins.Create 10, 10, "Ampel_1"
    Ampeln.Add AmpelEins, "Ampel_1"

           ' Erzeuge zweite Ampel, Ausführen der Objekt-Methode "Create"
    AmpelZwei.Create 50, 50, "Ampel_2"
    Ampeln.Add AmpelZwei, "Ampel_2"

    AmpelEins.RotAn                  ' Ausführen der Objekt-Methoden zum
    AmpelZwei.GelbAn                 ' Anschalten diverser Lichter.

    Farbe AmpelZwei.Kasten, "blau"   ' direkter Zugriff auf den Kasten
                                     ' der zweiten Ampel, um seine Farbe zu
                                     ' ändern

End Sub
```

3.7.3 Das Collection-Objekt

Eine Kollektion ist eine Sammlung von Objekten, die im Sinne einer *verketteten Liste* zusammengehalten werden. Man kann Objekte hinzufügen, herausnehmen oder ein einzelnes Objekt bearbeiten. Die Objekte können von verschiedenen Typen, auch selbst definierten Klassen sein.

Vereinbarung eines Collection Objektes.

Syntax	
`Dim` *`NameSammlung`* `As New Collection`	
Parameter	Verwendung
`NameSammlung`	Name des Collection Objektes
`New`	Zuweisung einer neuen Instanz schon bei der Definiton
`Collection`	Objekt Typ `Collection`

Objekt hinzufügen.

Syntax	
`NameSammlung``.Add Item, [`*`Key`*`, Before, After]`	
Parameter	Verwendung
`NameSammlung`	Name des Collection Objektes
`Item`	Objekt, das hinzugefügt werden soll
`Key`	Eindeutige Bezeichnung (String anstelle einer Indexzahl)
`Before`	Mitglied, vor dem eingefügt werden soll
`After`	Mitglied, nach dem eingefügt werden soll

Objekt löschen.

Syntax
`NameSammlung``.Remove Index`

Parameter	Verwendung
NameSammlung	Name des Collection Objektes
Index	Eindeutige Nummer oder Bezeichnung

Auf Objekt zugreifen.

Syntax	
NameSammlung.Item(Index)	
Parameter	Verwendung
NameSammlung	Name des Collection Objektes
Index	Eindeutige Nummer oder Bezeichnung .Index ist optional. Statt dessen kann die Klammer direkt nach *NameSammlung* geschrieben werden.

Anzahl der Objekte.

Syntax	
NameSammlung.Count	
Parameter	Verwendung
Count	Liefert die Anzahl der Mitglieder einer Kollection. Typ Long.

Beispiele. Sammlung aller Ampeln.
Bereits im Abschnitt zuvor wurde die Kollektion vereinbart:

```
Dim Ampeln As New Collection    ' Definition einer Sammlung aller Ampeln
```

Hinzufügen einzelner Ampeln zur Sammlung:

```
Ampeln.Add AmpelEins, "Ampel_1" ' Hinzufügen zur Sammlung "Ampeln"
                                ' AmpelEins   .. Objekt Variable
                                ' Ampel_1 .. eindeutige Bezeichnung, String
```

Weitere Beispiele zum Umgang mit Sammlungen, sowie andere Dienstprogramme:

```
Sub LöscheAmpeln()            ' alle Ampeln entfernen
    Dim i As Long
    For i = 1 To Ampeln.Count ' Schleife über alle Objekte
        Ampeln(1).Grün.Cut    ' Löschen der graf. Objekte
        Ampeln(1).Gelb.Cut
        Ampeln(1).Rot.Cut
        Ampeln(1).Kasten.Cut
        Ampeln.Remove 1       ' Entfernen aus der Sammlung
    Next
End Sub

Private Function CheckName(AmpelName As String) As Boolean
                       ' Überprüfen aller Objekt Bezeichnungen auf Existenz
    Dim Amp As Object

    CheckName = False             ' Voreinstellung: Objekt existiert nicht
    For Each Amp In Ampeln
        If Amp.Name = AmpelName Then  ' Objekt existiert,
            CheckName = True          ' Rückgabewert "True"
            Exit Function             ' Funktion verlassen.
        End If
    Next

End Function
```

```
Public Sub Farbe(Item As Object, Color As String)
'   direkter Zugriff auf graf. Objekt, um Farbe zu ändern
    Dim icol As Integer
    Select Case Color
        Case "rot":  icol = 10
        Case "gelb": icol = 13
        Case "grün": icol = 11
        Case "blau": icol = 12
        Case "grau": icol = 22
        Case Else:   icol = 23
    End Select
    Item.ShapeRange.Fill.ForeColor.SchemeColor = icol
End Sub
```

3.8 Das Range-Objekt - Zugriff auf Zellen

Aus der großen Vielzahl der Excel Objekte werden im folgenden die wichtigsten beschrieben, die sich mit dem Zugriff auf Zellen befassen. Informationen über weitere Objekte, Eigenschaften und Methoden gewinnt man aus der Online-Hilfe oder durch Aufzeichnung eines Makros.

Das Range-Objekt repräsentiert einen beliebigen Zellbereich. Damit kann auf einzelne oder mehrere, auch zusammenhängende Zellen zugegriffen werden. Die einzelne Zelle ist ein Sonderfall des Range-Objektes. Der zugriff auf Zellbereiche ist auf mehrere Arten möglich. Das Range-Objekt ist das wichtigste Objekt für den Datentransfer zwischen VBA-Prozeduren und der Excel-Oberfläche. Es wurde in den Beispielen bereits ständig verwendet und erscheint regelmäßig in aufgezeichneten Makros.

Syntax	
`Range("A1:A3")`	Direkte Adressierung
`Range("Umsatz")`	Zugriff über Bereichsnamen
`Range("A1")(2,4)`	Zugriff über Zeilen- und Spaltenindex
`ActiveCell` `Selection`	Zugriff über vordefinierte Objekte, die auf aktive Zellen verweisen

3.8.1 Cells-Eigenschaft

Syntax	
`Cells(`*`ZeilenIndex, SpaltenIndex`*`)`	
Liefert alle Zellen eines Tabellenblattes, Zugriff über indizes	
Parameter	Verwendung
`ZeilenIndex,` *`SpaltenIndex`*	Zellenindizes, Zeile, Spalte

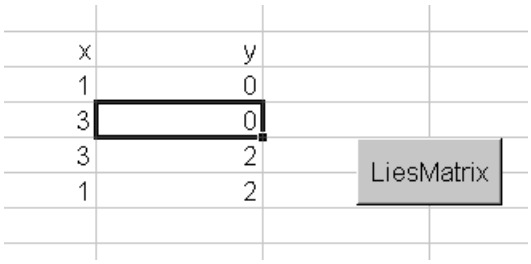
Beispiel.

```
Dim Wert As Integer

Cells(14, 4) = 123            ' Schreibe einen Wert in Zelle D14
Wert = Cells(14, 4)           ' Lies Wert aus Zelle D14
```

3.8.2 CurrentRegion- Eigenschaft

Syntax
`.CurrentRegion`
Liefert einen Bereich als Range-Objekt, der von leeren Zellen umgeben ist.

Beispiel. Von einem Excel-Tabellenblatt soll eine mehrspaltige Wertetabelle eingelesen werden. Es genügt, eine Zelle des Wertebereichs zu aktivieren. Mit `CurrentRegion` wird der gesamte Bereich bestimmt, der von leeren Zellen umgeben ist. Das VBA-Makro "LiesMatrix" ist einem Formular-Knopf zugewiesen.




