Harald Nahrstedt

Die Welt der VBA-Objekte

Was integrierte Anwendungen leisten können

Springer Vieweg

Die Welt der VBA-Objekte

Harald Nahrstedt

Die Welt der VBA-Objekte

Was integrierte Anwendungen leisten können

Springer Vieweg

Harald Nahrstedt
Möhnesee, Deutschland

ISBN 978-3-658-13890-5 ISBN 978-3-658-13891-2 (eBook)
DOI 10.1007/978-3-658-13891-2

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

Springer Vieweg

Lektorat: Thomas Zipsner

Gedruckt auf säurefreiem und chlorfrei gebleichtem Papier.

Springer Vieweg ist Teil von Springer Nature
Die eingetragene Gesellschaft ist Springer Fachmedien Wiesbaden GmbH

Vorwort

Warum dieses Buch

Als Dozent und Autor habe ich vielen Office-Anwendern die VBA-Entwicklungsumgebung, die in allen Anwendungen vorhanden ist, näher gebracht. Dabei ist mir immer wieder aufgefallen, dass die meisten Anwender zwar die Anwendung gut beherrschen und auch VBA-Prozeduren schreiben können, doch von Objekten und deren Handhabung wenig verstehen. Besonders fällt mir dies in Internet-Foren auf, in denen Anwender ihre VBA-Programmierprobleme schildern und auf Hilfe hoffen. Die Antworten werden dann oft von, manchmal auch ungeduldigen, sogenannten Experten geliefert, denen es an der objektbedingten Disziplin mangelt und sie damit mehr schaden als nutzen. Das Interesse an einer eigenen Weiterentwicklung von Prozeduren geht dadurch verloren. Dabei ist deren Einsatz in allen Office-Anwendungen eine wunderbare Möglichkeit zur Erweiterung der Funktionalität, besonders wenn eine Anwendung die Vorteile anderer Anwendungen nutzt und sie in die eigene Funktionalität integriert. So kommen die entscheidenden Vorteile der anderen Anwendungen erst richtig zur Geltung.

Kenntnisse in VBA setze ich bei diesem Buch voraus. Eine Einführung, die auch für Autodidakten sehr geeignet ist, enthält [1]. Dieses Buch führt in die Welt der Objektorientierten Programmierung (OOP) ein, insbesondere in die der Office-Anwendungen. Es werden Methoden der Entwicklung ebenso wie kleine Anwendungsbeispiele gezeigt. Interaktionen zwischen den Anwendungen sollen dem Leser Anreiz auf die eigene Entwicklung geben. Die Beispiele sind kurz gehalten und zeigen die ersten grundlegenden Schritte. Mein besonderer Appell an den Leser gilt einem sauberen strukturierten Programmierstil. Übersichtlichkeit und geringe Fehlerquoten sind das Ergebnis. Auf Kommentare in den Prozeduren habe ich verzichtet, um sie klein und übersichtlich zu halten. Außerdem soll sich der Leser die Zeit nehmen, alle Anweisungen zu studieren, um ihre Wirkungen zu verstehen. Dem Programmierer, der dann umfangreichere Prozeduren schreibt, empfehle ich von Kommentaren reichlich Gebrauch zu machen.

Nach wie vor weigere ich mich auch, die Prozeduren in VBA als Makros zu bezeichnen. In der Welt der Informatik werden logisch zusammenhängende und abgeschlossene Anweisungen als Prozeduren oder Funktionen bezeichnet. Die Programmierung unter

VBA hat daran einen verschwindend geringen Anteil. Als Makros lasse ich die Makros unter Access gelten, die in Form von umgangssprachlichen Anweisungssätzen geschrieben werden.

Versionen
Alle Anwendungen sind mit Office Version 2010 erstellt worden. Da dieses Buch die Methodik der Objektnutzung lehrt, sollten die Änderungen in den Objektstrukturen auch vom Leser erkennbar und umsetzbar sein. Dies gilt sowohl für vergangene wie auch für zukünftige Versionen. Da es sehr viele Kombinationen von Hard- und Software auf dem Markt gibt kann ich nicht garantieren, dass alle dargestellten Programm und Programmteile immer problemlos funktionieren. Ebenso kann ich keine Haftung für irgendwelche Folgen übernehmen, die sich aus dem Einsatz der Programmcodes ergeben. Ich bin mir jedoch sicher, dass alle vorgestellten Beispiele ohne große Änderungen einsetzbar sind.

Zum Aufbau
Das erste Kapitel befasst sich mit den Grundlagen der objektorientierten Entwicklung. Angefangen von der modellbasierten Sicht bis hin zu Objekten in der Entwicklungsumgebung von Office, die für alle Anwendungen Gültigkeit haben. In den weiteren Kapiteln werden dann die wichtigsten Objekte von Office-Anwendungen entsprechend dem zugrunde liegenden Objektmodell besprochen. Am Ende eines jeden Kapitels stehen einfache Interaktions-Beispiele zur eigenen Weiterentwicklung. Ich habe die nach meiner Meinung wichtigsten Anwendungen gewählt, um den mir gesetzten Rahmen nicht zu überschreiten. Das Buch hätte leicht den doppelten Umfang annehmen können, wenn man die Interaktionen mit Anwendungen außerhalb der Office-Welt dazu nimmt.

Danksagung
Ich danke all denen im Hause Springer Vieweg, die stets im Hintergrund wirkend, zum Gelingen dieses Buches beigetragen haben. Ein besonderer Dank gilt meinem Lektor Thomas Zipsner, der großen Einfluss auf Form und Inhalt dieses Buches genommen hat.

An den Leser
Dieses Buch soll auch zum Dialog zwischen Autor und Leser auffordern. Daher finden Sie sowohl auf der Homepage des Verlages www.springer-vieweg.de, wie auch auf der Homepage des Autors www.harald-nahrstedt.de ein Forum für ergänzende Programme, Anregungen und Kommentare.

Möhnesee, April 2016 Harald Nahrstedt

Inhaltsverzeichnis

Grundlagen

1

In diesem ersten Kapitel stelle ich die Grundlagen für eine modellbasierte Entwicklung vor, die für eine Objektorientierte Programmierung (OOP) unerlässlich sind und den Entwicklungsaufwand deutlich verringern. Die Programmstrukturen werden klarer und übersichtlicher, schon wegen der Verwendung von Objekten zur Abbildung realer Prozesse.

1.1 Modellbasierte Entwicklung

Wenn wir von modellbasierter Entwicklung sprechen, dann bilden wir Prozesse mit Objekten der realen Welt in unseren Programmen ab. Dabei benutzen wir nur die Eigenschaften und Methoden dieser Objekte, die zur Lösung unseres Projekts erforderlich sind. Wir erschaffen Modelle die eine Teilmenge der realen Welt darstellen. Wenn zum Beispiel eine Person in ein Krankenhaus kommt, dann wird zur Verwaltung ihres Aufenthalts das Modell Patient genutzt. Darin sind alle Daten enthalten die für den Krankenhausaufenthalt notwendig sind. Die Krankengeschichte (Anamnese), Name und Wohnort, Behandlungsdaten, Abrechnungsdaten und vieles mehr. Im Modell Patient sind nicht enthalten, welche Vorlieben die Person hat, welchen Hobbys sie nachgeht, wer zu ihrem Bekanntenkreis gehört, ihren Lebenslauf und so weiter. Es würde wahrscheinlich jeden Datenspeicher sprengen, wollte man alle ihre Daten erfassen wollen. Das Modell Patient ist eben nur eine Teilmenge des realen Objekts Person und die reicht für den Zweck der Behandlung vollkommen aus.

1.1.1 Die Klasse Aufgabe

Als einfaches Beispiel betrachten wir einmal das Objekt *Aufgabe*, von denen wir tagtäglich einige zu bewältigen haben. Tab. 1.1 zeigt eine typische Aufgabenliste in Kurzform.

H. Nahrstedt, *Die Welt der VBA-Objekte*, DOI 10.1007/978-3-658-13891-2_1

Tab. 1.1 To-Do-Liste

Aufgabe	Dringlichkeit	Zeitaufwand
Telefonat Müller	1	10 Min.
E-Mail Schmidt	4	15 Min.
MeetingPlan erstellen	2	2 Std.
Forum vorbereiten	3	4 Std.

Von all den dargestellten Aufgaben schauen wir uns die markierte Aufgabe etwas genauer an. Wir haben ein Objekt, nämlich eine konkrete individuelle Aufgabe (Tab. 1.2) die es zu erledigen gilt.

Tab. 1.2 Ausgewählte Aufgabe

Aufgabe	Dringlichkeit	Zeitaufwand	Datum	Uhrzeit
MeetingPlan erstellen	2	2 Std.	13.5.	14.00

Zu dieser Aufgabe gibt es noch eine abstrakte Darstellung in der Form, in der alle notwendigen Attribute zur Erledigung der Aufgabe angegeben werden, etwa wie in Abb. 1.1.

Abb. 1.1 Abstrakte Darstellung einer Aufgabe

```
Aufgabe
Titel = "MeetingPlan erstellen"
Dringlichkeit = 2 (wichtig)
Zeitaufwand = 2 h
Manager = "Herr Müller"
Termin = 13.5.
Uhrzeit = 14.00
Raum = "Venedig"
Agenda = Datei "C:\Planung\Meeting_13.5.txt"
Adressen = Datei "C:\Planung\Team.txt"
...
```

Auf der einen Seite gibt es einen Bauplan mit allen Angaben zur Aufgabe bis hin zur letzten Kleinigkeit, nach dem fast unendlich viele Aufgaben des gleichen Typs gestaltet werden können. Auf der anderen Seite gibt es eine abstrakte Darstellung der Aufgabe mit den Angaben die zur Durchführung notwendig sind (Abb. 1.2). Eine Aufzählung der Eigenschaften mit der Angabe des Datentyps für die Wertangabe.

Abb. 1.2 Attribute: Variable mit Typangabe

```
Aufgabe
Titel: Text
Dringlichkeit: Ganzzahl
Zeitaufwand: Ganzzahl
Planungstyp: Ganzzahl
Planungsdatei: Text
Manager: Text
Datum: Datumswert
Uhrzeit: Zeitwert
```

Die Angaben wie Raum, Agenda und Adressen sind nicht für jede Aufgabe relevant. Sie werden in eine Textdatei mit Struktur je nach Aufgabentyp ausgelagert. So lassen sich für andere Aufgaben andere Strukturen definieren.

1.1.2 Das Klassendiagramm

Werden nun noch die Methoden aufgeführt, wie diese Daten zu nutzen sind, so haben wir ein Modell *Aufgabe* für unsere Arbeitsorganisation erstellt. Eine grafische Darstellung unseres Modells in der Notation von UML (Unified Modeling Language) zeigt Abb. 1.3.

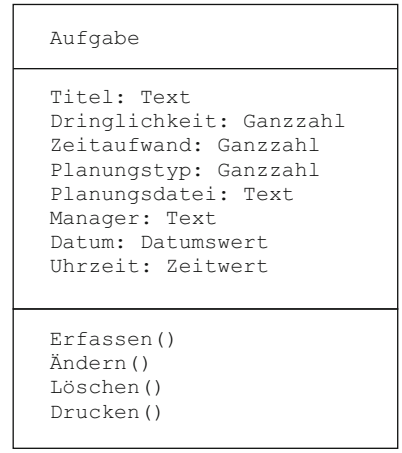


Abb. 1.3 Die Klasse Aufgabe

Die wichtigsten Klassendiagramm-Elemente stehen im Anhang 1, Tab. A1.1.

Ein solcher Bauplan eines Modells wird in der objektorientierten Programmierung (kurz OOP) als *Klasse* bezeichnet. Die Darstellung nach UML in Abb. 1.3 zeigt im ersten Feld den Klassennamen. Im zweiten Feld werden alle Eigenschaften (Attribute) der Klasse angegeben. Im dritten Feld folgen dann die Methoden (Member), die mit der Angabe möglicher Parameter in runden Klammern ( ) folgen.

1.1.3 Instanziierung

Wird eine Aufgabe der Liste hinzugefügt, dann bekommen die Attribute der Klasse Werte zugewiesen (Abb. 1.4) und wir haben ein eindeutiges Objekt. In der Programmierung sprechen wir von *Instanziierung* eines Objekts. Auf diese Art können mehrere Objekte instanziiert und mit den Methoden der Klasse genutzt werden.

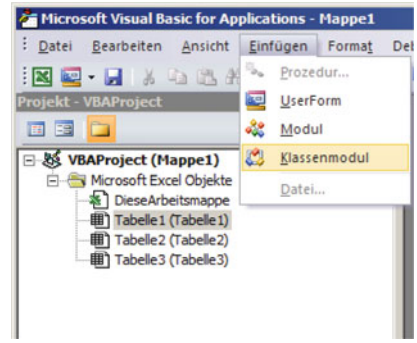
Abb. 1.4 Instanzspezifikation einer Aufgabe

```
ORG-13-05-ABC-12: Aufgabe

Titel: MeetingPlan erstellen
Dringlichkeit: 2
Zeitaufwand: 2 h
Planungstyp: 1
Planungsdatei: C:\Planung\Meeting_13.5.txt
Manager: Herr Müller
Termin: 13.05.
Uhrzeit: 14.00
```

1.1.4 Klassenmodule in VBA

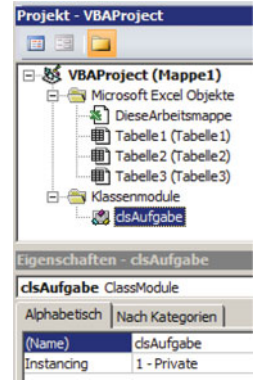
In VBA wird eine Klasse durch das Anlegen eines Klassenmoduls erzeugt. Dazu wird in der Entwicklungsumgebung unter dem Register *Einfügen* in der Auswahl das Klassenmodul (Abb. 1.5) gewählt.

Abb. 1.5 Einfügen eines Klassenmoduls im Projektexplorer



Über das Eigenschaftsfenster erhält das Klassenmodul einen sinnvollen Namen (Abb. 1.6), hier den Namen *Aufgabe*. Da es sich ebenfalls um ein Objekt handelt, be-

Abb. 1.6 Die Klasse Aufgabe anlegen



kommt es das Präfix *cls* für Klasse. Sie können aber auch ein beliebig anderes wählen, nur sollte es aussagekräftig sein und möglichst nicht oft geändert werden.

Im Code-Fenster der Klasse *clsAufgabe* werden die Eigenschaften und Methoden der Klasse definiert (Code 1.1). Der Bearbeiter, der Starttermin und die Uhrzeit werden wahrscheinlich am Anfang noch nicht bekannt sein, sie müssen später festgelegt werden.

Code 1.1 Die Klasse Aufgabe

```
Dim sKey       As String
Dim sTitel     As String
Dim iStufe     As Integer
Dim iZeit      As Integer
Dim iTyp       As Integer
Dim sDatei     As String
Sub Erfasse(Key, Titel, Stufe, Zeit, Typ, Datei)
   sKey = Key
   sTitel = Titel
   iStufe = Stufe
   iZeit = Zeit
   iTyp = Typ
   sDatei = Datei
End Sub
```

1.1.5 Die Klasse Person

Doch was nutzen die vielen definierten Aufgaben, wenn es niemanden gibt der sie erledigt. Womit wir auch schon bei der nächsten Klasse sind, der Klasse *Person* (Code 1.2).

Code 1.2 Die Klasse Person

```
Dim sKey          As String
Dim sName         As String
Dim sVorname      As String
Dim sAbteilung    As String
Sub Erfasse(Key, Name, Vorname, Abteilung)
   sKey = Key
   sName = Name
   sVorname = Vorname
   sAbteilung = Abteilung
End Sub
```

1.1.6 Assoziation

Beide Klassen *Aufgabe* und *Person*, bzw. ihre Objekte treten nun irgendwie in eine Beziehung. Im einfachsten Fall erledigt eine Person eine Aufgabe, besser gleich mehrere. Die Beziehung der Klassen wird als *Assoziation* bezeichnet und ebenfalls im Klassendiagramm (Abb. 1.7) als Linie zwischen den Klassen dargestellt.

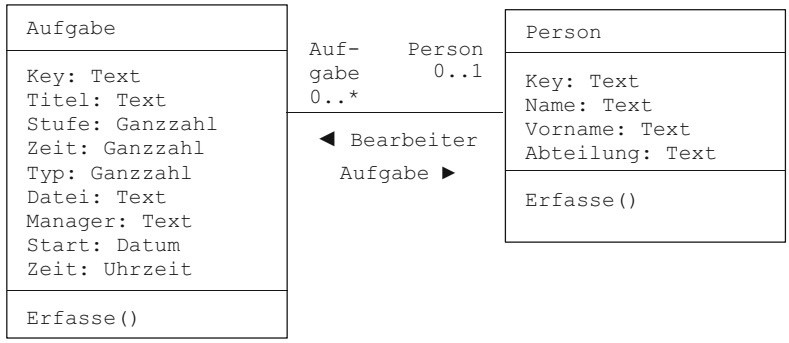


Abb. 1.7 Eine Assoziation der Klassen Aufgabe und Person

An den Enden der Linie werden häufig *Multiplizitäten* vermerkt. Sie geben an, wie viele Objekte einer Klasse in *Assoziation* zu den Objekten der anderen Klasse stehen. Im Beispiel muss eine Person nicht unbedingt eine Aufgabe erledigen, sie kann auch keine bis mehrere bearbeiten. Das wird durch die Angabe 0..* ausgedrückt und meint von Null bis beliebig viele. Oft wird auch nur * angegeben. Eine Aufgabe kann keinen oder einen Bearbeiter haben, was mit 0..1 gekennzeichnet wird. Eine Gruppe von Personen als Bearbeiter sieht unser Modell also zunächst einmal nicht vor. Dies nur der Einfachheit halber.

Oft werden zu den Multiplizitäten auch *Rollennamen* notiert, wie in diesem Fall *Bearbeiter* und *Aufgabe*. Auch die Assoziation selbst kann für die Richtung des Zugriffs auf die andere Instanz einen Namen haben. Die Richtung selbst wird durch einen kleinen Pfeil angedeutet.

1.1.7 Entity-Relationship-Model

Im Rahmen einer semantischen Datenmodellierung wird auch gerne ein *Entity-Relationship-Model* (kurz ERM) zur Darstellung von Objekt-Beziehungen benutzt. Das ERM erlaubt eine Typisierung der Objekte, ihrer Attribute und Beziehungen. Reale Objekte werden als *Entitäten*, ihre Eigenschaften als *Attribute* und ihre Beziehungen als *Relationship* bezeichnet. Alle Entitäten verfügen über einen eindeutigen Schlüssel (gelbe Formen).

Die wichtigsten ERM-Elemente stehen im Anhang 1, Tab. A1.2.

Das ERM für Aufgabe und Person zeigt Abb. 1.8. Solche Modelle bilden das Design für relationale Datenbanken. In Excel lassen sich Entitäten den *Worksheets* und Attribute den

Spalten der entsprechenden *Worksheets* zuordnen. Über die Methode *MSQuery* können dann Beziehungen definiert werden. Doch dies steht in einem anderen Buch [3].

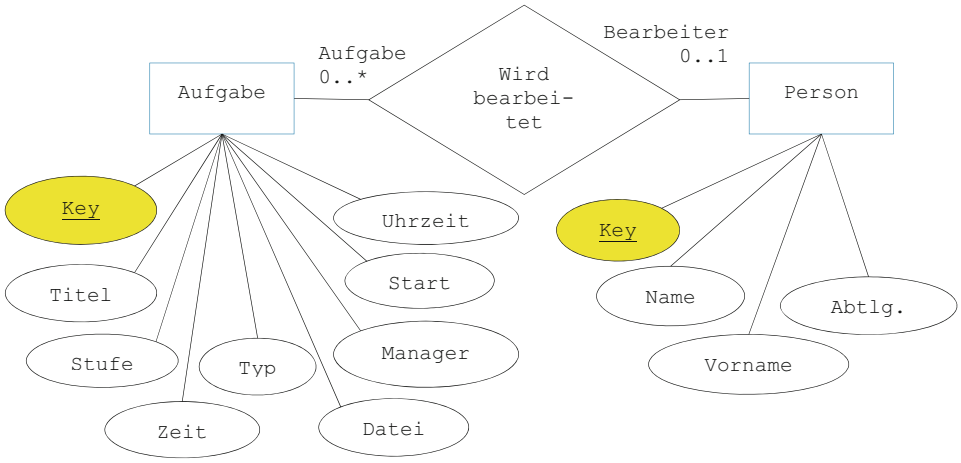


Abb. 1.8 Das Entity-Relationship-Model für Aufgabe und Person

Aus dem ERM wird anschaulich klar, dass über die Aufgabe und die Relation schnell auf den Bearbeiter zugegriffen werden kann, wenn es ihn gibt. Umgekehrt wird es schwieriger, da eine Person mehrere Aufgaben zur Bearbeitung bekommen kann. Die Entität Person müsste also mehrere Attribute *Aufgaben.Key* besitzen, doch wie viele? Die Einführung einer weiteren Entität *Planung* löst das Problem (Abb. 1.9). Den Code der Klasse *Planung* zeigt Code 1.3.

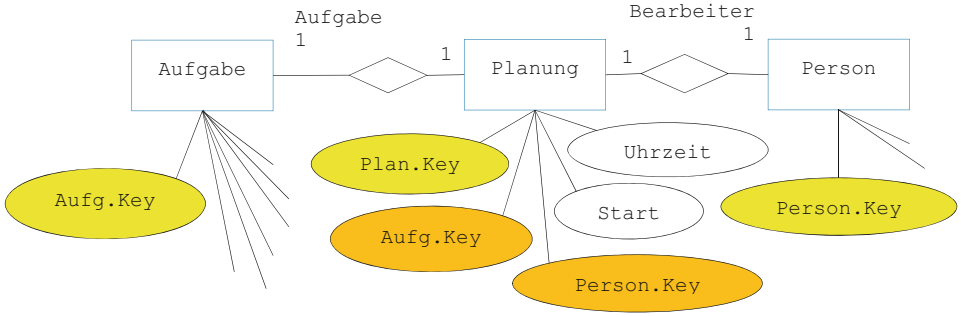


Abb. 1.9 Ein ERM-Ausschnitt von Aufgabe und Person in Beziehung über Planung

Jetzt gibt es eine 1:1-Beziehung zwischen Aufgabe und Planung und ebenso eine 1:1-Beziehung zwischen Person und Planung. Über eindeutige Schlüssel kann auf jede Entität zugegriffen werden.

Die Organisation der Bearbeitung liegt in der Klasse *Planung* und eine einzelne Planung ist mit einer Aufgabe und mit einer Person verbunden. Manager, Starttermin und Uhrzeit entfallen in der Klasse *Aufgabe*.

Code 1.3 Die Klasse Planung

```
Dim sKey       As String
Dim sTitel     As String
Dim iStufe     As Integer
Dim iZeit      As Integer
Dim iTyp       As Integer
Dim sDatei     As String

Sub Erfasse(Key, Titel, Stufe, Zeit, Typ, Datei)
   sKey = Key
   sTitel = Titel
   iStufe = Stufe
   iZeit = Zeit
   iTyp = Typ
   sDatei = Datei
End Sub
```

1.1.8 Sequenzdiagramm

Damit sind die Datenstrukturen unseres Modells *Aufgaben-Bearbeitung* festgelegt. Bleibt die Frage nach dem Ablauf. Hier wird gerne das *Sequenzdiagramm* (sequence diagram) genutzt. Ein weiteres Diagramm aus der Gruppe der UML-Verhaltensdiagramme. Es beschreibt die Nachrichten zwischen Objekten in einer bestimmten Szene und ebenso die zeitliche Reihenfolge.

Die wichtigsten Sequenzdiagramm-Elemente stehen im Anhang 1, Tab. A1.3. Die wichtigsten Teilszenen im Anhang 1, Tab. A1.4.

Im Sequenzdiagramm (Abb. 1.10) werden die Objekte in Rechtecken dargestellt. Sie haben Namen und durch einen Doppelpunkt getrennt die Angabe der Klasse. Ihre Lebenslinie verläuft senkrecht von oben nach unten. Das Ende ihrer Existenz kennzeichnen gekreuzte Linien. Im Beispiel wird eine Aufgabe A zu einem bestimmten Zeitpunkt definiert. Zu einem späteren Zeitpunkt wird die Aufgabe A zum Planungsobjekt X und wird dem Manager B zugeordnet. Nach Erledigung der Aufgabe wird die Fertigstellung an eine weitere Person C gemeldet. Damit hört das Planungsobjekt X auf zu existieren. Die Person C prüft die Erfüllung der Aufgabe A und löscht im besten Fall die Aufgabe. Stellt der Test Fehler fest, dann muss eine neue Aufgabe erstellt werden. Die Personen bleiben weiterhin existent, angedeutet durch eine unbegrenzte Lebenslinie.

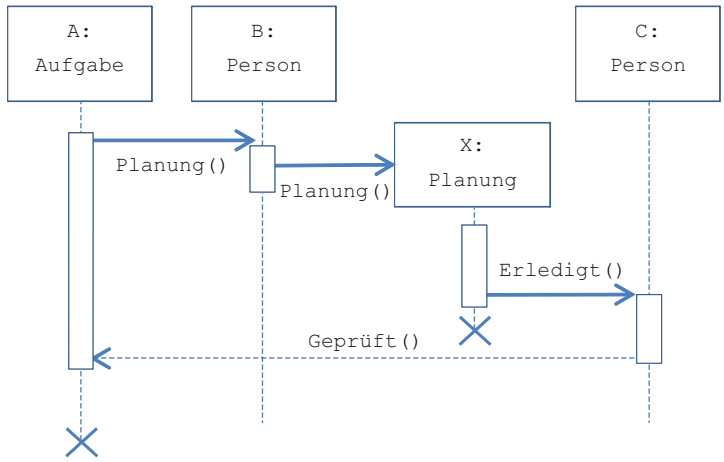


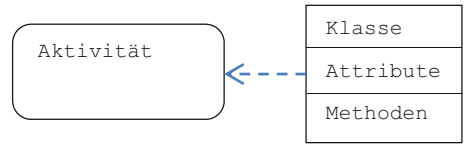
Abb. 1.10 Das Sequenzdiagramm zur Bearbeitung einer Aufgabe

Die Methoden *Planung()*, *Erledigt()* und *Geprüft()* werden in der Klasse *Planung* definiert.

1.1.9 Aktivitätsdiagramm

Während sich das Sequenzdiagramm primär um die Signale zwischen den Objekten kümmert, zeigt ein *Aktivitätsdiagramm* (activity diagram) (Abb. 1.11) primär den Datenfluss und die Nutzung der Methoden. Auch dieses Diagramm gehört zur Gruppe der Verhaltensdiagramme.

Abb. 1.11 Verbindung zwischen Klassendiagramm und Aktivität



Die wichtigsten Aktivitätsdiagramm-Elemente stehen im Anhang 1, Tab. A1.5.

Das Aktivitätsdiagramm ist eine objektorientierte Adaption des Flussdiagramms. Es beschreibt die Realisierung eines bestimmten Verhaltens durch ein System, indem es dafür den Rahmen und die geltenden Regeln vorgibt. Aktivitätsdiagramme dienen der Modellierung von dynamischen Abläufen und beschreiben Aktivitäten mit nicht-trivialem Charakter und den Fluss durch die Aktivitäten (Abb. 1.12). Es kann sowohl ein Kontrollfluss, als auch ein Datenfluss modelliert werden.

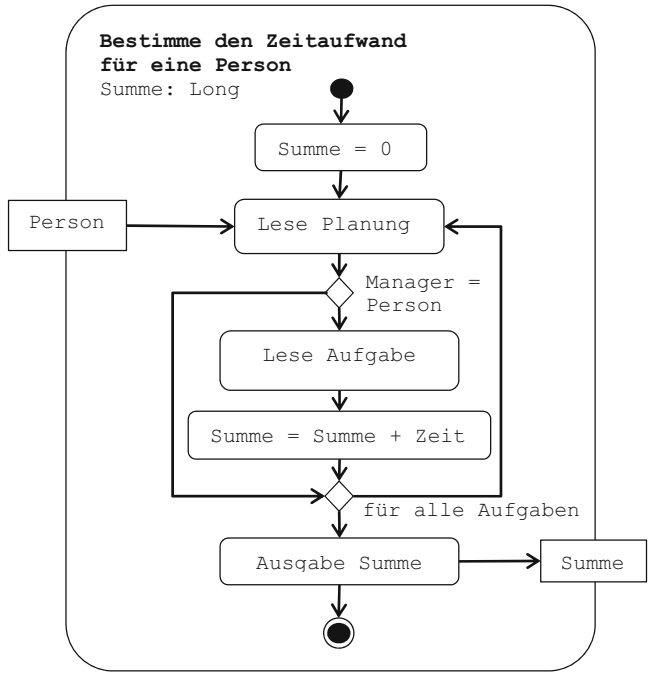


Abb. 1.12 Das Aktivitätsdiagramm zur Aufwandsbestimmung

Die Elemente eines Aktivitätsdiagramms können in ihrer Kombination mehr als es die Elemente von Flussdiagramm und Struktogramm können. So sind durch Verzweigungen des Kontrollflusses gleichzeitig parallel laufende Aktionen möglich. Eine Aktivität besteht aus Knoten (Aktionen, Kontrollknoten und Objektknoten) und Kanten (Pfeile), die den Kontrollfluss durch die Aktivität darstellen. In der linken oberen Ecke steht der Name der Aktivität. Darunter befinden sich die Parameter mit ihren Typangaben.

1.1.10 Anwendungsfalldiagramm

Auch das *Anwendungsfalldiagramm* (use case diagram) (Abb. 1.13) ist ein Verhaltensdiagramm und beschreibt das zu erwartende Verhalten eines Systems. Ein anderer Name ist *Nutzfalldiagramm*. Aus ihm lassen sich anschaulich die Anforderungen für die Software ableiten. Es hilft auch die vielen zum Teil sehr detaillierten Anforderungen in eine übersichtliche Ordnung zu fassen. So werden wichtige Funktionen der Software definiert und zueinander in Relation gesetzt. Die technische Realisierung wie z. B. die Programmiersprache spielt hier keine Rolle.

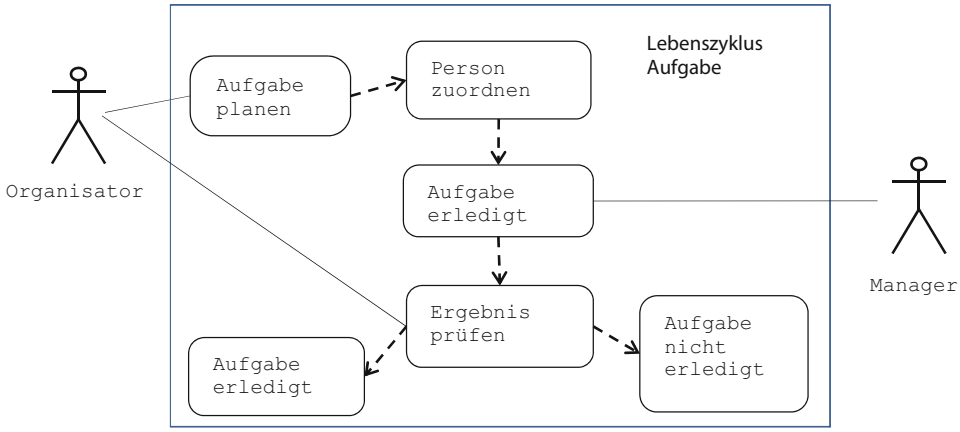


Abb. 1.13 Ein Anwendungsfalldiagramm zur Aufgabe

Die wichtigsten Anwendungsfalldiagramm-Elemente stehen im Anhang 1, Tab. A1.6.

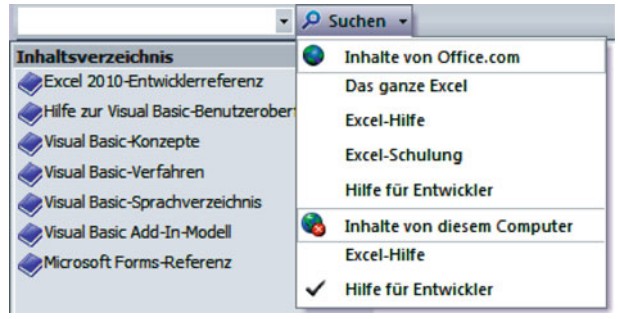
1.1.11 Hilfen in der VBA-Entwicklungsumgebung

Auch wenn ich in diesem Buch VBA-Kenntnisse voraussetze, so möchte ich zum Beginn noch einmal auf die Hilfen im System hinweisen.

In jeder Microsoft-Anwendung gibt es die Möglichkeit mit der F1-Taste ein Hilfefenster einzublenden. Darin können erklärende Texte zu anwendungsspezifische Themen aufgerufen werden. Eine solche Möglichkeit existiert auch in der VBA-Entwicklungsumgebung. Sie heißt dann z. B. Excel-Hilfe, je nach der Anwendung in der sie aufgerufen wird. Ich nenne sie fortan *VBA-Hilfe* und zwar für alle Anwendungen.

Nach dem Öffnen gibt es neben der Schaltfläche *Suchen* einen Pfeil zum Öffnen eines Pull-Down-Menüs. Dort wird die Version *Hilfe für Entwickler* angewählt (Abb. 1.14).

Abb. 1.14 VBA-Hilfe-Fenster



Im Inhaltsverzeichnis befindet sich unter der *Entwicklerreferenz* auch die *Objektmodellreferenz* der jeweiligen Anwendung. Sie beinhaltet alle Objekte der Anwendung mit ihren Eigenschaften, Methoden und oft auch erklärenden Beispielen.

Eine weitere Möglichkeit in der VBA-Entwicklungsumgebung ist der Objektkatalog. Er wird mit der F2-Taste aufgerufen (alternativ auch über das Register *Ansicht*). Im obersten Auswahlfenster können alle oder einzelne Objekt-Bibliotheken angewählt werden (Abb. 1.15).

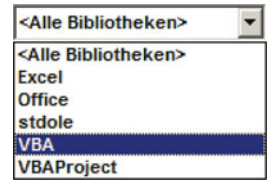


Abb. 1.15 Auswahl der Objekt-Bibliotheken im Objektkatalog

Im zweiten Auswahlfenster wird ein Suchbegriff für die ausgewählte Objekt-Bibliothek eingegeben und mit der *Enter*-Taste oder der Schalttaste *Fernglas* aufgerufen (Abb. 1.16).

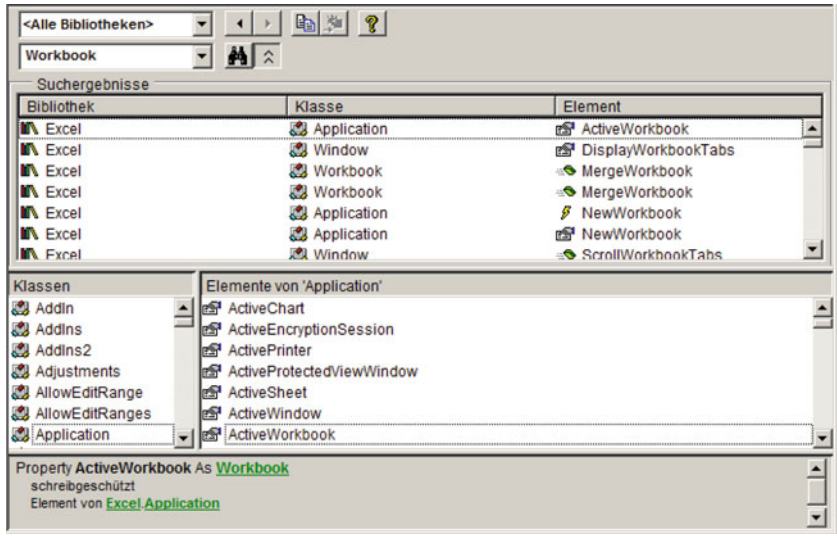


Abb. 1.16 Suchen im Objektkatalog

Für einen markierten Eintrag im Fenster *Suchergebnisse* kann mit der Schaltaste *Fragezeichen* weiterer Hilfstext eingeblendet werden. Auch dort gibt es oft erklärende Beispiele.

1.2 Objekte und Objektlisten

1.2.1 Module

Wir setzen das Modell *Aufgaben-Bearbeitung* fort, indem wir die Anwendung der Klassen besprechen, also die Instanziierung von Objekten der Klassen. Genauer mit der Methode *Erfasse* der Klasse *Aufgaben*, denn ohne Aufgaben keine Bearbeitung. Dazu nutzen wir ein Code-Modul mit dem Namen *modStart* (Abb. 1.17).

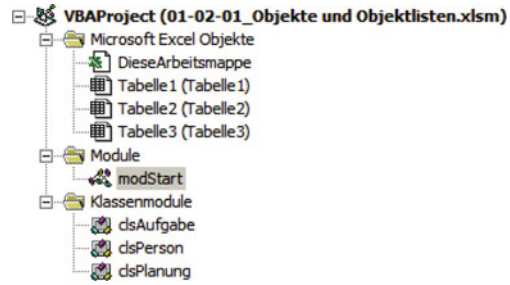


Abb. 1.17 Das Code-Modul Start

Die Erfassung der Aufgaben wird zunächst einmal *hard codiert*, d. h. die Daten werden direkt im Quellcode angegeben (Code 1.4). Später werden wir dazu *UserForm*-Objekte nutzen.

Code 1.4 Die Prozedur ErfasseAufgaben

```
Option Explicit

Sub ErfasseAufgaben()
   Dim objAufgabe As New clsAufgabe
   With Aufgabe
      .Erfasse "1", "Telefonat Müller", 1, 10, 0, "keine"
      .Erfasse "2", "Mail Hans Schmidt", 4, 15, 0, "keine"
      .Erfasse "3", _
         "Meeting Projekt", 2, 120, 1, "C:\Planung\Meeting_13_5.txt"
      .Erfasse "4", _
         "Forum News planen", 3, 240, 2, "C:\Foren\Forum_03.txt"
   End With
End Sub
```

1.2.2 Objektvariable

Objekte werden in der Programmierung durch Objektvariable dargestellt, ähnlich wie Variable für Datenwerte. Daher ist die erste Anweisung in der Prozedur die Definition der Objektvariablen *objAufgabe* und die gleichzeitige Instanziierung des Objekts. Die Anweisung *Dim As clsAufgabe* sorgt für die Definition und die Anweisung *New* für die Instanziierung. Ich benutze die Anweisung *New* zusammen mit der Definition nur sehr ungern, da mit der Instanziierung auch Events verbunden sind, die bei dieser Form nicht immer genutzt werden können.

Wir haben ein Objekt erzeugt und mit der Methode *Erfasse* der Klasse *Aufgabe* werden den Attributen Werte zugewiesen. Doch woher wissen wir dass dies so ist? Wir benötigen also eine Ausgabe der Attribute, die wir ebenfalls als Methode der Klasse *Aufgabe* konstruieren. Wir haben aber weder einen Drucker noch eine Oberfläche zur Ausgabe konstruiert. Die Entwicklungsumgebung verfügt über ein Direktfenster und eine Methode *Debug.Print* zur Ausgabe darin. Mithilfe der Methode *Zeige* in der Klasse *Aufgabe* wird die Ausgabe realisiert (Code 1.5).

Code 1.5 Die Methode Zeige der Klasse Aufgabe

```
Sub Zeige()
   Debug.Print sKey; " / "; sTitel, iStufe; " / "; iZeit, iTyp, sDatei
End Sub
```

Wir erweitern die Anwendung um die neue Methode und betrachten die Ausgabe im Direktfenster (Abb. 1.18).

```
 1 / Telefonat Müller          1  /  10   0 keine
 2 / Mail Hans Schmidt         4  /  15   0 keine
 3 / Meeting Projekt           2  /  120  1 C:\Planung\Meeting_13_5.txt
 4 / Forum News planen         3  /  240  2 C:\Foren\Forum_03.txt
 2 / Mail Hans Schmidt         4  /  15   0 keine
```

Abb. 1.18 Ausgabe der Personendaten im Direktfenster

Noch immer nutzen wir nur ein Objekt mit zeitlich unterschiedlichen Werten. Wollen wir jeweils ein Objekt mit den Werten erzeugen, so müssen wir es zum Zeitpunkt der Nutzung instanziieren. Objekte werden mit der *Dim* Anweisung definiert

```
'Syntax:
   Dim Objektvariable As Objekttyp

'Beispiel:
   Dim wshDaten as Worksheet
```

und mit der *Set* Anweisung

```
'Syntax:
   Set Objektvariable = New Objekttyp

'Beispiel:
   Set wshDaten = New Worksheet
```

zu jedem beliebigen Moment instanziiert. Da Objekte ihre Lebensdauer so lange behalten wie die Lebensdauer des Projekts und dabei sehr schnell viel Speicherplatz benötigen, sollte der Speicherplatz einer Objektvariablen nach dem Nutzungsende wieder freigegeben werden.

```
'Syntax:
   Set Objektvariable = Nothing

'Beispiel:
   Set wshDaten = Nothing
```

Es zeugt von einem guten Programmierstil diese Freigabe nicht zu vergessen. Die Methode *Zeige* wird in der Ausgabeform etwas verändert, um Platz zu sparen und die Prozedur zur Aufgabenerfassung erhält eine weitere Änderung (Code 1.6). Die Anweisung *Option Explicit* steht selbstverständlich immer in jedem Codemodul. Aus Platzgründen wird sie in allen folgenden Code-Beispielen nicht aufgeführt, sie steht aber in jedem Download-Beispiel.

Code 1.6 Die Prozedur ErfasseAufgaben – Version 3

```
Sub ErfasseAufgaben()
   Dim objAufgabe    As clsAufgabe

   Set objAufgabe = New clsAufgabe
   objAufgabe.Erfasse "1", _
      "Telefonat Müller", 1, 10, 0, "keine"
   Set objAufgabe = Nothing
   Set objAufgabe = New clsAufgabe
   objAufgabe.Erfasse "2", _
      "Mail Hans Schmidt", 4, 15, 0, "keine"
   Set objAufgabe = Nothing
   Set objAufgabe = New clsAufgabe
   objAufgabe.Erfasse "3", _
      "Meeting Projekt", 2, 120, 1, "C:\Planung\Meeting_13_5.txt"
```

```
   Set objAufgabe = Nothing
   Set objAufgabe = New clsAufgabe
   objAufgabe.Erfasse "4", _
      "Forum News planen", 3, 240, 2, "C:\Foren\Forum_03.txt"
   Set objAufgabe = Nothing
End Sub
```

Nun erzeugen wir mehrere Objekte der Klasse *Aufgabe* und weisen ihren Attributen auch Werte zu, doch wie können wir sie auseinander halten? Oder wie können wir auf einzelne Objekte zugreifen? Zunächst einmal wohl gar nicht.

1.2.3 Objektliste Collection

Hier kommen jetzt die Objektlisten ins Spiel. Dem Namen nach sind uns solche z. B. in Excel sehr bekannt wie Workbooks, Worksheets, Sheets, Columns, Rows, Cells, usw. erkennbar durch das „s“ am Ende des Namens. Für eigene Objekte stehen uns zwei Arten von Objektlisten zur Verfügung, *Collection* (Sammlung) und *Dictionary* (Wörterbuch).

Eigenschaften und Methoden der Klassen *Collection* und *Dictionary* stehen unter dem Klassennamen im Objektkatalog und in der VBA-Hilfe.

Um Objekte einer Objektliste anzusprechen, bedient man sich in der Programmierung der *For-Each-Next*-Anweisung.

```
'Syntax:
   For Each Objektvariable In Objektliste
      Anweisungen
   Next
'Beispiel:
   For Each wshBlatt in Thisworkbook.Worksheets
      MsgBox wshBlatt.Name
   Next
```

Wie der Name es schon ausdrückt ist ein Objekt der Klasse *Collection* eine Sammlung von Objekten und die einfachere Variante zur Klasse *Dictionary*. Die Sammlung verfügt über eine Methode zum Hinzufügen

```
'Syntax:
   Collection.Add (Item, Key, Before, After)

'Beispiel:
   colAufgaben.Add objAufgabe, "1"
```

und eine zum Entfernen

```
'Syntax:
   Collection.Remove (Index)

'Beispiel:
   colAufgaben.Remove (sAufgNr)
```

von Objekten in der Liste. Die Parameter der Methoden werden im Anhang 1, Tab. A1.7 und A1.8 beschrieben. Ebenso die Attribute der Objektlisten. Zu beachten ist, dass die Schlüssel als Text vorliegen. Daher sind auch die Schlüssel in den Klassen ebenfalls vom Typ *String*.

Mit der Methode *Item*

```
'Syntax:
   Collection.Item(Index)

'Beispiel:
   colAufgaben.Item("2")
```

erhalten wir Zugriff auf die einzelnen Objekte in der Liste. Die Anzahl der Objekte liefert das Attribut der Liste.

```
'Syntax:
   Collection.Count

'Beispiel:
   lMax = colAufgaben.Count
```

Nun können wir eine Sammlung *Aufgaben* anlegen (Code 1.7) und ihr unsere Objekte zuweisen. Während die einzelnen Objekte im Projekt immer wieder freigegeben werden, müssen die Objektlisten während der gesamten Lebensdauer des Projekts existent bleiben.

Code 1.7 Die Prozedur ErfasseAufgaben – Version 4

```
Sub ErfasseAufgaben()
   Dim objAufgabe    As clsAufgabe

   Set objAufgabe = New clsAufgabe
   objAufgabe.Erfasse "1", _
      "Telefonat Müller", 1, 10, 0, "keine"
   colAufgaben.Add objAufgabe, "1"
```

```
   Set objAufgabe = Nothing
   Set objAufgabe = New clsAufgabe
   objAufgabe.Erfasse "2", _
      "Mail Hans Schmidt", 4, 15, 0, "keine"
   colAufgaben.Add objAufgabe, "2"
   Set objAufgabe = Nothing
   Set objAufgabe = New clsAufgabe
   objAufgabe.Erfasse "3", _
      "Meeting Projekt", 2, 120, 1, "C:\Planung\Meeting_13_5.txt"
   colAufgaben.Add objAufgabe, "3"
   Set objAufgabe = Nothing
   Set objAufgabe = New clsAufgabe
   objAufgabe.Erfasse "4", _
      "Forum News planen", 3, 240, 2, "C:\Foren\Forum_03.txt"
   colAufgaben.Add objAufgabe, "4"
   Set objAufgabe = Nothing
End Sub
```

Diese Version definiert und instanziiert ein Collection-Objekt *colAufgaben.* Danach werden alle erzeugten Objekte zugewiesen. In einer Programmschleife erfolgt die Ausgabe der Attribute aller Objekte. Zum Schluss wird dann das Objekt mit dem Key ″2″ noch einmal ausgegeben. Die Ausgaben beweisen, dass die Objekte auch nach der Zuweisung weiterhin existieren. Die Sammlung vermerkt die Klasse des Objekts und damit seine Methoden und Attribute.

1.2.4 Gültigkeitsbereiche Private und Public

Es bleibt noch die Frage des Gültigkeitsbereichs zu klären. Betrachten wir dazu eine einfache Anordnung von Testprozeduren in einem Code-Modul.

```
Dim Wert As Integer

Sub Test()
   Test1
   Test2
   Debug.Print "3: "; Wert
End Sub
Sub Test1()
   Wert = 1
   Debug.Print "1: "; Wert
End Sub
Sub Test2()
   Dim Wert As Integer
```

```
   Wert = 2
   Debug.Print "2: "; Wert
End Sub
```

Auf Modulebene wird die Variable *Wert* definiert, ebenso eine Variable gleichen Namens in der Prozedur *Test2*. Mit dem Aufruf der Prozedur *Test* werden die Prozeduren *Test1* und *Test2* durchlaufen und danach wird noch einmal der Wert ausgegeben. Das Ergebnis ist

```
1: 1
2: 2
3: 1
```

Bei gleichen Variablennamen hat die Variable in einer Prozedur Vorrang vor der übergeordneten Variable auf Modulebene. Außerhalb des Code-Moduls verlieren beide ihre Gültigkeit. Auf Modulebene kann die Variable auch mit der Anweisung *Private* definiert werden und besitzt den gleichen Gültigkeitsbereich, eben nur die Modulebene.

```
'Syntax:
   Private Variablenname As Datentyp

'Beispiel:
   Private sName As String
```

Erst mit der Definition

```
'Syntax:
   Public Variablenname As Datentyp

'Beispiel:
   Public dDaten As Double
```

auf Modulebene erhält die Variable ihren Gültigkeitsbereich für das gesamte Projekt.

Wir möchten die Aufgabendaten ja nicht nur in einem Code-Modul nutzen, sondern im gesamten Projekt. Dafür ist es notwendig, dass die Sammlung auch in allen Bereichen existiert (Code 1.8).

Mit der Angabe

```
Option Private Module
```

im Kopf eines Codemoduls werden alle Prozeduren automatisch in den Modus *Private* gesetzt, auch wenn sie nur *Sub* oder *Public Sub* gesetzt sind. Der Vorteil ist, dass sie im gesamten Projekt aufgerufen werden können.

Code 1.8 Die Prozedur ErfasseAufgaben – Version 5

```
Public colAufgaben   As Collection

Sub ErfasseAufgaben()
   Dim objAufgabe    As clsAufgabe

   Set objAufgabe = New clsAufgabe
   objAufgabe.Erfasse "1", _
      "Telefonat Müller", 1, 10, 0, "keine"
   colAufgaben.Add objAufgabe, "1"
   Set objAufgabe = Nothing
   Set objAufgabe = New clsAufgabe
   objAufgabe.Erfasse "2", _
      "Mail Hans Schmidt", 4, 15, 0, "keine"
   colAufgaben.Add objAufgabe, "2"
   Set objAufgabe = Nothing
   Set objAufgabe = New clsAufgabe
   objAufgabe.Erfasse "3", _
      "Meeting Projekt", 2, 120, 1, "C:\Planung\Meeting_13_5.txt"
   colAufgaben.Add objAufgabe, "3"
   Set objAufgabe = Nothing
   Set objAufgabe = New clsAufgabe
   objAufgabe.Erfasse "4", _
      "Forum News planen", 3, 240, 2, "C:\Foren\Forum_03.txt"
   colAufgaben.Add objAufgabe, "4"
   Set objAufgabe = Nothing
End Sub
```

Mit der Definition *Public* im Modul außerhalb einer Prozedur erreichen wir den gesamten Gültigkeitsbereich. Die Ausgabe bekommt eine eigene Prozedur im Code-Modul (Code 1.9). Der Aufruf der Prozeduren *ErfasseAufgabe* und *LeseAufgaben* bringt nun das gleiche Ergebnis.

Code 1.9 Die Prozedur LeseAufgaben

```
Sub LeseAufgaben()
   Dim objAufgabe    As clsAufgabe

'lesen aller Aufgaben und zeigen
   For Each objAufgabe In colAufgaben
      objAufgabe.Zeige
   Next

'Zugriff auf Aufgabe Nr. 2 als Beispiel
   Debug.Print colAufgaben.Item("2").Zeige
```

```
   Set objAufgabe = Nothing
End Sub
```

Aufgelöst werden die Sammlungen erst mit einer *Nothing*-Anweisung (Code 1.10).

Code 1.10 Die Prozedur LöscheAufgabenSammlung

```
Sub LöscheAufgabenSammlung()
   Set colAufgaben = Nothing
   Set colPersonen = Nothing
   Set colPlanungen = Nothing
End Sub
```

1.2.5 Auto_Open und Auto_Close

Es wäre sinnvoll, die Sammlung mit dem Öffnen der Arbeitsmappe anzulegen und auch wieder mit dem Schließen zu löschen. Eine Excel-Arbeitsmappe verfügt über zwei festgelegte Events die sich dafür sehr gut eignen. Mit dem Öffnen einer Arbeitsmappe wird die Methode *Auto_Open* aufgerufen und ihre Anweisungen, falls es solche gibt, werden ausgeführt. Ebenso gibt es die Methode *Auto_Close*, die ihre Anweisungen vor dem Schließen der Arbeitsmappe ausführt. Später werden wir sehen, dass das Workbook über zwei ähnliche Methoden *Workbook_Open* und *Workbook_BeforeClose* verfügt.

Für unser Beispiel benutzen wir ein weiteres Code-Modul *modAuto* in dem diese Prozeduren definiert werden (Code 1.11). Als Hinweis auf die Funktionalität werden Hinweistexte angezeigt. Außerdem benötigen wir noch zwei weitere Sammlungen *colPersonen* und *colPlanungen*.

Code 1.11 Das Codemodul Auto

```
Option Explicit

Public colAufgaben   As Collection
Public colPersonen   As Collection
Public colPlanungen  As Collection

Sub Auto_Open()
   MsgBox "Sammlungen werden angelegt!"
   Set colAufgaben = New Collection
   Set colPersonen = New Collection
   Set colPlanungen = New Collection
   ErfasseAufgaben
End Sub
```

```
Sub Auto_Close()
   MsgBox "Sammlungen werden gelöscht!"
   Set colAufgaben = Nothing
   Set colPersonen = Nothing
   Set colPlanungen = Nothing
End Sub
```

Mit dem Start werden ebenfalls einige Personen *hard codiert* erzeugt, so dass die Grundlagen für eine Planung vorliegen (Code 1.12). Die Klasse *Personen* bekommt eine Methode *Zeige*.

Code 1.12 Die Prozedur ErfassePersonen

```
Sub ErfassePersonen()
   Dim objPerson     As clsPerson

   Set objPerson = New clsPerson
   objPerson.Erfasse "1", "Gost", "Petra", "Planung"
   colPersonen.Add objPerson, "1"
   Set objPerson = Nothing
   Set objPerson = New clsPerson
   objPerson.Erfasse "2", "Brenner", "Hans", "Management"
   colPersonen.Add objPerson, "2"
   Set objPerson = Nothing
   Set objPerson = New clsPerson
   objPerson.Erfasse "3", "Hebbel", "Maria", "Marketing"
   colPersonen.Add objPerson, "3"
   Set objPerson = Nothing
End Sub
```

Auf dieser so erreichten Konstruktion können wir nun ebenfalls *hard codiert* die Planung einer Aufgabe realisieren und protokollieren. Entsprechend dem zuvor gezeigten Sequenzdiagramm. Dort haben wir die Prozeduren *Planen* (Code 1.13), *Terminieren* und *Prüfen* definiert.

Code 1.13 Die Methoden Planen und Zeige in der Klasse Planung

```
Sub Planen(Key, Aufgabe, Person, Start, Uhrzeit)
   sKey = Key
   sAufgabe = Aufgabe
   sPerson = Person
   If Not Start = "00.00.0000" Then dStart = Start
   If Not Uhrzeit = "00:00" Then dUhrzeit = Uhrzeit
End Sub
```

```
Sub Zeige()
   Debug.Print sKey; " / "; sAufgabe; " / "; sPerson, dStart, dUhrzeit
End Sub
```

Das Planen wird in einem eigenen Modul *modPlanung* über die Prozedur *PlaneAufgaben* (Code 1.14) realisiert.

Code 1.14 Die Prozedur PlaneAufgaben im Modul Planung

```
Sub PlaneAufgaben()
   Dim objPlan       As clsPlanung

   Set objPlan = New clsPlanung
   objPlan.Planen "1", "1", "2", "15.05.2015", "14:00"
   objPlan.Zeige
   colPlanungen.Add objPlan, "1"
   Set objPlan = Nothing
   Set objPlan = New clsPlanung
   objPlan.Planen "2", "2", "1", "15.05.2015", "14:00"
   objPlan.Zeige
   colPlanungen.Add objPlan, "2"
   Set objPlan = Nothing
   Set objPlan = New clsPlanung
   objPlan.Planen "3", "4", "2", "16.05.2015", "08:00"
   objPlan.Zeige
   colPlanungen.Add objPlan, "3"
   Set objPlan = Nothing
   Set objPlan = New clsPlanung
   objPlan.Planen "4", "3", "3", "16.05.2015", "10:00"
   objPlan.Zeige
   colPlanungen.Add objPlan, "4"
   Set objPlan = Nothing
End Sub
```

Das Protokoll zum Planungsablauf steht im Direktfenster (Abb. 1.19).

```
1 / Telefonat Müller        1  /  10     0          keine
2 / Mail Hans Schmidt       4  /  15     0          keine
3 / Meeting Projekt         2  /  120    1          C:\Planung\Meeting_13_5.txt
4 / Forum News planen       3  /  240    2          C:\Foren\Forum_03.txt
2 / Mail Hans Schmidt       4  /  15     0          keine

1 / Gost      Petra         Planung
2 / Brenner   Hans          Management
3 / Hebbel    Maria         Marketing
2 / Brenner   Hans          Management

1 / 1 / 2     15.05.2015    14:00:00
2 / 2 / 1     15.05.2015    14:00:00
3 / 4 / 2     16.05.2015    08:00:00
4 / 3 / 3     16.05.2015    10:00:00
```

Abb. 1.19 Das Protokoll zur Planung

Nun soll eine kleine Auswertung folgen. In einer Schleife sollen die verplanten Zeiten der Personen ausgegeben werden. Eine weitere Prozedur im neuen Modul *Planung* erledigt diese Aufgabe. Doch zuvor müssen wir noch die Eigenschaftsfunktionen einführen.

1.2.6 Property-Funktionen

Bisher haben wir immer nur auf die vorhandenen Listen zugegriffen. Doch wie können wir den Wert eines einzelnen Attributes ändern oder auch lesen? Der Zugriff auf Eigenschaftswerte eines Objekts einer Klasse wird über die Eigenschaftsprozeduren

```
'Syntax:
   Property Let (Wert setzen)
   Property Get (Wert auslesen)

'Beispiel:
   Property Let Key(Key As String)
   Property Get Key() As String
```

gesteuert (Property = Eigenschaft). Damit also die Planzeit einer Aufgabe geändert werden kann und auch die Werte anderer Attribute, erhalten die Klassen die erforderlichen Prozeduren. Die Property-Funktionen der Klasse *Aufgaben* (Code 1.15) stehen stellvertretend für alle anderen Klassen.

Code 1.15 Die Property-Funktionen Let und Get der Klasse Aufgabe zum Schreiben und Lesen

```
Property Let Key(Key As String)
   sKey = Key
End Property
```

```
Property Let Titel(Titel As String)
   sTitel = Titel
End Property

Property Let Stufe(Stufe As Integer)
   iStufe = Stufe
End Property

Property Let Zeit(Zeit As Integer)
   iZeit = Zeit
End Property

Property Let Typ(Typ As Integer)
   iTyp = Typ
End Property

Property Let Datei(Datei As String)
   sDatei = Datei
End Property

Property Get Key() As String
   Key = sKey
End Property

Property Get Titel() As String
   Titel = sTitel
End Property

Property Get Stufe() As Integer
   Stufe = iStufe
End Property

Property Get Zeit() As Integer
   Zeit = iZeit
End Property

Property Get Typ() As Integer
   Typ = iTyp
End Property

Property Get Datei() As String
   Datei = sDatei
End Property
```

Jetzt kann die Auswertung der Planzeiten geschrieben werden (Code 1.16).

Code 1.16 Die Prozedur zur Bestimmung der personenbezogenen Planzeiten

```
Sub BestimmePlanzeiten()
   Dim objPerson     As clsPerson
   Dim objPlan       As clsPlanung
   Dim objAufgabe    As clsAufgabe
   Dim lZeit         As Long

   Debug.Print vbLf; "Planzeiten:"
   For Each objPerson In colPersonen
      lZeit = 0
      For Each objPlan In colPlanungen
         If objPlan.Person = objPerson.Key Then
            Set objAufgabe = colAufgaben.Item(objPlan.Aufgabe)
            lZeit = lZeit + objAufgabe.Zeit
         End If
      Next
      Debug.Print objPerson.Name, lZeit
   Next
End Sub
```

Nach dem Start steht das Protokoll ebenfalls im Direktfenster (Abb. 1.20).

Es bleibt noch darauf hinzuweisen, dass die Property-Funktionen *Let* und *Get* den gleichen Namen besitzen dürfen, was ansonsten für Prozeduren und Funktionen nicht erlaubt ist.

Abb. 1.20 Das Ergebnis der personenbezogenen Planzeiten im Direktfenster

```
Planzeiten:
Gost            15
Brenner         250
Hebbel          120
```

1.2.7 Konstruktor und Destruktor

Mitunter sollen einem Objekt bei der Instanziierung Startwerte zugewiesen werden. In allen üblichen objektorientierten Programmiersprachen verfügen Klassen über eine Methode die beim Anlegen eines Objekts (Konstruktor) und bei der Freigabe (Destruktor) durchlaufen wird. So auch in VBA und sie lauten

```
Sub Class_Initialize()
   Anweisungen Konstruktor
End Sub
```

```
Sub Class_Terminate()
   Anweisungen Destruktor
End Sub
```

Es ist denkbar, dass bei der Planung das Startdatum und die Uhrzeit nicht bekannt sind. So könnte bei der Instanziierung das aktuelle Datum und die aktuelle Zeit gesetzt werden. Nur wenn bei der Planung auch ein Datum und/oder eine Uhrzeit angegeben wird, dann werden diese übernommen. Inzwischen hat unsere Startprozedur den in Code 1.17 dargestellten Inhalt und erledigt alle Instanziierungen.

Code 1.17 Die erweiterte Version von Auto_Open

```
Sub Auto_Open()
   MsgBox "Sammlungen werden angelegt!"
   Set colAufgaben = New Collection
   Set colPersonen = New Collection
   Set colPlanungen = New Collection
   ErfasseAufgaben
   ErfassePersonen
   LeseAufgaben
   LesePersonen
   PlaneAufgaben
End Sub
```

Die Methode *Planen* der Klasse *Planung* erhält Abfragen zu Datum und Uhrzeit. Der Konstruktor schreibt bei der Instanziierung die aktuellen Werte in Datum und Uhrzeit (Code 1.18).

Code 1.18 Die erweiterte Methode Planen und der Konstruktor der Klasse Planung

```
Sub Planen(Key, Aufgabe, Person, Start, Uhrzeit)
   sKey = Key
   sAufgabe = Aufgabe
   sPerson = Person
   If Not Start = "00.00.0000" Then dStart = Start
   If Not Uhrzeit = "00:00" Then dUhrzeit = Uhrzeit
End Sub

Sub Class_Initialize()
   If dStart = 0 Then _
      dStart = Format(Now, "DD.MM.YYYY")
   If dUhrzeit = 0 Then _
      dUhrzeit = Format(Now, "hh:mm")
End Sub
```

Zum Test ergänzen wir das Modul *Planung* um eine weitere Prozedur, die eine weitere Aufgabe und eine neue Person instanziiert, diese dann verplant und danach eine neue Planzeitauswertung startet (Code 1.19).

Code 1.19 Eine neue Prozedur zur Ergänzung und Auswertung

```
Sub NeueAufgabePlanen()
   Dim objAufgabe     As clsAufgabe
   Dim objPerson      As clsPerson
   Dim objPlan        As clsPlanung

   Set objAufgabe = New clsAufgabe
   objAufgabe.Erfasse "5", "Algorithmus entwerfen", 1, 300, 0, "keine"
   colAufgaben.Add objAufgabe, "5"
   Set objAufgabe = Nothing
   Set objPerson = New clsPerson
   objPerson.Erfasse "4", "Hoffmann", "Klaus", "EDV"
   colPersonen.Add objPerson, "4"
   Set objPerson = Nothing
   Set objPlan = New clsPlanung
   objPlan.Planen "5", "5", "4", "00.00.0000", "00:00"
   objPlan.Zeige
   colPlanungen.Add objPlan, "5"
   Set objPlan = Nothing
   BestimmePlanzeiten
End Sub
```

Das Protokoll steht nach Ausführung im Direktfenster (Abb. 1.21).

```
1 / Telefonat Müller        1  /  10      0          keine
2 / Mail Hans Schmidt       4  /  15      0          keine
3 / Meeting Projekt         2  /  120     1          C:\Planung\Meeting_13_5.txt
4 / Forum News planen       3  /  240     2          C:\Foren\Forum_03.txt
2 / Mail Hans Schmidt       4  /  15      0          keine

1 / Gost      Petra         Planung
2 / Brenner   Hans          Management
3 / Hebbel    Maria         Marketing
2 / Brenner   Hans          Management

1 / 1 / 2     15.05.2015    14:00:00
2 / 2 / 1     15.05.2015    14:00:00
3 / 4 / 2     16.05.2015    08:00:00
4 / 3 / 3     16.05.2015    10:00:00
5 / 5 / 4     06.06.2015    07:28:00

Planzeiten:
Gost          15
Brenner       250
Hebbel        120
Hoffmann      300
```

Abb. 1.21 Das Protokoll zur neuen Ergänzung

Es fehlt zum Schluss noch die Prozedur zur Terminierung einer Aufgabe. Auch sie wird im Modul *Planung* realisiert. Zur Vereinfachung wird die Aufgabe nach Bestätigung durch den Sachbearbeiter gelöscht, so dass damit Terminierung und Prüfung zusammen durchgeführt werden.

Abb. 1.22 zeigt das Struktogramm des Algorithmus und Code 1.20 den Code der Prozedur. Der Dialog wird über die Funktionen *MsgBox* und *InputBox* realisiert.

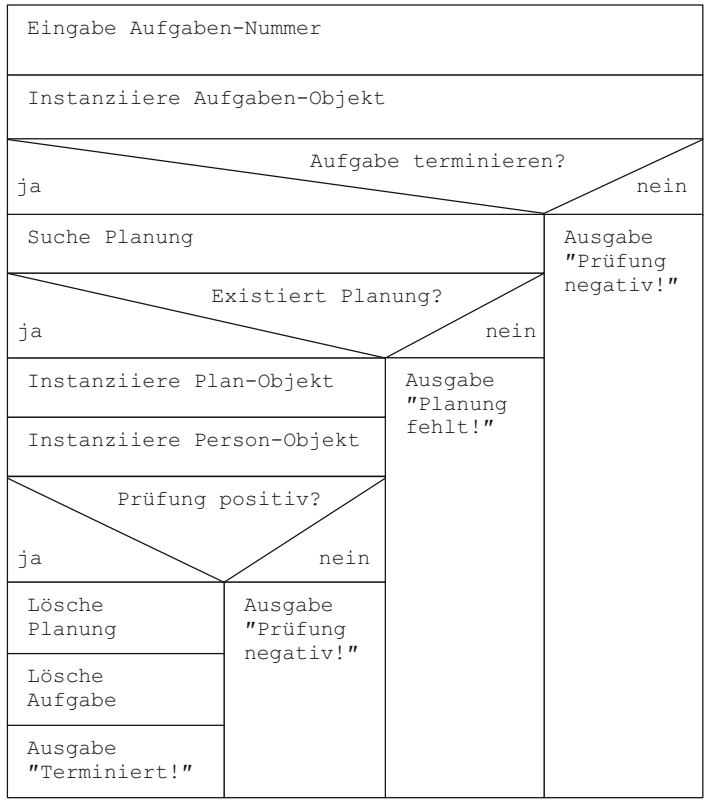


Abb. 1.22 Der Algorithmus zur Terminierung als Struktogramm

Code 1.20 Die Prozedur zur Terminierung einer Aufgabe

```
Sub AufgabeTerminieren()
   Dim objPerson     As clsPerson
   Dim objPlan       As clsPlanung
   Dim objAufgabe    As clsAufgabe
   Dim lZeit         As Long
   Dim sAufgNr       As String
   Dim sPlanNr       As String
   Dim vAntwort      As Variant
```

```
   sAufgNr = InputBox("Welche Aufgabe terminieren" & vbLf _
      & "Bitte AufgabenNr. angeben: ")
   Set objAufgabe = colAufgaben.Item(sAufgNr)
   vAntwort = MsgBox("Aufgabe: " & objAufgabe.Titel & _
      " - terminieren?", vbQuestion + vbYesNo)
   If vAntwort = vbYes Then
   'Planung finden
      sPlanNr = ""
      For Each objPlan In colPlanungen
         If objPlan.Aufgabe = sAufgNr Then
            sPlanNr = objPlan.Key
         End If
      Next
      If sPlanNr = "" Then
         MsgBox "Fehler!"
         Exit Sub
      Else
         Set objPlan = colPlanungen.Item(sPlanNr)
         Set objPerson = colPersonen.Item(objPlan.Person)
         vAntwort = MsgBox(objPerson.Name & ": bitte prüfen!" & _
            vbLf & "Ist der Test positiv?", vbQuestion + vbYesNo)
         If vAntwort = vbYes Then
            colPlanungen.Remove (sPlanNr)
            Set objPlan = Nothing
            colAufgaben.Remove (sAufgNr)
            Call LeseAufgaben
            MsgBox objAufgabe.Titel & " gelöscht!"
         Else
            MsgBox "Die Prüfung war negativ!" & vbLf & _
               "Bitte Aufgabe weiter bearbeiten!", vbCritical
         End If
      End If
   Else
      MsgBox "Die Prüfung war negativ!" & vbLf & _
         "Bitte Aufgabe weiter bearbeiten!", vbCritical
   End If
End Sub
```

Im ersten Schritt wird die Nummer der Aufgabe abgefragt, die gelöscht werden soll. Die Funktion *InputBox* übernimmt diese Aufgabe. Dann wird zum entsprechenden Objekt in der Sammlung Aufgaben eine Objektvariable instanziiert. Noch einmal wird die Bestätigung zum Löschvorgang abgefragt. Das übernimmt die Funktion *MsgBox*. Mit einer positiven Bestätigung erfolgt der Suchlauf nach dem Eintrag in der Planung mittels einer *For-Each-Next*-Schleife. Bei einer *Dictionary*-Sammlung wäre der Zugriff direkt möglich. Ist die Aufgabe vorhanden, dann werden die zugehörigen Objekte *Plan* und *Person*

instanziiert. Noch einmal erfolgt eine Bestätigungsabfrage zur Löschabfrage, bevor die Objekte in den Sammlungen endgültig gelöscht werden.

Bis jetzt haben wir ein Minimum an Dialog benutzt und Fehler durch falsche Handhabung ignoriert. Dies ändert sich mit der Einführung von Formularen. Wurde der Programmcode bisher in Codemodule geschrieben, die nur über ein Codefenster verfügen, so besitzen Formulare zusätzlich noch eine Oberfläche.

1.3 Formulare und Steuerelemente

Formulare und Steuerelement erlauben die Entwicklung eigener Dialogfenster. Dadurch erhöht sich die Benutzerfreundlichkeit einer Anwendung um ein Vielfaches.

1.3.1 Formulare

Bisher haben wir die Eingaben *hard codiert* und die Ausgaben im Direktfenster durchgeführt. Nun soll das Handling eine sinnvolle Form bekommen. Für einen gefälligen Dialog benutzen wir das Formular-Objekt *UserForm*.

Formulare werden ebenso wie Module in ein Projekt eingefügt. Allerdings werden sie im Projektordner Formulare gesammelt (Abb. 1.23). Ein Formular besitzt wie ein Modul ein Code-Fenster. Zusätzlich aber auch eine Oberfläche, auf der Steuerelemente aus der Werkzeugsammlung platziert werden können.

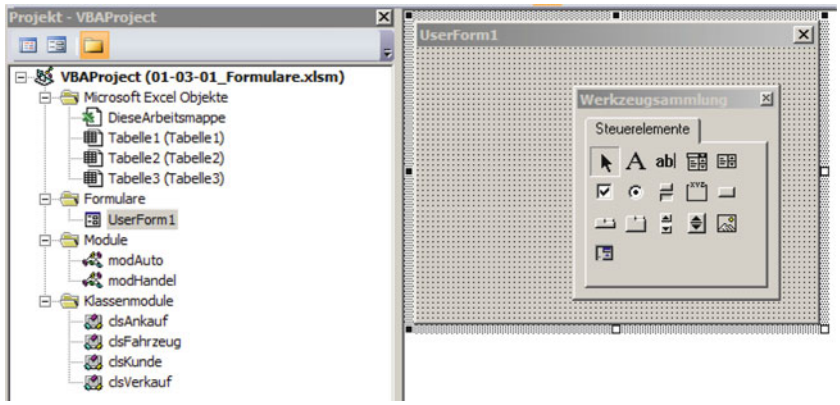


Abb. 1.23 Ein Formular im Projekt mit Oberfläche und Werkzeugsammlung

Die wichtigsten ActiveX-Steuerelemente sind im Anhang 1, Tab. A1.9 dargestellt.

Im Projekt-Explorer befinden sich oben links (siehe Abb. 1.23) zwei Schaltflächen, mit denen zwischen der Darstellung des Code-Fensters und der Darstellung der Oberfläche umgeschaltet werden kann.

Unser Hauptformular für die Planung bekommt den Namen *frmPlanung* (Abb. 1.24). Aufgerufen wird es in der Prozedur *Auto_Open* (Code 1.21). Die Anweisung *Load* lädt das Formular in den Arbeitsspeicher. Danach können bereits Wertzuweisungen durchgeführt werden. Doch sichtbar wird das Formular erst mit der Methode *Show*. Zu sehen ist danach lediglich ein Fenster mit der für Windows üblichen Schließfläche oben rechts. Ein Klick darauf beendet die Darstellung. Ebenso sichtbar ist ein kleines Fenster mit dem Namen Werkzeugsammlung. Es enthält Steuerelemente, mit denen wir später die Oberfläche des Formulars gestalten.