```
Sub LiesMatrix()
    Dim CR As String: CR = Chr(13) & Chr(10)
    Dim MatrixBereich As Range
    Dim MatrixElement As Range
    Dim ElementWert As Variant
    Dim i As Long

    Set MatrixBereich = ActiveCell.CurrentRegion        ' Bereich festlegen

    i = 0
    For Each MatrixElement In MatrixBereich  ' Zellen im Bereich abarbeiten
        i = i + 1
        ElementWert = MatrixElement.Value
        If IsNumeric(ElementWert) Then
            MsgBox "Element " & i & ": " & CStr(ElementWert) & CR & _
                   "ist vom Typ " & TypeName(ElementWert)
        End If
    Next
End Sub
```

3.8.3 Count-Eigenschaft

Syntax
`Count`
Anzahl der Zellen innerhalb eines Bereichs

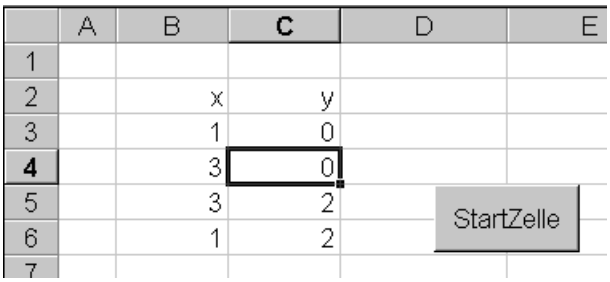
Beispiel. Bestimme die Anzahl der Zellen im ausgewählten Bereich.

```
Sub ZähleZellen()
    Dim MatrixBereich As Range
    Dim Anzahl As Long

    Set MatrixBereich = ActiveCell.CurrentRegion
    Anzahl = MatrixBereich.Count

    MsgBox "So viel sind's: " & CStr(Anzahl)
End Sub
```

3.8.4 Column- und Row-Eigenschaften

Syntax
`Column` `Row`
Liefert Spalten- bzw. Zeilennummer der ersten Zelle im ausgewählten Bereich.

Beispiel. Bestimme die Startzelle (2,2) für das dargestellte Beispiel.




```
Sub StartZelle()
    Dim MatrixBereich As Range
    Dim Row As Long, Column As Long

    Set MatrixBereich = ActiveCell.CurrentRegion
    Row = MatrixBereich.Row
    Column = MatrixBereich.Column

    MsgBox "Startzelle(" & CStr(Row) & "," & CStr(Column) & ")"
End Sub
```

3.8.5 `Columns`- und `Rows`-Eigenschaften

Syntax	
`Columns(`*n*`)` `Rows(`*n*`)`	
Liefert eine Sammlung mit den Spalten bzw. Zeilen eines Bereiches..	
Parameter	Verwendung
n	n-te Spalte/Zeile des ausgewählten Bereichs

Beispiel. Zeige alle Elemente auf der Zeile des markierten Bereichs innerhalb von `CurrentRegion`.

```
Sub Zeile()
    Dim GanzeZeile As Range
    Dim ZeilenElement As Range
    Dim ElementWert As Variant
    Dim Nmber As Long

    ' Bestimme die Zeilennummer innerhalb des Bereichs aus der Differenz
    ' der absoluten Tabellenadressen von aktiver Zelle und Bereichsanfang

    Number = ActiveCell.Row - ActiveCell.CurrentRegion.Row + 1

    Set GanzeZeile = ActiveCell.CurrentRegion.Rows(Number).Cells

    For Each ZeilenElement In GanzeZeile
        ElementWert = ZeilenElement.Value
        MsgBox CStr(ElementWert)
    Next
End Sub
```

3.8.6 `EntireColumn`- und `EntireRow`-Eigenschaften

Syntax
`EntireColumn` `EntireRow`
Liefert komplette Spalten bzw. Zeilen des ausgewählten Bereichs.

Beispiel. Markiere Spalte B.

```
Sub MarkColumnB()

    Range("B2").EntireColumn.Select

End Sub
```

3.8.7 `Offset`-Eigenschaft

Syntax	
`Offset(`*`rows, columns`*`)`	
Liefert ein `Range`-Objekt, das um eine Anzahl von Zeilen und Spalten gegenüber dem aktiven `Range`-Objekt verschoben ist	
Parameter	Verwendung
`rows` *`columns`*	Verschiebungsinkremente in Zeilen- und Spaltenrichtung

Beispiel. Ausgehend von Zelle B12 markiere Zelle G15.

```
Sub SelectOffset()
    Range("B12").Offset(3, 5).Select
End Sub
```

3.8.8 `Resize`-Eigenschaft

Syntax	
`Resize(`*`rows, columns`*`)`	
Verändert die Größe des Bereichs um eine beliebige Anzahl von Zeilen und Spalten	
Parameter	Verwendung
`columns` *`rows`*	Anzahl Spalten bzw. Zeilen

Beispiel. Markiere eine 3x3-Matrix, ausgehend von aktiver Zelle.

```
Sub MarkMatrix3x3()
    ActiveCell.Resize(3,3)
End Sub
```

3.8.9 `Value`-Eigenschaft

Syntax
`Value`
Standardeigenschaft, die den Wert einer Zelle zurückgibt. Muß nicht explizit genannt werden.

Beispiel.

```
Dim Wert As Variant

Wert = Range("C12").Value
Wert = Range("C12")          ' Zugriff ohne explizite Nennung von "Value"
```

3.8.10 `Select`-Methode

Syntax
`.Select`
Markiert den Bereich auf dem Tabellenblatt

Beispiel.

```
Range("A1:B4").Select
```

3.8.11 `Clear` und `ClearContents`-Methoden

Syntax
`Clear` `ClearContents`
Löscht Inhalt, Format und Kommentar einer Zelle Löscht Inhalt einer Zelle

3.8.12 Beispiel: Matrix einlesen, speichern und zurückschreiben

```
Sub MatrixInOut()

    ' Einlesen, Speichern und Zurückschreiben einer Matrix vom Typ Long

    Dim MatrixIn As Range
    Dim MatrixOut As Range
    Dim MatrixElement As Range
    Dim Rows As Long, Cols As Long
    Dim i As Long, j As Long

    Set MatrixIn = ActiveCell.CurrentRegion  ' Wähle Eingabebereich
    Rows = MatrixIn.EntireRow.Count          ' Zeilen und Spalten
    Cols = MatrixIn.EntireColumn.Count       ' des Eingabebereichs

                                             ' Speicher dimensionieren
    ReDim Matrix(1 To Rows, 1 To Cols) As Long

                                           ' Matrix einlesen und speichern
    i = 1: j = 1
    For Each MatrixElement In MatrixIn
        Matrix(i, j) = CLng(MatrixElement.Value)
        j = j + 1
        If j > Cols Then
            i = i + 1
            j = 1
        End If
    Next

                                ' Startzelle des Ausgabebereichs festlegen,
                                ' eine Zeile unterhalb des Eingabebereichs
    Set MatrixOut = _
                Cells(MatrixIn.Row, MatrixIn.Column).Offset(Rows + 1, 0)

                                ' Matrix ausgeben
    For i = 1 To Rows
        For j = 1 To Cols
            MatrixOut.Offset(i - 1, j - 1).Value = Matrix(i, j)
        Next j
    Next i

End Sub
```

3.9 Zeitgesteuerte Aufführung von Routinen

Mit der Methode `OnTime()` können Ereignisse zu einem bestimmten Zeitpunkt aktiviert werden. Während der Wartezeit gibt das Programm die Kontrolle an den Anwender zurück, d.h. der Computer ist freigegeben und man kann in der Zwischenzeit etwas anderes mit Excel machen.

Beispiel. Zeitgesteuerte Ampelschaltung. Vom Datenblatt wird die Dauer der verschiedenen Ampelphasen in Sekunden eingelesen. Die Zeitsteuerung startet mit der Rot-Phase. Dabei wird das Rotlicht angeschaltet und der "Timer" für die Gelb-Phase mit der Dauer der Rot-Phase gestartet. Die Gelb-Phase startet den Timer für die Grün-Phase, usw. Das Vorgehen wird im Programm-Code am besten erklärt.

```
Dim Zeitschaltung As Boolean     ' Projektvar.: Indikator für Zeitschaltung
                                 ' die Timer werden nur gestartet wenn der
                                 ' Indikator gesetzt ist.
Dim Scheduled                    ' Name der nächsten Phase

Sub ZeitSchalt_Start()           ' Zeitschaltung starten,
                                 ' mit Rot-Phase beginnen
    Dim phs As Long

    Zeitschaltung = True         ' Indikator setzen

    phs = Range("Rotphase").Value  ' Länge der Rot-Phase lesen
    Halt                           ' Ampel auf Rot schalten
                                   ' Name der nächsten Phase setzen
    Scheduled = "ZeitSchalt_Achtung_Fahrt"

                                   ' Timer für nächste Phase starten
    Application.OnTime _
        earliesttime:=Now + TimeValue("00:00:" & phs), _
        procedure:=Scheduled, _
        schedule:=True

End Sub

Sub ZeitSchalt_Achtung_Fahrt()      ' Rot-Gelb-Phase
    Dim phs As Long
                                    ' Prozedur wird nur ausgeführt,
                                    ' wenn Indikator gesetzt
    If Not Zeitschaltung Then Exit Sub

    phs = Range("Gelbphase").Value  ' Länge der Gelb-Phase lesen
    AchtungFahrt                    ' Ampel auf Rot-Gelb schalten
                                    ' Name der nächsten Phase setzen
    Scheduled = "ZeitSchalt_grün"

                                    ' Timer für nächste Phase starten
    Application.OnTime _
        earliesttime:=Now + TimeValue("00:00:" & phs), _
        procedure:=Scheduled, _
        schedule:=True

End Sub

Sub ZeitSchalt_grün()                   ' Grün-Phase
    Dim phs As Long
    Dim Richtung As Long

    If Not Zeitschaltung Then Exit Sub  ' Indikator ?

    phs = Range("Grünphase").Value      ' Länge der Grünphase
    Richtung = Range("Richtung").Value  ' Fahrtrichtung für Auto lesen
```

```
    Fahrt                                   ' Ampel auf Grün schalten

    Auto_fährt phs, Richtung                ' Auto für die Dauer der Grün-
                                            ' Phase fahren lassen
    Auto_Wenden                             ' Auto wenden

    ZeitSchalt_Achtung_Halt                 ' Gelb-Phase starten

End Sub
Sub ZeitSchalt_Stop()                       ' Zeitschaltung ausschalten

    Zeitschaltung = False       ' Indikator = False unterbindet
                                ' dass die nächste Prozedur gestartet wird,
                                ' damit ist die Prozesskette unterbrochen
End Sub
```

Nichtdargestellte Prozeduren sind entsprechend oder können als Makro aufgezeichnet werden (z.B. AutoWenden).

Aufgabe. Erstellen Sie die fehlenden Prozeduren.

4 Programmelemente

4.1 Programmaufbau

Sequentielle Programme. Traditionelle, klassische Art zu programmieren. Die Programme bestehen aus einem *Hauptprogramm*, das im *Stapelbetrieb* gestartet wird und die Kontrolle über den weiteren Programmablauf übernimmt. Das Hauptprogramm ruft Unterprogramme (Prozeduren und Funktionen) auf. Die sequentielle Programmierung für große Probleme, die längere Zeit unbeaufsichtigt bearbeitet werden müssen Stand der Technik.

Ereignis- und Objektorientierte Programme. Es existiert kein Hauptprogramm. Die Reihenfolge der Problembearbeitung wird vom Anwender gesteuert, indem er Ereignisse auslöst. Die Programme bestehen aus einer Zusammenstellung von Prozeduren und Funktionen in *Modulen*.

Elemente eines Moduls bzw. Programmes.

Modul 1	**Modul 2**
Deklarationsteil	**Deklarationsteil**

Moduloptionen
`Option Explicit`
`Option Base 1`
 	
Konstanten
`Const Konst_1 As Long = 145`
 	
Variablen
`Dim Var_1 As Double`
 	
Records
`Type Rec_1`
`  Element1 As String`
`  Element2 As Single`
`End Type`
	Moduloptionen
	`Option Explicit`
	`Option Base 1`
	
	Konstanten
	`Const Konst_1 As Long = 145`
	
	Variablen
	`Dim Var_1 As Double`
	
	Records
	`Type Rec_1`
	`  Element1 As String`
	`  Element2 As Single`
	`End Type`

Prozeduren	**Prozeduren**

`Sub Prozedurname(...)`
`  Deklarationen`
`  Anweisungen`
`  ...`
`End Sub`
	`Sub Prozedurname(...)`
	`  Deklarationen`
	`  Anweisungen`
	`  ...`
	`End Sub`

Funktionen	**Funktionen**

`Function Fktname(...) As` *`Typ`*
`  Deklarationen`
`  Anweisungen`
`  ...`
`  Fktname = ...`
`End Function`
	`Function Fktname(...) As` *`Typ`*
	`  Deklarationen`
	`  Anweisungen`
	`  ...`
	`  Fktname = ...`
	`End Function`

4.2 Prozeduren und Funktionen

Prozeduren und Funktionen werden bei Bedarf mit ihrem Namen aufgerufen und abgearbeitet. Funktionen liefern im Gegensatz zu Prozeduren einen Rückgabewert, der direkt zugewiesen werden kann. Die Rückgabe mit Prozeduren erfolgt indirekt über die Parameterliste. Funktionen könne deshalb innerhalb einer Anweisung verwendet werden.

Gültigkeitsbereiche.

Private Prozedur/Funktion	Kann nur innerhalb eines Moduls verwendet werden. Schlüsselworte: `Private Sub` *`ProzedurName`*`(...)` `Private Function` *`FktName(...)`* `As` *`Typ`*
Öffentliche Prozedur/Funktion	Kann von allen Modulen des Projektes verwendet werden. Voreinstellung. `Public Sub` *`ProzedurName`*`(...)` `Public Function` *`FktName(...)`* `As` *`Typ`*

4.3 Variablen

Variablen sind Programmelemente, die während eines Programmlaufes unterschiedliche Werte annehmen können. Für jede Variable wird Speicherplatz im Arbeitsspeicher des Computers reserviert. Die Größe des Speicherplatzes wird vom vereinbarten Datentyp der Variablen bestimmt. Variablen werden mit einem Namen angesprochen.

Gültigkeitsbereiche und Gültigkeitsdauer.

Art	Gültigkeitsbereich	Gültigkeitsdauer	Deklaration
Projekt-variable	Im gesamten Projekt. Können von überall angesprochen werden.	Während der gesamten Laufzeit. Inhalte bleiben erhalten.	Im Deklarationsteil eines Moduls. Schlüsselwort `Public`.
Modul-variable	Im gesamten Modul. Können von allen Prozeduren und Funktionen im Modul angesprochen werden. Andere Module haben keinen Zugriff.	Während der gesamten Laufzeit. Inhalte bleiben erhalten.	Im Deklarationsteil eines Moduls. Schlüsselwort `Private`.
Lokale Variable	Nur in der Prozedur/ Funktion, in der sie vereinbart wird. Anderen Prozeduren nicht zugänglich.	Nur solange, wie die Prozedur ausgeführt wird. Beim nächsten Aufruf ist der letzte Inhalt nicht mehr vorhanden.	Im Deklarationsteil einer Prozedur/Funktion. Schlüsselwort `Dim`.
Statische Variable	Wie lokale Variable.	Während der gesamten Laufzeit.	Im Deklarationsteil einer Prozedur/Funktion. Schlüsselwort `Static`.

4.4 Konstanten

Feste Zahlenwerte sollte man Konstanten zuweisen und nicht direkt als Zahl programmieren. Bei Änderungen muß dann nur die Konstantenzuweisung geändert werden. Konstanten werden in den Deklarationsteilen von Modul und Prozedur vereinbart. Der Wert einer Konstanten läßt sich während des Programmlaufes nicht ändern. Der Wert bleibt während des gesamten Programmlaufes erhalten.

Gültigkeitsbereiche.

Projekt-konstante	Definition im Deklarationsbereiches eines Moduls `Public Const` *`KonstantenName`* `As` *`Typ = Wert`*
Modul-konstante	Definition im Deklarationsbereiches eines Moduls `Private Const` *`KonstantenName`* `As` *`Typ = Wert`*
Lokale Konstante	Definition in Prozedur/Funktion `Const` *`KonstantenName`* `As` *`Typ = Wert`*

4.5 Kontrollstrukturen

Ein Programm besteht aus einer Folge von Anweisungen. In seiner einfachsten Form enthält ein Programm Anweisungen, die der Reihe nach ausgeführt werden (Folge). Anweisungen können jedoch auch mehrmals (Wiederholung) oder gar nicht, als Folge einer Auswahl, ausgeführt werden. Die Kontrollstrukturen *Folge*, *Wiederholung* und *Auswahl* sind die Hauptbestandteile eines Programmes.

Der strukturierte Entwurf eines Programmes ist Gegenstand des *Software-Engineering*. Beim Entwurf werden verschiedene Hilfsmittel und Methoden eingesetzt, um insbesondere die Kontrollstrukturen anschaulich darstellen und entwerfen zu können (siehe nächstes Kapitel).

4.5.1 Die Folge (Sequenz)

Eine Sequenz ist eine Folge von Anweisungen, die nacheinander und jede genau einmal abgearbeitet werden. Ein rein sequentieller Programmablauf ist die einfachste Struktur eines Programmes.

4.5.2 Die Auswahl (Alternative)

Der weitere Programmablauf ist von einer oder mehreren Bedingungen abhängig. Man unterscheidet:

Einseitige Auswahl. Nach einer Bedingungsabfrage werden Anweisungen ausgeführt oder nicht.