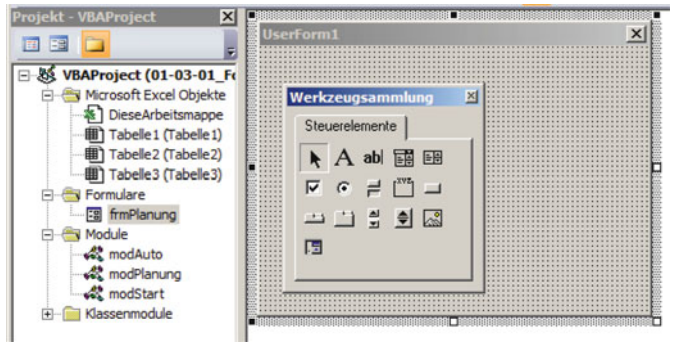


Abb. 1.24 Die neue UserForm Planung mit Werkzeugsammlung

Code 1.21 Die neue Version der Prozedur Auto_Open im Modul Auto

```
Sub Auto_Open()
   Set colAufgaben = New Collection
   Set colPersonen = New Collection
   Set colPlanungen = New Collection
   ErfasseAufgaben
   ErfassePersonen
   Load frmPlanung
   frmPlanung.Show
End Sub
```

Im Eigenschaftsfenster erhält das Formular unter der Eigenschaft *Caption* die Bezeichnung *frmPlanung*.

1.3.2 Schaltflächen

Das erste Steuerelement auf dem Formular soll nun das Schaltflächen-Objekt *CommandButton* sein, mit der das Formular geschlossen wird. Mit der Maus wird die

Schaltfläche in der Werkzeugsammlung angeklickt und dann auf die Formularfläche gezogen (Abb. 1.25).

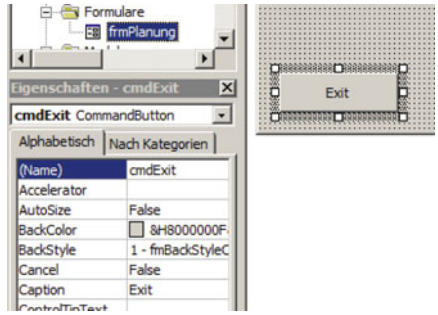


Abb. 1.25 Die Schaltfläche Exit auf dem Formular Planung

Steuerelemente sind ebenfalls Objekte und verfügen damit auch über Eigenschaften und Methoden. Zu einem markierten Steuerelement werden die Eigenschaften im Eigenschaftsfenster angezeigt und können dort auch verändert werden. Die Eigenschaften *Name* und *Caption* werden für das Beispiel auf *cmdExit* und *Exit* geändert.

Mit einem Doppelklick auf die Schaltfläche *Exit* schaltet der Explorer auf die Darstellung des Code-Fensters um und wir sehen eine Methode des Steuerelements mit dem Namen *cmdExit_Click*.

1.3.3 Ereignisse

Steuerelemente registrieren eine Vielzahl von Ereignissen mit entsprechenden Prozeduren, den Ereignisprozeduren auch *Events* genannt. Der Prozedurname einer Ereignisprozedur besteht aus der Konstruktion

```
'Syntax:
   Private Sub Objektname_Ereignis()

'Beispiel:
   Private Sub cmdExit_Click()
```

Wird ein Steuerelement im Code-Fenster ausgewählt (Abb. 1.26), dann können dort auch zugehörige Ereignisse ausgewählt werden. Mit dem Anklicken des Ereignisses installiert sich im Code-Fenster die entsprechende Event-Funktion.

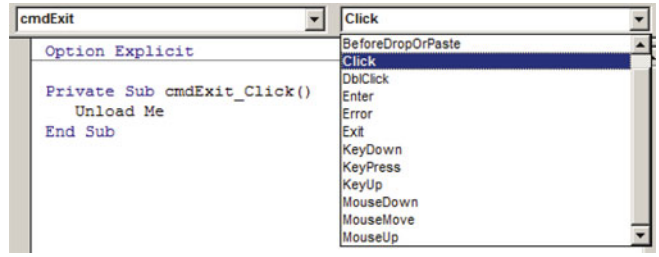


Abb. 1.26 Die Event-Funktion cmdExit_Click im Code-Fenster des Formulars Planung

Eine UserForm wird mit der Anweisung Unload geschlossen.

```
'Syntax:
   Unload Formularname
   Unload Me 'nur im Codefenster des Formulars

'Beispiel:
   Unload frmAufgabe
```

Da sich die Event-Funktion im Code-Fenster des Formulars befindet, kann statt des Formularnamens auch *Me* angegeben werden. Damit betitelt sich das Objekt selbst, in dem sich der Code befindet. Mit dem Aufruf der Excel-Mappe wird auch das Formular gestartet. Es liegt also nahe dieses Formular für alle weiteren Aktionen zu nutzen.

1.3.4 Listenfelder

Natürlich wollen wir wieder die vorhandenen Kunden angezeigt bekommen. Dazu benutzen wir das Listenfeld-Objekt *ListBox* (Abb. 1.27). Es bekommt den Namen *lbxKunden* und einen Platz auf dem Formular.

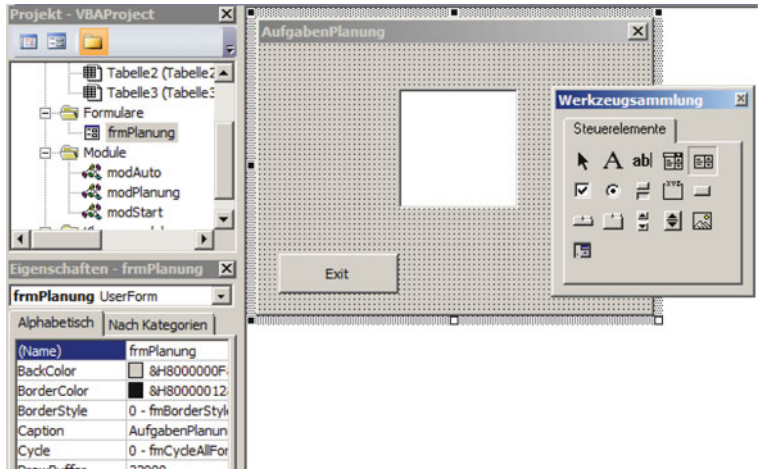


Abb. 1.27 Die ListBox lbxAufgaben auf dem Formular Planung

Code 1.22 Die Prozedur Auto_Open mit Initialisierung des Formulars Planung

```
Sub Auto_Open()
   Dim objAufgabe    As clsAufgabe

   Set colAufgaben = New Collection
   Set colPersonen = New Collection
   Set colPlanungen = New Collection

   ErfasseAufgaben 'erstelle Aufgaben
   ErfassePersonen 'erstelle Personen

   Load frmPlanung 'lade Formular
   With frmPlanung.lbxAufgaben
      .Clear 'lösche Listbox
      For Each objAufgabe In colAufgaben
         .AddItem objAufgabe.Key & _
         " / " & objAufgabe.Stufe & _
         " - " & objAufgabe.Titel
      Next
   End With
   frmPlanung.Show
End Sub
```

In unserer Startprozedur *Auto_Open* erfassen wir unsere Aufgaben in der Prozedur *ErfasseAufgaben* (im Modul *modStart*) und danach existiert auch eine Objektliste der Aufgaben. Mit der Anweisung *Load frmPlanung* existiert unser Formular bereits im Arbeitsspeicher einschließlich unserer ListBox *lbxAufgaben*. Wir können bereits hier unsere Listbox füllen (Code 1.22).

Dazu nutzen wir die eingerichteten Properties der Klasse *Aufgaben*. Die ListBox bekommt auf dem Formular noch eine Überschrift durch ein Beschriftungsfeld. Das wird im Eigenschaftsfenster noch etwas farblich angepasst.

Mit jedem Neustart der Arbeitsmappe wird nun das Formular geöffnet und zeigt in der Listbox die hard codierten Aufgabendaten (Abb. 1.28).

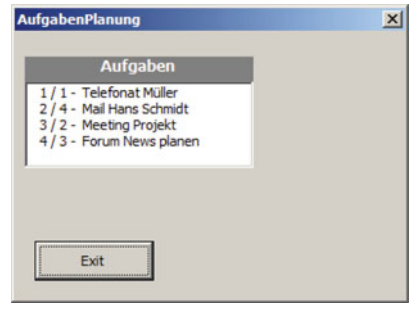


Abb. 1.28 Das Formular Planung nach dem Start

1.3.5 Textfelder

Doch damit ist es nicht getan, denn es müssen vorhandene Aufgaben auch geändert, gelöscht oder neu erfasst werden. Wir erstellen ein neues Formular-Objekt für die Aufgaben-Daten (Abb. 1.29). Es bekommt den Namen *frmAufgabe* und die Textfelder *tbxNum*, *tbxTitel, tbxStufe, tbxZeit, tbxTyp* und *tbxDatei*. Beschriftungsfelder davor beschreiben den Inhalt der TextBoxen. Da sie in der Programmierung nicht auftauchen, müssen wir ihnen auch keine Namen geben.

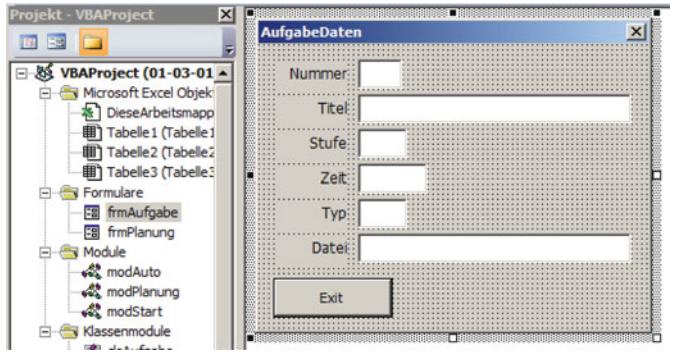


Abb. 1.29 Das neue Formular Aufgabe

Außerdem erhält das Formular wieder eine Schaltfläche *Exit* mit der gleichen Funktionalität wie die Schaltfläche im Formular *frmPlanung*. Aufgerufen werden soll dieses Formular aus dem Startformular. Wir könnten dies wieder mit einer Schaltfläche erreichen, doch wir wählen diesmal ein Event der ListBox, nämlich einen *Doppelklick* auf einen Eintrag in der Listbox *lbxAufgaben*. Damit kennzeichnen wir gleichzeitig die Aufgabe, deren Daten bearbeitet werden sollen. Die Daten selbst holen wir uns aus der Objektliste (Code 1.23).

Code 1.23 Die Event-Prozedur zum Doppelklick auf der ListBox Aufgaben

```
Private Sub lbxAufgaben_DblClick(ByVal Cancel As MSForms.ReturnBoolean)
   Dim objAufgabe     As clsAufgabe
   Dim sText          As String
   Dim sNum           As String

   Load frmAufgabe
   sText = lbxAufgaben.List(lbxAufgaben.ListIndex)
   sNum = Trim(Left(sText, InStr(sText, "/") - 1))
   Set objAufgabe = colAufgaben.Item(sNum)
   With frmAufgabe
      .tbxNum = objAufgabe.Key
      .tbxTitel = objAufgabe.Titel
```

```
        .tbxStufe = objAufgabe.Stufe
        .tbxZeit = objAufgabe.Zeit
        .tbxTyp = objAufgabe.Typ
        .tbxDatei = objAufgabe.Datei
    End With
    frmAufgabe.Show
    Set objAufgabe = Nothing
End Sub
```

Das Ergebnis eines Doppelklicks auf den zweiten Eintrag der ListBox zeigt Abb. 1.30. Die persönliche Gestaltung der Oberfläche überlasse ich dem Leser. Mir geht es in erster Linie um die Funktionalität.

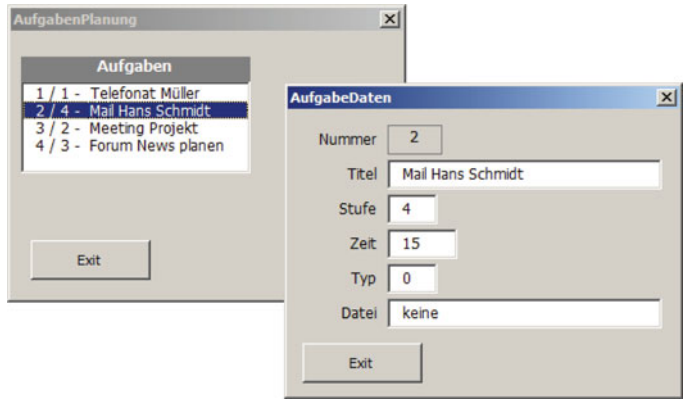


Abb. 1.30 Der Aufruf des Formulars Aufgabe

Nun müssen wir noch die zuvor beschriebenen Funktionen realisieren. Beginnen wir mit der Änderung. Dazu erhält das Formular Aufgabe eine weitere Schaltfläche *cmdSpeichern*, die die Inhalte der Textboxen an die Objektliste überträgt auf dem Weg über ein instanziiertes Objekt (Code 1.24). Eigentlich darf die Aufgabennummer nicht änderbar sein. Wir können dies dadurch erreichen, dass wir statt einer *Textbox* ein Bezeichnungsfeld-Objekt *Label* verwenden.

Code 1.24 Speicher-Prozedur im Formular Aufgabe

```
Private Sub cmdSave_Click()
    Dim objAufgabe     As clsAufgabe
    Dim sText          As String
    Dim sNum           As String
    sNum = Trim(tbxNum)
```

```
'Fehlerhandling
   On Error Resume Next
   Set objAufgabe = colAufgaben.Item(sNum)
   If Err.Number > 0 Then
      Set objAufgabe = New clsAufgabe          'instanziiere Aufgabe
      objAufgabe.Erfasse sNum, "Neu", 0, 0, 0, ""
                                               'schreibe Defaultwerte
      colAufgaben.Add objAufgabe, sNum         'Aufgabe in Sammlung
      frmPlanung.lbxAufgaben.AddItem sNum      'Aufgabe in Liste
      frmPlanung.lbxAufgaben.ListIndex = _
        frmPlanung.lbxAufgaben.ListCount - 1   'Listenindex auf count-1
   End If
   On Error GoTo 0

'Daten übernehmen und speichern
   With objAufgabe
      .Key = tbxNum
      .Titel = tbxTitel
      .Stufe = tbxStufe
      .Zeit = tbxZeit
      .Typ = tbxTyp
      .Datei = tbxDatei
   End With

'Listeneintrag
   frmPlanung.lbxAufgaben.List(frmPlanung.lbxAufgaben.ListIndex) = _
      sNum & " / " & tbxStufe & " - " & tbxTitel
   Set objAufgabe = Nothing
   Unload Me
End Sub
```

Ähnlich handhaben wir den Löschvorgang einer Aufgabe mit einer Schaltfläche (Code 1.25).

Code 1.25 Lösch-Prozedur im Formular Aufgabe

```
Private Sub cmdDelete_Click()
   Dim objAufgabe    As clsAufgabe
   Dim sText         As String
   Dim sNum          As String

   sNum = Trim(tbxNum)
   Set objAufgabe = colAufgaben.Item(sNum)
   colAufgaben.Remove (sNum)
   frmPlanung.lbxAufgaben.RemoveItem _
      (frmPlanung.lbxAufgaben.ListIndex)
```

```
   Set objAufgabe = Nothing
   Unload Me
End Sub
```

Mit der Löschprozedur sind wir in der Lage auch alle Aufgaben aus unserer Liste zu streichen. Dann aber führt ein Doppelklick in der ListBox zu einer Fehlermeldung. Dieser Fehler muss durch eine Kontrollabfrage abfangen werden (Code 1.26).

Code 1.26 Die erweiterte Event-Prozedur zum Doppelklick auf der ListBox Aufgaben

```
Private Sub lbxAufgaben_DblClick(ByVal Cancel As MSForms.ReturnBoolean)
   Dim objAufgabe    As clsAufgabe
   Dim sText         As String
   Dim sNum          As String
   Dim iNum          As Integer

   iNum = lbxAufgaben.ListIndex
   If iNum < 0 Then
      MsgBox "Fehler bei der Auswahl!"
      Exit Sub
   End If

   Load frmAufgabe
   sText = lbxAufgaben.List(lbxAufgaben.ListIndex)
   sNum = Trim(Left(sText, InStr(sText, "/") - 1))
   Set objAufgabe = colAufgaben.Item(sNum)
   With frmAufgabe
      .tbxNum = objAufgabe.Key
      .tbxTitel = objAufgabe.Titel
      .tbxStufe = objAufgabe.Stufe
      .tbxZeit = objAufgabe.Zeit
      .tbxTyp = objAufgabe.Typ
      .tbxDatei = objAufgabe.Datei
   End With
   frmAufgabe.Show
   Set objAufgabe = Nothing
End Sub
```

Der Fehler tritt auch auf, wenn wir in der ListBox außerhalb der Einträge einen Doppelklick ausführen, was aber nur beim Start und mit wenigen Einträgen passieren kann. Danach liegt der Fokus immer auf einem Eintrag, soweit vorhanden. Zur Eingabe einer neuen Aufgabe führen wir eine Schaltfläche *cmdNeueAufgabe* auf dem Startformular ein (Abb. 1.31). Die Event-Prozedur zum Klick zeigt Code 1.27. Zur Eingabe selbst benutzen

wir weiterhin das Formular *Aufgabe* und auch die Schaltfläche zum Speichern. Allerdings müssen wir die Event-Funktion noch anpassen (Code 1.28).

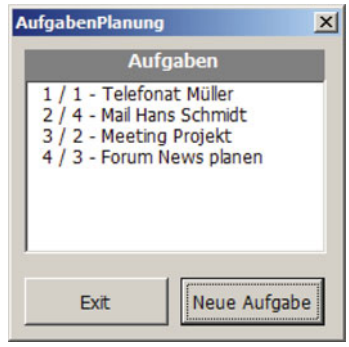


Abb. 1.31 Neue Version des Formulars Planung

Code 1.27 Die Event-Prozedur der Schaltfläche NeueAufgabe

```
Private Sub cmdNeueAufgabe_Click()
   Dim sNum          As String
   Dim iNum          As Integer

   iNum = colAufgaben.Count
   If iNum < 0 Then
      MsgBox "Fehler bei der Auswahl!"
      Exit Sub
   End If
   sNum = Trim(Str(iNum + 1))
   Load frmAufgabe
      frmAufgabe.tbxNum = sNum
   frmAufgabe.Show
End Sub
```

Code 1.28 Die veränderte Event-Funktion Speichern

```
Private Sub cmdSave_Click()
   Dim objAufgabe    As clsAufgabe
   Dim sText         As String
   Dim sNum          As String
   sNum = Trim(tbxNum)
```

```
'Fehlerhandling
   On Error Resume Next
   Set objAufgabe = colAufgaben.Item(sNum)
   If Err.Number > 0 Then
      Set objAufgabe = New clsAufgabe          'instanziiere Aufgabe
      objAufgabe.Erfasse sNum, "Neu", 0, 0, 0, ""
                                               'schreibe Defaultwerte
      colAufgaben.Add objAufgabe, sNum         'Aufgabe in Sammlung
      frmPlanung.lbxAufgaben.AddItem sNum      'Aufgabe in Liste
      frmPlanung.lbxAufgaben.ListIndex = _
        frmPlanung.lbxAufgaben.ListCount - 1   'Listenindex auf count-1
   End If
   On Error GoTo 0

'Daten übernehmen und speichern
   With objAufgabe
      .Key = tbxNum
      .Titel = tbxTitel
      .Stufe = tbxStufe
      .Zeit = tbxZeit
      .Typ = tbxTyp
      .Datei = tbxDatei
   End With

'Listeneintrag
   frmPlanung.lbxAufgaben.List(frmPlanung.lbxAufgaben.ListIndex) = _
      sNum & " / " & tbxStufe & " - " & tbxTitel
   Set objAufgabe = Nothing
   Unload Me
End Sub
```

1.3.6 Kombinationsfelder

Für manche Eingaben lassen sich Vorgaben definieren. In diesem Beispiel wollen wir die Stufen der Wichtigkeit von 1 (sehr wichtig) bis 6 (sehr unwichtig) vorgeben. Dazu benutzen wir das Kombinationsfeld-Objekt *ComboBox*, bei dem die Eingabewerte aus einer Liste ausgewählt werden können. Ein Kombinationsfeld ist im Prinzip ein Listenfeld mit festen Werten zur Auswahl, gekoppelt mit einem Textfeld zur Eingabe.

Wir ersetzen die Textbox *tbxStufe* durch ein Kombinationsfeld gleichen Namens. Das Formular *Aufgabe* verfügt wie jedes andere Formular über die Event-Funktion *UserForm_Initialize*. In dieser füllen wir das Kombinationsfeld (Code 1.29).

Code 1.29 Das Initialisierungs-Event der UserForm Aufgabe

```
Private Sub UserForm_Initialize()
   With tbxStufe
      .Clear
      .AddItem "1"
      .AddItem "2"
      .AddItem "3"
      .AddItem "4"
      .AddItem "5"
      .AddItem "6"
   End With
End Sub
```

Nach dem Aufruf des Formulars kann im Eingabefeld zur Stufe eine Auswahlliste aufgeklappt und mit einem Klick ein Wert ausgewählt werden (Abb. 1.32).

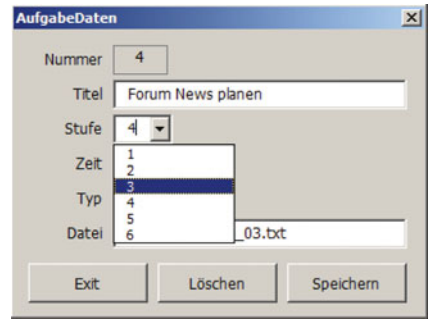


Abb. 1.32 Die Auswahlliste zur Stufenwahl

Die Verwaltung der Personen und Planungen muss ähnlich programmiert werden. Immer wird die Oberfläche eines Formulars mit den erforderlichen Steuerelementen belegt und deren relevante Ereignisprozeduren eingebunden.

Bevor wir uns mit den eigentlichen Aktionen der Aufgaben-Bearbeitung beschäftigen, wollen wir uns eine weitere Welt in die Programmierung holen, die Anwendungsobjekte. Denn bisher funktioniert unser System zwar, doch nach dem Ausschalten sind alle Änderungen verloren und wir beginnen wieder bei null. Das passiert nicht, wenn wir die Anwendungs-Objekte mit ins Boot holen. Sie besitzen ebenfalls eine Darstellung, deren Inhalte im Gegensatz zum Formular gespeichert werden können und damit eine Archivierung der aktuellen Daten ermöglichen.

1.4 Klassen und Bibliotheken

1.4.1 Bibliotheken

Bisher haben wir alle Klassen als gegeben vorausgesetzt, wie z. B. Formulare oder Steuerelemente. Dass wir sie aber direkt nutzen können liegt an der Grundeinstellung, hier einer Excel-Anwendung. Diese verfügt bereits über Verweise auf Bibliotheks-Klassen *Libraries*, in denen die verwendeten Klassen definiert sind. Unter dem Register *Extras* der Entwicklungsumgebung findet sich der Eintrag *Verweise*, der das Dialogfenster *Verweise – VBAProjekt* (Abb. 1.33) öffnet.

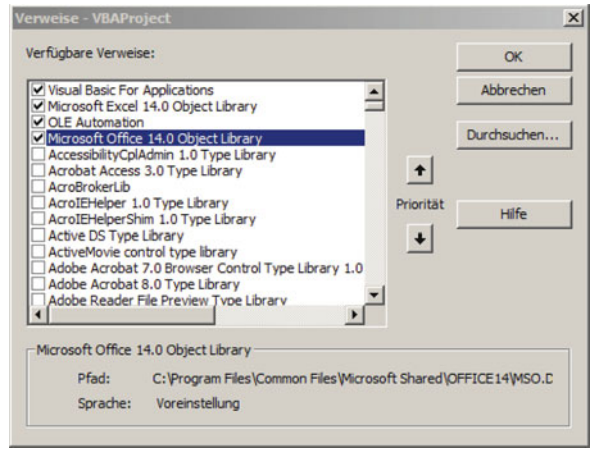


Abb. 1.33 Dialogfenster zur Verwaltung der genutzten Bibliotheken

Dort sind einige Bibliotheken bereits angewählt, so dass wir ihre Klassen in den Code-Fenstern nutzen können. Welche wir für eine Nutzung verwenden wollen, richtet sich nach dem Projektzweck.

1.4.2 Objektkatalog

Die angewählten Bibliotheken finden sich auch im Objektkatalog wieder, der uns als Nachschlagewerk in der Entwicklungsumgebung unter dem Register *Ansicht* zur Verfügung steht (Abb. 1.34).

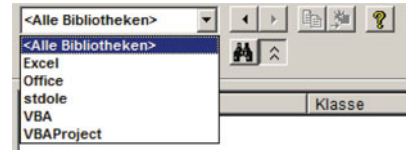


Abb. 1.34 Auswahl der Bibliotheken im Objektkatalog

Im zweiten Textfeld kann ein Suchbegriff eingegeben werden, hier *UserForm* (Abb. 1.35). In der Tabelle darunter werden die gefundenen Einträge dargestellt. So erfahren wir, dass die *UserForm* ein Element der Klasse *VBA.Global* ist und die Klasse über die Methoden *Load* und *Unload* verfügt.

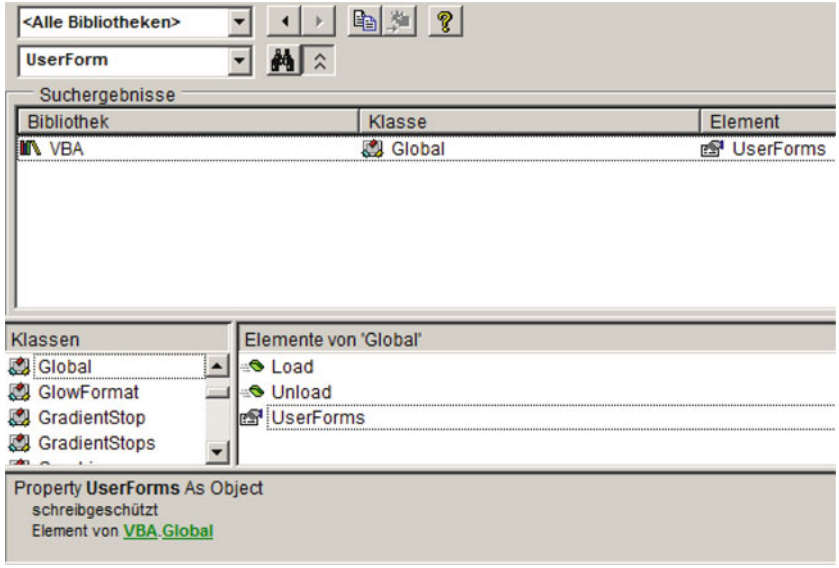


Abb. 1.35 Suchergebnisse im Objektkatalog

Wird ein Eintrag in Suchergebnisse markiert, so lassen sich mit der Schaltfläche *?* weitere Informationen abrufen. Dort wird die Syntax ebenso wie Parameter und die Anwendung erklärt. Meistens sind auch einfach Beispiele angegeben.

Im Objektkatalog wird der Typ der Elemente durch ein Symbol gekennzeichnet. In Abb. 1.36 sind die verwendeten Symbole und ihre Bedeutung wiedergegeben.

Abb. 1.36 Verwendete Symbole im Objektkatalog

Wenn wir nun Elemente aus der Word-Bibliothek in Excel nutzen wollen und durch den VBA-Editor Hilfe erwarten, dann müssen wir die Bibliothek unter Verweise anwählen (Abb. 1.37).

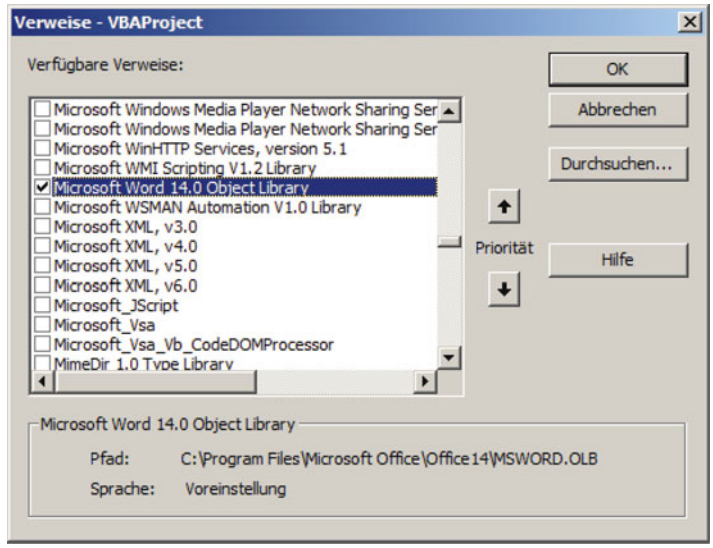


Abb. 1.37 Auswahl der Word-Bibliothek

Danach stehen uns die Klassen und Objekte dieser Bibliothek zur Verfügung. Bereits bei der Definition werden sie vorgegeben.

1.4.3 Intellisense-Funktion

Die VBA-Entwicklungsumgebung verfügt über eine logische Programmierhilfe, die als Intellisense-Funktion bezeichnet wird. Geben wir in ein Code-Fenster etwas Text ein, so können wir mit den Tasten STRG+LEERTASTE eine Hilfe zu den Möglichkeiten einblenden und in dieser Liste mit einem Doppelklick eine Auswahl treffen (Abb. 1.38).

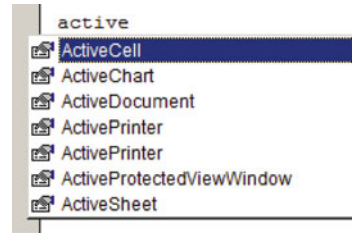


Abb. 1.38 Auswahl mit der Intellisense-Funktion (STRG+LEERTASTE)

Die Intellisense-Funktion wird auch wirksam wenn sich die Eingabemöglichkeiten auf eine Auswahl beziehen. Bei der Definition von Variablen und Objekten z. B. werden die möglichen Datentypen zur Auswahl eingeblendet (Abb. 1.39).

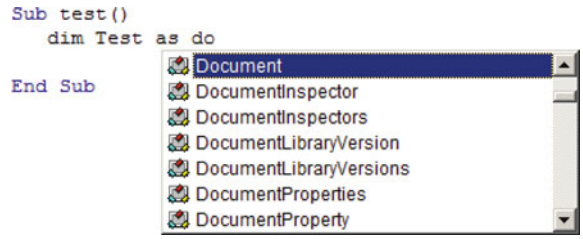


Abb. 1.39 Automatische Auswahl mit der Intellisense-Funktion

Und da wir zuvor einen Verweis auf die Word-Bibliothek erstellt haben, wird auch das Word-Objekt *Document* zur Auswahl angeboten. Doch die Intellisense-Funktion hilft noch weiter, wenn das definierte Objekt im Programmcode verwendet wird. Dann werden zum Objekt als Auswahl mögliche Unterobjekte, Eigenschaften und Methoden eingeblendet (Abb. 1.40).

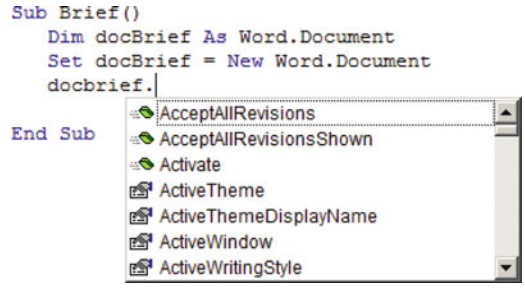


Abb. 1.40 Objektbezogene Auswahl mit der Intellisense-Funktion

Solange wir mit einer Anwendung arbeiten genügt es nur den Objektnamen des Objekttyps anzugeben, z. B. in Word nur *Document*. Nutzen wir Objekte einer weiteren Anwendung dann werden wir, der sauberen Definition wegen, immer auch den Bibliotheksnamen davor setzen, also z. B. *Word.Document*.

1.4.4 Frühe Bindung

Die objektbezogene Auswahl ist nur möglich, wenn in der Definition der entsprechende Objekttyp angegeben wurde. Diese Art der Bindung zwischen Objektvariable und Objekttyp wird als *Frühe Bindung (early binding)* bezeichnet. Auf der einen Seite unterstützt sie damit die Auswahl bei der Programmierung durch die Intellisense-Funktion, auf der anderen Seite setzt sie den Verweis auf die Word-Bibliothek voraus. Ebenso ist auch die Nutzung von Konstanten aus der Bibliothek möglich.

Die Prozedur *Brief1* (Code 1.30) erzeugt ein Word-Dokument mit Früher Bindung und aktiviert es zur Eingabe. Eine MessageBox stoppt den Ablauf für eine Eingabe. Doch

Vorsicht bei der Eingabe. Die Enter-Taste setzt den Ablauf fort. Die VBA-Konstanten *vbQuestion* und *vbYesNo* stehen für ein Fragezeichen und die Schaltflächen *Ja* und *Nein*. Die Wahl *Ja* speichert den Text unter dem angegebenen Pfad und Namen im Format *wdFormatText* als Textdokument *Test1.txt*.

Wird die Prozedur ohne den Verweis auf die Word-Bibliothek gestartet, dann kommt es zu einer Fehlermeldung. Daher nutzt man für die Entwicklung gerne die *Frühe Bindung* und stellt am Ende auf die *Späte Bindung* um.

Code 1.30 Die Prozedur erzeugt ein Word-Dokument mit Früher Bindung

```
Sub Brief1()
   Dim docBrief As Document
   Dim vAntwort As Variant

   Set docBrief = New Document
   docBrief.Activate
   vAntwort = MsgBox("Inhalt speichern?", vbQuestion + vbYesNo)
   If vAntwort = vbYes Then
      docBrief.SaveAs2 _
         Filename:="C:\Temp\Test1", _
         FileFormat:=wdFormatText
   End If
   Set docBrief = Nothing
End Sub
```

1.4.5 Späte Bindung

Bei der *Späten Bindung (late binding)* wird eine Objektvariable in der Definition mit dem allgemeinen Datentyp *Object* bestimmt. Erst bei der Instanziierung wird der endgültige Datentyp bekannt gegeben. Ein Verweis auf die Bibliothek ist dabei nicht notwendig. Dadurch gibt es auch keine Unterstützung durch die Intellisense-Funktion. Ebenso muss die Hierarchie der Objekte der jeweiligen Anwendung beachtet werden (Code 1.31).

Code 1.31 Die Prozedur erzeugt ein Word-Dokument mit Später Bindung

```
Sub Brief2()
   Dim docBrief As Object
   Dim vAntwort As Variant

   Set docBrief = CreateObject("Word.Document")
   docBrief.Activate
   vAntwort = MsgBox("Inhalt speichern?", 36)
```

```
   If vAntwort = vbYes Then
      docBrief.SaveAs2 _
         Filename:="C:\Temp\Test2", _
         FileFormat:=2
   End If
   Set docBrief = Nothing
End Sub
```

Die VBA-Konstanten *vbQuestion* und *vbYesNo* könnten noch benutzt werden, da die VBA-Objekt-Bibliothek immer vorhanden ist. Da sie aber die Werte *vbQuestion=32* und *vbYesNo=2* besitzen, lässt sich auch ihre Summe eintragen. Die Word-Konstante *wdFormatText=2* muss aber in jedem Fall ersetzt werden, da es die Anbindung der Word-Bibliothek nicht gibt.

Für alle nachfolgenden Beispiele wird die frühe Bindung verwendet.

1.4.6 Verweise handhaben

VBA wäre nicht VBA, wenn es nicht eine programmierbare Lösung für das Problem der Bindung gäbe. Die Prozedur (Code 1.32) zeigt zunächst einmal alle vorhandenen Verweise im Direktfenster. Erscheint beim Start der *Laufzeitfehler '1004' : Der programmatische Zugriff auf das Visual Basic-Projekt ist nicht sicher*, so muss zuvor unter *Optionen / Sicherheitscenter / Einstellungen für das Sicherheitscenter* unter *Einstellungen für Makros* die Option *Zugriff auf das VBA-Projektobjektmodell vertrauen* gewählt werden.

Code 1.32 Die Prozedur zeigt vorhandene Verweise

```
Sub ShowReferences()
   Dim objBook As Object
   Dim refAll  As Object
   Dim refIst  As Object

   For Each objBook In Workbooks
      Set refAll = objBook.VBProject.References
      For Each refIst In refAll
         Debug.Print objBook.Name, _
            refIst.Name, _
            refIst.Type, _
            refIst.GUID , _
            refIst.FullPath
      Next
   Next
End Sub
```

Die damit ausgegebene GUID (**g**lobal **u**nique **id**entifier) ist auf allen Rechnern weltweit gültig. Diese Methode ist sicherer als mit den Installationspfaden zu arbeiten. Die müssen nicht immer auf allen Rechnern stimmen.

Die Prozedur (Code 1.33) prüft, ob alle Verweise in Ordnung sind.

Code 1.33 Die Prozedur prüft die vorhandenen Verweise

```
Sub VerifyReferences()
   Dim objRef As Object

   For Each objRef In ThisWorkbook.VBProject.References
      If objRef.IsBroken Then _
         MsgBox "Der Verweis auf " & objRef.Name & _
         " ist defekt.", vbExclamation, "Achtung"
   Next
End Sub
```

Sollen Verweise auf Objektbibliotheken installiert werden, so ist zunächst zu prüfen ob sie nicht bereits installiert sind. Sind sie gesetzt, dann sollte geprüft werden ob sie in Ordnung sind. Sind sie nicht in Ordnung, dann müssen sie gelöscht und wieder neu gesetzt werden (Code 1.34).

Code 1.34 Die Prozedur prüft und setzt Verweise

```
Private Sub InstallReferences()
   Dim objRef        As Object

   Dim sRefGUID(2)   As String
   Dim iCount        As Integer
   'Office / MSO.DLL
   sRefGUID(1) = "{2DF8D04C-5BFA-101B-BDE5-00AA0044DE52}"
   'Word / MSWORD.OLB
   sRefGUID(2) = "{00020905-0000-0000-C000-000000000046}"

   'check bevor
   For Each objRef In ThisWorkbook.VBProject.References
      If objRef.isbroken Then
         objRef.Remove
      Else
         For iCount = 1 To 2
            If objRef.GUID = sRefGUID(iCount) Then
               sRefGUID(iCount) = ""
            End If
         Next iCount
      End If
   Next
```

```
   'set references
   For iCount = 1 To 2
      If Not sRefGUID(iCount) = "" Then
         ThisWorkbook.VBProject.References.AddFromGuid _
            GUID:=sRefGUID(iCount), Major:=2, Minor:=0
      End If
   Next iCount

   'check after
   For Each objRef In ThisWorkbook.VBProject.References
      If objRef.isbroken Then
         MsgBox "Der Verweis auf " & objRef.Name & _
            " ist defekt.", vbExclamation, "Achtung"
      End If
   Next

   ShowReferences
End Sub
```

So wie die Verweise installiert werden, lassen sie sich auch wieder deinstallieren. Dazu wird die Methode *Remove* verwendet (Code 1.35).

Code 1.35 Die Prozedur löscht vorhandene Verweise

```
Sub DeinstallReferences()
   Dim objRef        As Object

   Dim sRefGUID(2)   As String
   Dim iCount        As Integer
   'Office / MSO.DLL
   sRefGUID(1) = "{2DF8D04C-5BFA-101B-BDE5-00AA0044DE52}"
   'Word / MSWORD.OLB
   sRefGUID(2) = "{00020905-0000-0000-C000-000000000046}"

   'check bevor
   On Error Resume Next
   For Each objRef In ThisWorkbook.VBProject.References
      For iCount = 1 To 2
         If objRef.GUID = sRefGUID(iCount) Then
            ThisWorkbook.VBProject.References.Remove objRef
         End If
      Next iCount
   Next
   ShowReferences
End Sub
```

1.4.7 Windows API

Bibliotheken mit ihren Klassen erlauben die objektorientierte Programmierung (OOP) von Objekten einer oder mehrerer Office-Anwendungen. Über das Application-Objekt ist auch ein Zugriff auf Teile des Betriebssystems möglich. Allerding in geringem Maße. Das *Windows API* (*Application Programming Interface*) bietet eine große Anzahl an Funktionen, die einen Zugriff auf kleinste Details des Betriebssystems erlauben.

Darin liegt aber auch eine große Gefahr. Mit dem Zugriff über die API werden oft Standard-Sicherheitsvorkehrungen umgangen, die einen kompletten Absturz verhindern. Eine Speicherung des Programmcodes vor dem Test ist also unabdingbar. Die meisten API-Funktionen sind für die C und C++ Programmierung geschrieben, erlauben aber auch VBA den Zugriff. Dadurch gibt es Unterschiede zu den Aufrufen von VBA-Funktionen. Zum Arbeiten mit der API gehört also eine umfassende Dokumentation. Auf dem Markt gibt es zahllose Bücher zu diesem Thema.

An dieser Stelle sollen ein paar einfache Beispiele die Möglichkeiten zeigen, die in der API stecken. Wenn wir später eine API-Funktion benötigen, dann wird sie dort umfassend erklärt. API-Funktionen müssen immer vor ihrer Nutzung deklariert werden. Dies erledigt die Anweisung *Declare* und die Konstruktion erinnert uns an C/C++. Auch hier kommen wir ohne einen Bibliothekshinweis nicht aus (kernel32). Das erste Beispiel liest die Zeit in Millisekunden, die eine Programmschleife benötigt um 1.000.000-mal die API-Funktion GetTickCount auszuführen (Code 1.36).

Code 1.36 Eine API-Funktion liest den TimeTicker

```
Private Declare Function GetTickCount Lib "kernel32" () As Long

Public Sub Zeitverlauf()
   Dim lTimeStart    As Long
   Dim lTimeEnd      As Long
   Dim lLoop         As Long

   lTimeStart = GetTickCount
   For lLoop = 1 To 1000000
      lTimeEnd = GetTickCount
   Next lLoop
   Debug.Print (lTimeEnd - lTimeStart) / 1000; " Sec"
End Sub
```

Etwas komplexer aber dennoch relativ einfach ist die Konstruktion zur Bestimmung der Auflösung des ersten Bildschirms (Code 1.37).

Code 1.37 API-Funktionen bestimmen die Bildschirmauflösung

```
Declare Function GetDeviceCaps Lib "gdi32" _
   (ByVal hdc As Long, ByVal nIndex As Long) As Long
Declare Function GetDC Lib "user32" _
   (ByVal hwnd As Long) As Long
Declare Function ReleaseDC Lib "user32" _
   (ByVal hwnd As Long, ByVal hdc As Long) As Long

Const HORZRES = 8
Const VERTRES = 10

Public Sub ScreenSize()
   MsgBox ScreenResolution()
End Sub

Function ScreenResolution()
   Dim lRval   As Long
   Dim lDc     As Long
   Dim lHSize  As Long
   Dim lVSize  As Long

   lDc = GetDC(0&)
   lHSize = GetDeviceCaps(lDc, HORZRES)
   lVSize = GetDeviceCaps(lDc, VERTRES)
   lRval = ReleaseDC(0, lDc)
   ScreenResolution = lHSize & " x " & lVSize
End Function
```

Das nächste Beispiel zeigt den Zugriff auf die Windows-Registry und liest dort den Username (Code 1.38). Zum Abschluss bestimmt eine API-Konstruktion die Seriennummer der Festplatte (Code 1.39).

Code 1.38 Eine API-Funktion bestimmt den Username aus der Registry

```
Declare Function GetUserName Lib "advapi32.dll" _
   Alias "GetUserNameA" (ByVal lpBuffer As String, _
   nSize As Long) As Long

Sub ShowUserName()
   Dim sBuffer    As String * 100
   Dim lBuffer    As Long

   lBuffer = 100
   GetUserName sBuffer, lBuffer
   MsgBox Left(sBuffer, lBuffer - 1)
End Sub
```

Code 1.39 Eine API-Funktion bestimmt die Seriennummer der Festplatte

```
Declare Function GetVolumeInformationA Lib "kernel32" _
   (ByVal lpRootPathName As String, _
   ByVal lpVolumeNameBuffer As String, _
   ByVal nVolumeNameSize As Long, _
   lpVolumeSerialNumber As Long, _
   lpMaximumComponentLength As Long, _
   lpFileSystemFlags As Long, _
   ByVal lpFileSystemNameBuffer As String, _
   ByVal nFileSystemNameSize As Long) As Long

Public Sub SerienNummer()
   Dim lSerialNumber As Long

   GetVolumeInformationA "C:\", _
      vbNullString, 0, SerialNumber, 0, 0, vbNullString, 0
   MsgBox SerialNumber
End Sub
```

1.4.8 Component Object Model

Das *Component Object Model* (COM) erlaubt u. a. den objektorientierten Zugriff auf Computerdienste mit VBA. COM ist damit eine spezielle API die mehrere Klassen enthält, die unter anderem den Zugriff mit VBA erlaubt.

Als Beispiel soll eine Datei Test.txt aus dem Verzeichnis C:\Temp\Quelle in das Verzeichnis C:\Temp\Ziel verschoben werden (Code 1.40).

Code 1.40 Verschieben eines Files mit dem COM

```
Public Sub MoveFile()
   Dim objFso     As Object
   Dim objFile    As Object

'Instanziierung eines File-System-Objekts
   Set objFso = CreateObject("Scripting.FileSystemObject")
'Instanziierung des Unterobjekts objFile
   Set objFile = objFso.GetFile("C:\Temp\Quelle\Test.txt")
'Anwendung der Methode Move der Klasse FSO
   objFile.Move ("C:\Temp\Ziel\")
   MsgBox "File moved!"

   Set objFile = Nothing
   Set objFso = Nothing
End Sub
```

1.5 Formen

1.5.1 Standardformen

Die Objektliste *Shapes* beinhaltet die Zeichenobjekte einer Oberfläche, wie zum Beispiel alle Formen in einem Excel-Worksheet. In fast allen Anwendungen lassen sich Shapes einsetzen. Sie befinden sich im Register Einfügen in der Gruppe Illustrationen unter Formen (Abb. 1.41). Ich verwende sie hier stellvertretend für alle anderen Zeichenobjekte.

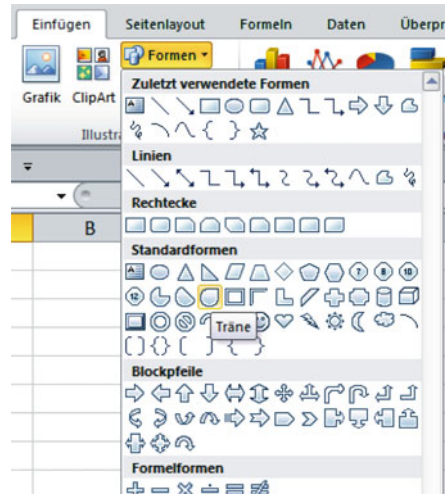


Abb. 1.41 Das Auswahlfenster für Formen

1.5.2 Formen hinzufügen

Die Methode *AddShape* erstellt eine neue Form auf der Oberfläche und hat die Syntax

```
'Syntax:
   Worksheet.Shapes.AddShape(Type, Left, Top, Width, Height)

'Beispiel:
   wshTab.Shapes.AddShape (1, 20, 50, 100, 50)
```

mit dem ersten Parameter *Type*, der die Form des Shapes bestimmt. Die VBA-Hilfe liefert unter dem Suchbegriff *MsoAutoShapeType-Aufzählung* eine Liste aller Typen. So kennzeichnet *msoShapeRectangle* ein Rechteck und hat den Wert 1. Für eine späte Bindung macht es Sinn, den Wert anzugeben. Hier steht noch einmal der Typ, später nur noch die Konstante.

Die Prozedur (Code 1.41) erzeugt mit der Methode *AddShape* ein Rechteck auf dem Arbeitsblatt *Tabelle1*, mit der Position in Pixel, von links 20, von oben 50 und den Maßen Breite 200, Höhe 100.

Code 1.41 Eine Prozedur zur Erzeugung von Shapes

```
Sub ErzeugeRechteck()
  Dim wshTab     As Worksheet
  Dim shpForm    As Shape

  LöscheAlleShapes
  Set wshTab = ThisWorkbook.Worksheets("Tabelle1")

'erzeuge Rechteck
'  1. Param. Formtyp statt msoShape... lassen sich auch
'  die direkten Werte setzen:
'  1 - Rechteck
'  2 - Raute us.w
  Set shpForm = wshTab.Shapes.AddShape _
     (msoShapeRectangle, 20, 50, 100, 60)
  With shpForm
  'Bezeichnung
     .Name = "Rechteck 1"
  'Rahmen
     With .Line
        .DashStyle = msoLineSolid
        .ForeColor.RGB = RGB(100, 100, 100)
     End With
     With .Fill
        With .ForeColor
           .RGB = RGB(200, 200, 200)
           .TintAndShade = 0
           .Brightness = 0
        End With
        .Transparency = 0
        .Solid
        .Visible = True
     End With
  'Text
     With .TextFrame
        With .Characters
           .Text = "TEST"
           With .Font
              .Name = "Arial"
              .ColorIndex = 1
              .Size = 14
```

```
                .Bold = True
            End With
        End With
        .HorizontalAlignment = xlHAlignCenter
        .VerticalAlignment = xlVAlignCenter
    End With
  End With
  Set wshTab = Nothing
End Sub
```

Gleichzeitig werden einige Attribute gesetzt. So erhält die Form einen Namen und einen durchgezogene Linie als Rand. Hintergrund und Vordergrund bekommen Farben nach der RGB-Funktion mit den Parametern (Rot-Anteil, Grün-Anteil, Blau-Anteil). Textrahmen und Text lassen sich ebenso gestalten (Abb. 1.42). Die gesetzten Attribute zeigen nur eine kleine Anzahl von vielen Möglichkeiten. Über den Objektkatalog lassen sich alle anderen erfragen. Bei dieser Gelegenheit habe ich die Schachtelung von *With*-Anweisung einmal ausgiebig genutzt.

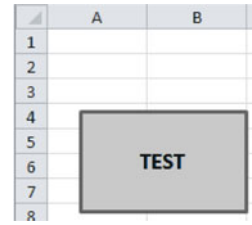


Abb. 1.42 Ein rechteckiges Shape auf dem Worksheet

Wie bei allen Objektlisten sind Formen auch über die *For-Each-Next*-Schleife ansprechbar, so wie in der Prozedur (Code 1.42), die alle Formen eines aktiven Sheets löscht.

Code 1.42 Die Prozedur löscht alle Formen auf dem aktiven Sheet

```
Sub LöscheAlleShapes()
  Dim wshTab     As Worksheet
  Dim shpForm    As Shape

  Set wshTab = ActiveSheet
  For Each shpForm In wshTab.Shapes
    shpForm.Delete
  Next
End Sub
```

1.5.3 Formstile

Die Prozedur (Code 1.43) erzeugt alle Formstile *ShapeStile* mit den Werten von 1 bis 183 (Abb. 1.43). Lediglich der Wert 138 ist ungültig. Der jeweilige Stil wird durch eine Zufallszahl von 1 bis 43 erzeugt.

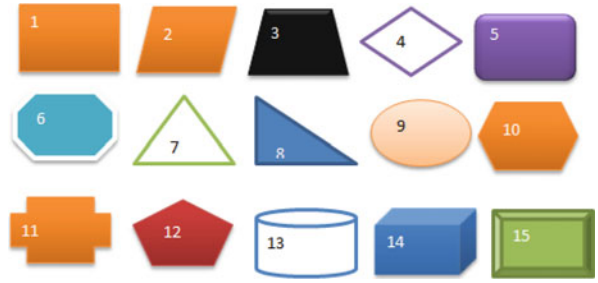


Abb. 1.43 Die ersten fünfzehn Shape-Stile

Code 1.43 Die Prozedur erzeugt alle Formstile auf dem Worksheet Tabelle1

```
Sub ErzeugeAlleFormen()
   Dim wshTab      As Worksheet
   Dim shpForm     As Shape
   Dim iCount      As Integer
   Dim lVPos       As Long

   Randomize
   LöscheAlleShapes
   Set wshTab = ThisWorkbook.Worksheets("Tabelle1")
   lVPos = 10
   For iCount = 1 To 183
      If Not iCount = 138 Then
         Set shpForm = wshTab.Shapes.AddShape _
            (iCount, 20, lVPos, 60, 40)
         With shpForm
            .TextFrame.Characters.Text = iCount
            .ShapeStyle = Int(Rnd() * 42 + 1)
         End With
         Set shpForm = Nothing
         lVPos = lVPos + 50
      End If
   Next
   Set wshTab = Nothing
End Sub
```

1.5.4 Formengruppen

Mehrere Formen lassen sich zum Gruppen-Objekt *Groups* zusammenfassen. Die Gruppe erbt die Attribute und Methoden der Formen. Eine Definition der Form, die eine Gruppe bildet, wird durch die Struktur

```
'Syntax:
   Worksheet.Shapes.Range(Array(Shape1, Shape2, ...)).Select

'Beispiel:
   wshTab.Shapes.Range(Array("L1", "T1")).Select
```

vorbereitet. Über das Unterobjekt *ShapeRange* der Selektion wird dann die Methode *Group* angewendet.

```
'Syntax:
   Selection.ShapeRange.Group
```

Die Methode *Regroup* stellt eine ehemals bestehende Gruppe wieder her, die zuvor mit *Ungroup* aufgelöst wurde. Das Beispiel (Code 1.44) fasst drei Rechtecke zu einer Gruppe zusammen.

Code 1.44 Die Prozedur bildet eine Gruppe

```
Sub ErzeugeShapeGruppe()
   Dim wshTab     As Worksheet
   Dim shpForm    As Shape

   Set wshTab = ThisWorkbook.Worksheets("Tabelle2")
   LöscheAlleShapes

   With wshTab.Shapes.AddShape(1, 20, 50, 200, 100)
      .Name = "R1"
   End With
   With wshTab.Shapes.AddShape(1, 20, 150, 100, 50)
      .Name = "R2"
   End With
   With wshTab.Shapes.AddShape(1, 120, 150, 100, 50)
      .Name = "R3"
   End With
```

```
'Gruppe
   wshTab.Shapes.Range(Array("R1", "R2", "R3")).Select
   Set shpForm = Selection.ShapeRange.Group
   shpForm.Select
   shpForm.Name = "G1"
   With shpForm
   'Rahmen
      .Line.DashStyle = 1
      With .Fill
         .BackColor.RGB = RGB(164, 0, 0)
         .ForeColor.RGB = RGB(164, 164, 164)
         .Visible = msoTrue
         .ForeColor.TintAndShade = 0
         .Transparency = 0
         .Solid
      End With
   'Textrahmen
      'schreibt Text in alle Gruppenelemente
      .TextFrame2.TextRange.Text = "TEST"
      With .TextFrame
         .HorizontalAlignment = xlHAlignCenter
         .VerticalAlignment = xlVAlignCenter
      End With
   End With

   'hängt Text an vorhandenen Text eines Shapes
   With wshTab.Shapes("R1").TextFrame2.TextRange
      .Text = .Text & " - R1"
      .Characters.Font.Size = 14
      .Characters.Font.Bold = True
   End With

   Set shpForm = Nothing
   Set wshTab = Nothing
End Sub
```

Abb. 1.44 zeigt das Ergebnis der Gruppierung. Die Gruppe wie eine Form behandelt und verfügt auch über einen Drehpunkt. Eine Textzuweisung an die Gruppe, wird an alle Elemente weitergegeben. Aber es lassen sich auch immer noch einzelne Formen der Gruppe ansprechen.

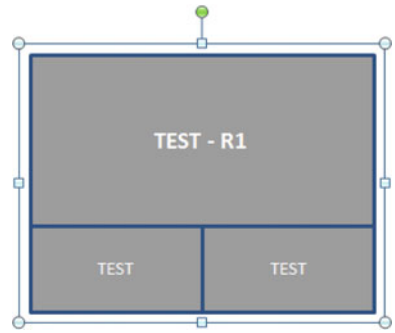


Abb. 1.44 Drei gruppierte Rechtecke bilden eine neue Form

1.5.5 Verbinder

Das Verbinder-Objekt *Connector* ist zunächst einmal wiederum ein Form-Objekt *Shape*. Mithilfe der Methoden *BeginConnect* und *EndConnect* können die Enden der Verbinder den Verbindungspunkten von Formen zugeordnet werden. Die Verbindung bleibt auch bei einer Verschiebung einer Form bestehen und wird automatisch angepasst. Ein Rechteck zum Beispiel verfügt über vier Verbindungspunkte (Abb. 1.45).

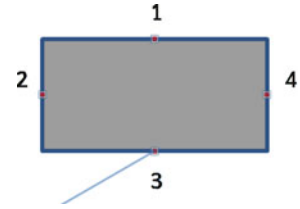


Abb. 1.45 Die Verbindungspunkte und ihr Werte für eine Rechteckform

Die Methode *AddConnector* hat die Syntax

```
'Syntax:
   Worksheet.Shapes.AddConnector(Type, BeginnX, BeginnY, EndeX, EndeY)

'Beispiel:
   wshTab.Shapes.AddConnector (1, 10, 10, 10, 10)
```

und erzeugt einen Verbinder. Mit dem ersten Parameter *Type*, wird die Form des Verbinders bestimmt (Anhang 1, Tab. A1.10). Doch erst mit der Angabe der zu verbindenden Objekte über die Syntax kommt die Verbindung zustande.

```
'Syntax:
   With VerbinderShape
      .Name = "Shapename"
      .ConnectorFormat.BeginConnect Shape, Verbindungspunkt
      .ConnectorFormat.EndConnect Shape, Verbindungspunkt
   End With
```

```
'Beispiel:
   With shpLine
      .Name = "L1"
      .ConnectorFormat.BeginConnect shpRect1, 4
      .ConnectorFormat.EndConnect shpRect2, 2
   End With
```

Das Beispiel (Code 1.45) verbindet zwei Rechtecke und erzeugt zum Verbinder ein Textfeld, welches dann mit dem Verbinder zu einer Gruppe gefügt wird (Abb. 1.46).

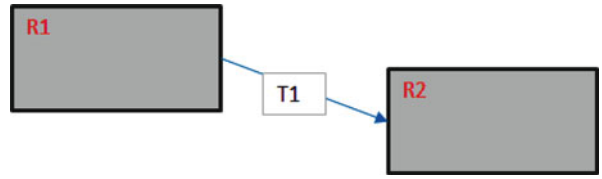


Abb. 1.46 Die Verbindung zweier Rechtecke mit einem zugeordneten Textfeld

Code 1.45 Die Prozedur verbindet zwei Formen über einen Verbinder mit Text

```
Sub VerbindeShapesMitText()
   Dim wshTab      As Worksheet
   Dim shpRect1    As Shape
   Dim shpRect2    As Shape
   Dim shpLine1    As Shape
   Dim shpLine2    As Shape
   Dim shpText1    As Shape
   Dim lLeft       As Long
   Dim lTop        As Long

   Set wshTab = ThisWorkbook.Worksheets("Tabelle3")
   LöscheAlleShapes

'Rechteck 1
   Set shpRect1 = wshTab.Shapes.AddShape _
      (1, 20, 50, 100, 50)
   With shpRect1
      .Name = "R1"
      .Fill.ForeColor.RGB = RGB(200, 200, 200)
      .Line.ForeColor.RGB = RGB(100, 100, 100)
      .TextFrame2.TextRange.Text = "R1"
      .TextFrame.Characters.Font.ColorIndex = 3
   End With
```

```
'Rechteck 2
   Set shpRect2 = wshTab.Shapes.AddShape _
      (1, 200, 80, 100, 50)
   With shpRect2
      .Name = "R2"
      .Fill.ForeColor.RGB = RGB(200, 200, 200)
      .Line.ForeColor.RGB = RGB(100, 100, 100)
      .TextFrame2.TextRange.Text = "R2"
      .TextFrame.Characters.Font.ColorIndex = 3
   End With
'Verbinder
   Set shpLine1 = wshTab.Shapes.AddConnector _
      (1, 10, 10, 10, 10)
   With shpLine1
      .Name = "L1"
      .ConnectorFormat.BeginConnect shpRect1, 4
      .ConnectorFormat.EndConnect shpRect2, 2
      .Line.EndArrowheadStyle = 2
      lLeft = .Left + .Width / 4
      lTop = .Top + .Height / 4
   End With
'Textfeld mittig zum Verbinder
   Set shpText1 = wshTab.Shapes.AddTextbox _
      (msoTextOrientationHorizontal, lLeft, lTop, 30, 20)
   With shpText1
      .Name = "T1"
      .TextFrame2.TextRange.Text = "T1"
   End With
'Text und Verbinder gruppieren
'folgt nicht einer Verschiebung
   wshTab.Shapes.Range(Array("L1", "T1")).Select
   Selection.ShapeRange.Group
   Set shpText1 = Nothing
   Set shpLine1 = Nothing
   Set shpRect1 = Nothing
   Set shpRect2 = Nothing
   Set wshTab = Nothing
End Sub
```

Bei Verschiebung eines Rechtecks bleibt der Verbinder am Verbindungspunkt. Leider folgt das Textfeld trotz Gruppierung nicht dem Verbinder. Soll das Textfeld als Bezeichnung der Verbindung dem Verbinder folgen, dann ist das Beispiel (Code 1.46) eine Lösung (Abb. 1.47).

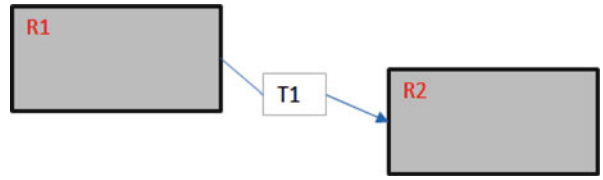


Abb. 1.47 Die Verbindung zweier Rechtecke über ein Textfeld

Code 1.46 Die Prozedur nutzt zur Verbindung von Formen ein Textfeld als Verbinder

```
Sub VerbindeShapesÜberText()
   Dim wshTab      As Worksheet
   Dim shpRect1    As Shape
   Dim shpRect2    As Shape
   Dim shpLine1    As Shape
   Dim shpLine2    As Shape
   Dim shpText1    As Shape
   Dim lLeft       As Long
   Dim lTop        As Long

   Set wshTab = ThisWorkbook.Worksheets("Tabelle3")
   LöscheAlleShapes

'Rechteck 1
'  1. Param. Formtyp:   1 - Rechteck
'                       2 - Raute us.w
   Set shpRect1 = wshTab.Shapes.AddShape _
      (1, 20, 50, 100, 50)
   With shpRect1
      .Name = "R1"
      .Fill.ForeColor.RGB = RGB(200, 200, 200)
      .Line.ForeColor.RGB = RGB(100, 100, 100)
      .TextFrame2.TextRange.Text = "R1"
      .TextFrame.Characters.Font.ColorIndex = 3
   End With

'Rechteck 2
   Set shpRect2 = wshTab.Shapes.AddShape _
      (1, 200, 80, 100, 50)
   With shpRect2
      .Name = "R2"
      .Fill.ForeColor.RGB = RGB(200, 200, 200)
      .Line.ForeColor.RGB = RGB(100, 100, 100)
      .TextFrame2.TextRange.Text = "R2"
      .TextFrame.Characters.Font.ColorIndex = 3
   End With
```

```
'Textfeld mittig zwischen den Rechtecken
   lLeft = shpRect1.Left + shpRect1.Width + _
      (shpRect2.Left - (shpRect1.Left + shpRect1.Width)) / 4
   lTop = (shpRect1.Top + shpRect1.Height / 2) + _
      (shpRect2.Top - shpRect1.Top) / 4
   Set shpText1 = wshTab.Shapes.AddTextbox _
      (msoTextOrientationHorizontal, lLeft, lTop, 30, 20)
   With shpText1
      .Name = "T1"
      .TextFrame2.TextRange.Text = "T1"
   End With

'Verbinder zwischen Rechteck1 und Textfeld
   Set shpLine1 = wshTab.Shapes.AddConnector _
      (1, 10, 10, 10, 10)
   With shpLine1
      .Name = "L1"
      .ConnectorFormat.BeginConnect shpRect1, 4
      .ConnectorFormat.EndConnect shpText1, 2
      .Line.EndArrowheadStyle = 1
   End With

'Verbinder zwischen Textfeld und Rechteck2
   Set shpLine2 = wshTab.Shapes.AddConnector _
      (1, 10, 10, 10, 10)
   With shpLine2
      .Name = "L2"
      .ConnectorFormat.BeginConnect shpText1, 4
      .ConnectorFormat.EndConnect shpRect2, 2
      .Line.EndArrowheadStyle = 2
   End With
   Set shpText1 = Nothing
   Set shpLine1 = Nothing
   Set shpRect1 = Nothing
   Set shpRect2 = Nothing
   Set wshTab = Nothing
End Sub
```

Im Gegensatz zur oben genannten Prozedur zur Löschung aller Shapes, löscht die Prozedur (Code 1.47) nur die vorhandenen Verbindungen durch Abfrage der Eigenschaft *Connector*. Statt einer *For-Each-Next*-Schleife wird hier die Eigenschaft *Count* der Shapes auf dem aktiven Arbeitsblatt zum Aufbau einer *For-Next*-Schleife genutzt.

Code 1.47 Die Prozedur löscht alle Verbinder auf dem aktiven Worksheet

```
Sub LöscheAlleVerbindungen()
   Dim wshTab      As Worksheet
   Dim iCount      As Integer

   Set wshTab = ThisWorkbook.ActiveSheet
   With wshTab.Shapes
      For iCount = .Count To 1 Step -1
         With .Item(iCount)
            If .Connector Then .Delete
         End With
      Next
   End With
   Set wshTab = Nothing
End Sub
```

1.5.6 Netzplan

Als Anwendungsbeispiel betrachten wir eine Tätigkeitsfolge in der Fertigung (Tab. 1.3). Die Dauer zur Herstellung eines Produkts richtet sich nach der Fertigungszeit der einzelnen Bauteile. Es soll ein Netzplan generiert werden, in dem die Positionen des Plans Knoten (Shapegruppe) und ihr Erreichen durch Verbinder mit der Angabe der Dauer (Textfeld) dargestellt werden.

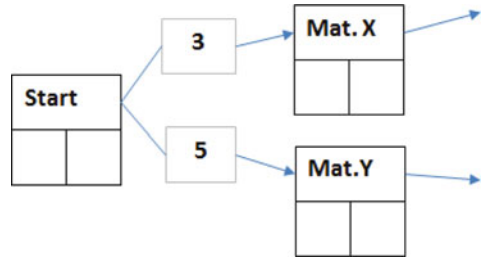
Tab. 1.3 Ein einfaches Beispiel für einen Arbeitsplan

Nr.	Bezeichnung	Vor	Nach	Dauer	FAZ	SAZ
1	Start		2/3	0		
2	Material X	1	4/5	3		
3	Material Y	1	6/7	5		
4	Teil A fräsen	2	8	4		
5	Teil B hobeln	2	9	7		
6	Teil C drehen	3	9	4		
7	Teil D biegen	3	10	5		
8	Teil A bohren	4	10	5		
9	Gruppe montieren	5/6	10	6		
10	Produkt montieren	7/8/9		1		

Aus modellbasierter Sicht haben wir es mit zwei Objekten zu tun, den Knoten (bestehend aus Formen) und deren Verbindern (bestehend aus einem Verbinder und einem Textfeld).

In Abb. 1.48 ist der Entwurf der Darstellung, allgemein als Netzplan bezeichnet, zu sehen.

Abb. 1.48 Netzplan-Modell



Der erste Schritt ist die Definition der Klassen für die verwendeten Objekte. Diesmal wollen wir die Eigenschaftswerte öffentlich definieren und sie somit allgemein zugänglich machen. Wir ersparen uns damit die Property-Funktionen.

Die Klasse der *Knoten* sieht grundsätzlich drei rechteckige Shapes vor, deren Größen durch konstante Werte festgelegt sind. Die Methode *Create* der Klasse erzeugt einen Knoten aus den vorliegenden Eigenschaften (Code 1.48).

Code 1.48 Die Klasse Knoten

```
Public Enum Maße
   lBreite = 30
   lHöhe = 30
End Enum
Public sNum    As String
Public sTitel  As String
Public lPosX   As Long
Public lPosY   As Long
Public sZeit   As String

Sub Create()
   Dim shpR1      As Shape
   Dim shpR2      As Shape
   Dim shpR3      As Shape
   Dim shpGroup   As Shape

   Set shpR1 = ActiveSheet.Shapes.AddShape _
      (1, lPosX, lPosY, 2 * lBreite, lHöhe)
   With shpR1
      .Name = "R1_" & sNum
      .TextFrame.Characters.Text = sNum
      .TextFrame.Characters.Font.ColorIndex = 1
      .Fill.ForeColor.RGB = RGB(255, 255, 255)
   End With
```

```
    Set shpR2 = ActiveSheet.Shapes.AddShape _
       (1, lPosX, lPosY + lHöhe, lBreite, lHöhe)
    With shpR2
       .Name = "R2_" & sNum
       .TextFrame.Characters.Text = sZeit
       .TextFrame.Characters.Font.ColorIndex = 1
       .Fill.ForeColor.RGB = RGB(255, 255, 255)
    End With

    Set shpR3 = ActiveSheet.Shapes.AddShape _
       (1, lPosX + lBreite, lPosY + lHöhe, lBreite, lHöhe)
    With shpR3
       .Name = "R3_" & sNum
       .TextFrame.Characters.Font.ColorIndex = 1
       .Fill.ForeColor.RGB = RGB(255, 255, 255)
    End With

    ActiveSheet.Shapes.Range(Array _
       (shpR1.Name, shpR2.Name, shpR3.Name)).Select
    Set shpGroup = Selection.ShapeRange.Group
    shpGroup.Name = "G_" & sNum
    With shpGroup
       With .Line
          .DashStyle = 1
          .Weight = 0.25
       End With
       With .TextFrame
          .HorizontalAlignment = xlHAlignCenter
          .VerticalAlignment = xlVAlignCenter
       End With
    End With
    Set shpR1 = Nothing
    Set shpR2 = Nothing
    Set shpR3 = Nothing
    Set shpGroup = Nothing
End Sub
```

Die zweite Klasse *Line* (Code 1.49) instanziiert die erforderlichen Verbinder.

Code 1.49 Die Klasse Linie

```
Public sNum1   As String
Public sNum2   As String
Public iZeit   As Integer
```

```
Sub Create()
   ActiveSheet.Shapes.AddConnector _
      (1, 10, 10, 10, 10).Select
   Selection.Name = "L_" & sNum1 & "-" & sNum2
   Selection.ShapeRange.ConnectorFormat. _
      BeginConnect ActiveSheet.Shapes("R1_" & sNum1), 4
   Selection.ShapeRange.ConnectorFormat. _
      EndConnect ActiveSheet.Shapes("R1_" & sNum2), 2
   Selection.ShapeRange.Line.EndArrowheadStyle = 2
   'Pfeilgestaltung
   'Selection.ShapeRange.Line.EndArrowheadLength = msoArrowheadLong
   'Selection.ShapeRange.Line.EndArrowheadWidth = msoArrowheadWide
End Sub
```

Die Auswertung erfolgt in zwei Schritten. Im ersten Schritt wird der vorliegende Arbeitsplan in einzelne Verbindungen aufgelöst. Dann erfolgt die Analyse der Verbindungen.

Zuerst die Bestimmung der Verbindungen aus dem Arbeitsplan als Pseudocode (Code 1.50). Ein *Pseudocode* dient nicht zur maschinellen Umsetzung mittels Compiler, sondern dient nur der Veranschaulichung des Algorithmus. Er enthält natürliche Sprachanteile in der Struktur höherer Programmiersprachen. Ein Pseudocode ist an keine Technologie und Programmiersprache gebunden und so kompakter und leicht verständlicher als ein Programmcode. Er ist eine Alternative zum Flussdiagramm, Struktogramm und anderen grafischen Formen der Algorithmendarstellung.

Code 1.50 Der Pseudocode zur Bestimmung der Verbindungen

```
setze Index auf 0
for Pos = 1 bis Ende // für alle Positionen im Plan
   Num1 = Nr(Pos)
   Text = Nach(Pos)
   do solange Text ein / enthält
      Num2 = erste Zahl in Text vor /
      entferne erste Zahl in Text
      setze Index+1
      merke Verbindung(Index): Num1 -> Num2, Zeit(Num2)
   loop
   Num2 = Zahl in Text
   setze Index+1
   merke Verbindung(Index): Num1 -> Num2, Zeit(Num2)
next
```

Nachfolgend ist die Analyse als Pseudocode dargestellt (Code 1.51).

Code 1.51 Der Pseudocode zur Darstellung

```
setze Index auf 1
do
   do // gehe vorwärts
      lese Verbindung(Index)
     if Verbindung(Index) nicht bearbeitet then
         if Knoten1 nicht vorhanden then
            erstelle Knoten1
         else
            PosX = Knoten1.PosX
         end if
         if Knoten2 nicht vorhanden then
            erstelle Knoten2
         end if
         erstelle Verbindung
         setze Index+1
      end if
   loop solange Verbindung(Index) existiert

   do // gehe rückwärts
      setze Index-1
   loop solange bis Verbindung(Index) nicht bearbeitet
      or Index > 0
loop solange Index > 0
```

Damit folgt die Auswertung zum Netzplan für eine Vorwärtsrechnung (Code 1.52).