```
If Bedingung Then
    Anweisungen
    ...
End If
```

Zweiseitige Auswahl. In Abhängigkeit einer Bedingung wird eine von zwei alternativen Anweisungsfolgen abgearbeitet.

```
If Bedingung Then
    Anweisungsfolge 1
    ...
Else
    Anweisungsfolge 2
    ...
End If
```

Mehrstufige Auswahl. In Abhängig verschiedener Bedingungen wird eine von mehreren Anweisungsfolgen abgearbeitet.

```
If Bedingung 1 Then
    Anweisungsfolge 1
    ...
Else If Bedingung 2 Then
    Anweisungsfolge 2
    ...
Else If Bedingung 3 Then
    Anweisungsfolge 3
    ...
Else
    Anweisungsfolge n
    ...
End If
```

Mehrseitige Auswahl (Fallauswahl). Die mehrseitige Auswahl ist vom Wert einer Variable abhängig.

```
Select Case Selektor
    Case Auswahl 1
        Anweisungsfolge 1
    Case Auswahl 2
        Anweisungsfolge 2
    Case Auswahl 3
        Anweisungsfolge 3
    Case Else
        Anweisungsfolge n
End Select
```

Die mehrseitige ist der mehrstufigen Auswahl vorzuziehen, wenn eine einzige Variable auf unterschiedliche Werte abgefragt werden soll. Die mehrseitige Auswahl ist dann übersichtlicher.

Eine gut strukturierte Auswahl ist dadurch gekennzeichnet, daß

- Die Fälle nach ihrer Häufigkeit angeordnet sind. Der häufigste Fall steht an erster Stelle, der seltenste Fall am Ende.
- Der letzte `Else`-Zweig für unvorhergesehene Fälle oder Fehlermeldungen verwendet wird und somit eine korrekte Abarbeitung der Auswahl möglich ist.
- Die Anweisungen innerhalb von Auswahlblöcken möglichst kurz und einfach gehalten sind (Lesbarkeit!). Im Zweifel lieber ein Unterprogramm mehr schreiben.

4.5.3 Die Wiederholung (Iteration)

Wiederholungen werden auch als *Schleifen* bezeichnet. Eine Schleife besteht aus einer *Schleifensteuerung* und einem *Schleifenkörper*.

die Anzahl der Wiederholungen ist ...		
zahlenmäßig bekannt	von Bedingungen abhängig	
Zählergesteuerte Wiederholung	**Bedingte Wiederholung**	
	Kopfgesteuerte Wiederholung	**Fußgesteuerte Wiederholung**

4.5.4 Zählergesteuerte Wiederholung

```
For Zähler = Startwert To Endwert Step Schrittweite
    Anweisungen
    ...
Next Zähler
```

4.5.5 Bedingte Wiederholung

Kopfgesteuert. Solange die Bedingung am Kopf der Schleife erfüllt ist, wird die Schleife durchlaufen. Es kann sein, dass die Schleife nicht durchlaufen wird (*abweisende Schleife*).

```
Do While|Until Bedingung
    Anweisungen
    ...
Loop
```

Fußgesteuert. Nach dem Durchlaufen der Anweisungen im Schleifenkörper wird entschieden, ob die Schleife ein weiteres Mal abgearbeitet wird. Die Schleife wird in jedem Fall mindestens einmal abgearbeitet (*nicht abweisende Schleife*).