Code 1.52 Netzplan zur Vorwärtsrechnung

```
Sub CreateAll()
   Dim wshTab      As Worksheet
   Dim shpTemp     As Shape
   Dim colKnoten   As Collection
   Dim objKtn      As clsKnoten
   Dim objLine     As clsLine
   Dim lRow        As Long
   Dim lBegin      As Long
   Dim lEnd        As Long
   Dim lMax        As Long
   Dim sNum        As String
   Dim sNach       As String
   Dim sMkr()      As String   'VerbindungsArray
   Dim iMkr        As Integer  'VerbindungsIndex
```

```
   Dim iMax       As Integer
   Dim iDa        As Integer
   Dim sNum1      As String
   Dim sNum2      As String
   Dim sNumx      As String
   Dim lX         As Long
   Dim lY         As Long
   Dim sZeit      As String
   Dim iZeit      As Integer
   Dim iZeit1     As Integer
   Dim iZeit2     As Integer

   DeleteAll
   Set wshTab = ThisWorkbook.ActiveSheet
   Set colKnoten = New Collection
'Auflösung in einzelnen Verbindungen
   lBegin = 4  'Zeile mit erstem Eintrag
   lEnd = 13   'Zeile mit letztem Eintrag
   lMax = lEnd - lBegin + 1
   ReDim sMkr(4, lMax) As String 'Merker
   iMkr = 0
   For lRow = lBegin To lEnd
      sNum = Right("0" & Trim(Str(Val(Cells(lRow, 2)))), 2)
      sNach = Trim(Cells(lRow, 5))
      Do While InStr(sNach, "/") > 0
         iMkr = iMkr + 1
         If iMkr > UBound(sMkr, 2) Then
            ReDim Preserve sMkr(4, iMkr) As String
         End If
         sMkr(1, iMkr) = sNum
         sMkr(2, iMkr) = Right("0" & Trim(Str(Val(sNach))), 2)
         sMkr(3, iMkr) = 1
         sMkr(4, iMkr) = Cells(3 + Val(sNach), 6)
         'Debug.Print iMkr, sMkr(1, iMkr); " -> "; sMkr(2, iMkr)
         sNach = Right(sNach, Len(sNach) - InStr(sNach, "/"))
      Loop
      iMkr = iMkr + 1
      If iMkr > UBound(sMkr, 2) Then
         ReDim Preserve sMkr(4, iMkr) As String
      End If
      sMkr(1, iMkr) = sNum
      sMkr(2, iMkr) = Right("0" & Trim(Str(Val(sNach))), 2)
      sMkr(3, iMkr) = 1
      sMkr(4, iMkr) = Cells(3 + Val(sNach), 6)
      'Debug.Print iMkr, sMkr(1, iMkr); " -> "; sMkr(2, iMkr)
```

```
   Next lRow
   iMax = UBound(sMkr, 2)
'Backtracking vorwärts und rückwärts
   lX = 100
   lY = 300
   iMkr = 1
   On Error Resume Next
   Do
      Do
      'vorwärts
         'ist Verb. nicht bearbeitet
         If sMkr(3, iMkr) > 0 Then
            sNum1 = sMkr(1, iMkr)
            sNum2 = sMkr(2, iMkr)
            'Knoten1 vorhanden ?
            Set objKtn = colKnoten.Item("G_" & sNum1)
            If objKtn.lPosX = 0 Then
               'erstelle Knoten1
               Set objKtn = New clsKnoten
               With objKtn
                  .sTitel = sNum1
                  .sNum = sNum1
                  .lPosX = lX
                  .lPosY = lY
                  .Create
               End With
               colKnoten.Add objKtn, "G_" & sNum1
               Err.Number = 0
            Else
               'übernehme x-Position
               lX = objKtn.lPosX
               iZeit = Val(objKtn.sZeit)
            End If
            Set objKtn = Nothing
            'Knoten2 vorhanden?
            Set objKtn = colKnoten.Item("G_" & sNum2)
            If objKtn.lPosX = 0 Then
               Set objKtn = New clsKnoten
               With objKtn
                  .sTitel = sNum2
                  .sNum = sNum2
                  .lPosX = lX + 200
                  .lPosY = lY
                  .sZeit = iZeit + Val(sMkr(4, iMkr))
                  .Create
```

```
            End With
            colKnoten.Add objKtn, "G_" & sNum2
         Else
            iZeit1 = Val(objKtn.sZeit)
            iZeit2 = iZeit + Val(sMkr(4, iMkr))
            If iZeit2 > iZeit1 Then
               objKtn.sZeit = Str(iZeit2)
               wshTab.Shapes("R2_" & sNum2).TextFrame _
                  .Characters.Text = Str(iZeit2)
            End If
         End If
         Set objKtn = Nothing
         'Verbinder
         Set objLine = New clsLine
         With objLine
            .sNum1 = sNum1
            .sNum2 = sNum2
            .iZeit = Val(sMkr(4, iMkr))
            .Create
         End With

         sMkr(3, iMkr) = 0 'Verb. bearbeitet
         'suche nächste Verb. von Knoten2, wenn vorhanden
         Do While (iMkr < iMax) And Not (sNum2 = sMkr(1, iMkr))
            iMkr = iMkr + 1
         Loop
      Else
         iMkr = iMkr + 1
      End If
   Loop While iMkr < iMax
   'gehe wieder rückwärts bis Verb. nicht bearbeitet
   lY = lY + 100
   Do
      iDa = 0
      iMkr = iMkr - 1
      sNum1 = sMkr(1, iMkr)
      For Each objKtn In colKnoten
         sNumx = objKtn.sNum
         If sNumx = sNum1 Then
            If sMkr(3, iMkr) = 1 Then
               iDa = iMkr
               Exit For
            End If
         End If
      Next
```

```
         If iDa > 0 Then iMkr = iDa
      Loop While iMkr > 0 And iDa = 0
   Loop While iMkr > 0
   Set objKtn = Nothing
   Set wshTab = Nothing
   Set colKnoten = Nothing
End Sub

Sub DeleteAll()
   Dim objShape As Shape

   For Each objShape In ActiveSheet.Shapes
      objShape.Delete
   Next
End Sub
```

Das Ergebnis der Vorwärtsbestimmung zeigt Abb. 1.49. Es gibt auch noch die Rückwärtsbestimmung, bei der nach der Vorwärtsbewegung in den Verbindungen, diese von der letzten Verbindung aus wieder rückwärts durchlaufen werden. Eine schöne Aufgabe für den Leser.

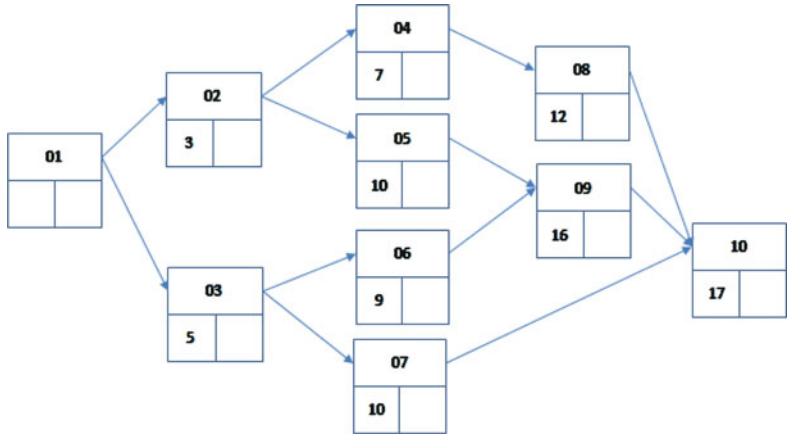


Abb. 1.49 Netzplan in einer vereinfachten Version

Die so erreichte Darstellung nimmt immer den größten errechneten Wert für den Knoten, allerdings schwer zu erkennen, da die Aktionszeiten nicht dargestellt sind. Wenn wir die Klasse der Verbinder um ein Textfeld erweitern, dann erhalten wir auch diese Angabe.

Die neue Klasse der Verbinder (Code 1.53), mit der dann auch gleich die neue Darstellung erreicht wird.

Code 1.53 Die Klasse Verbinder

```
Public sNum1   As String
Public sNum2   As String
Public iZeit   As Integer

Sub Create()
   Dim lLeft   As Long
   Dim lTop    As Long
   Dim lL1     As Long
   Dim lL2     As Long
   Dim lT1     As Long
   Dim lT2     As Long
   Dim lW1     As Long

'Textbox
   lL1 = ActiveSheet.Shapes("R1_" & sNum1).Left
   lL2 = ActiveSheet.Shapes("R1_" & sNum2).Left
   lT1 = ActiveSheet.Shapes("R1_" & sNum1).Top
   lT2 = ActiveSheet.Shapes("R1_" & sNum2).Top
   lW1 = ActiveSheet.Shapes("R1_" & sNum1).Width
   lLeft = lL1 + lW1 + (lL2 - (lL1 + lW1)) / 3
   lTop = (lT1 + lW1 / 2) + (lT2 - lT1) / 3
   ActiveSheet.Shapes.AddTextbox _
      (1, lLeft, lTop, 30, 20).Select
   With Selection
      .Name = "T_" & sNum1 & "_" & sNum2
      .Characters.Text = Str(iZeit)
   End With
'erster Verbindungsteil
   ActiveSheet.Shapes.AddConnector _
      (1, 10, 10, 10, 10).Select
   Selection.Name = "L1_" & sNum1 & "-" & sNum2
   Selection.ShapeRange.ConnectorFormat. _
      BeginConnect ActiveSheet.Shapes("R1_" & sNum1), 4
   Selection.ShapeRange.ConnectorFormat. _
      EndConnect ActiveSheet.Shapes _
      ("T_" & sNum1 & "_" & sNum2), 2
   Selection.ShapeRange.Line.EndArrowheadStyle = 1
'zweiter Verbindungsteil
   ActiveSheet.Shapes.AddConnector _
      (1, 10, 10, 10, 10).Select
   Selection.Name = "L2_" & sNum1 & "-" & sNum2
   Selection.ShapeRange.ConnectorFormat. _
      BeginConnect ActiveSheet.Shapes _
      ("T_" & sNum1 & "_" & sNum2), 4
```

```
   Selection.ShapeRange.ConnectorFormat. _
      EndConnect ActiveSheet.Shapes ("R1_" & sNum2), 2
   Selection.ShapeRange.Line.EndArrowheadStyle = 2
End Sub
```

In der neuen Darstellung (Abb. 1.50) ist dann auch relativ leicht der sogenannte *Kritische Pfad*, in Rot dargestellt, zu erkennen.

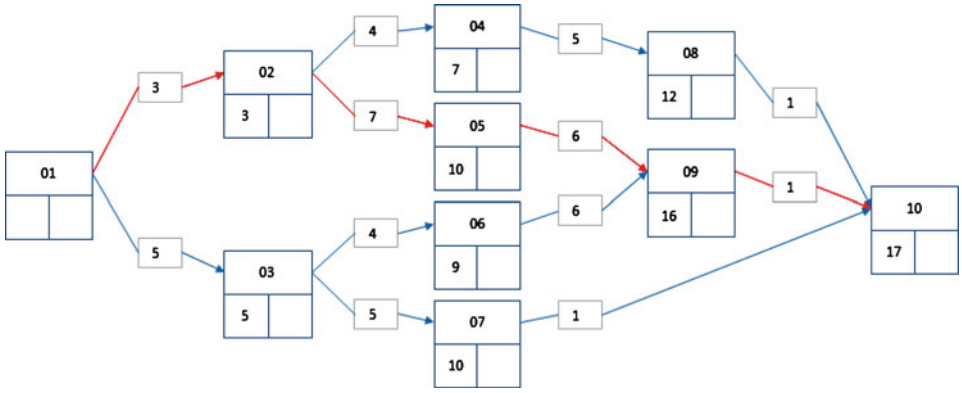


Abb. 1.50 Netzplan mit Zeitangaben und kritischem Pfad

Excel-Objekte

2

Dieses Kapitel nutze ich zur Beschreibung über den Umgang mit VBA-Objekten im Allgemeinen und den Excel-Objekten im Besonderen. Die Excel-Anwendung dient in der Praxis immer mehr als Arbeitswerkzeug zur schnellen Auswertung. Sie spielt im Office-Alltag eine wichtige Rolle im Umgang mit Datenmengen und ist auch in meinem Buch von zentraler Bedeutung.

2.1 Excel-Anwendungen

Das Anwendungs-Objekt *Application* ist das oberste Objekt der Excel-Objekt-Hierarchie (Abb. 2.1). Dank der Multitaskingfähigkeit von Windows können mehrere Anwendungen quasi gleichzeitig aktiv sein.

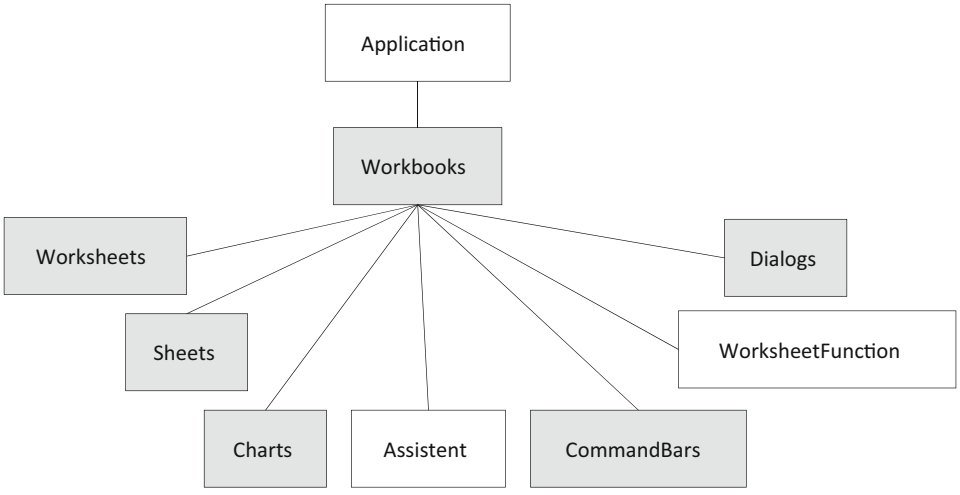


Abb. 2.1 Die wichtigsten Unterobjekte und Objektlisten (*grau*) des Excel-Application-Objekts

H. Nahrstedt, *Die Welt der VBA-Objekte*, DOI 10.1007/978-3-658-13891-2_2

Im Objektkatalog befindet sich eine ausführliche Darstellung der Excel-Objektmodellreferenz. Mit ihr kann durch die Dokumentation navigiert werden. Ebenso gibt es eine Aufstellung aller Elemente des Anwendungs-Objekts und die der untergeordneten Objekte.

2.1.1 Eigenschaften

Betrachten wir nachfolgend einige der wichtigsten Eigenschaften (*Attribute*) des Anwendungs-Objekts. Dazu gehört mit Sicherheit die boolesche Eigenschaft *Visible*, die zu jedem Objekt mit einer Oberfläche gehört. Mit den Wertzuweisungen *False* und *True* kann das Objekt verborgen und wieder sichtbar werden (Code 2.1).

Code 2.1 Die Prozedur schaltet die Eigenschaft Visible

```
Sub ApplicationVisible()
   Application.Visible = False
   MsgBox "Anwendung nicht sichtbar! Weiter mit OK."
   Application.Visible = True
End Sub
```

Auch Informationen zum System lassen sich aus den Eigenschaften abfragen (Code 2.2).

Code 2.2 Die Prozedur zeigt einige Systemeigenschaften

```
Sub ApplicationInfos()
   Dim sInfo As String

   With Application
      sInfo = "Version: " & .Version & vbCrLf
      sInfo = sInfo & "Betriebssystem: " & .OperatingSystem & vbCrLf
      sInfo = sInfo & "SystemPfad: " & .Path & vbCrLf
      MsgBox sInfo, vbInformation + vbOKOnly, "Systeminfo"
   End With
End Sub
```

In der Prozedur (Code 2.3) werden 1 Million Pseudozufallszahlen erzeugt und in die Zellen des Objekts *Worksheet* geschrieben, in dessen Code-Fenster die Prozedur steht. Dadurch kann die Angabe des übergeordneten Objektstruktur entfallen.

Code 2.3 Die Prozedur unterbindet laufende Bildschirmaktualisierungen

```
Sub ErzeugeZufallszahlen()
   Dim iZeile     As Integer
   Dim iSpalte    As Integer
   Dim dx         As Double
   Dim dt         As Double
```

```
   Randomize (Timer) 'Start des Zufallszahlengenerators
   dx = Timer
   Cells.ClearContents
   Application.ScreenUpdating = False
   dt = Timer
   For iZeile = 1 To 1000
      For iSpalte = 1 To 1000
         dx = Rnd(dx)
         Cells(iZeile, iSpalte) = dx
      Next iSpalte
   Next iZeile
   dt = Timer - dt
   Application.ScreenUpdating = True
   MsgBox "Laufzeit: " & dt
End Sub
```

Nach jeder Ausgabe in eine Zelle wird der Bildschirm aktualisiert. Das kostet viel Zeit. Mit der booleschen Eigenschaft *Application.ScreenUpdating* und deren Wert *False* wird die Aktualisierung unterbunden, bis sie wieder den Wert *True* bekommt. Auch das lästige Bildschirmflackern entfällt.

Eine weitere wichtige boolesche Eigenschaft ist *DisplayAlerts*. Mit dem Wert *False* wird die Anwendung angehalten, keine Warnmeldungen einzublenden während der Programmcode ausgeführt wird. Im Beispiel (Code 2.4) schließt das aktive Workbook ohne eine Frage nach der Speicherung. Die Excel-Anwendung bleibt geöffnet.

Code 2.4 Die Prozedur unterbindet Sytemmeldungen

```
Sub ApplicationWorkbookClose()
   Application.DisplayAlerts = False
   ActiveWorkbook.Close
   Application.DisplayAlerts = True
End Sub
```

Mit dem Start einer Excel-Anwendung ist eine automatische Auswertung aller Formeln vorgegeben. Verfügt eine Anwendung über sehr viele Formeln, so kann es zu Geschwindigkeitsverlusten kommen. Mit der Eigenschaft *Calculation* lässt sich dieses Verhalten umstellen. Dazu verwenden wir ein Formular mit Namen *frmCalculation* (Abb. 2.2). Aufgerufen wird das Formular durch die Prozedur (Code 2.5) in einem Code-Modul.

Code 2.5 Start des Formulars zur Umschaltung der Calculation-Eigenschaft

```
Sub LoadCalculation()
   Load frmCalculation
   frmCalculation.Show vbModeless
End Sub
```

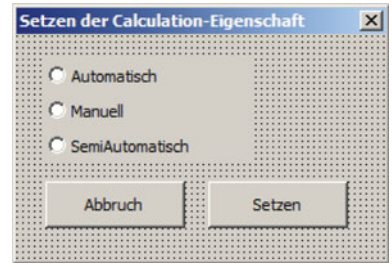


Abb. 2.2 Eine UserForm für die Einstellung

Die Prozeduren im Code-Fenster der *UserForm* stehen im Code 2.6.

Code 2.6 Die Prozeduren im Formular zur Umschaltung

```
Option Explicit

Private Sub cmdAbbruch_Click()
   Unload Me
End Sub

Private Sub cmdSetzen_Click()
   If obnAuto Then
      Application.Calculation = xlCalculationAutomatic
   ElseIf obnManu Then
      Application.Calculation = xlCalculationManual
   Else
      Application.Calculation = xlCalculationSemiautomatic
   End If
End Sub

Private Sub UserForm_Initialize()
   If Application.Calculation = xlCalculationAutomatic Then
      obnAuto = True
   ElseIf Application.Calculation = xlCalculationManual Then
      obnManu = True
   Else
      obnSemi = True
   End If
End Sub
```

Über die Eigenschaft *Assistance* des Application-Objekts kann das Hilfefenster eingeblendet werden.

Code 2.7 Aufruf des Hilfefensters aus einer Prozedur

```
Sub ApplicationAssistance()
   Application.Assistance.ShowHelp ("Windows")
End Sub
```

2.1.2 Methoden

Mit der Methode *FindFile* lassen sich aus der Anwendung weitere Anwendungen öffnen (Code 2.8). Sie ist eine boolesche Funktion und liefert bei gelungenem Start den Wert *True* ansonsten *False*. Damit kann auf das Ergebnis unterschiedlich reagiert werden.

Code 2.8 Schema einer FindFile-Anwendung

```
Dim bEvent As Boolean
bEvent = Application.FindFile
If bEvent Then
   'Aktionen bei geöffneter Anwendung
Else
   'Aktionen bei Abbruch
End If
```

Nicht immer ist ein schneller Ablauf für eine Prozedur gefragt. Es kann auch sinnvoll sein der Anwendung einen Kurzschlaf zu geben, so wie im Beispiel (Code 2.9). Ein Info-Fenster wird für eine kurze Zeit eingeblendet ohne dass die Prozedur abgebrochen wird. Die Methode *Wait* erwartet als Parameter die Zeitangabe für den Fortlauf. In der Prozedur werden auch ein paar Eigenschaften des Fensters gesetzt. Am Ende wird die Methode *Close* benutzt, die es ebenfalls für jede Anwendung gibt.

Code 2.9 Schema einer Wait-Anwendung

```
Sub NewWindowsWait()
   Dim wndFenster    As Window
   Dim iAntwort      As Integer

'instanziiert neues Fenster
   Set wndFenster = ActiveWindow.NewWindow
'setzt neue Eigenschaftswerte
   With wndFenster
      .WindowState = xlNormal 'Fenstergröße
      .Top = 20
      .Left = 20
      .Width = 400
      .Height = 300
   End With
```

```
'setzt Wartezeit auf 10 Sekunden
   Application.Wait (Now + TimeValue("0:00:10"))
'schließt Fenster
   wndFenster.Close
'löscht Verweis auf das Objekt
   Set wndFenster = Nothing
End Sub
```

Eine interessante Methode ist *OnTime*, mit der zu einem bestimmten Zeitpunkt eine Prozedur gestartet werden kann, quasi ein Start mit Verzögerung und auch schon ein wenig Ereignissteuerung.

```
'Syntax:
   OnTime (When, Name [, Tolerance])
   'When gibt die Zeit an
   'Name gibt den Prozedurnamen an
   'Tolerance verzögert bis Word beendet
'Beispiel:
   appExcel.OnTime When:="12:00:10", Name:="CopyMap"
```

2.1.3 Ereignisse

Um auf Ereignisse (*Events*) der Anwendung zu reagieren, bedarf es eines kleinen Tricks, denn in der Objektwahl eines Code-Fensters, ob nun beim Workbook oder beim Worksheet, findet sich das Anwendungs-Objekt nicht.

Erst mit den Anweisungen im Code-Fenster der Arbeitsmappe (Code 2.10) und der Ausführung der Prozedur *CreateAppObj* existiert in der Objektwahl des Codefensters auch das Objekt *appEvent*.

Code 2.10 Eine Objektvariable für die Anwendung und ihre Ereignisse anlegen

```
Dim WithEvents appEvent As Application
Private Sub CreateAppObj()
   Set appEvent = Application
End Sub
```

Damit finden sich in der nebenstehenden Ereigniswahl des Objekts auch eine Menge Ereignisse wie z. B. *AfterCalculate* (Code 2.11).

Code 2.11 Eine Ereignis-Prozedur zum Anwendungs-Objekt

```
Private Sub appEvent_AfterCalculate()
   MsgBox "Berechnung erfolgt!"
End Sub
```

Immer wenn eine Formel in einem Worksheet ausgewertet wird, wird auch diese Prozedur durchlaufen, wie sich leicht testen lässt. Ja sogar nach einer Inhaltsänderung in einer Zelle werden alle Formeln überprüft.

2.1.4 Windows

Windows ist eine Objektliste aller Fenster des aktiven Anwendungs-Objekts. Während die Prozedur *CreateWindow* (Code 2.12) mit jedem Aufruf ein Window-Objekt erzeugt,

Code 2.12 Die Prozedur erstellt ein neues Fenster

```
Sub CreateWindow()
   Dim wndFenster    As Window
   Dim iAntwort      As Integer

   Set wndFenster = ActiveWindow.NewWindow
   Set wndFenster = Nothing
End Sub
```

zeigt die Prozedur *ShowAllWindows* (Code 2.13) die *Caption*-Eigenschaft (Text in der Kopfzeile der Anwendung) aller geöffneten Fenster an.

Code 2.13 Die Prozedur liest alle geöffneten Fenster-Überschriften

```
Sub ReadAllWindows()
   Dim wndTest As Window

   For Each wndTest In Application.Windows
      MsgBox wndTest.Caption
   Next
End Sub
```

2.1.5 Dialoge

Dialogs ist eine Objektliste aller Dialog-Fenster, die in Windows existieren. Daher kann kein Dialog-Objekt gelöscht oder neu hinzugefügt werden.

Über das Attribut *Dialogs.Count* lässt sich die Anzahl verfügbarer Dialog-Objekte ermitteln. Dargestellt werden Dialog-Objekte mit der *Show*-Methode.

```
'Syntax:
   Application.Dialogs(Dialogfeldkonstante).Show

'Beispiel:
   Application.Dialogs(xlDialogFindFile).Show
```

Einzelne Dialog-Objekte lassen sich über *Dialogs* (Dialogfeldkonstante) ansprechen. Der Name der Dialogfeldkonstanten setzt sich aus der Präfix *xlDialog* und dem Namen des Dialogfeldes zusammen. So lautet beispielsweise die Konstante für das Dialogfeld „Datei suchen“ *xlDialogFindFile*.

Dialogfeld-Konstante und -Parameter stehen in Anhang 2, Tab. A2.1.

Excel-VBA stellt zwei Methoden bereit, um Dateinamen mithilfe der normalen Datei-Dialoge zu ermitteln:

- *GetOpenFilename()* für die Auswahl einer vorhandenen Datei und
- *GetSaveAsFilename()* für die Angabe einer neuen Datei unter Beachtung gültiger Pfade und Benennung.

Beide Dialogfenster unterscheiden sich optisch nicht voneinander. Der Unterschied besteht darin, dass *GetSaveAsFilename()* die Angabe eines neuen, noch nicht vorhandenen Dateinamens erlaubt. Bei *GetOpenFilename()* muss die Datei vorhanden sein.

Nachfolgend eine einfache Möglichkeit zum Datei-Öffnen-Dialog.

Code 2.14 Die Prozedur öffnet das Datei-Dialogfenster und wertet die Antwort aus

```
Sub ÖffneDatei()
    Dim varDatei As Variant
    varDatei = Application.GetOpenFilename()
    If varDatei = False Then
        MsgBox "Abgebrochen!", vbInformation
    Else
        MsgBox "Wahl:" & vbCrLf & varDatei
    End If
End Sub
```

2.1.6 Statusleiste und Tabellenfunktionen

Als Statusleisten-Objekt *StatusBar* wird die Zeile im unteren Rand des Anwendungs-Fensters bezeichnet. In ihr lässt sich eine Vielzahl Informationen einblenden. Im Kontextmenü der Statuszeile werden sie angewählt. Aber es lässt sich auch eigener Text einblenden, wie z. B. den Fortschritt in einer Programmschleife melden (Code 2.15).

Code 2.15 Die Prozedur schreibt den Programmfortschritt als Wert in die Statuszeile

```
Sub ShowStatusPercent()
   Dim lLoop   As Long
   Const lMax  As Long = 100
   Dim dZeit   As Double
```

```
   For lLoop = 1 To lMax
      Application.StatusBar = Format(lLoop / lMax, "#0%")
      dZeit = Timer + 0.1
      Do Until Timer > dZeit
      Loop
   Next lLoop
End Sub
```

Der Text kann ebenso als optische Anzeige verwendet werden, wie in dem nachfolgenden Beispiel (Code 2.16).

Code 2.16 Die Prozedur schreibt den Programmfortschritt grafisch in die Statuszeile

```
Sub ShowStatusBar()
   Dim lLoop      As Long
   Dim lB1        As Long
   Dim lB2        As Long
   Const lMax     As Long = 100
   Const lWidth   As Long = 50
   Dim dZeit      As Double

   For lLoop = 1 To lMax
      lB1 = Int(lWidth * lLoop / lMax)
      lB2 = Int(lWidth * (lMax - lLoop) / lMax)
      Application.StatusBar = _
         Format(lLoop / lMax, "#0%") & " " & _
         WorksheetFunction.Rept(ChrW(9609), lB1) & _
         WorksheetFunction.Rept(ChrW(9605), lB2)
      dZeit = Timer + 0.1
      Do Until Timer > dZeit
      Loop
   Next lLoop
   Application.StatusBar = False
End Sub
```

Dieses Beispiel verwendet ein weiteres Unterobjekt des Anwendungs-Objekts, das Tabellenfunktions-Objekt *WorksheetFunction*. Es ist ein Container für alle Arbeitsblattfunktionen in Excel. Allerding unterscheiden sich die Funktionsnamen teilweise. So lautet die Funktion *Ersetzen* im Arbeitsblatt als *WorksheetFunction*-Element *Replace*. Eine Übersicht aller Funktionen liefert die VBA-Hilfe unter dem Begriff *Elemente des WorksheetFunction-Objekts*.

Anhang 2, Tab. A2.2 ist eine hilfreiche Übersicht abgeleiteter mathematischer Funktionen.

2.2 Excel-Arbeitsmappen

Das Arbeitsmappen-Objekt *Workbook* ist ein einzelnes Excel-Objekt, das auch als Datei gespeichert wird. Es ist ein Element in der Objektliste *Workbooks*.

Soll eine Arbeitsmappe aus der Liste der *Workbooks* ausgewählt werden, so geschieht dies durch Angabe eines Index als fortlaufende Nummer aller vorhandenen *Workbooks*, die aber kaum bekannt ist. Einfacher geht es durch die Angabe des Namens.

```
'Syntax:
Workbooks.(Item(IndexNr oder Name))

'Beispiele:
Workbooks.Item(1)
Workbooks.Item("Mappe1.xls")
```

Später wird noch gezeigt, wie man einer solch selektierten Arbeitsmappe eine Objektvariable zuordnen kann.

In der VBA-Hilfe befinden sich unter *Elemente des Workbook-Objekts* eine Aufstellung aller Eigenschaften, Methoden und Ereignisse des *Workbooks*-Objekts.

2.2.1 Eigenschaften

Ein sehr wichtiges Attribut einer Arbeitsmappe ist der *Name*. Die Prozedur (Code 2.17) zeigt den Namen der aktuellen Arbeitsmappe (Abb. 2.3), die eine Besonderheit unter den Arbeitsmappen darstellt und durch den Titel *ThisWorkbook* angesprochen werden kann.

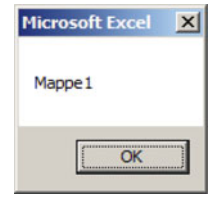


Abb. 2.3 Name der aktuellen Arbeitsmappe

Code 2.17 Die Prozedur gibt den Namen der aktiven Arbeitsmappe aus

```
Sub ActiveWorkbookName ()
   Dim objBook As Workbook

   Set objBook = ThisWorkbook
   MsgBox objBook.Name
End Sub
```

Abb. 2.4 Pfad der aktuellen Arbeitsmappe

Eine weitere wichtige Eigenschaft ist *Path*, der Pfad (Abb. 2.4) unter dem eine Arbeitsmappe gespeichert ist (Code 2.18).

Code 2.18 Die Prozedur gibt den Pfad der aktiven Arbeitsmappe aus

```
Sub ActiveWorkbookPath ()
   Dim objBook As Workbook

   Set objBook = ThisWorkbook
   MsgBox objBook.Path
End Sub
```

Eine wichtige boolesche Eigenschaft ist *Saved,* sie gibt an ob der Inhalt des Workbooks seit der letzten Speicherung geändert wurde. Sie ist *True* wenn keine Änderung vorliegt, andernfalls *False*. Soll ein Workbook nicht gespeichert werden, so kann dieser Wert im Programm auf *True* gesetzt werden. Die Prozedur (Code 2.19) erinnert an eine Speicherung, wenn in der Arbeitsmappe eine Änderung erfolgte.

Code 2.19 Die Prozedur erinnert an eine Speicherung

```
Sub ActiveWorkbookSaved ()
   Dim objBook As Workbook

   Set objBook = ThisWorkbook
   If Not objBook.Saved Then
      MsgBox "Bitte Änderungen speichern!"
   End If
   Set objBook = Nothing
End Sub
```

2.2.2 Methoden

Da es unter dem Anwendungs-Objekt die Objektliste *Workbooks* gibt, sind natürlich auch mehrere Arbeitsmappen in einer Anwendung möglich. Die Methode *Add* (Code 2.20) erzeugt eine neue Arbeitsmappe innerhalb der Anwendung (Abb. 2.5).

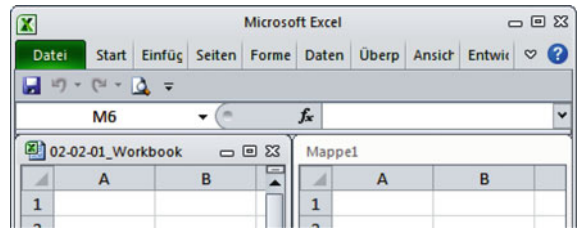


Abb. 2.5 Zwei Arbeitsmappen in der Anwendung

Code 2.20 Die Prozedur erzeugt eine neue Arbeitsmappe innerhalb der Anwendung

```
Sub NewWorkbook()
   Dim objBook As Workbook

   Set objBook = Application.Workbooks.Add
   Set objBook = Nothing
End Sub
```

Mit der Angabe von Pfad und Namen einer vorhandenen Arbeitsmappe wird diese geöffnet (Code 2.21).

Code 2.21 Die Prozedur erzeugt eine neue Arbeitsmappe innerhalb der Anwendung

```
Sub NewWorkbookOpen ()
   Dim objBook    As Workbook
   Dim sPfad      As String
   Dim sName      As String

   sPfad = "C:\Temp\"
   sName = "Mappe2"
   Set objBook = Application.Workbooks.Add(sPfad & sName)
   Set objBook = Nothing
End Sub
```

Doch in diesem Fall (Abb. 2.6) sollte der mögliche Fall einer nicht vorhandenen Arbeitsmappe mittels Fehlerbehandlung (Abb. 2.7) abgefangen werden (Code 2.22).

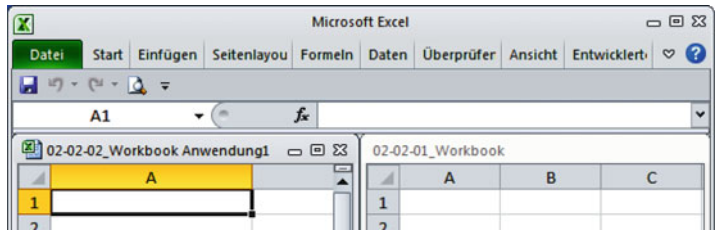


Abb. 2.6 Zwei Arbeitsmappen in einer Anwendung

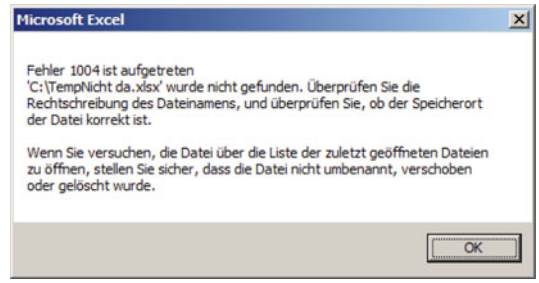


Abb. 2.7 Gewollte Fehlermeldungen

Code 2.22 Die Prozedur prüft die Existenz einer Arbeitsmappe

```
Sub NewWorkbookMissing ()
   Dim objBook    As Workbook
   Dim sPfad      As String
   Dim sName      As String

   sPfad = "C:\Temp\"
   sName = "Mappe2"
   On Error Resume Next
   Set objBook = Application.Workbooks.Add(sPfad & sName)
   If Err.Number > 0 Then
      MsgBox "Fehler " & Err.Number & " ist aufgetreten" & _
         vbLf & Err.Description
   End If
   Set objBook = Nothing
End Sub
```

Eine vorhandene Arbeitsmappe lässt sich per VBA auch wieder schließen, so dass nur noch die Anwendung bleibt, falls es die einzige Arbeitsmappe im Anwendungs-Objekt ist (Code 2.23).

Code 2.23 Die Prozedur schließt die Arbeitsmappe mit dem Code

```
Sub ThisWorkbookClose()
   Dim objBook    As Workbook

   Set objBook = ThisWorkbook
   objBook.Close
   Set objBook = Nothing
End Sub
```

Es wird auch nur die Arbeitsmappe *ThisWorkbook* geschlossen, in der sich die Prozedur befindet. Bei mehreren geöffneten Arbeitsmappen schließt die Prozedur (Code 2.24) immer die Arbeitsmappe, die gerade aktiv ist (*ActiveWorkbook*).

Code 2.24 Die Prozedur schließt die aktive Arbeitsmappe

```
Sub ActiveWorkbookClose()
   Dim objBook    As Workbook

   Set objBook = ActiveWorkbook
   objBook.Close
   Set objBook = Nothing
End Sub
```

Eine geschlossene Arbeitsmappe lässt sich über das Anwendungs-Objekt öffnen. Dazu wird die Methode *Open* der Objektliste *Workbooks* benutzt.

Code 2.25 Die Prozedur öffnet eine vorhandene Arbeitsmappe

```
Sub WorkbookOpen()
   Dim objBook    As Workbook
   Dim sPfad      As String
   Dim sName      As String

   sPfad = "C:\Temp\"
   sName = "02-02-02_Workbook Anwendung"
   Set objBook = Application.Workbooks.Open(sPfad & sName)

   Set objBook = Nothing
End Sub
```

Eine sehr elegante Lösung ist die Verwendung der Methode *Show* eines Dialogs-Objekts. Hier wird nur der Ordner angegeben und der Dateityp. Die Auswahl erfolgt in einem Dialogfeld.

Code 2.26 Die Prozedur öffnet ein Dialogfenster zur Auswahl

```
Sub WorkbookDialog()
   Dim objBook    As Workbook
   Dim sPfad      As String
   Dim sName      As String
   Dim bReturn    As Boolean

   sPfad = "C:\Temp\"
   sName = "*.xlsx"
   bReturn = Application.Dialogs(xlDialogOpen).Show(sPfad & sName)
   If Not bReturn Then
      MsgBox "Auswahl fehlerhaft!"
   End If
   Set objBook = Nothing
End Sub
```

Um eine Arbeitsmappe im laufenden Programm zu speichern, bedient man sich der Methode *Save*.

```
'Syntax:
   Workbook.Save

'Beispiel:
   objBook.Save
```

Diese Methode setzt einen Namen und einen Ordner für die Ablage voraus. Bei einer neu erstellten Arbeitsmappe ist der Name MappeX, mit X für eine laufende Nummer und der Standard-Ablageordner wird in den Optionen der Mappe festgelegt. Für die Speicherung unter einem neu gewählten Namen und einem frei gewählten Ordner ist die Methode *SaveAs* gedacht.

Code 2.27 Die Prozedur speichert eine neue Arbeitsmappe unter dem vorgegebenen Pfad

```
Sub WorkbookSaveAs()
   Dim objBook    As Workbook
   Dim sPfad      As String
   Dim sName      As String

   Set objBook = Application.Workbooks.Add
   sPfad = "C:\Temp\"
   sName = "Test"
   objBook.SaveAs Filename:=sPfad & sName
   Set objBook = Nothing
End Sub
```

2.2.3 Ereignisse

Wichtige Ereignisse *Events* der Arbeitsmappe sind *Open* und *BeforeClose*. Ihre Ereignisprozeduren werden beim Öffnen einer Excel-Arbeitsmappe und vor deren Schließen aufgerufen. Damit können wir diese Ereignisse statt der Prozeduren *Auto_Open* und *Auto_Close* verwenden.

Code 2.28 Die Workbook-Events der Arbeitsmappe zur Aufgabenverwaltung

```
Private Sub Workbook_Open()
   Dim objAufgabe    As clsAufgabe

   Set colAufgaben = New Collection
   Set colPersonen = New Collection
```

```
   Set colPlanungen = New Collection
   ErfasseAufgaben 'erstelle Aufgaben
   ErfassePersonen 'erstelle Personen
   Load frmPlanung 'lade Formular
   With frmPlanung.lbxAufgaben
      .Clear 'lösche Listbox
      For Each objAufgabe In colAufgaben
         .AddItem objAufgabe.Key & _
         " / " & objAufgabe.Stufe & _
         " - " & objAufgabe.Titel
      Next
   End With
   frmPlanung.Show
End Sub
Private Sub Workbook_BeforeClose(Cancel As Boolean)
   Set colAufgaben = Nothing
   Set colPersonen = Nothing
   Set colPlanungen = Nothing
End Sub
```

Die nachfolgenden Objekte sind eigentlich Unterobjekte des Anwendungs-Objekts, aber erst im Zusammenhang mit diesen Ereignissen kommen sie richtig zum Einsatz.

2.2.4 Befehlsleisten und Steuerelemente

Bei dem Befehlsleisten-Objekt *CommandBars* handelt es sich um eine Objektliste, deren Objekte erst in Zusammenhang mit dem Arbeitsmappen-Objekt alle nutzbar sind. Es sind Elemente des Menübandes und sie befinden sich als Steuerelement-Objekte *Controls* ebenfalls in einer Objektliste (Abb. 2.8).

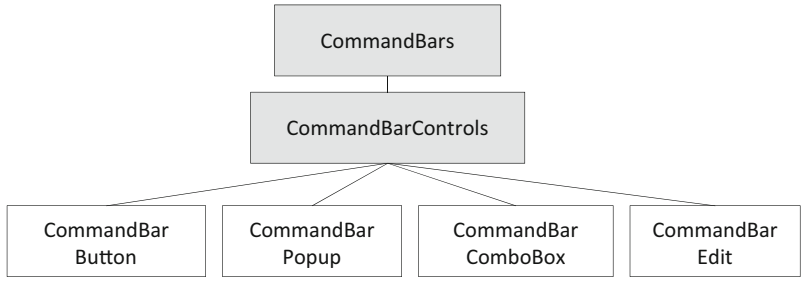


Abb. 2.8 Die Hierarchie der CommandBars-Objekte

Wie alle Objekte einer Objektliste sind auch sie über die *For-Each-Next*-Schleife zu erreichen. Die nachfolgende Prozedur (Code 2.29) ermittelt alle vorhandenen Befehls-

leisten-Objekte und deren Steuerelemente. Ihre Bezeichnungen werden im Direktfenster ausgegeben (Abb. 2.9).

Abb. 2.9 Protokoll der gefundenen Controls

Code 2.29 Die Prozedur gibt alle vorhandenen Controls im Direktfenster aus

```
Sub ErmittleCommandBarObjekte()
   Dim objMenuBars   As CommandBars
   Dim objMenuBar    As CommandBar
   Dim objMenu       As CommandBarControl
   Set objMenuBars = Application.CommandBars
   For Each objMenuBar In objMenuBars
      Debug.Print "--- BAR: " & objMenuBar.Name
      For Each objMenu In objMenuBar.Controls
         Debug.Print "CONTROL: " & objMenu.Caption
      Next
   Next
   Set objMenuBars = Nothing
End Sub
```

Zuerst wird in der Prozedur die Objektliste mit einer Objektvariablen *TempMenuBar* belegt. Es ist sinnvoll hier auch das Eltern-Objekt *Application* anzugeben. Dann werden in einer Schleife alle im Menüband vorhandenen *CommandBar*-Objekte der Objektliste *CommandBars* gelesen. Zu jedem *CommandBar*-Objekt werden in einer weiteren Schleife alle vorhandenen *CommandBarControl*-Objekte aus der Objektliste der *CommandBar-Controls* gelesen.

Im Direktfenster finden wir bei den Control-Bezeichnungen das Zeichen &, das als Steuerelement dient und im Menü den Buchstaben hinter dem Zeichen unterstreicht, um so auf seine zusätzliche Funktionalität hinzuweisen. Einträge mit einem unterstrichenen Buchstaben können mit der Tastenkombination *ALT+Buchstabe* aufgerufen werden, als Alternative zum Mausklick.

Der erste Eintrag im Direktfenster ist vom Typ *Bar* und trägt die Bezeichnung *Worksheet Menu Bar*. Alle Leisten können mit der Eigenschaft *Enable* aus- und eingeblendet werden.

Code 2.30 Prozeduren zum Aus- und Einblenden des Worksheet-Menübandes

```
Sub WorksheetMenuBarAusblenden()
    CommandBars("Worksheet Menu Bar").Enabled = False
    Application.DisplayFullScreen = True
    ActiveWorkbook.Protect Windows:=True
End Sub

Sub WorksheetMenuBarEinblenden()
    ActiveWorkbook.Unprotect
    CommandBars("Worksheet Menu Bar").Enabled = True
    Application.DisplayFullScreen = False
End Sub
```

Die Eigenschaft *DisplayFullScreen* des Anwendungs-Objekts sorgt für eine Darstellung auf dem gesamten Bildschirm. Mit der Eigenschaft *Protect* der Arbeitsmappe wird das Fenster vor Veränderungen geschützt. Die zweite Prozedur nimmt diese Einstellungen wieder zurück.

Wie alle Objektlisten verfügt auch die der *CommandBars* über die *Add*-Methode, so dass eigene Einträge dem Menüband hinzugefügt werden können. Alle eigenen Einträge sollten eine Kennung bei der Instanziierung erhalten, so dass die Controls mit dem Beenden auch wieder entfernt werden können. Die Arbeitsmappe erhält eine Definition einer konstanten Variablen mit dem entsprechenden Menütitel.

Der Menütitel sollte auf die Verwendung der eigenen Menüeinträge hinweisen, denn er erscheint auch im Menüband in der Eigenschaft *Caption* des *CommandBar*-Objekts. Ähnlich wie im Menüband zusammengehörige Funktionen in einer Gruppe zusammengefasst sind, sollten auch die eigenen Einträge als Gruppe zusammengefasst werden. Es passiert oft, dass auch noch andere Einträge hinzukommen und dann alles etwas unübersichtlich wird. Die Prozedur (Code 2.31) erzeugt einen Rahmen mit Eintrag und einem Popup-Kontrollelement als Gruppenelement zur Aufnahme weiterer *Controls* für Aktionen (Abb. 2.10).

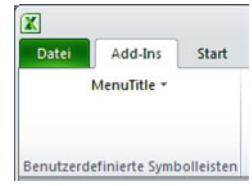


Abb. 2.10 Eigener Menüeintrag als Popup-Control unter dem Register Add-In

Code 2.31 Die Prozedur zeigt eine Grundstruktur für einen Eintrag im Menüband

```
Const sMenuName As String = "MenuTitel"

Private Sub MenuTemplate()
   Dim objMenuBar       As CommandBar
   Dim objMenuGroup     As CommandBarControl
```

```
    RemoveMenu
    Set objMenuBar = Application.CommandBars.Add _
        (sMenuName, msoBarTop)
    objMenuBar.Visible = True

    Set objMenuGroup = objMenuBar.Controls.Add _
        (Type:=msoControlPopup, Temporary:=False)
    With objMenuGroup
        .Caption = sMenuName
        .Tag = sMenuName
        .TooltipText = "Aktion wählen ..."
    End With

    Set objMenuBar = Nothing
    Set objMenuGroup = Nothing
End Sub
```

Der erste Aufruf im *MenuTemplate* ist die Prozedur *RemoveMenu* (Code 2.32), mit der eventuell vorhandene Steuerelemente mit der Tag-Eigenschaft *sMenuName* gelöscht werden, bevor die neuen Steuerelemente wieder installiert werden. So werden doppelte Einträge vermieden.

Code 2.32 Die Prozedur entfernt eigene Einträge im Menüband

```
Private Sub RemoveMenu()
    Dim objMenuBar       As CommandBar
    Dim objMenuControl   As CommandBarControl

    For Each objMenuBar In Application.CommandBars
        For Each objMenuControl In objMenuBar.Controls
            If objMenuControl.Tag = sMenuName Then
                objMenuControl.Delete
            End If
        Next
        If objMenuBar.Name = sMenuName Then
            objMenuBar.Delete
        End If
    Next
    Set objMenuBar = Nothing
    Set objMenuControl = Nothing
End Sub
```

Die drei wichtigsten Steuerelemente für ein Anwendungsmenü sind eine Schaltfläche *msoControlButton*, ein Eingabefeld *msoControlEdit* und ein Auswahlfeld *msoControl-*

ComboBox. Die Prozedur (Code 2.33) zeigt die Anwendung dieser Steuerelemente mit einfachen Textausgaben.

Code 2.33 Die Prozedur zeigt die verschiedenen Controltypen für einen Eintrag im Menüband

```
Option Explicit
Const sMenuName As String = "MenuTitle"

'--- Button ---
Private Sub Meldung()
   MsgBox sMenuName & " ausgeführt!"
End Sub

'--- Eingabefeld ---
Sub Übergabe()
   Dim objControl As CommandBarComboBox

   On Error Resume Next
   Set objControl = _
      Application.CommandBars(sMenuName).Controls("Eingabe")
   MsgBox objControl.Caption & " / " & objControl.Text
   On Error GoTo 0
   Set objControl = Nothing
End Sub

'--- Auswahlfeld ---
Sub Auswahl()
   Dim objControl As CommandBarComboBox

   On Error Resume Next
   For Each objControl In _
      Application.CommandBars(sMenuName).Controls
      If objControl.Caption = "EingabeWahl" Then
         MsgBox objControl.Caption & " / " & objControl.Text
      End If
   Next
   On Error GoTo 0
   Set objControl = Nothing
End Sub

'--- Erzeuge Menü ---
Private Sub CreateMenu()
   Dim objMenuBar       As CommandBar
   Dim objMenuGroup     As CommandBarControl
   Dim objMenuButton    As CommandBarControl
```

```
    Dim objMenuBox        As CommandBarControl
    Dim objMenuCombo      As CommandBarComboBox

    RemoveMenu

'--- Auswahlliste / Menükopf ---
    Set objMenuBar = Application.CommandBars.Add _
       (sMenuName, msoBarTop)
    objMenuBar.Visible = True

    Set objMenuGroup = objMenuBar.Controls.Add _
       (Type:=msoControlPopup, Temporary:=False)
    With objMenuGroup
       .Caption = sMenuName
       .Tag = sMenuName
       .TooltipText = "Aktion wählen ..."
    End With

'--- Button ---
    Set objMenuButton = objMenuGroup.Controls.Add _
       (Type:=msoControlButton, Temporary:=True)
    With objMenuButton
       .BeginGroup = False
       .Caption = "Meldung ausgeben"
       .FaceId = 59
       .OnAction = "Meldung"
       .Style = msoButtonIconAndCaption
       .TooltipText = "Schaltfläche betätigen"
       .Tag = "Aktion"
    End With

'--- Eingabefeld ---
    Set objMenuBox = objMenuBar.Controls.Add _
       (Type:=msoControlEdit, Temporary:=True)
    With objMenuBox
       .BeginGroup = False
       .Text = "Eingabe"
       .Caption = "Eingabe"
       .OnAction = "Übergabe"
       .TooltipText = "Text eingeben und bestätigen"
       .Tag = "Eingabe"
    End With
```

```
'--- Auswahlfeld ---
   Set objMenuCombo = objMenuBar.Controls.Add _
      (Type:=msoControlComboBox, Temporary:=True)
   With objMenuCombo
      .BeginGroup = True
      .Text = "Auswahl"
      .Caption = "EingabeWahl"
      .OnAction = "Auswahl"
      .TooltipText = "Auswahl treffen"
      .Tag = "Auswahl"
      .AddItem "Test1"
      .AddItem "Test2"
      .AddItem "Test3"
      .AddItem "Test4"
      .AddItem "Test5"
      .AddItem "Test6"
      .AddItem "Test7"
      .AddItem "Test8"
      .AddItem "Test9"
   End With

   Set objMenuBar = Nothing
   Set objMenuGroup = Nothing
   Set objMenuBar = Nothing
   Set objMenuBox = Nothing
   Set objMenuCombo = Nothing
End Sub
```

Mit den Möglichkeiten des Befehlsleisten-Objekts stellen wir unsere Anwendung nun so um, dass der Start aus einem Menüeintrag erfolgt. Um nicht die Übersichtlichkeit zwischen Aufrufen und Ereignissen zu verlieren, erstellen wir vorher noch ein Aktivitätsdiagramm.

2.2.5 Aufgabenplanung (Fortsetzung)

Das Aktivitätsdiagramm wurde bereits in Abschn. 1.1.9 besprochen und seine Elemente sind in Anhang 1, Tab. A1.5 beschrieben.

Abb. 2.11 zeigt die Initialisierung der Excel-Anwendung, die das Anwendungs-Objekt instanziiert und darin das Arbeitsmappen-Objekt. Über das Ereignis *Open* der Arbeitsmappe wird ein Menü erzeugt. Mit einem Klick auf die Schaltfläche *AufgabenPlanung* werden die erforderlichen Sammlungen erzeugt, Startdaten generiert und ein Formular mit einer Aufgabenliste und zwei Schaltflächen erzeugt. Die Schaltfläche *Exit* schließt das Formular wieder und löst die Sammlungen auf.

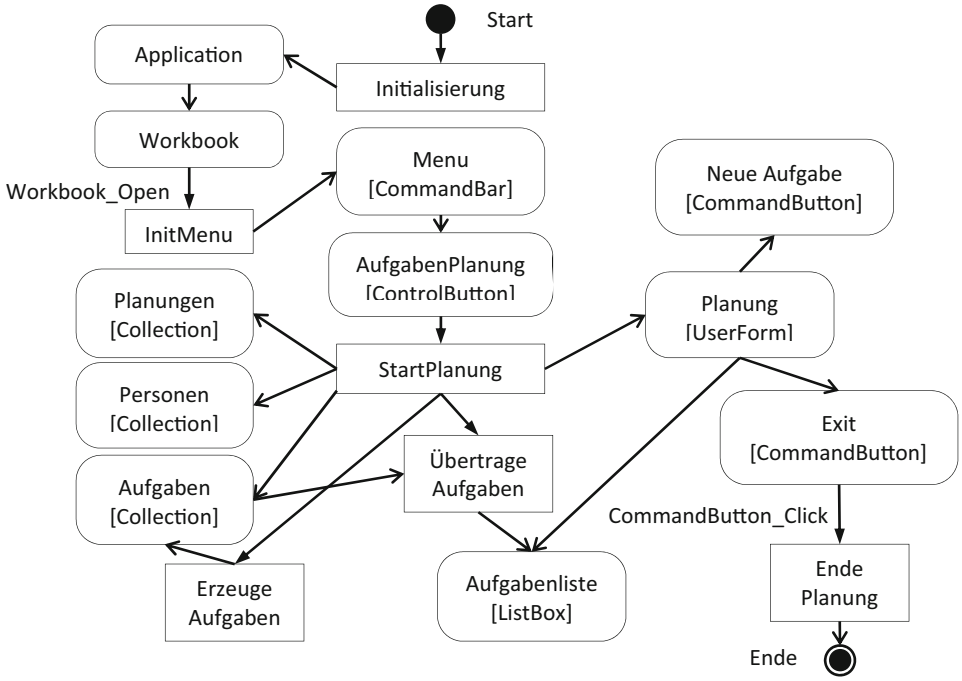


Abb. 2.11 Aktivitätsdiagramm zum Start der Aufgaben-Planung

Mit dem Schließen der Arbeitsmappe werden die restlichen Objekte zerstört (Abb. 2.12).

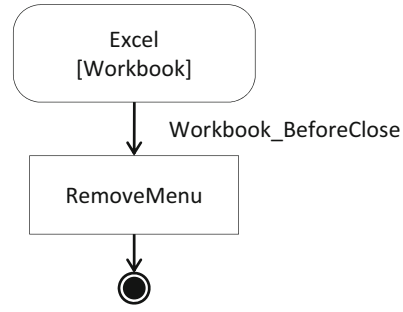


Abb. 2.12 Aktivitätsdiagramm zum Beenden der Aufgaben-Planung

Mithilfe des Befehlsleisten-Objekts und ihrer Steuerelement-Objekte wird die Anwendung gestartet.

Code 2.34 Die Menüprozeduren zur Aufgaben-Planung

```
Const sMenuName As String = "AufgabenPlanung"

Private Sub Workbook_Open()
   InitMenu
End Sub

Private Sub Workbook_BeforeClose(Cancel As Boolean)
   RemoveMenu
End Sub

Public Sub NeuStart()
   Workbook_Open
End Sub

Private Sub InitMenu()
   Dim objMenuBar       As CommandBar
   Dim objMenuGroup     As CommandBarControl
   Dim objMenuButton    As CommandBarControl

   Call RemoveMenu
   Set objMenuBar = Application.CommandBars.Add _
      (sMenuName, msoBarTop)
   objMenuBar.Visible = True

   Set objMenuGroup = objMenuBar.Controls.Add _
      (Type:=msoControlPopup, Temporary:=False)
   With objMenuGroup
      .Caption = sMenuName
      .Tag = sMenuName
      .TooltipText = "Aktion wählen ..."
   End With

   Set objMenuButton = objMenuGroup.Controls.Add _
      (Type:=msoControlButton, Temporary:=True)
   With objMenuButton
      .BeginGroup = False
      .Caption = "AufgabenPlanung"
      .FaceId = 59
      .OnAction = "PlanungsStart"
      .Style = msoButtonIconAndCaption
      .TooltipText = "AufgabenPlanung starten"
      .Tag = "AufgabenPlanung"
   End With
```

```
   Set objMenuBar = Nothing
   Set objMenuGroup = Nothing
   Set objMenuButton = Nothing
End Sub

Private Sub RemoveMenu()
   Dim objMenuBar       As CommandBar
   Dim objMenuControl   As CommandBarControl

   For Each objMenuBar In Application.CommandBars
      For Each objMenuControl In objMenuBar.Controls
         If objMenuControl.Tag = sMenuName Then
            objMenuControl.Delete
         End If
      Next
      If objMenuBar.Name = sMenuName Then
         objMenuBar.Delete
      End If
   Next

   Set objMenuBar = Nothing
   Set objMenuControl = Nothing
End Sub
```

Um unsere erste Anwendung wirklich effektiv zu gestalten, fehlt noch das wichtige Excel-Objekt *Worksheet*, das wir im nächsten Kapitel kennen lernen. Denn immer noch sind mit dem Schließen der Anwendung alle Daten gelöscht.

2.2.6 Dokument-Eigenschaften

Mit der Erstellung einer Arbeitsmappe werden automatisch zwei Bereiche zur Speicherung von Dokument-Eigenschaften (Document Properties) reserviert.

- Die *BuiltInDocumentProperties* ist eine Liste aller Eigenschaften die mit der Generierung angelegt werden, wie *Title, Author, Creation date*, usw.
- Die *CustomDocumentProperties* ist eine Liste von benutzerdefinierten Eigenschaften. Hier können Schlüsselwörtern angelegt und ihnen Inhalte zugewiesen werden. Natürlich können nur die benutzerdefinierten Schlüsselwörter auch wieder gelöscht werden. Die nachfolgende Prozedur gibt alle vorhandenen DocumentProperties auf das aktuelle Worksheet aus (Code 2.35).

Code 2.35 Die Prozedur schreibt alle vorhandenen Document-Properties in das aktuelle Arbeitsblatt

```
Sub ReadDocumentProperties()
   Dim objBook    As Object
   Dim objTab     As Object
   Dim iRow       As Integer

   Set objBook = ThisWorkbook
   Set objTab = objBook.ActiveSheet
   objTab.Activate
   Cells.Select
   Selection.ClearContents
   Range("A1").Value = "Built in Document Properties"
   Range("E1").Value = "Custom Document Properties"
   Range("A2").Value = "Name"
   Range("B2").Value = "Inhalt"
   Range("C2").Value = "Typ"
   Range("E2").Value = "Name"
   Range("F2").Value = "Inhalt"
   Range("G2").Value = "Typ"

   On Error Resume Next
   With objBook.BuiltinDocumentProperties
      For iRow = 1 To .Count
         objTab.Cells(iRow + 2, 1).Value = .Item(iRow).Name
         objTab.Cells(iRow + 2, 2).Value = .Item(iRow).Value
         objTab.Cells(iRow + 2, 3).Value = .Item(iRow).Type
         If Err.Number <> 0 Then
            objTab.Cells(iRow + 2, 2).Value = CVErr(xlErrNA)
            Err.Clear
         End If
      Next iRow
   End With
   With objBook.CustomDocumentProperties
      For iRow = 1 To .Count
         objTab.Cells(iRow + 2, 5).Value = .Item(iRow).Name
         objTab.Cells(iRow + 2, 6).Value = .Item(iRow).Value
         objTab.Cells(iRow + 2, 7).Value = .Item(iRow).Type
         If Err.Number <> 0 Then
            objTab.Cells(iRow + 2, 6).Value = CVErr(xlErrNA)
            Err.Clear
         End If
      Next iRow
   End With
```

```
   objTab.Range("A1:G2").Select
   Selection.Font.Bold = True
   objTab.Columns("A:G").EntireColumn.AutoFit
   On Error GoTo 0
   Set objTab = Nothing
End Sub
```

Auf dem aktuellen Blatt *ActiveSheet* können die Inhalte der *BuiltinDocumentProperties* verändert werden. Ebenso die der *CustomDocumentProperties*. Hier können auch neue Einträge hinzugefügt werden. Die Prozedur (Code 2.36) überträgt diese dann an die aktuelle Arbeitsmappe zurück.

Code 2.36 Die Prozedur überträgt die Daten auf dem Arbeitsblatt in die DocumentProperties

```
Sub WriteDocumentProperties()
   Dim objBook    As Object
   Dim objTab     As Object
   Dim iRow       As Integer
   Dim iRowMax    As Integer

   Set objBook = ThisWorkbook
   Set objTab = objBook.ActiveSheet
   If IsEmpty(objTab.Range("A1")) Then
      MsgBox "Bitte zuerst die Eigenschaften einlesen!"
      Exit Sub
   End If

   On Error Resume Next
   iRowMax = objTab.UsedRange.Rows.Count
   With objBook.BuiltinDocumentProperties
      For iRow = 3 To iRowMax
         If Not IsEmpty(objTab.Cells(iRow, 1)) Then
            If IsError(objTab.Cells(iRow, 2)) = False Then
               .Item(objTab.Cells(iRow, 1).Value) = _
                  objTab.Cells(iRow, 2).Value
            End If
         End If
      Next iRow
   End With
   With objBook.CustomDocumentProperties
      For iRow = 3 To iRowMax
         If IsEmpty(objTab.Cells(iRow, 5)) Then
            .Add Name:=objTab.Cells(iRow, 5).Value, _
               LinkToContent:=False, Type:=msoPropertyTypeDate, _
```

```
                Value:=objTab.Cells(iRow, 6).Value
            End If
        Next iRow
    End With
    MsgBox "Die Daten wurden übertragen, bitte prüfen!"
End Sub
```

Für die *CustomDocumentProperties* fehlen noch die Prozeduren um einzelne Eigenschaften zu setzen, zu ändern und auch wieder zu löschen. Die Prozedur (Code 2.37) ändert den Inhalt einer *CustomDocumentProperty*. Nicht vorhandene werden neu angelegt.

Code 2.37 Die Prozedur verwaltet ein Document-Property

```
Sub CreateCustomProperty()
    Dim sName       As String
    Dim sInhalt     As String

    sName = InputBox("Eigenschaftsname = ")
    sInhalt = InputBox("Inhalt von " & sName & " = ")
    On Error Resume Next
    ActiveWorkbook.CustomDocumentProperties.Add _
        Name:=sName, LinkToContent:=False, _
        Type:=msoPropertyTypeString, Value:=sInhalt
    If Err.Number = 0 Then
        MsgBox sName & " mit Inhalt" & vbLf & _
            sInhalt & " erstellt!", vbOKOnly & vbInformation
    Else
        MsgBox sName & " nicht erstellt!", vbOKOnly & vbCritical
    End If
    On Error GoTo 0
End Sub
```

Die Prozedur (Code 2.38) löscht einzelne *CustomDocumentProperties*.

Code 2.38 Die Prozedur löscht ein Document-Property

```
Sub DeleteCustomProperty()
    Dim sName       As String

    sName = InputBox("Dokumentname = ", "LÖSCHVORGANG")

    On Error Resume Next
```

```
   ActiveWorkbook.CustomDocumentProperties(sName).Delete
   If Err.Number = 0 Then
      MsgBox sName & " gelöscht!", vbOKOnly & vbInformation
   Else
      MsgBox sName & " nicht gelöscht!", vbOKOnly & vbCritical
   End If

   On Error GoTo 0

End Sub
```

2.2.7 Symbole als FaceIDs

Bereits bei den Steuerelementen im Menüband werden Symbole aus der Symbol-Objektliste *Icons* verwendet. Ihre Zuweisung erfolgt über die Eigenschaft *FaceID*. Die Prozedur (Code 2.39) zeigt alle möglichen Windows-Symbole im Menüband und ihren Code als ToolTipText. Dieser wird immer dann angezeigt, wenn sich die Maus auf dem Objekt befindet.

Code 2.39 Die Prozedur zeigt alle möglichen Windows-Icons als Menüband

```
Dim cmbSymbolList    As CommandBar
Dim lZaehler         As Long

Sub MenuSymbols()
   Dim cmbControl    As CommandBar
   Dim ctlForward    As CommandBarButton
   Dim ctlBack       As CommandBarButton
   Dim ctlEnd        As CommandBarButton

   Set cmbSymbolList = CommandBars.Add(Name:="Symbol-Liste", _
      Temporary:=True)
   With cmbSymbolList
      .Top = 200
      .Left = 250
   End With
   lZaehler = 1

   On Error Resume Next
   CommandBars("Control").Delete
   On Error GoTo 0
```

```
   Set cmbControl = CommandBars.Add(Name:="Control", _
      Temporary:=True)
   Set ctlBack = cmbControl.Controls.Add(Type:=msoControlButton)
   With ctlBack
      .Caption = "Back"
      .FaceId = 155
      .OnAction = "ListBack"
   End With
   Set ctlEnd = cmbControl.Controls.Add(Type:=msoControlButton)
   With ctlEnd
      .Caption = "Break"
      .Style = msoButtonCaption
      .OnAction = "ListBreak"
   End With
   Set ctlForward = cmbControl.Controls.Add(Type:=msoControlButton)
   With ctlForward
      .Caption = "Forward"
      .FaceId = 156
      .OnAction = "ListForward"
   End With
   With cmbControl
      .Top = 150
      .Left = 200
      .Visible = True
      .Protection = msoBarNoChangeVisible + msoBarNoCustomize
   End With
   NewList 0, 499
End Sub

Private Sub ListForward()
   NewList 0, 499
End Sub

Private Sub ListBack()
   If lZaehler <= 501 Then Exit Sub
   NewList -1000, -501
End Sub

Private Sub ListBreak()
   cmbSymbolList.Delete
   Set cmbSymbolList = Nothing
   CommandBars("Control").Delete
End Sub
```

```
Private Sub NewList(lStart As Long, lEnde As Long)
   Dim ctl As CommandBarControl
   Dim i As Long
   Dim x As Integer
   Dim E As Integer
   Dim B
   x = 0
   E = 0
   On Error GoTo Ende
   With CommandBars("Control").Controls(3)
      .Caption = "Forward"
      .Enabled = True
   End With
   cmbSymbolList.Visible = False
   For Each ctl In cmbSymbolList.Controls
      ctl.Delete
   Next ctl
   For i = lZaehler + lStart To lZaehler + lEnde
      Set ctl = cmbSymbolList.Controls.Add(Type:=msoControlButton)
      With ctl
         .FaceId = i
         .TooltipText = i
      End With
      If x = 1 Then
         ctl.Delete
         If E = 0 Then E = i
         Exit For
      End If
      x = 0
   Next i
   lZaehler = lZaehler + lEnde + 1
   If E = 0 Then E = i
   With cmbSymbolList
      .Width = 600
      .Height = 300
      .Name = "Symbol-Liste _
         (" & lZaehler - 500 & " - " & E - 1 & ")"
      .Visible = True
      .Protection = msoBarNoChangeVisible + msoBarNoCustomize
   End With
```

```
   x = 0
   If lZaehler <= 501 Then
      With CommandBars("Control").Controls(1)
         .Caption = ""
         .Enabled = False
      End With
   Else
      With CommandBars("Control").Controls(1)
         .Caption = "Back"
         .Enabled = True
      End With
   End If
   Exit Sub
Ende:
   With CommandBars("Control").Controls(3)
      .Caption = ""
      .Enabled = False
   End With
   x = 1
   Resume Next
End Sub
```

2.3 Excel-Blätter

Eine Excel-Arbeitsmappe enthält mindestens ein Blatt-Objekt *Sheet*. *Sheets* ist eine Objektsammlung aller Blätter. Während unter *Sheets* alle Blätter einer Arbeitsmappe aufgeführt werden, sowohl Tabellenblätter als auch Diagramme, werden in der Objektsammlung *Worksheets* nur alle Tabellenblätter verwaltet.

2.3.1 Blätter und Arbeitsblätter

Die Prozedur (Code 2.40) zeigt die Anwendung der Objektlisten *Sheets* und *Worksheets*.

Code 2.40 Prozeduren zum Lesen der Sheets und Worksheets

```
Option Explicit

'liest alle Sheets in der Anwendung
'und gibt deren Namen im Direktfenster aus
'zu den Scheets gehört auch ein ChartSheet!
```

```
Sub ApplicationSheets()
   Dim objTest As Object

   For Each objTest In Application.Sheets
      Debug.Print objTest.Name
   Next
End Sub

'liest alle Worksheets in der Anwendung
'und gibt deren Namen im Direktfenster aus
'dazu gehören keine ChartScheets!
Sub ApplicationWorksheets()
   Dim objTest As Object

   For Each objTest In Application.Worksheets
      Debug.Print objTest.Name
   Next
End Sub
```

Um den Unterschied festzustellen, sollte die Testmappe wenigstens ein Diagramm als Blatt besitzen (Abb. 2.13).

Abb. 2.13 Sheet Diagramm1 und Worksheets Tabelle1 bis Tabelle3

Eigenschaften und Methoden des Blatt-Objekts und des Arbeitsblatt-Objekts stehen unter der Excel-Objektmodellreferenz in der VBA-Hilfe.

2.3.2 Eigenschaften

Eine interessante Eigenschaft eines Arbeitsblatts ist seine Sichtbarkeit. Sie kann auf sichtbar (visible), geschützt (hidden) und sehr geschützt (very hidden) gesetzt werden. Die nachfolgende Prozedur schaltet zwischen diesen Werten mit jedem Aufruf um.

Code 2.41 Die Prozedur schaltet zwischen den Werten der Sichtbarkeit um

```
Sub SwitchVisible()
   Dim wshDaten As Worksheet
```

```
   Set wshDaten = ThisWorkbook.Worksheets("Daten")
   With wshDaten
      If .Visible = xlSheetVisible Then
         .Visible = xlSheetHidden
      ElseIf .Visible = xlSheetHidden Then
         .Visible = xlSheetVeryHidden
      Else
         .Visible = xlSheetVisible
      End If
   End With
   Set wshDaten = Nothing
End Sub
```

Der Unterschied zwischen *hidden* und *very hidden* liegt in der Handhabung auf der Oberfläche. Hidden-geschützte Arbeitsblätter können über das Kontextmenü wieder eingeblendet werden, very-hidden-geschützte Arbeitsblätter nicht. Sie bleiben für den Anwender verborgen.

Eine weitere Eigenschaft wird durch ihren Aufruf umgeschaltet, die Rede ist vom *Autofilter* (Code 2.42).

Code 2.42 Die Prozedur schaltet den Autofilter ein/aus

```
Sub SwitchAutofilter()
   Dim wshDaten As Worksheet
   Set wshDaten = ThisWorkbook.Worksheets("Liste")
   wshDaten.UsedRange.AutoFilter
   Set wshDaten = Nothing
End Sub
```

Durch die Angabe eines Unterbereichs, wie hier die Spalten B und C, lässt sich der Bereich beschränken.

Code 2.43 Die Prozedur schaltet den Autofilter für die Spalten B und C ein/aus

```
Sub AutofilterBC()
   Dim wshDaten As Worksheet
   Set wshDaten = ThisWorkbook.Worksheets("Liste")
   wshDaten.Range("B:C").AutoFilter
   Set wshDaten = Nothing
End Sub
```

Beliebt ist auch die Angabe des benutzen Bereichs *UsedRange*. Dieses Unterobjekt liefert für viele Anwendungen die Anzahl benutzter Zeilen und Spalten aus der Eigenschaft *Count*.

Code 2.44 Anweisungen zur Bestimmung der benutzten Zeilen/Spalten

```
Dim wshDaten   As Worksheet
Dim lRow       As Long
Dim lCol       As Long

Set wshDaten = ThisWorkbook.Worksheets("Daten")

lRow = wshDaten.UsedRange.Rows.Count    'Anzahl Zeilen
lCol = wshDaten.userrange.Columns.Count 'Anzahl Spalten
```

Aber Achtung! Der Zähler beginnt in der Zeile bzw. Spalte mit der ersten Belegung, und das ist nicht in jedem Fall die erste Zeile, bzw. erste Spalte.

Die auszuwertende Spalte für den Autofilter kann durch einen Parameter des Autofilters gesetzt werden. Ebenso ein Filterkriterium.