```
Do
    Anweisungen
    ...
Loop While|Until Bedingung
```

5 Software Engineering

Software Engineering ist die systematische Verwendung von Methoden und Werkzeugen zur Herstellung und Anwendung von Software mit dem Ziel der Rationalisierung bei gleichzeitiger Qualitätssteigerung.

Die Notwendigkeit des Software Engineering ergibt sich aus den Folgen der sich ständig erneuernden Technologien der Informationsverarbeitung, damit unter Zeit- und Kostendruck dennoch ein funktionstüchtiges und wirtschaftlich herstellbares Softwareprodukt entstehen kann.

Die Methoden des Software Engineering sind auch für kleinere Programmieraufgaben des Alltags sinnvoll und gewinnbringend einsetzbar. Sie helfen bei allen Phasen der Programmentwicklung:

- Problemanalyse und Anforderungs-Definition
- Programm-Entwurf
- Implementierung (Programmierung)
- Test der Software
- Installation der Software

5.1 Anforderungen an Software

Software muß im wesentlichen den beiden Anforderungen der *Benutzerakzeptanz* und der *Ausbaufähigkeit* genügen. Die beiden Kriterien lassen sich weiter unterteilen:

Benutzerakzeptanz	Ausbaufähigkeit
> Zuverlässigkeit	> Flexibilität; Anpassungsfähigkeit vorh. Programmteile an neue Anforderungen
> Korrektheit (der Erbegnisse)	> Adaptibilität; Ausbaufähigkeit durch Hinzufügen
> Robustheit; Ausweisung von ungültigen Eingaben, nicht erlaubter Ausgaben, Erkennen von Störungen	> Portabilität; unter verschiedenen Betriebs- und Hardwaresystemen ausführbar
> Effektivität; Realisierung der Benutzeranforderungen	
> Effizienz; in Ausführungszeit in Speicherbedarf	

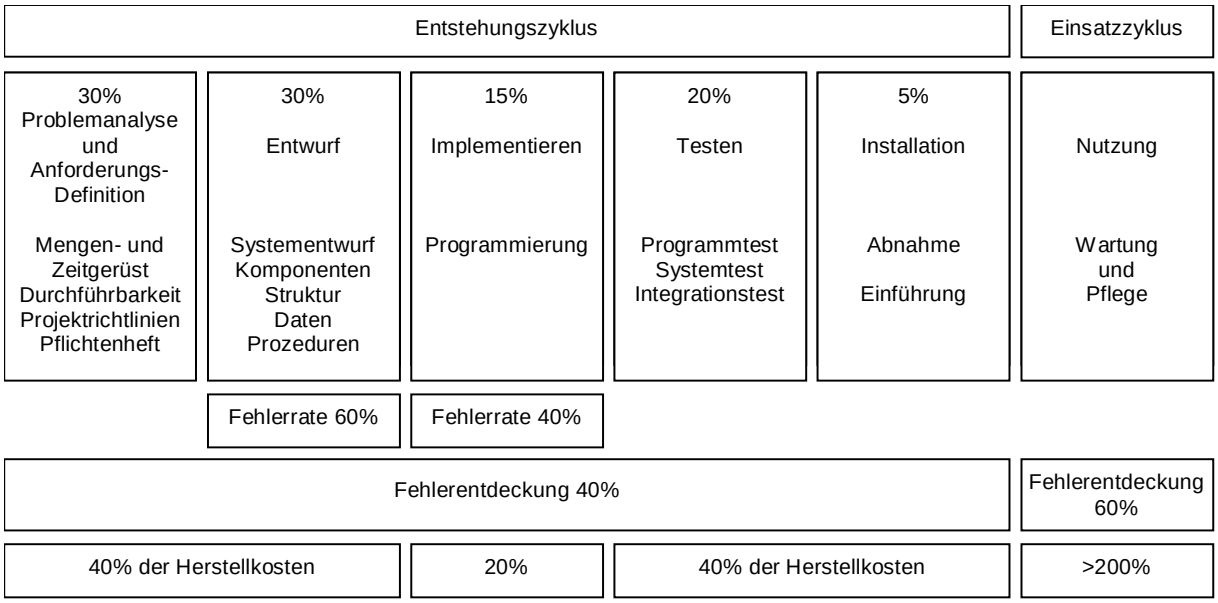
5.2 Software-Lebenszyklus

Software ist ein dynamisches Produkt. Insbesondere lehrt die Erfahrung:

- Programme sind nie fehlerfrei
- Programme ändern sich ständig
- Programme werden mit zunehmendem Alter unstrukturierter
- Ein Programmpaket wächst im Laufe seines Alters

Für die Zusammenhänge zwischen Aufwand, Kosten, Fehlerentstehung und Fehlerbeseitigung im Lebenszyklus eines Programmes gibt das folgende Diagramm Auskunft, das die Ergebnisse empirischer Untersuchungen zusammenfaßt. Bemerkenswert ist insbesondere, dass bereits in der Analysephase, also bereits **vor** der eigentlichen Programmierung bereits 40% der Kosten und 60% der Fehler entstehen. Von allen Fehlern werden in der Entstehungsphase in der Regel nur 40% gefunden, der überwiegende Rest erst während der Anwendung mit allen unangenehmen Konsequenzen für Anwender und Entwickler. Dagegen wächst der Aufwand für die Fehlerbeseitigung von Phase zu Phase.

Die Analyse dieser Zusammenhänge macht die Notwendigkeit einer methodischen und strukturierten Vorgehensweise bei der Software-Entwicklung deutlich. Die damit zusammenhängenden Techniken sind im Software-Engineering zusammengefaßt.



5.3 Prinzipien des Software-Engineering

Das Software-Engineering folgt der Vorstellung, ein komplexes Problem in einfacher zu verstehende Teilaufgaben aufzulösen, um damit möglichst umfassend und fehlerfrei behandeln zu können. Dieser Ansatz führt zu folgenden Prinzipien:

- **Prinzip der Modularisierung**
  Umfangreiche Probleme werden in kleinere, überschaubare und in sich geschlossene Teilbereiche zerlegt. Hochkomplexe Aufgabenstellungen werden schrittweise verfeinert, bis sie dem Erkenntnisumfang entsprechen.
- **Prinzip der hierarchischen Strukturierung**
  Die Abhängigkeiten der Teilaufgaben (Moduln) relativ zueinander sollen streng hierarchisch sein, d.h. von übergeordneten Begriffen zu untergeordneten führen *(Top-Down-Entwurf)*.
  Gegensatz: *Bottom-Up-Entwurf*. Hier werden Einzelaufgaben zu einer Gesamtaufgabe zusammengefaßt. "Vom Kleinen zum Großen". In der Praxis werden beide Methoden zusammen eingestzt, damit der Bezug zur Gesamtaufgabe nicht verloren geht.
  Up-Down-Entwurf: Bei dieser Strukturierungsmethode wird die Gesamtaufgabe top-down verfeinert und Teilaufgaben bottom-up vergröbert. Kritische Teilaufgaben können somit zuerst angegangen und getestet werden.
- **Prinzip der strukturierten Programmierung**
  Beschränkung auf möglichst wenige logische Beschreibungselemente (bei Ablaufstrukturen auf die drei Grundbausteine Folge, Auswahl und Wiederholung) und auf rein lineare Denkprozesse (ein Block nach dem anderen).

5.4 Methoden des Software-Engineering

Die genannten Prinzipien werden von verschiedenen Methoden umgesetzt. Spezielle Software (*CASE-Tools, Computer Aided Software Engineering*) unterstützen dies. U.a. kennt man folgende Methoden:

- **Hierarchische Funktionsgliederung**
  Die geforderten Funktionen werden streng hierarchisch nach Oberbegriffen gegliedert.
- **Entscheidungstabelle**
  Beschränkung auf die Prozedurstruktur "Auswahl".
- **Programmablaufplan**