Code 2.45 Die Prozedur schaltet den Autofilter mit Parameterangaben

```
Sub AutofilterFieldCriteria()
   Dim wshDaten As Worksheet

   Set wshDaten = ThisWorkbook.Worksheets("Liste")
   wshDaten.UsedRange.AutoFilter Field:=2, Criteria1:="Druckfedern"
   Set wshDaten = Nothing
End Sub
```

Für das Kriterium kann auch der Inhalt eines Zellbereichs gewählt werden, z. B. Criteria1:=Range(″H1:H4″). Ebenso können mehrere Kriterien durch Operatoren wie die wichtigen *xlAnd* oder *xlOr* verknüpft werden.

Code 2.46 Die Prozedur schaltet den Autofilter mit mehreren Kriterien

```
Sub AutofilterOr()
   Dim wshDaten As Worksheet

   Set wshDaten = ThisWorkbook.Worksheets("Liste")
   wshDaten.UsedRange.AutoFilter Field:=2, _
      Criteria1:="Druckfedern", Operator:=xlOr, _
      Criteria2:="Zugfedern"
   Set wshDaten = Nothing
End Sub
```

Die Konstruktion mit einem Vergleichsoperator zeigt die Prozedur (Code 2.47).

Code 2.47 Die Prozedur schaltet den Autofilter mit einem bedingten Kriterium

```
Sub AutofilterOperator()
   Dim wshDaten As Worksheet

   Set wshDaten = ThisWorkbook.Worksheets("Liste")
   wshDaten.UsedRange.AutoFilter Field:=3, Criteria1:=">=1000"
   Set wshDaten = Nothing
End Sub
```

Die Konstruktion eines Platzhalters (wildcard) ist ebenfalls möglich und zeigt im nachfolgenden Beispiel alle Einträge die am Ende der Bezeichnung den Text *federn* aufweisen, wie Druckfedern, Zugfedern, Tellerfedern, etc.

Code 2.48 Die Prozedur schaltet den Autofilter mit einem Platzhalter im Kriterium

```
Sub AutofilterWildCard()
   Dim wshDaten As Worksheet

   Set wshDaten = ThisWorkbook.Worksheets("Liste")
   wshDaten.UsedRange.AutoFilter Field:=2, Criteria1:="=*federn"
   Set wshDaten = Nothing
End Sub
```

Eine weitere wichtige Eigenschaft ist die Sortierung (Code 2.49).

Code 2.49 Die Prozedur erstellt eine sortierte Liste nach getrennten Bereichen

```
Sub SortList()
   Dim wshDaten As Worksheet

   Set wshDaten = ThisWorkbook.Worksheets("Liste")
   With wshDaten.Sort
      .SortFields.Clear
      .SortFields.Add Key:=Range("B:B"), _
         SortOn:=xlSortOnValues, _
         Order:=xlAscending, _
         DataOption:=xlSortNormal
      .SortFields.Add Key:=Range("C:C"), _
         SortOn:=xlSortOnValues, _
         Order:=xlAscending, _
         DataOption:=xlSortNormal
      .Header = xlYes
      .SetRange Range("B:B", "C:C")
      .Apply
   End With
   Set wshDaten = Nothing
End Sub
```

Da diese Sort-Regeln gespeichert werden, sollte man die Argumente bei jeder Sortierung neu anlegen. Daher also erst die Methode *Clear* bevor neue *Fields* festgelegt werden. Bei *SetRange* hätte ich auch den Bereich B:C angeben können, doch so wird sichtbar, wie man getrennte Bereiche behandelt.

2.3.3 Methoden

Eine Möglichkeit eine Sortierung im Dialog durchzuführen, bietet die Dialog-Objektliste *Dialogs*.

Code 2.50 Die Prozedur erstellt eine sortierte Liste über ein Dialogfenster

```
Sub SortDialog()
   Application.Dialogs(xlDialogSort).Show
End Sub
```

Um ein neues Blatt zu erstellen und dieses zur Auflistung hinzuzufügen, verwenden wir wieder die Methode *Add*. Im folgenden Beispiel werden in der aktiven Arbeitsmappe zwei neue Tabellen hinter der zweiten Tabelle eingefügt.

```
'Syntax:
   Worksheets.Add (Before, After, Count, Type)

'Beispiel:
   Worksheets.Add Count:=2, After:=Worksheets(2)
```

Angesprochen werden die Objekte in der Worksheets-Objektliste mit *Worksheets(Index)*, wobei *Index* der Name oder die Indexnummer des Blattes ist, um ein einzelnes Arbeitsblatt-Objekt zurückzugeben. Im folgenden Beispiel wird das Arbeitsblatt *Tabelle1* aktiviert.

```
'Syntax:
   Worksheets(Item).Activate

'Beispiel:
   Worksheets("Tabelle1").Activate
```

2.3.4 Diagramme

Ein Diagramm-Objekt *Chart* ist ein Element der Objektliste *Charts* in einem Arbeitsmappen-Objekt. Es kann als eingebettetes Objekt einem Arbeitsblatt oder als separates Diagramm einem Blatt zugehören. Diagramme wirken in gleicher Weise wie Arbeitsblät-

ter, nur besitzen sie andere Ereignisse. Es existieren auch keine übergeordneten Ereignisse in der Arbeitsmappe und in der Anwendung. Diagramme sind Sonderlinge im Eventsystem von Excel.

Um Ereignisse auf einem Diagramm zu erhalten, müssen wir uns der gleichen Methode wie bei der Anwendung bedienen und eine Objektvariable mitsamt ihren Ereignissen instanziieren. Das Beispiel (Code 2.51) zeigt diese Möglichkeit. Der Code befindet sich im Code-Fenster der Arbeitsmappe. Mit der Aktivierung von Diagramm1 werden einige Eigenschaften des Diagramm-Objekts ausgegeben.

Code 2.51 Aufruf von Chart-Events über eine Chart-Objektvariable mit Ereignissen

```
Public WithEvents chtZeiten As Chart

Private Sub Workbook_Open()
   Set chtZeiten = ThisWorkbook.Charts(1)
End Sub

Private Sub chtZeiten_Activate()
   MsgBox "Chart-Info " & vbCrLf & _
      "Type:         " & TypeName(chtZeiten) & vbCrLf & _
      "Name:         " & chtZeiten.Name & vbCrLf & _
      "Parent Type: " & TypeName(chtZeiten.Parent) & vbCrLf & _
      "Parent Name: " & chtZeiten.Parent.Name
End Sub
```

Eine Methode zum Verschieben von Diagrammen auf einem Blatt ist *Move*. Die Prozedur (Code 2.52) verschiebt *Diagramm1* an das Ende aller Blätter und nutzt dazu die Eigenschaft *Count* des Blatt-Objekts.

Code 2.52 Die Prozedur verschiebt Diagramm1 an das Ende aller Blätter

```
Private Sub MoveChart()
   Charts("Diagramm1").Move after:=Sheets(Sheets.Count)
End Sub
```

Die Positionierung eines eingebetteten Diagramms auf einem Arbeitsblatt zeigt die Prozedur (Code 2.53), dabei werden die Maße in Pixel angegeben. Voraussetzung für die Wirksamkeit ist, dass das Diagramm auf dem Arbeitsblatt vorher markiert wird.

Code 2.53 Die Prozedur positioniert ein eingebettetes Diagramm

```
Sub ChartPosition()
   Dim chtDaten As Chart
```

```
   Set chtDaten = ActiveChart
   With chtDaten.ChartArea
      .Left = 100
      .Width = 375
      .Top = 75
      .Height = 225
   End With
   Set chtDaten = Nothing
End Sub
```

Der Zugriff auf eine Datenreihe im Diagramm erfolgt über das Objekt *SeriesCollection*. Auch dies ist eine Objektsammlung und kann aus mehreren Datenreihen bestehen. Eine Datenreihe wird somit wieder über einen Index aufgerufen. Die Prozedur (Code 2.53) verändert einige Eigenschaften der ersten (und hier auch einzigen) Datenreihe. Ebenso lassen sich einzelne Punkte der Datenreihe ansprechen.

Code 2.54 Die Prozedur verändert einige Eigenschaften von Diagramm1

```
Private Sub SetDiagramAttributes ()
   Dim chtDaten As Chart

   Set chtDaten = Charts("Diagramm1")
   With chtDaten
      '.ChartType = xlLine             'Liniendiagramm
      '.ChartType = xl3DPie            'Kuchendiagramm
      'andere Möglichkeit über die Aktivierung
      .Activate
      With ActiveChart
         .ChartType = xl3DColumnStacked 'Balkendiagramm
         .HasTitle = True
         .ChartTitle.Text = "Datenreihe"
      End With
      With .SeriesCollection(1) '1. Datenreihe
         .Format.Fill.ForeColor.RGB = rgbGray
         '6. Punkt der Datenreihe
         .Points(6).Format.Fill.ForeColor.RGB = rgbRed
      End With
   End With
   Set chtDaten = Nothing
End Sub
```

Weitere Eigenschaften und Methoden werden im Objektkatalog dargestellt und noch in späteren Beispielen beschrieben.

2.3.5 Aufgabenplanung (Fortsetzung)

Im nächsten Schritt der Aufgabenplanung werden die Objektsammlungen durch Datenblätter ersetzt und damit die Daten dauerhaft gespeichert. Wir beginnen mit dem Worksheet Aufgaben (Abb. 2.14).

	A	B	C	D	E	F
1	Key	Titel	Stufe	Zeit	Typ	Datei
2	1	Telfonat Müller	1	10	0	
3	2	Mail Hans Schmidt	4	15	0	
4	3	Meeting Projekte	2	120	1	C:\Planung\Meeting_13_5.txt
5	4	Forum News planen	3	240	2	C:\Foren\Forum_03.txt

Abb. 2.14 Startdaten im Worksheet Aufgaben

Der Start der Anwendung beginnt mit dem Laden des Formulars *Planung* im Modul *modAuto* (Code 2.55).

Code 2.55 Start des Formulars Planung aus dem Modul Auto

```
Option Explicit
Option Private Module

Private Sub PlanungsStart()
   Load frmPlanung 'lade Formular
   frmPlanung.Show
End Sub
```

Die ListBox *lbxAufgabe* auf dem Formular *Planung* bezieht ihre Daten nun vom Datenblatt. Das Ereignis *Activate* des Formulars lädt jedes Mal die Aufgabendaten neu vom Datenblatt in die ListBox.

Code 2.56 Mit dem Ereignis Aktivierung werden die Aufgabendaten geladen

```
Private Sub UserForm_Activate()
   LadeAufgaben
End Sub
```

Die eigentliche Ladeprozedur steht im Modul *Start* an der Stelle wo vorher die Aufgabendaten generiert wurden (Code 2.57).

Code 2.57 Die Prozedur zum Laden der Aufgabendaten im Modul Start

```
Sub LadeAufgaben()
   Dim wshAufgaben   As Worksheet
   Dim lRowMax       As Long
   Dim lRow          As Long
```

```
   Set wshAufgaben = ThisWorkbook.Worksheets("Aufgaben")
   lRowMax = wshAufgaben.UsedRange.Rows.Count
   With frmPlanung.lbxAufgaben
      .Clear 'lösche Listbox
      For lRow = 2 To lRowMax 'lese Aufgaben
         .AddItem wshAufgaben.Cells(lRow, 1) & _
         " / " & wshAufgaben.Cells(lRow, 3) & _
         " - " & wshAufgaben.Cells(lRow, 2)
      Next
   End With
   Set wshAufgaben = Nothing
End Sub
```

Da die Verwaltung der Daten auf dem Datenblatt erfolgt, ist eine Prozedur zum Beenden der Planung nicht mehr erforderlich. Das Event *lbxAufgaben_DblClick* muss auf das Datenblatt umgestellt werden.

Code 2.58 Das Ereignis Doppelklick auf die Aufgabenliste öffnet das Formular Aufgaben mit den relevanten Daten

```
Private Sub lbxAufgaben_DblClick _
   (ByVal Cancel As MSForms.ReturnBoolean)
   Dim wshAufgaben   As Worksheet
   Dim lRowMax       As Long
   Dim lRow          As Long
   Dim sText         As String
   Dim sNum          As String
   Dim iNum          As Integer

   iNum = lbxAufgaben.ListIndex
   If iNum < 0 Then
      MsgBox "Fehler bei der Auswahl!"
      Exit Sub
   End If
   sText = lbxAufgaben.List(iNum)                  'liest ListBox
   sNum = Trim(Left(sText, InStr(sText, "/") - 1)) 'bestimmt Num
   lRow = Val(sNum) + 1

   Set wshAufgaben = ThisWorkbook.Worksheets("Aufgaben")
   lRowMax = wshAufgaben.UsedRange.Rows.Count
   If lRow > lRowMax Then
      MsgBox "Aufgaben-Nr. ungültig!"
      Exit Sub
   End If
```

```
   Load frmAufgabe
   With frmAufgabe 'übergebe Aufgabendaten
      .tbxNum = wshAufgaben.Cells(lRow, 1)
      .tbxTitel = wshAufgaben.Cells(lRow, 2)
      .tbxStufe = wshAufgaben.Cells(lRow, 3)
      .tbxZeit = wshAufgaben.Cells(lRow, 4)
      .tbxTyp = wshAufgaben.Cells(lRow, 5)
      .tbxDatei = wshAufgaben.Cells(lRow, 6)
   End With
   frmAufgabe.Show
   Set wshAufgaben = Nothing
End Sub
```

Nachfolgend die Klick-Events des Formulars *Aufgabe* (Code 2.59 und 2.60).

Code 2.59 Die Ereignis-Prozedur Speichern des Formulars Aufgabe

```
Private Sub cmdSave_Click()
   Dim wshAufgaben   As Worksheet
   Dim lRowMax       As Long
   Dim lRow          As Long
   Dim sText         As String
   Dim sNum          As String
   Dim iNum          As Integer

   Set wshAufgaben = ThisWorkbook.Worksheets("Aufgaben")
   lRowMax = wshAufgaben.UsedRange.Rows.Count
   sNum = Val(tbxNum)
   lRow = Val(sNum) + 1
   If lRow > lRowMax + 1 Then
      MsgBox "Aufgaben-Nr. ungültig!"
      Exit Sub
   End If
   With wshAufgaben
      .Cells(lRow, 1) = Val(tbxNum)
      .Cells(lRow, 2) = tbxTitel
      .Cells(lRow, 3) = Val(tbxStufe)
      .Cells(lRow, 4) = Val(tbxZeit)
      .Cells(lRow, 5) = Val(tbxTyp)
      .Cells(lRow, 6) = tbxDatei
      .Columns("A:F").AutoFit 'Spaltenbreite anpassen
   End With
   Set wshAufgaben = Nothing
   LadeAufgaben
   Unload Me
End Sub
```

Code 2.60 Die Ereignis-Prozedur Löschen des Formulars Aufgabe

```
Private Sub cmdDelete_Click()
   Dim wshAufgaben   As Worksheet
   Dim lRowMax       As Long
   Dim lRow          As Long
   Dim sText         As String
   Dim sNum          As String
   Dim iNum          As Integer

   Set wshAufgaben = ThisWorkbook.Worksheets("Aufgaben")
   lRowMax = wshAufgaben.UsedRange.Rows.Count
   sNum = Val(tbxNum)
   lRow = Val(sNum) + 1
   If lRow > lRowMax + 1 Then
      MsgBox "Aufgaben-Nr. ungültig!"
      Exit Sub
   End If
   wshAufgaben.Rows(lRow).Select
   Selection.Delete
   Set wshAufgaben = Nothing
   LadeAufgaben
   Unload Me
End Sub
```

Der weitere Ausbau der Anwendung sieht für alle Objekte ein Datenblatt vor und entsprechend benötigt das Formular *Planung* drei Auswahllisten samt Schaltflächen für neue Eingaben. Doch diese Datenverwaltung wurde ja zur Auswertung der Planzeiten konstruiert. Die Prozedur liegt bereits vor. Lediglich eine Ausgabe fehlt. Die Daten zur Auswertung kommen nicht mehr aus den *Collections* sondern aus den Arbeitsblättern (Code 2.61). Aufgerufen wird die Analyse über eine weitere Schaltfläche. Die Oberfläche des Formulars *Planung* bekommt ein weiteres Listenfeld, das als Protokollfeld dient und so wie vorher das Direktfenster funktioniert. Ebenso eine Schaltfläche für den Start (Abb. 2.15).

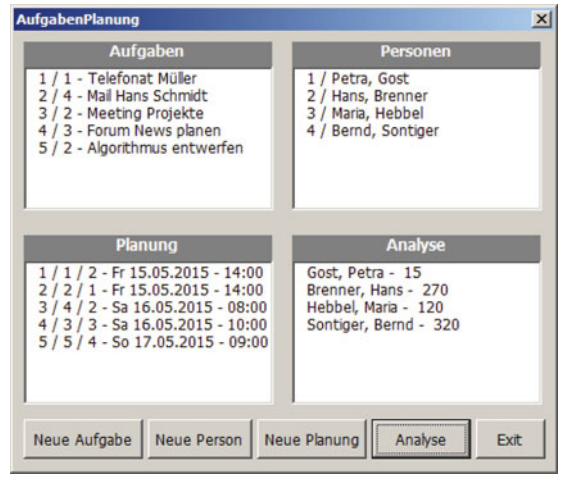


Abb. 2.15 Die neue Oberfläche des Formulars Planung

Code 2.61 Die Prozedur zur Auswertung der Planungszeiten

```
Private Sub cmdAnalyse_Click()
   Dim wshPlanungen  As Worksheet
   Dim wshAufgaben   As Worksheet
   Dim wshPersonen   As Worksheet
   Dim lRowPln       As Long
   Dim lRowPlnMax    As Long
   Dim lRowAuf       As Long
   Dim lRowPer       As Long
   Dim lRowPerMax    As Long
   Dim lZeit         As Long
   Dim sName         As String

   Set wshPlanungen = ThisWorkbook.Worksheets("Planungen")
   Set wshAufgaben = ThisWorkbook.Worksheets("Aufgaben")
   Set wshPersonen = ThisWorkbook.Worksheets("Personen")
   lRowPlnMax = wshPlanungen.UsedRange.Rows.Count
   lRowPerMax = wshPersonen.UsedRange.Rows.Count

   lbxAnalyse.Clear
   For lRowPer = 1 To lRowPerMax - 1
      sName = wshPersonen.Cells(lRowPer + 1, 2) & ", " & _
         wshPersonen.Cells(lRowPer + 1, 3)
      lZeit = 0
      For lRowPln = 2 To lRowPlnMax
         If Val(wshPlanungen.Cells(lRowPln, 3)) = lRowPer Then
            lRowAuf = Val(wshPlanungen.Cells(lRowPln, 2))
            lZeit = lZeit + Val(wshAufgaben.Cells(lRowAuf + 1, 4))
         End If
```

```
      Next lRowPln
      lbxAnalyse.AddItem sName & " - " & Str(lZeit)
   Next lRowPer

   Set wshPlanungen = Nothing
   Set wshAufgaben = Nothing
   Set wshPersonen = Nothing
End Sub
```

Eine Ausweitung dieser Anwendung macht erst Sinn, wenn wir andere Anwendungen aus der Office-Welt mit einbinden können.

2.4 Excel-Zellen und ihre Sammlungen

2.4.1 Zellen

Durch die tabellenförmige Einteilung des Arbeitsblattes ergibt sich als kleinstes Element das Zellen-Objekt *Cells*. Eine Zelle wird aus der Objektliste *Cells* unter Angabe des Zeilen- und Spalten-Index ausgewählt.

```
'Syntax:
   Worksheet.Cells (Zeilenindex, Spaltenindex)

'Beispiel:
   ActiveSheet.Cells (3, 4)
```

Die Inhalte aller Zellen eines Arbeitsblattes lassen sich mit der Methode *ClearCells* löschen. Hier sind es alle Zellen des aktiven Blattes (Code 2.62).

Code 2.62 Die Prozedur löscht alle Inhalte des aktiven Blattes

```
Sub DeleteContentUsedRange()
   ActiveSheet.UsedRange.ClearContents
End Sub
```

Dabei bleiben Formatierungen jedoch erhalten. Die Prozedur (Code 2.63) löscht den Inhalt einschließlich Formatierung des aktiven Blattes.

Code 2.63 Die Prozedur löscht alle Inhalte und Formatierungen des aktiven Blattes

```
Sub DeleteAllCells ()
   ActiveSheet.Cells.Delete
End Sub
```

Die wichtigste Eigenschaft einer Zelle ist ihr Inhalt, der sowohl als Wert wie auch als Formel zugewiesen werden kann. Das Beispiel (Code 2.64) setzt den Inhalt von Zelle B3 auf 23 und weißt der Zelle C5 eine Zufallszahl zu. Außerdem bekommt die Zelle C5 einen Kommentar und ihre Hintergrundfarbe wird auf Gelb gesetzt. Die Schrift in C5 wird auf Arial mit der Schriftgröße 12 und fett gesetzt (Code 2.64). Zuletzt wird die Zelle markiert.

Code 2.64 Beispiel für Zuweisungen an Zellen

```
Sub CellAttributes()
   With ActiveSheet
      .Cells.Delete
      .Cells(3, 2).Value = 23
      .Cells(5, 3).Formula = "=Rand()"
      .Cells(5, 3).AddComment 'erzeugt einen Kommentar
      .Cells(5, 3).Comment.Text Text:="Die Zufallszahl ändert sich!"
      .Cells(5, 3).Interior.ColorIndex = 6
      .Cells(5, 3).Font.Name = "Arial"
      .Cells(5, 3).Font.Size = 12
      .Cells(5, 3).Font.Bold = True
      .Cells(5, 3).Select
   End With
End Sub
```

2.4.2 Zellbereiche

Interessant werden Zellen erst im Verbund, dem Zellbereichs-Objekt *Range*. Ein Bereichs-Objekt ist ebenfalls ein Unterobjekt des Arbeitsblatt-Objekts und besteht per Definition aus einer oder mehreren Zellen.

```
'Syntax:
   Objekt.Range("Spaltenbuchstabe & Zeilennummer")

'Beispiel:
   ActiveSheet.Range("A1")
```

Für einen zusammenhängenden Bereich aus mehreren Zellen werden die Zelladressen oben links und unten rechts des Bereichs angegeben.

```
'Syntax:
   Objekt.Range("Zelladresse oben links":"Zelladresse unten rechts")

'Beispiele:
   ActiveSheet.Range("A1:D8").Value = 13
   ActiveSheet.Range("A1", "D8").Value = 13
```

Eine weitere Möglichkeit ist es, einen Zellbereich über das Unterobjekt *Cells* zu definieren.

```
'Syntax:
   Objekt.Range(Cells(Row,Column))

'Beispiel:
   ActiveSheet.Range(Cells(1, 1)).Value = 13
```

Diese Möglichkeit besteht auch für einen zusammenhängenden Zellbereich.

```
'Syntax:
   Objekt.Range(Cells(Row1,Column1),Cells(Row2,Column2))

'Beispiel:
   ActiveSheet.Range(Cells(1, 1), Cells(8, 4)).Value = 10
```

Eigenschaften und Methoden des *Range*-Objekts zeigen der Objektkatalog und die VBA-Hilfe.

2.4.3 Zellmarkierungen

So wie ein Arbeitsblatt oder andere Objekte mit der Methode *Select* markiert werden können, geht dies auch für einen Zellbereich.

```
'Syntax:
   RangeObject.Select

'Beispiel:
   rngDaten.Select
```

Damit entsteht automatisch ein Objekt *Selection*. Es existiert so lange, bis eine neue Selektion erfolgt oder die Anwendung schließt. Die Adresse eines markierten Bereichs kann über die Eigenschaft *Address* erfragt werden und liefert die Bereichsangabe abhängig von den Parametern. Für den *ReferenceStyle* gibt es die Angabe xlA1 für die übliche Adressangabe Spalte & Zeile, für die Angabe xlR1C1 eine relative Adressangabe wie sie z. B. auch der Makrorecorder verwendet. Die Parameter *RowAbsolute* und *ColumnAbsolute* erlauben die Angabe in relativen oder absoluten Adressen.

Code 2.65 Die Prozedur gibt die Adresse einer Selektion aus

```
Sub AdressSelection()
   MsgBox Selection.Address _
      (ReferenceStyle:=xlA1, _
      RowAbsolute:=False, _
      ColumnAbsolute:=False)
End Sub
```

Ein Objekt *Selection* verfügt annähernd über die gleichen Eigenschaften und Methoden wie das Objekt *Range*. Es wird in der Programmierung aber weniger oft verwendet, da über die Intellisense-Funktion keine Informationen zum Objekt angeboten werden.

2.4.4 Eigenschaften und Methoden

Sollen mehrere Bereiche angesprochen werden, dann hilft die Methode *Union* des Anwendungs-Objekts. Das Beispiel (Code 2.66) zeigt, wie die Bereiche einer Objektvariablen vom Typ *Range* zugeordnet werden und deren Zellen danach die Formel für Zufallszahlen zugewiesen bekommen.

Code 2.66 Die Prozedur füllt die angegebenen Zellbereiche mit Zufallszahlen

```
Sub FillRanges()
   Dim rngDaten As Range
   Set rngDaten = Union(Range("B3:B8"), Range("D7:D10"))
   rngDaten.Formula = "=Rand()"
   Set rngDaten = Nothing
End Sub
```

Auch der Zellbereich verfügt über ein Unterobjekt *UsedRange*, mit dessen Hilfe Zeilen und Spalten des genutzten Zellbereichs abgefragt werden können.

Code 2.67 Auswertung des genutzten Zellbereichs

```
Sub RangeUsedRange()
   MsgBox "Zeilen = " & ActiveSheet.UsedRange.Rows.Count
   MsgBox "Spalten = " & ActiveSheet.UsedRange.Columns.Count
End Sub
```

Die Anwendung der Prozedur *ZellbereicheFüllen* liefert 8 Zeilen und 3 Spalten. Das ist genau der rechteckige Bereich, in dem beide Zellbereiche liegen. Erkennbar wird dieser Bereich mit der Prozedur (Code 2.68), die den Zellen eine Farbe über den Eigenschaftsbereich *Interior* und der Eigenschaft *ColorIndex* zuweist. Zu beiden liefert der Objektkatalog

wichtige Hinweise. Es ist sinnvoll, zum Objekt *UsedRange* auch das übergeordnete Objekt zu nennen.

Code 2.68 Die Prozedur füllt den genutzten Bereich mit Farbe

```
Sub UsedRangeColor()
   ActiveSheet.UsedRange.Interior.ColorIndex = 7
End Sub
```

Eine wichtige Eigenschaft ist *Offset*. Sie verschiebt den Bereich des Objekts um die angegebene Anzahl Zeilen und Spalten.

```
'Syntax:
   Range.Offset(Zeilen,Spalten)

'Beispiel:
   UsedRange.Offset(rowOffset:=1, columnOffset:=2)
```

Die Anwendung der Eigenschaft (Code 2.69) kann sehr komplex werden. In der Eigenschaft *Formula* von Zellen werden Formeln gespeichert die deren Wert bestimmen. Die Darstellung der Formel entspricht ihrer Darstellung in der Bearbeitungsleiste, also auch mit dem Gleichheitszeichen.

Code 2.69 Die Prozedur verschiebt einen Zellbereich

```
Sub OffsetRange()
   Dim wshDaten   As Worksheet
   Dim rngDaten   As Range
   Dim rngNeu     As Range

'erzeugt Zellbereich mit Zufallszahlen
   Set wshDaten = ThisWorkbook.Worksheets("Daten")
   Set rngDaten = wshDaten.Range("B3:D8")
   rngDaten.Formula = "=Rand()"

'verschiebt Zellbereich
   Set rngNeu = rngDaten.Offset(rowOffset:=3, columnOffset:=5)
   rngNeu.Formula = "=Rand()"
   Set rngDaten = Nothing
   Set rngNeu = Nothing
End Sub
```

Ebenso wichtig sind die Methoden *Copy* und *PasteSpecial*. Mit der Methode *Copy* wird der Inhalt eines Zellbereichs in die Zwischenablage kopiert. Die Methode *PasteSpecial*

fügt den Inhalt beginnend mit der angegebenen Zelle, die als die oben links gilt, ein. Hier die Zelle in Zeile 15 und Spalte C.

Code 2.70 Die Prozedur kopiert den markierten Zellbereich

```
Sub CopyUsedRange()
   ActiveSheet.UsedRange.Copy
   ActiveSheet.Range("C15").PasteSpecial
End Sub
```

Zum Abschluss sollen noch die Methode *Merge* und die Eigenschaft *MergeCells* besprochen werden. Die Prozedur (Code 2.71) erstellt aus den Zellen des zuvor erstellten Objekts *UsedRange* eine zusammengefasste Zelle. Haben die Zellen zuvor verschiedene Werte, dann gibt es zuerst eine Meldung, dass nur ein Wert übernommen werden kann und alle anderen verloren gehen. Nach der Bestätigung ist der Bereich zu einer Zelle geworden und wird auch so behandelt.

Code 2.71 Die Prozedur fasst die Zellen eines Zellbereichs zu einer Zelle zusammen

```
Sub MergeUsedRange()
   ActiveSheet.UsedRange.Merge
End Sub
```

Die Eigenschaft *MergeCells* liefert den Wert *True*, wenn sich im markierten Bereich zusammengefasste Zellen befinden, andernfalls ist der Wert *False*. Wir markieren vor dem Aufruf der nachfolgenden Prozedur einen Bereich und erhalten mit dem Aufruf die Auswertung.

2.4.5 Zeilen und Spalten

Alle Zellen einer Zeile oder Spalte können über die Objektlisten *Rows* (Zeilen) und *Columns* (Spalten) angesprochen werden. Für einzelne Zeilen und Spalten ergibt sich nachfolgende Schreibweise.

```
'Syntax:
   Rows(Zeile)
   Columns(Spalte)
'Beispiel:
   Rows(5)
   Columns(2)
```

Das Beispiel (Code 2.72) füllt die 5. Zeile und 2. Spalte des aktiven Arbeitsblattes mit Zufallszahlen zwischen 0 und 100.

Code 2.72 Die Prozedur füllt die 5. Zeile und 2. Spalte mit Zufallszahlen zwischen 0 und 100

```
Sub WriteToRow5Column2()
   Dim rngZeile   As Range
   Dim rngSpalte  As Range

   Randomize
   ActiveSheet.UsedRange.Delete
   Set rngZeile = ActiveSheet.Rows(5)
   Set rngSpalte = ActiveSheet.Columns(2)
   rngZeile.Formula = "=INT(100*Rand())"
   rngSpalte.Formula = "=INT(100*Rand())"

   Set rngZeile = Nothing
   Set rngSpalte = Nothing
End Sub
```

Werden mehrere Zeilen oder Spalten zusammengefasst, so werden die Indizes als Text vorgegeben.

```
'Syntax:
   Rows("von Zeile:bis Zeile")
   Columns("von Spalte:bis Spalte")

'Beispiel:
   Rows("5:7")
   Columns("B:E")
```

Mehrere Zeilen- und Spaltenbereiche können wieder mit Hilfe der Methode *Union* zusammengefasst werden (Code 2.73).

Code 2.73 Die Prozedur beschreibt mehrere getrennte Zeilen- und Spaltenbereiche

```
Sub WriteToRowsColumns()
   Dim rngZeilen   As Range
   Dim rngSpalten  As Range

   Randomize
   ActiveSheet.Cells.Delete
   Set rngZeilen = Union(Rows("5:7"), Rows("10:11"))
   Set rngSpalten = Union(Columns("A:B"), Columns("D:E"), _
         Columns("G:H"))
   rngZeilen.Formula = "=INT(100*Rand())"
   rngSpalten.Formula = "=INT(100*Rand())"
```

```
   Set rngZeilen = Nothing
   Set rngSpalten = Nothing
End Sub
```

Eine wichtige Methode für Zeilen und Spalten ist das Aus- und Einblenden. Es erfolgt über das Objekt *Selection* eines zuvor ausgewählten Zellbereichs.

```
'Syntax:
   RangeObject.Select
   Selection.EntireRow.Hidden = True       ' Zeile(n) ausblenden
   Selection.EntireRow.Hidden = False      ' Zeile(n) einblenden
   Selection.EntireColumn.Hidden = True    ' Spalte(n) ausblenden
   Selection.EntireColumn.Hidden = False   ' Spalte(n) ausblenden

'Beispiele:
   Rows("29:29").Select
   Selection.EntireRow.Hidden = True       'blendet Zeile 29 aus
   Columns("I:J").Select
   Selection.EntireColumn.Hidden = True    'blendet Spaltenbereich I:J
                                           'aus
```

Abschließend sollen noch Zeilenhöhe und Spaltenbreite gesetzt werden. Oft ist auch die Anpassung an den Inhalt erforderlich (Code 2.74).

Code 2.74 Die Prozedur setzt Zeilenhöhe, Spaltenbreite und passt den Spaltenbereich D:H dem Inhalt an

```
Sub AdaptionRowsColumns()
   ActiveSheet.Rows("1:2").Select
   Selection.RowHeight = 20
   ActiveSheet.Columns("A:C").Select
   Selection.ColumnWidth = 30
   ActiveSheet.Columns("D:H").Select
   Selection.Columns.AutoFit
End Sub
```

Die bisher besprochenen Objekte lassen nun auch komplexere Funktionen zu. Als Beispiel stehen auf einem Arbeitsblatt mehrere Datenreihen (Abb. 2.16).

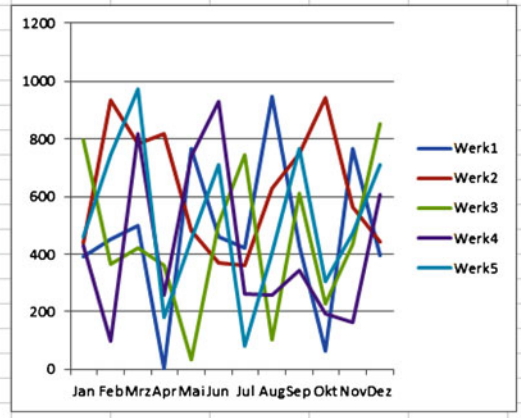
Abb. 2.16 Datenreihen auf einem gleichnamigen Arbeitsblatt

	A	B	C	D	E	F
1	Monat	Werk1	Werk2	Werk3	Werk4	Werk5
2	Jan	389	438	797	432	459
3	Feb	450	933	367	97	744
4	Mrz	498	782	420	816	974
5	Apr	2	816	363	257	182
6	Mai	768	482	33	742	451
7	Jun	461	370	507	930	710
8	Jul	423	360	744	264	80
9	Aug	945	629	104	256	405
10	Sep	430	748	609	343	768
11	Okt	62	943	227	192	307
12	Nov	765	563	436	163	473
13	Dez	395	444	854	607	708

Die Prozedur (Code 2.75) bestimmt die Anzahl vorhandener Spaltenwerte im Arbeitsblatt *Datenreihen* und erstellt ein eingebettetes Liniendiagramm (Abb. 2.17).

Abb. 2.17 Ergebnis der Auswertung und Verschiebung

	A	B	C	D	E	F
1	Monat	Werk1	Werk2	Werk3	Werk4	Werk5
2	Jan	389	438	797	432	459
3	Feb	450	933	367	97	744
4	Mrz	498	782	420	816	974
5	Apr	2	816	363	257	182
6	Mai	768	482	33	742	451
7	Jun	461	370	507	930	710
8	Jul	423	360	744	264	80
9	Aug	945	629	104	256	405
10	Sep	430	748	609	343	768
11	Okt	62	943	227	192	307
12	Nov	765	563	436	163	473
13	Dez	395	444	854	607	708
14						
15						
16						
17						
18						
19						
20						
21						
22						
23						
24						
25						
26						
27						
28						
29						
30						
31						



Code 2.75 Die Prozedur erstellt ein Liniendiagramm aus Datenreihen

```
Sub ColumnsEmbeddedChart()
   Dim wshDaten   As Worksheet
   Dim chtDaten   As ChartObject
   Dim rngXAxes   As Range
   Dim lRowMax    As Long
   Dim lRow       As Long
   Dim lColMax    As Long
   Dim lCol       As Long

'Bestimme den benutzten Bereich
   Set wshDaten = ThisWorkbook.Worksheets("Datenreihen")
   With wshDaten
      lRowMax = .UsedRange.Rows.Count
      lColMax = .UsedRange.Columns.Count
   End With
   Set rngXAxes = Range(Cells(2, 1), Cells(lRowMax, 1))

'Erstelle Diagramm
   Set chtDaten = wshDaten.ChartObjects.Add(20, 200, 400, 250)
   With chtDaten.Chart
      .ChartType = xlLine
      For lCol = 2 To lColMax
         With .SeriesCollection.NewSeries
            .Values = rngXAxes.Offset(, lCol - 1)
            .XValues = rngXAxes
            .Name = wshDaten.Cells(1, lCol)
         End With
      Next lCol
   End With

   Set rngXAxes = Nothing
   Set chtDaten = Nothing
   Set wshDaten = Nothing
End Sub
```

Mitunter möchte man die Position, die hier mit Pixeln angegeben wurde, lieber einem Zellbereich zuordnen. Die Prozedur (Code 2.76) verschiebt das Diagramm in einen vorgegebenen Zellbereich.