  Enthält die drei logischen Ablaufstrukturen Folge, Auswahl und Schleife, jedoch ohne Zwang zur hierarchischen Strukturierung oder strukturierten Programmierung.
- **Struktogramm**
  Enthält die drei logischen Ablaufstrukturen wie der Programmablaufplan, aber mit dem Zwang zur strukturierten Programmierung und der Möglichkeit der hierarchischen Strukturierung.
- **Datenflußplan**
  Beschränkung auf die Beschreibung von Datenstrukturen, die zur Verarbeitung benötigt werden

oder als Ergebnis der Verarbeitung folgen.

> **SA/SADT** (Structured Analysis / Structured Analysis and Design Technique)
Hierarchisch gegliederte Strukturanalyse als Aktivitäten- oder Datenmodell. Grafische Darstellung der Zusammenhänge als Rechtecke mit Pfeilen.

> **Jackson Strukturierte Programmierung**
Hierarchisch gegliederter, sich an den Daten orientierender Programmentwurf.

> **Petri-Netz**
Zusätzliche Möglichkeit der Beschreibung nebenläufiger, dynamischer Prozesse.

5.5 Entscheidungstabelle

Die Entscheidungstabelle ist für die sichere Auswahl von Handlungsalternativen geeignet. Sie gestattet, die Voraussetzungen (Bedingungen = Wenn) zu formulieren, unter denen bestimmte Maßnahmen (Aktionen = Dann) zu ergreifen sind. Eine Entscheidungstabelle besteht diesen infolgedessen aus:

> **Wenn-Bereich**
In den Zeilen werden die Bedingungen beschrieben. In den Spalten der *Entscheidungsregeln* werden sie mit logisch UND verknüpft. Voraussetzungen können erfüllt sein (J, 1), nicht erfüllt (N, 0) oder irrelevant (—) sein.

> **Dann-Bereich**
In den Zeilen des Dann-Bereichs werden die Maßnahmen als Entscheidung für eine Aktion angegeben. Ausführen der Aktion: X; Nichtausführen: —; Mehrere Aktionen werden mit logisch UND verknüpft.

Entscheidungstabellen sind besonders dafür geeignet, komplexe Probleme mit vielen Voraussetzungen, Regeln und Aktionen klar zu strukturieren, um vor allem Widersprüche und Redundanzen zu identifizieren.

Redundanz. Führen unterschiedliche Regeln zu derselben Aktion, liegt Redundanz vor. Die Entscheidungstabelle wird *konsolidiert*, indem die redundanten Regeln zu einer Hauptregel zusammengefaßt werden.

Widerspruch. Führen gleiche Entscheidungsregeln zu verschiedenen Aktionen, liegt ein Widerspruch vor. Regeln und Voraussetzungen müssen überprüft werden.

SONST- oder ELSE-Regel. Diese Regel legt, welche sonstige Maßnahme zu ergreifen ist, wenn keine der formulierten Regeln zutrifft. Sie dient üblicherweise der Fehlerbehandlung.

Vorgehensweise. Es empfiehlt sich folgende Vorgehensweise:

1. Formulierung der Bedingungen (*Wenn*)
2. Formulierung der Aktionen (*Dann*)
3. Vollständige Formulierung der Regeln
   Spaltenweises Zuordnen der Regeln zu Aktionen
4. Prüfen der Tabelle auf Widerspruchsfreiheit
5. Prüfen auf Redundanz, ggf. konsolidieren

Beispiel. Verkehrsverhalten an einer Kreuzung mit Ampelregelung.

Bedingungen:	Ampel rot	
	Ampel gelb	
	Ampel grün	
	Ampel blinkt	
	Ampel ist defekt	
Aktionen:	Anhalten	
	Anfahren	
	Fahren	
	Verkehrszeichen beachten	
	Polizei verständigen	
Regeln	R1: rot	-> Halten
	R2: rot und gelb	-> Anfahren
	R3: gelb	-> Halten
	R4: grün	-> Fahren
	R5: Ampel blinkt	-> Verkehrszeichen beachten

R6: Ampel defekt -> Polizei verständigen

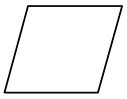
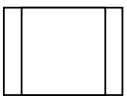
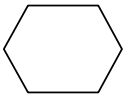
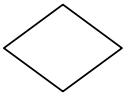
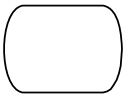
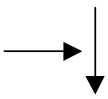
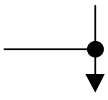
Daraus ergibt sich folgende Ereignistabelle. Sie ist frei von Widersprüchen und Redundanzen.

		Regeln					
		R1	R2	R3	R4	R5	Else
Wenn (Beding. B)	B1 Ampel rot	J	J	N	N	N	
	B2 Ampel gelb	N	J	J	N	N	
	B3 Ampel grün	N	N	N	J	N	
	B4 Ampel blinkt	N	N	N	N	J	
	B5 Ampel defekt	N	N	N	N	N	
Dann (Aktionen A)	A1 Halten	X	—	X	—	—	—
	A2 Anfahren	—	X	—	—	—	—
	A3 Fahren	—	—	—	X	—	—
	A4 Verkehrszeichen beachten	—	—	—	—	X	—
	A5 Polizei verständigen	—	—	—	—	—	X

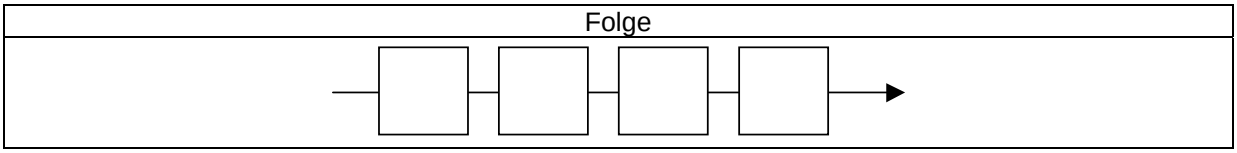
5.6 Programmablaufplan nach DIN 66001, PAP

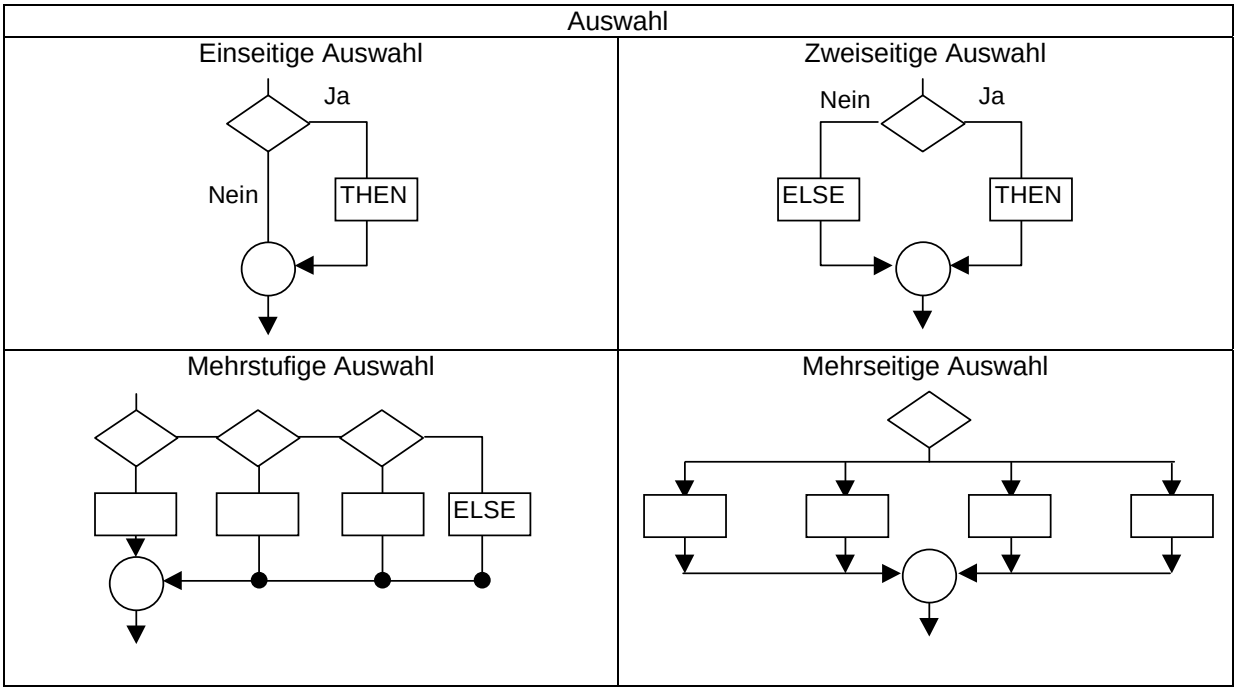
PAP beschreibt mit Sinnbildern symbolisch den Ablauf oder die Reihenfolge logischer Operationen. Der Zusammanhang wird durch Ablauflinien dargestellt.

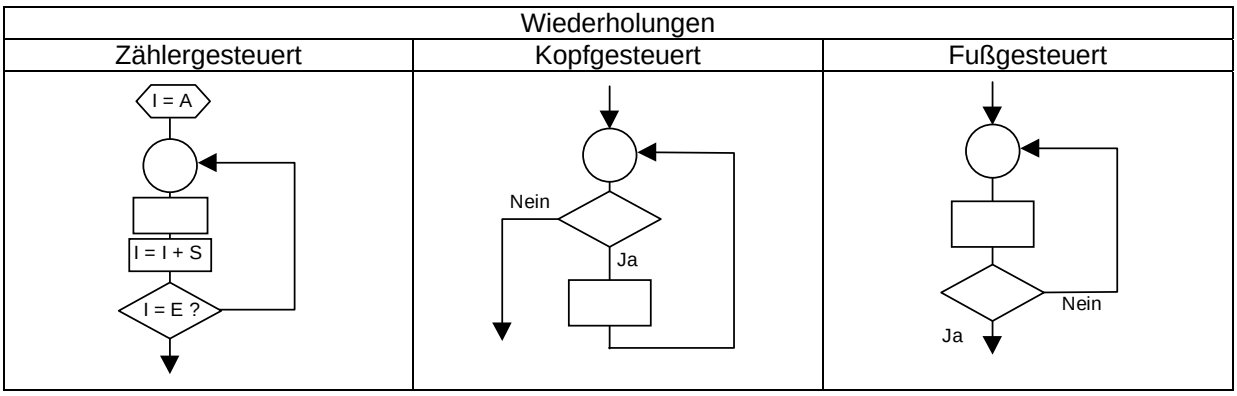
Es werden u.a. folgende Sinnbilder verwendet:

Sinnbild	Bedeutung	Bemerkungen
	Eingabe, Ausgabe	Darstellung der Ein- und Ausgabe durch externe Geräte.
	Operation	Allgemeine Operationen, häufig für arthmetische Rechen-operationen.
	Unterprogramm	Ein in sich geschlossenes Programmteil. Detailierung an anderer Stelle.
	Programm-modifikation	Vorbereitung von Operationen, z.B. Änderung von Zähler-zuständen, Initialisierungen.
	Verzweigung	
	Grenzstelle	Anfang, Ende oder Zwischenhalt.
	Übergangs- oder Verbindungsstelle	Der Übergang kann von meheren Stellen aus, aber nur zu einer Eingangsstelle hin erfolgen.
	Ablauflinien	Vorzugsrichtungen: von oben nach unten, von links nach rechts.
	Zusammenführung	Zwei sich kreuzende Ablauflinien bedeuten keine Zusam-menführung.

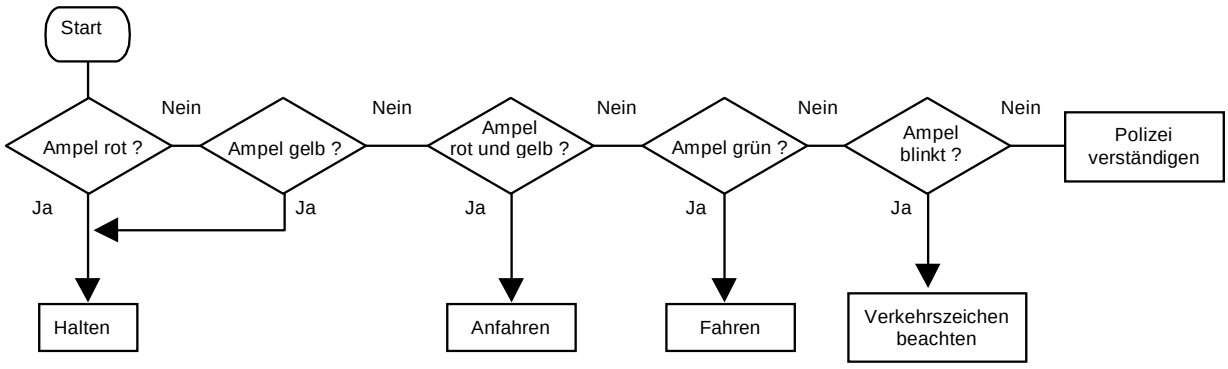
Die elementaren logischen Ablaufstrukturen:







Beispiel. Ampelschaltung.



5.7 Struktogramm und Pseudocode

Das Ziel der strukturierten Programmierung ist eine zuverlässige, lesbare und wartungsfreundliche Software. Dies geschieht durch die Beachtung folgender Vorschriften:

- **>** Hierachische Problemgliederung mit der Möglichkeit der schrittweisen Verfeinerung (*Top-Down Entwurf*).
- **>** Beschränkung der elementaren logischen Ablaufstrukturen auf die oben angeführten Varianten der Strukturen *Folge, Auswahl* und *Wiederholung*. Logische Trennung zwischen Auswahl und Wiederholung.
- **>** Zweipoligkeit der Reihung der elementaren logischen Ablaufstrukturen (ein Eingang - ein Ausgang).

Die von dieser Konzeption gekennzeichnete Ablauflogik wird *Struktogramm* genannt und ist nach DIN 66261 genormt.

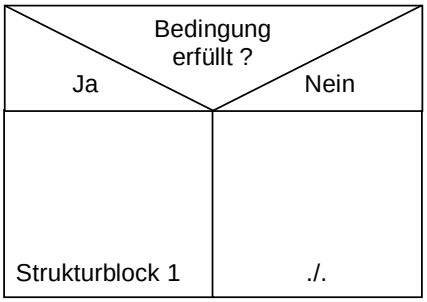
Der *Pseudocode* setzt die garafischen Struktogrammsymbole in Sprache um. Er formuliert die logischen Abläufe im Sinne der strukturierten Programmierung in der Weise mit programmsprachenähnlichen Begriffen, dass sie fast codierbar sind. VBA ist eine strukturierte Programmiersprache.

Strukturblöcke. Struktogramme sind aus logischen Einheiten, den *Strukturblöcken* aufgebaut. Sie bestehen lediglich aus den drei elementaren Ablaufstrukturen *Folge, Auswahl* und *Wiederholung*. Die Strukturblöcke werden von oben nach unten durchlaufen.

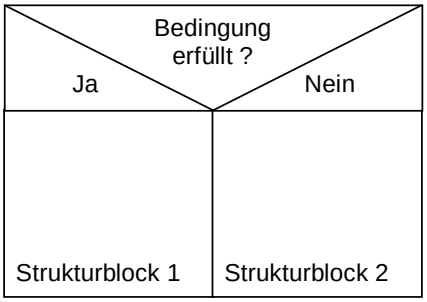
Prinzip der Zweipoligkeit. Strukturblöcke dürfen sich nicht überlappen. Jeder Strukturblock ist in sich geschlossen und befindet sich entweder völlig außerhalb oder völlig innerhalb anderer Strukturblöcke. Die wichtigste Regel ist: **Jeder Strukturblock hat nur einen Eingang und einen Ausgang.** Das Prinzip der Zweipoligkeit verhindert das Springen zwischen nicht aneinander grenzenden Strukturblöcken (das fehleranfällige Merkmal einer nicht-strukturierten Programmierung). Das hat zur Folge, dass nicht alle Flußdiagramme eindeutig in Struktogramme umgewandelt werden können.

Verwendete Symbole und elementare Strukturblöcke im Struktogramm und im Pseudocode.

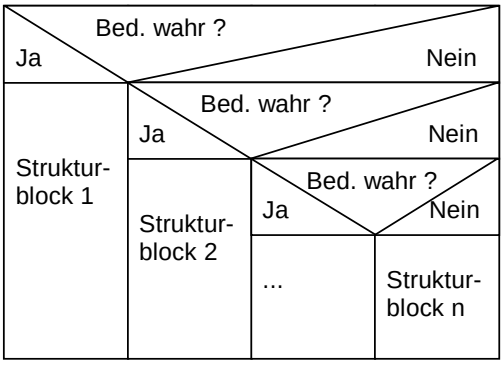
Folge	
Struktogramm	Pseudocode
1. Strukturblock 2. Strukturblock ... Letzter Strukturblock	1. Anweisung 2. Anweisung ... ... letzte Anweisung

Einseitige Auswahl	
Struktogramm	Pseudocode
	IF *Bedingung erfüllt* THEN *Ja-Anweisung* ENDIF

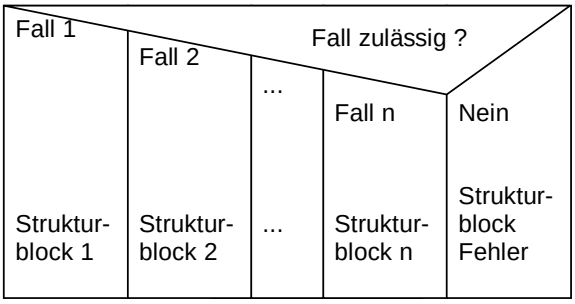
Zweiseitige Auswahl

Struktogramm	Pseudocode
	IF *Bedingung erfüllt* THEN *Ja-Anweisung* ELSE *Nein-Anweisung* ENDIF

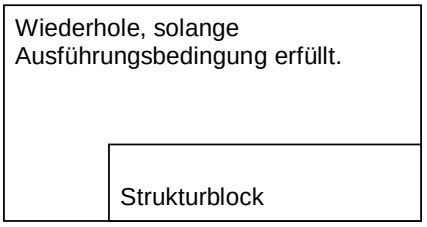
mehrstufige Auswahl

	IF *Bedingung1 erfüllt* THEN
	Anweisungen 1
	ELSE IF *Bedingung2 erfüllt* THEN
	Anweisungen 2
	ELSE IF ... THEN
	...
	ELSE
	Anweisungen n
	ENDIF

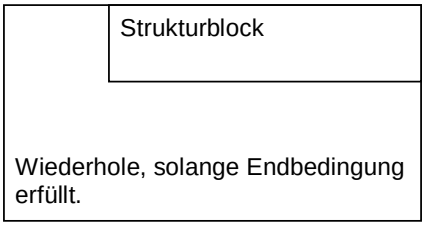
mehrseitige Auswahl

	CASE *Ausdruck* OF Fall 1: *Anweisungen 1* Fall 2: *Anweisungen 2* ... Fall n: *Anweisungen n* OTHERCASE ENDCASE

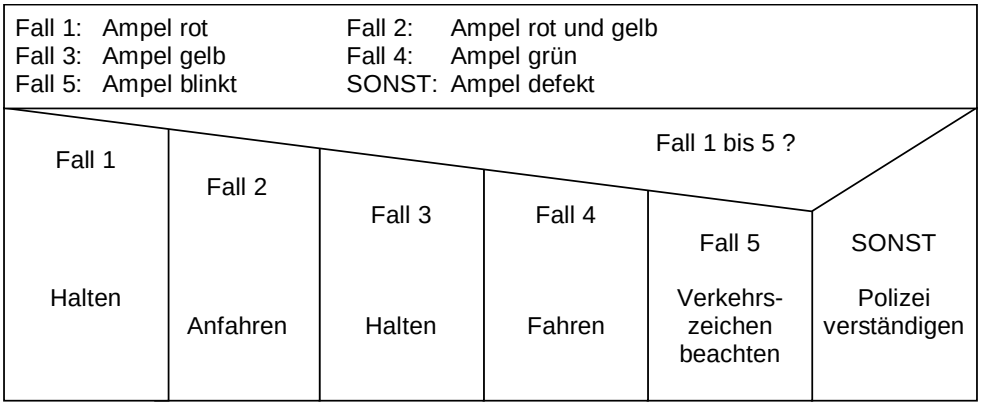
Kopfgesteuerte Wiederholung (abweisende Schleife)

	DO WHILE *Bedingung erfüllt* *Anweisungen* ... LOOP

Fußgesteuerte Wiederholung (nicht abweisende Schleife)

	DO *Anweisungen* ... LOOP WHILE *Bedingung erfüllt*

Beispiel. Verkehrsverhalten an der Ampel.

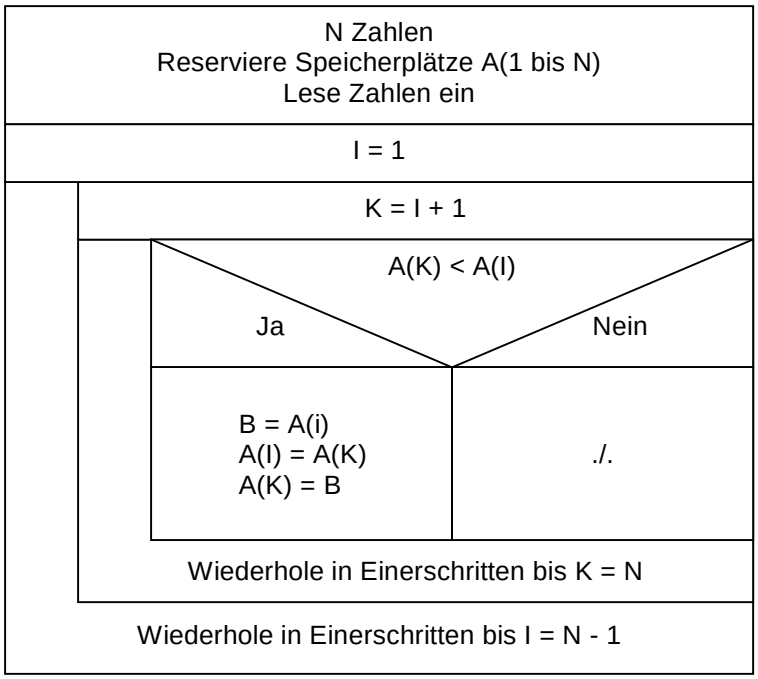


Beispiel. Sortieren von Zahlen nach ihrer Größe: von klein nach groß.
Erläuterung des Sortiervorgangs an einem Beispiel mit 6 Zahlen mit der Ausgangsreihenfolge A(i) = {100, 11, 10, 15, 1, 7} auf den Positionen i = {1, 2, 3, 4, 5, 6}. Das Problem wird von zwei ineinander geschachtelten Schleifen gelöst. Die äußere Schleife (Index i) geht der Reihe nach jede Position von links nach rechts durch. In der zweiten Schleife (Index k) werden alle rechts der Zahl A(i) stehenden Zahlen A(k) darauf überprüft, ob sie kleiner sind als A(i). Ist dies der Fall, werden die Zahlen umkopiert. Dieser Vorgang wird für die ersten 5 Positionen wiederholt, dann ist die Sortierung abgeschlossen.

Veranschaulichung der Sortierabfolge:

i	k	Reihenfolge					
Ausgangsfolge		100	11	10	15	1	7
1	2	11	100	10	15	1	7
	3	10	100	11	15	1	7
	4	10	100	11	15	1	7
	5	1	100	11	15	10	7
	6	1	100	11	15	10	7
2	3	1	11	100	15	10	7
	4	1	11	100	15	10	7
	5	1	10	100	15	11	7
	6	1	7	100	15	11	10
3	4	1	7	15	100	11	10
	5	1	7	11	100	15	10
	6	1	7	10	100	15	11
4	5	1	7	10	15	100	11
	6	1	7	10	11	100	15
5	6	1	7	10	11	15	100

Das Struktogramm:



Der Pseudocode:

```
Speicherplatz reservieren
Einlesen
I = 1
DO UNTIL I = N - 1
    K = I + 1
    DO UNTIL K = N
        IF A(K) < A(I)
        THEN  Tausche A(I) mit A(K)
    END DO
END DO
```

Aufgabe. Programmieren Sie den Sortieralgorithmus in VBA. Benutzen Sie die `Range`-Methoden, um eine beliebig große Anzahl von Zahlen einzulesen, die in einer Spalte auf dem Tabellenblatt angeordnet sind. Geben Sie das Ergebnis auf dem Tabellenblatt aus.

6 Literatur

Ulrich Cluber	*VBA-Programmierung in EXCEL 97.* ECON Taschenbuch Verlag, Düsseldorf, München, 1998.
Patrizia S. Prudenzi	*VBA mit Excel 97 lernen.* 1. Auflage, Addison-Wesley, Bonn, 1997.
Peter Becker	*VBA 5, Schnellübersicht.* Markt & Technik, Haar bei München, 1997.
Michael Kofler	*VBA-Programmierung mit Excel 97.* Addison-Wesley, Bonn, 1997.
RRZN	*Excel 97 - Automatisierung, Programmierung.* 2. Auflage, Regionales Rechenzentrum für Niedersachsen, Universität Hannover, 1998; © Herdt-Verlag, Nackenheim, 1998.
Gerhard Brosius et.al.	*VBA-Programmierung.* 1. Auflage, Hrsgb.: Herdt-Verlag, Nackenheim, © Addison Wesley, Bonn, 1998.
Ekbert Hering	*Software Engineering.* Dritte Auflage, Vieweg, Braunschweig / Wiesbaden, 1992.