Code 2.76 Die Prozedur verschiebt ein Diagramm(1) in einen Zellbereich

```
Sub DiagramPosition()
   Dim chtDaten      As ChartObject
   Dim rngPosition   As Range
```

```
    Set chtDaten = ActiveSheet.ChartObjects(1)
    Set rngPosition = ActiveSheet.Range("B15:F30")

    With chtDaten.Chart.ChartArea
        .Left = rngPosition.Left
        .Width = rngPosition.Width
        .Top = rngPosition.Top
        .Height = rngPosition.Height
    End With
End Sub
```

2.4.6 Aufgabenplanung (Fortsetzung)

Nun sind wir in der Lage die Planungszeiten so auszuwerten, dass sie auf einem Arbeitsblatt eingetragen und mittels Diagramm visualisiert werden (Code 2.77). Das Ergebnis zeigt Abb. 2.18.

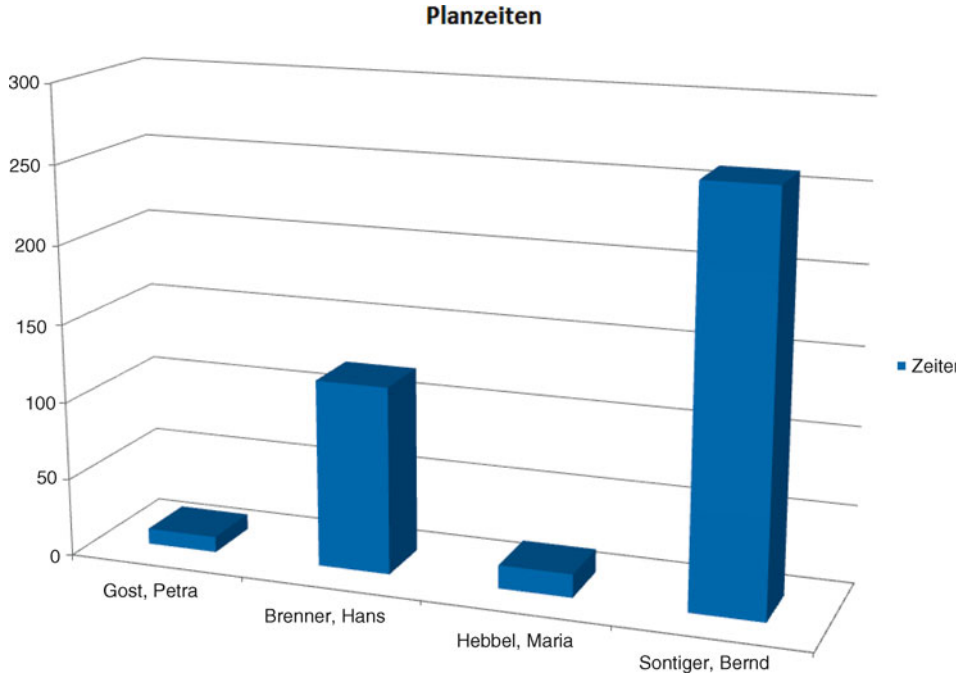


Abb. 2.18 Darstellung der Gesamtplanzeiten aller Personen

Code 2.77 Die Prozedur wertet die vorhandenen Planungen aus und erstellt ein Diagramm

```
Sub BestimmePlanzeiten()
   Dim wshPersonen   As Worksheet
   Dim wshPlanungen  As Worksheet
   Dim wshAufgaben   As Worksheet
   Dim wshPlanzeiten As Worksheet
   Dim chtZeiten     As Chart
   Dim lRowMax1      As Long
   Dim lRowMax2      As Long
   Dim lRow1         As Long
   Dim lRow2         As Long
   Dim lSumme        As Long
   Dim iAufgabe      As Integer
   Dim sName         As String

   Set wshPersonen = ThisWorkbook.Worksheets("Personen")
   Set wshPlanungen = ThisWorkbook.Worksheets("Planungen")
   Set wshAufgaben = ThisWorkbook.Worksheets("Aufgaben")
   Set wshPlanzeiten = ThisWorkbook.Worksheets("Planzeiten")

'alte Auswertung löschen
   On Error Resume Next
   Application.DisplayAlerts = False
   ThisWorkbook.Sheets("PlanzeitenAnalyse").Delete
   Application.DisplayAlerts = True
   On Error GoTo 0
   With wshPlanzeiten
      lRowMax1 = .UsedRange.Rows.Count
      .Range(.Cells(2, 1), .Cells(lRowMax1, 2)).Clear
   End With

'Auswertung
   lRowMax1 = wshPersonen.UsedRange.Rows.Count
   lRowMax2 = wshPlanungen.UsedRange.Rows.Count
   For lRow1 = 2 To lRowMax1
      sName = wshPersonen.Cells(lRow1, 2) & ", " & _
         wshPersonen.Cells(lRow1, 3)
      lSumme = 0
```

```
      For lRow2 = 2 To lRowMax2
         If wshPersonen.Cells(lRow1, 1) = _
            wshPlanungen.Cells(lRow2, 3) Then
            iAufgabe = Val(wshPlanungen.Cells(lRow2, 2))
            lSumme = lSumme + Val(wshAufgaben.Cells(iAufgabe, 4))
         End If
      Next lRow2
      wshPlanzeiten.Cells(lRow1, 1) = sName
      wshPlanzeiten.Cells(lRow1, 2) = lSumme
   Next lRow1

'Diagramm
   lRowMax1 = wshPlanzeiten.UsedRange.Rows.Count
   Set chtZeiten = ThisWorkbook.Charts.Add
   With chtZeiten
      .Activate
      .Name = "PlanzeitenAnalyse"
      With ActiveChart
         .ChartType = xl3DColumnStacked
         .HasTitle = True
         .ChartTitle.Text = "Planzeiten"
         .SetSourceData Source:=Sheets("Planzeiten").UsedRange
      End With
   End With
   Set chtZeiten = Nothing
   Set wshAufgaben = Nothing
   Set wshPlanungen = Nothing
   Set wshPersonen = Nothing
End Sub
```

2.5 Excel-Interaktionen

2.5.1 Erzeugnis-Strukturen und Stücklisten

In diesem Kapitel wird die Wechselwirkung zwischen Excel-Objekten an einem Praxisbeispiel verdeutlicht. Wir haben bereits die Darstellung von Netzplänen realisiert. Grundlage für diese und auch für Fertigungsunterlagen sind Erzeugnisstrukturen, die anhand eines Erzeugnisbaumes dargestellt werden können (Abb. 2.19).

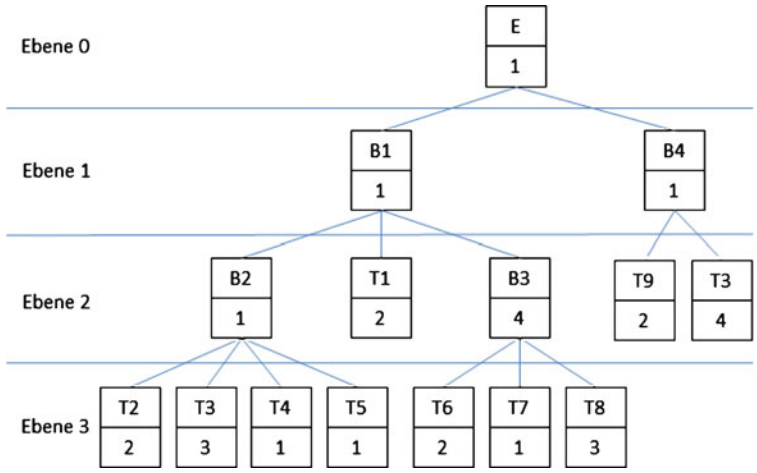
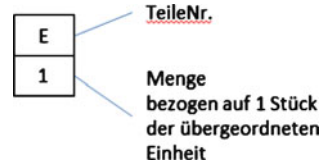


Abb. 2.19 Beispiel einer Erzeugnisstruktur

Die Elemente eines Strukturbaumes zeigt Abb. 2.20.

Abb. 2.20 Element einer Erzeugnisstruktur



Anhand vorliegender Erzeugnisbäume lassen sich verschiedene Arten von Stücklisten erzeugen. Als Stücklisten werden Verzeichnisse in Listenform verstanden, aus denen hervorgeht, aus welchen Baugruppen und Einzelteilen ein Erzeugnis besteht. Stücklisten verschiedener Formen finden sich in Industriezweigen wie:

- Strukturlisten (Maschinenbau),
- Bauvorschriften (Hoch- und Tiefbau),
- Rezepturen (Chemische Industrie),
- Zutatenlisten (Lebensmittelindustrie).

Neben vielen Misch- und Sonderformen werden hauptsächlich folgende Stücklistenformen genutzt:

- Strukturstückliste,
- Baukastenstücklisten,
- Aufzählungsstücklisten,
- Mengenstücklisten.

Eine Strukturstückliste enthält alle Merkmale (Abb. 2.21), aus denen sich die anderen Formen ableiten lassen. In fortlaufender Folge werden Baugruppen und Einzelteile ebenso wie die Fertigungsstruktur beschrieben.

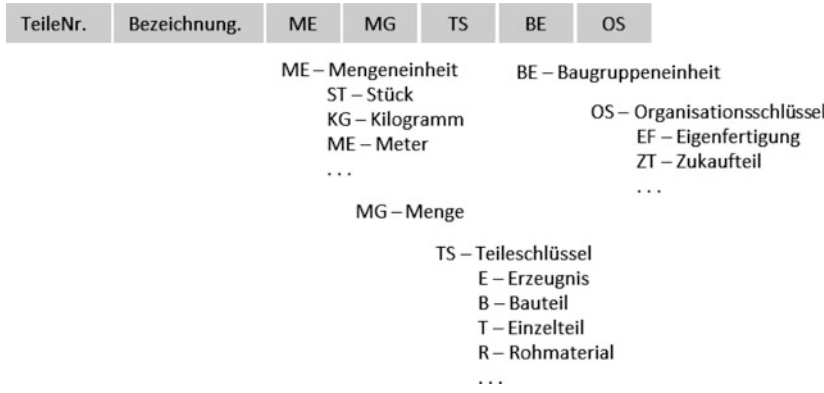


Abb. 2.21 Elemente einer Strukturstückliste

Wir verwenden die Erzeugnisstruktur als Strukturstückliste (Abb. 2.22) in einer Arbeitsmappe und erstellen daraus eine neue Arbeitsmappe (in einer weiteren Excel-Anwendung) mit den resultierenden Baukastenstücklisten als Arbeitsblätter, ebenso eine Mengenstückliste.

Abb. 2.22 Strukturstückliste zum Beispiel

	A	B	C	D	E	F	G
1	**TeileNr.**	**Bezeichnung**	**ME**	**MG**	**TS**	**BE**	**OS**
2	E1	Erzeugnis 1	St	1	E		
3	B1	Baugruppe 1	ST	1	B	E1	
4	B2	Baugruppe 2	ST	1	B	B1	
5	T2	Einzelteil 2	ST	2	T	B2	EF
6	T3	Einzelteil 3	ST	3	T	B2	EF
7	T4	Einzelteil 4	ST	1	T	B2	EF
8	T5	Einzelteil 5	ST	1	T	B2	ZT
9	T1	Einzelteil 1	ME	1	R	B1	EF
10	B3	Baugruppe 3	ST	4	B	B1	
11	T6	Einzelteil 6	ST	2	T	B3	EF
12	T7	Einzelteil 7	ST	1	T	B3	EF
13	T8	Einzelteil 8	ST	3	T	B3	EF
14	B4	Baugruppe 4	St	1	B	E1	
15	T9	Einzelteil 9	ST	2	T	B4	ZT
16	T3	Einzelteil 3	ST	4	T	B4	EF

Gesteuert wird die Verwaltung aus der Strukturstückliste über Einträge im Menüband (Code 2.78) nach gewohntem Muster.

Code 2.78 Die Prozeduren im Codefenster der Arbeitsmappe

```
Const sMenuName As String = "ErzeugnisStruktur"

Private Sub Workbook_Open()
   InitMenu
End Sub

Private Sub Workbook_BeforeClose(Cancel As Boolean)
   RemoveMenu
End Sub

Private Sub InitMenu()
   Dim objMenuBar       As CommandBar
   Dim objMenuGroup     As CommandBarControl
   Dim objMenuButton    As CommandBarControl

   Call RemoveMenu
   Set objMenuBar = Application.CommandBars.Add _
      (sMenuName, msoBarTop)
   objMenuBar.Visible = True

   Set objMenuGroup = objMenuBar.Controls.Add _
      (Type:=msoControlPopup, Temporary:=False)
   With objMenuGroup
      .Caption = sMenuName
      .Tag = sMenuName
      .TooltipText = "Aktion wählen ..."
   End With

   Set objMenuButton = objMenuGroup.Controls.Add _
      (Type:=msoControlButton, Temporary:=True)
   With objMenuButton
      .BeginGroup = False
      .Caption = "Baukastenstücklisten"
      .FaceId = 577
      .OnAction = "BaukastenStücklisten"
      .Style = msoButtonIconAndCaption
      .TooltipText = "Baukastenstücklisten erstellen"
      .Tag = "Baukastenstücklisten"
   End With

   Set objMenuButton = objMenuGroup.Controls.Add _
      (Type:=msoControlButton, Temporary:=True)
   With objMenuButton
      .BeginGroup = False
```

```
      .Caption = "Mengenstückliste"
      .FaceId = 584
      .OnAction = "Mengenstückliste"
      .Style = msoButtonIconAndCaption
      .TooltipText = "Mengenstückliste erstellen"
      .Tag = "Mengenstückliste"
   End With

   Set objMenuButton = objMenuGroup.Controls.Add _
      (Type:=msoControlButton, Temporary:=True)
   With objMenuButton
      .BeginGroup = False
      .Caption = "Erzeugnisstruktur"
      .FaceId = 523
      .OnAction = "Erzeugnisstruktur"
      .Style = msoButtonIconAndCaption
      .TooltipText = "Erzeugnisstruktur erstellen"
      .Tag = "Erzeugnisstruktur"
   End With

   Set objMenuBar = Nothing
   Set objMenuGroup = Nothing
   Set objMenuButton = Nothing
End Sub

Private Sub RemoveMenu()
   Dim objMenuBar       As CommandBar
   Dim objMenuControl   As CommandBarControl

   For Each objMenuBar In Application.CommandBars
      For Each objMenuControl In objMenuBar.Controls
         If objMenuControl.Tag = sMenuName Then
            objMenuControl.Delete
         End If
      Next
      If objMenuBar.Name = sMenuName Then
         objMenuBar.Delete
      End If
   Next

   Set objMenuBar = Nothing
   Set objMenuControl = Nothing
End Sub
```

In diesem Beispiel soll nun erstmals eine weitere Excel-Anwendung erstellt werden und beide sollen ansprechbar sein. Die Konstruktion erfolgt über Objektvariable (Code 2.79).

Code 2.79 Generierung einer neuen Excel-Anwendung

```
Sub CreateApp
   Dim appStr  As Application
   Dim appBau  As Application

   Set appStr = ThisWorkbook.Application
   Set appBau = CreateObject("Excel.Application")
   appBau.Visible = True
   MsgBox appStr.Caption

   appBau.Quit
   Set appBau = Nothing
   Set appStr = Nothing
End Sub
```

Die Anwendung der aktuellen Arbeitsmappe, die den Code in einem Modul enthält, wird über das Objekt *ThisWorkbook* angesprochen. Eigentlich müsste es ja *ActiveWorkbook* heißen, so wie *ActiveSheet* und *ActiveChart*, aber die Programmierer haben es anders gewollt. Über eine *Messagebox* wird zur Kontrolle die Eigenschaft *Caption* ausgegeben.

Eine weitere Anwendung, die noch nicht vorhanden ist, wird über die Methode *CreateObject* mit Angabe der Klasse erzeugt.

```
'Syntax:
   CreateObject (Class As String, [Servername As String])

'Beispiel:
   CreateObject ("Excel.Application")
```

Anders ist es bei bestehenden Anwendungen, sie werden über die Methode *GetObject* aufgerufen.

```
'Syntax:
   GetObject ([PathName], [Class])

'Beispiel:
   GetObject ("C:\Temp\Demo.xlsx")
```

Eine so erzeugte Anwendung muss noch für die Darstellung freigegeben werden. Ähnlich wie bei den Formularen befindet sich die Anwendung nur im Arbeitsspeicher. Das hat den großen Vorteil, dass spätere Integrationen von anderen Anwendungen nicht immer unbe-

dingt sichtbar sein müssen. Soll die Anwendung nach der Ausführung wieder geschlossen werden, dann kommt noch die Methode *Quit* zur Anwendung.

Wer das obige Demoprogramm *CreateApp* ausführt, wird feststellen, dass sich zwar die Excel-Anwendung öffnet, aber nicht zusammen mit einer Arbeitsmappe, wie es beim Standardaufruf einer Excel-Anwendung üblich ist. Doch eine neue Arbeitsmappe innerhalb der Anwendung ist über die Add-Methode der Objektliste *Workbooks* schnell generiert (Code 2.80).

Code 2.80 Generierung einer neuen Excel-Anwendung und einer neuen Arbeitsmappe

```
Sub CreateAppAndBook()
   Dim appStr  As Application
   Dim appBau  As Application
   Dim wbkBau  As Workbook

   Set appStr = ThisWorkbook.Application
   Set appBau = CreateObject("Excel.Application")
   appBau.Visible = True
   Set wbkBau = appBau.Workbooks.Add

   wbkBau.Close
   appBau.Quit
   Set appBau = Nothing
   Set appStr = Nothing
End Sub
```

Der Pseudocode zur Generierung von Baukastenstücklisten ist schnell geschrieben (Code 2.81).

Code 2.81 Pseudocode zur Generierung von Baukastenstücklisten

```
erzeuge neue Baukastenstücklisten-Arbeitsmappe
for alle Strukturstücklisten-Elemente
   if Element eine Baugruppe
      erzeuge neues Baukastenstücklisten-Arbeitsblatt
      for alle Strukturstücklisten-Elemente
         außer sich selbt
         if BE gleich Baugruppe
            trage Element in die Liste ein
         end
      next
   end
next
```

Damit folgt dann die Lösung (Code 2.82), die wie alle Beispiele nur in einer minimalistischen Version behandelt wird.

Code 2.82 Die Prozedur erzeugt Baukastenstücklisten aus einer Strukturstückliste

```
Sub BaukastenStücklisten()
   Dim appStr  As Application
   Dim appBau  As Application
   Dim wbkBau  As Workbook
   Dim wshStr  As Worksheet
   Dim wshNeu  As Worksheet
   Dim lRowMax As Long
   Dim lRow1   As Long
   Dim lRow2   As Long
   Dim lCol    As Long
   Dim lRowNeu As Long
   Dim sName   As String

   Set appStr = ThisWorkbook.Application
   Set wshStr = appStr.Worksheets("Strukturstückliste")
   lRowMax = wshStr.UsedRange.Rows.Count
'neue BaukastenStücklisten-Arbeitsmappe
   Set appBau = CreateObject("Excel.Application")
   appBau.Visible = True
   Set wbkBau = appBau.Workbooks.Add
   For lRow1 = 2 To lRowMax
      If wshStr.Cells(lRow1, 5) = "E" Or _
         wshStr.Cells(lRow1, 5) = "B" Then
         sName = wshStr.Cells(lRow1, 1)
         With wbkBau
            Set wshNeu = .Worksheets.Add _
               (after:=.Worksheets(.Worksheets.Count))
         End With
         With wshNeu
            .Name = sName
            .Range("A1:G1").Value = _
               Array("TeileNr.", "Bezeichnung", "ME", "MG", _
                  "TS", "BE", "OS")
            .Range("A1:G1").Font.Bold = True
            lRowNeu = 1
            For lRow2 = 2 To lRowMax
               If Not lRow2 = lRow1 And _
                  wshStr.Cells(lRow2, 6) = sName Then
                  lRowNeu = lRowNeu + 1
                  For lCol = 1 To 7
                     .Cells(lRowNeu, lCol) = _
```

```
                        wshStr.Cells(lRow2, lCol)
                    Next lCol
                End If
            Next lRow2
        End With
    End If
  Next lRow1
'Löschung aller leeren Tabellen
  For Each wshNeu In wbkBau.Worksheets
    If Left(wshNeu.Name, 7) = "Tabelle" Then
      wshNeu.Delete
    End If
  Next

  'wbkBau.Close
  'appBau.Quit
  Set wshStr = Nothing
  Set appBau = Nothing
  Set appStr = Nothing
End Sub
```

Bei einer Mengenstückliste muss darauf geachtet werden, dass gleiche Einzelteile in verschiedenen Baugruppen mit unterschiedlichen Mengen eingesetzt werden. Die Mengenstückliste soll in unserem Beispiel in der gleichen Arbeitsmappe wie die Strukturstückliste erstellt werden (Code 2.83). Ist sie bereits vorhanden, dann muss sie zuvor gelöscht werden.

Code 2.83 Die Prozedur erzeugt eine Mengenstückliste aus einer Strukturstückliste

```
Sub Mengenstückliste()
  Dim wshStr     As Worksheet
  Dim wshMng     As Worksheet
  Dim lRowMax    As Long
  Dim lRow1      As Long
  Dim lRow2      As Long
  Dim lRowNeu    As Long
  Dim lCol       As Long
  Dim lSumme     As Long
  Dim sKey       As String

  Set wshStr = ThisWorkbook.Worksheets("Strukturstückliste")
  lRowMax = wshStr.UsedRange.Rows.Count

'löscht vorhandene Mengenstückliste, falls vorhanden
'DisplayAlerts verhindert Meldung beim Löschen
```

```
   On Error Resume Next
   Application.DisplayAlerts = False
   ThisWorkbook.Worksheets("Mengenstückliste").Delete
   Application.DisplayAlerts = True
   On Error GoTo 0
'erzeugt neue Mengenstückliste
   Set wshMng = ThisWorkbook.Worksheets.Add
   With wshMng
      .Name = "Mengenstückliste"
      .Range("A1:D1").Value = _
         Array("TeileNr.", "Bezeichnung", "ME", "MG")
      .Range("A1:D1").Font.Bold = True
      .Columns("A:B").AutoFit
      .Columns("C:D").ColumnWidth = 5
   End With
'Auswertung
   lRowNeu = 1
   For lRow1 = 2 To lRowMax
      If wshStr.Cells(lRow1, 5) = "T" Or _
         wshStr.Cells(lRow1, 5) = "R" Then
      'Bestimme Gesamtmenge für diese Position
         sKey = wshStr.Cells(lRow1, 6)
         lSumme = Val(wshStr.Cells(lRow1, 4))
         Do
            For lRow2 = 2 To lRowMax
               If wshStr.Cells(lRow2, 1) = sKey Then
                  lSumme = lSumme * Val(wshStr.Cells(lRow2, 4))
                  sKey = wshStr.Cells(lRow2, 6)
                  Exit For
               End If
            Next lRow2
         Loop While Not sKey = ""
      'finde Einzelteil in der Mengenstückliste
         sKey = wshStr.Cells(lRow1, 1)
         For lRow2 = 2 To lRowNeu
            If wshMng.Cells(lRow2, 1) = sKey Then
               wshMng.Cells(lRow2, 4) = _
               Val(wshMng.Cells(lRow2, 4)) + lSumme
               sKey = ""
               Exit For
            End If
         Next lRow2
      'neuer Eintrag
         If Not sKey = "" Then
            lRowNeu = lRowNeu + 1
```

```
            For lCol = 1 To 3
               wshMng.Cells(lRowNeu, lCol) = _
                  wshStr.Cells(lRow1, lCol)
            Next lCol
            wshMng.Cells(lRowNeu, 4) = lSumme
         End If
      End If
   Next lRow1
   Set wshMng = Nothing
   Set wshStr = Nothing
End Sub
```

Es bleibt noch die Erzeugnisstruktur (Code 2.84) mit ähnlichen Strukturen wie beim Netzplan.

Code 2.84 Die Prozedur erzeugt eine grafische Erzeugnisstruktur

```
Sub Erzeugnisstruktur()
   Dim wshStrukt     As Worksheet
   Dim wshErzeug     As Worksheet
   Dim objElement    As clsElement
   Dim objLine       As clsLine
   Dim colElements   As Collection
   Dim lRowMax       As Long
   Dim lRow          As Long
   Dim lX            As Long
   Dim lY            As Long
   Dim bStart        As Boolean

   Set colElements = New Collection
   Set wshStrukt = ThisWorkbook.Worksheets("Strukturstückliste")
   lRowMax = wshStrukt.UsedRange.Rows.Count

'erzeuge Worksheet
   With ThisWorkbook
      On Error Resume Next
      Application.DisplayAlerts = False
      .Sheets("Erzeugnisstruktur").Delete
      Application.DisplayAlerts = True
      On Error GoTo 0
      Set wshErzeug = .Worksheets.Add _
         (after:=.Worksheets(.Worksheets.Count))
   End With
   wshErzeug.Application.DisplayFormulaBar = False
   wshErzeug.Name = "Erzeugnisstruktur"
```

```
   wshErzeug.Activate
   With ActiveWindow
      .DisplayGridlines = False
      .DisplayHeadings = False
   End With

'Auswertung
   lX = 100
   lY = 100
   bStart = True
   For lRow = 2 To lRowMax
      Set objElement = New clsElement
      With objElement
         .sTeileNr = wshStrukt.Cells(lRow, 1)
         .sTeileZu = wshStrukt.Cells(lRow, 6)
         .sMenge = wshStrukt.Cells(lRow, 4)
         If wshStrukt.Cells(lRow, 5) = "T" Or _
            wshStrukt.Cells(lRow, 5) = "R" Then
            .sTeileNr = .sTeileNr & .sTeileZu
         End If
         If bStart = True Then
            If wshStrukt.Cells(lRow, 5) = "T" Or _
               wshStrukt.Cells(lRow, 5) = "R" Then
               bStart = False
            End If
         Else
            lX = lX + 50
            If wshStrukt.Cells(lRow, 5) = "B" Then
               bStart = True
            End If
         End If
         If .sTeileZu = "" Then
            .lPosY = lY
         Else
            .lPosY = colElements.Item(.sTeileZu).lPosY + 100
         End If
         .lPosX = lX
         .Create
         If Not .sTeileZu = "" Then
            Set objLine = New clsLine
            objLine.sKey1 = .sTeileZu
            objLine.sKey2 = .sTeileNr
            objLine.Create
            Set objLine = Nothing
         End If
```

```
        End With
        colElements.Add objElement, objElement.sTeileNr
        Set objElement = Nothing
    Next lRow
    Set wshErzeug = Nothing
    Set wshStrukt = Nothing
End Sub
```

Das Ergebnis der Erzeugnisstruktur (Abb. 2.23) kann nach der Erzeugung manuell verändert werden.

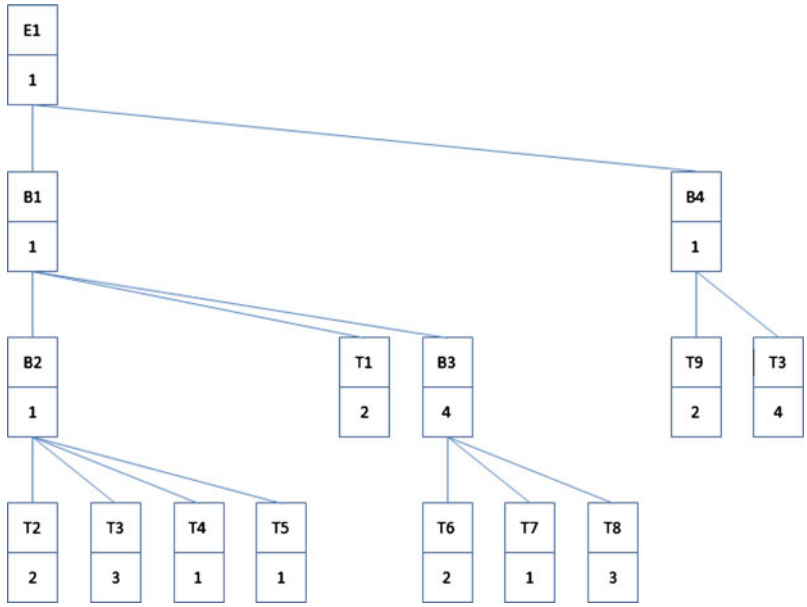


Abb. 2.23 Erzeugnisstruktur als Ergebnis

2.5.2 Daten konsolidieren

Die Methode ‚Daten konsolidieren' in Excel erlaubt die Zusammenführung gleicher Tabellenwerte aus unterschiedlichen Arbeitsmappen ohne dazu nur ein VBA-Codeschnipsel zu benutzen. Auch sind gleiche Inhalte keine Voraussetzung, aber die Tabellen benötigen eine gleiche wohlgeordnete Struktur. Die ist bei Tabellen wie in Abb. 2.24 schon nicht mehr gegeben.

Abb. 2.24 Zwei unterschiedliche Tabellen mit gleichen Datenstrukturen

Produkt	Bauteil	Anzahl	Std
A	1	7	3,5
C	2	1	7,5
A	2	1	4,3
B	3	0	6,9
C	1	17	9,5
B	1	4	2,5
E	1	14	2,8
B	2	2	4,4
D	1	7	4,8
D	2	19	1,4

Anzahl	Produkt	Bauteil	Std	Pers.Nr.
4	A	1	2,5	101
2	B	1	3,5	113
3	B	3	7,2	108
12	C	1	10,2	101
2	C	2	4,5	115
6	D	1	4,2	112
14	D	2	1,2	120
10	E	1	3,1	121

Der nachfolgende Pseudocode zeigt, wie mit Hilfe einer Programmierung die Daten doch noch zueinander finden (Code 2.85).

Code 2.85 Pseudocode zur Generierung von Baukastenstücklisten

```
Lege ein Array mit relevanten Spaltenbezeichnungen an
for alle vorhandenen Arbeitsmappen im gleichen Ordner
   öffne Arbeitsmappe
   bestimme Spaltenpositionen
   for jede Zeile mit Daten
      instanziiere Datenobjekt
      if Datenobjekt in Sammlung vorhanden
         bestimme Zeitsumme
      else
         lege Datenobjekt in Sammlung an
      end
   next
next
for alle Datenobjekte in Sammlung
   gib Daten auf neuem Arbeitsblatt aus
next
```

Die nachfolgende Prozedur öffnet alle Arbeitsmappen in einem vorgegeben Verzeichnis (Code 2.86).

Code 2.86 Die Prozedur öffnet alle Arbeitsmappen in einem Verzeichnis

```
Sub OpenAllWorkbooks()
   Dim sPfad As String
   Dim sFile As String

   sPfad = "C:\Temp\"
   sFile = Dir(sPfad & "*.xlsx")
```

```
   Do While Not sFile = ""
      Workbooks.Open sPfad & sFile
      sFile = Dir()
   Loop
End Sub
```

Die Methode *Dir* der Klasse VBA.FileSystem liefert den Namen einer Datei oder eines Verzeichnisses, die mit dem Parameter der Methode übereinstimmt.

```
'Syntax:
   Dir([PathName][, Attribute])

'Beispiel:
   Dir("C:\Temp\*.xlsx", vbHidden)
   'liefert alle versteckten Arbeitsmappen ohne Makros
   'vbNormal ist Standard
```

Die nachfolgende Prozedur (Code 2.87) öffnet alle Datenblätter und sucht nach den Spaltenüberschriften, die in einem Array vorgegeben sind. Der Spaltenindex wird gespeichert und bei der Auswertung angewendet. In eine Kollektion werden alle Datenobjekte samt ihrer Werte gesammelt. Da nur ein Schlüssel erlaubt ist, werden Produkt und Bauteil zu einem Index zusammengefasst. Das Ergebnis wird auf einem neuen Arbeitsblatt ausgegeben, das mit Datum und Uhrzeit gekennzeichnet ist.

Code 2.87 Die Prozedur konsolidiert Daten aus verschiedenen Datenmappen

```
Option Explicit
Option Base 1

Sub AllWorkbooksData()
   Dim wbkDaten   As Workbook
   Dim wshDaten   As Worksheet
   Dim wshSumme   As Worksheet
   Dim objDaten   As clsDaten
   Dim colDaten   As Collection
   Dim colItem    As Variant
   Dim sPfad      As String
   Dim sFile      As String
   Dim sIdx       As String
   Dim sZeit      As String
   Dim sAnzahl    As String
   Dim dZeit      As Double
   Dim lAnzahl    As Long
```

```
   Dim lRowMax    As Long
   Dim lRow       As Long
   Dim lColMax    As Long
   Dim lCol       As Long
   Dim lMax       As Long
   Dim lNum       As Long
   Dim sSpalte()  As Variant
   Dim lSpalte()  As Variant
   Dim sAddress   As Variant

   sSpalte = Array("Produkt", "Bauteil", "Anzahl", "Std")
   lMax = UBound(sSpalte)
   ReDim lSpalte(lMax)

'Daten lesen und summieren
   Set colDaten = New Collection
   sPfad = "C:\Temp\"
   sFile = Dir(sPfad & "*.xlsx")
   Do While Not sFile = ""
      Set wbkDaten = Workbooks.Open(sPfad & sFile)
      With wbkDaten
         For Each wshDaten In .Worksheets
            lRowMax = wshDaten.UsedRange.Rows.Count
            lColMax = wshDaten.UsedRange.Columns.Count

         'bestimme Spalten
            For lCol = 1 To lColMax
               For lNum = 1 To lMax
                  If wshDaten.Cells(1, lCol) = sSpalte(lNum) Then
                     lSpalte(lNum) = lCol
                  End If
               Next lNum
            Next lCol

         'erzeuge Objekte
            For lRow = 2 To lRowMax
               If wshDaten.Cells(lRow, 1) > 0 Then
                  Set objDaten = New clsDaten
                  sIdx = wshDaten.Cells(lRow, lSpalte(1)) & _
                     wshDaten.Cells(lRow, lSpalte(2))
                  objDaten.sKey = sIdx
                  sAnzahl = wshDaten.Cells(lRow, lSpalte(3))
                  lAnzahl = Val(sAnzahl)
                  sZeit = wshDaten.Cells(lRow, lSpalte(4))
                  dZeit = Val(Replace(sZeit, ",", "."))
```

```
                If ExistItem(colDaten, sIdx) Then
                   Set objDaten = colDaten.Item(sIdx)
                   objDaten.lAnzahl = objDaten.lAnzahl + lAnzahl
                   objDaten.dZeit = objDaten.dZeit + dZeit
                Else
                   objDaten.lAnzahl = lAnzahl
                   objDaten.dZeit = dZeit
                   colDaten.Add objDaten, sIdx
                End If
             End If
          Next lRow
       Next
       .Close
    End With
    sFile = Dir()
 Loop

'Summen in neues Arbeitsblatt ausgeben
 With ThisWorkbook
    Set wshSumme = .Worksheets.Add _
       (after:=.Worksheets(.Worksheets.Count))
 End With

 With wshSumme
    .Name = Format(Now, "DD.MM.YYYY_hh.mm")
    .Range("A1:C1").Value = _
       Array("Produkteil", "Anzahl", "Std")
    .Range("A1:C1").Font.Bold = True
    .Columns("A:C").AutoFit
    lRow = 1
    For Each colItem In colDaten
       Set objDaten = colItem
       lRow = lRow + 1
       .Cells(lRow, 1) = objDaten.sKey
       .Cells(lRow, 2) = objDaten.lAnzahl
       .Cells(lRow, 3) = objDaten.dZeit
    Next
    sAddress = wshSumme.UsedRange.Address
    sAddress = Replace(sAddress, "$", "")
 End With

'Sortierung
 With wshSumme.Sort
    .SortFields.Clear
    .SortFields.Add Key:=Range("A2"), SortOn:=xlSortOnValues, _
```

```
         Order:=xlAscending, DataOption:=xlSortNormal
      .SetRange Range(sAddress)
      .Header = xlYes
      .MatchCase = False
      .Orientation = xlTopToBottom
      .SortMethod = xlPinYin
      .Apply
   End With
End Sub

Function ExistItem(colAll As Collection, sKey As Variant) As Boolean
   Dim objDaten As clsDaten
   ExistItem = True
   On Error GoTo NotFound
   Set objDaten = colAll.Item(sKey)
   On Error GoTo 0
   Set objDaten = Nothing
   Exit Function

NotFound:
   ExistItem = False
End Function
```

Die Angabe *Option Base 1* sorgt dafür, dass der Index des Arrays mit Eins beginnt.

```
'Syntax:
   Option Base {0 | 1}

'Beispiel:
   Option Base 1  'stellt die Untergrenze von Arrays in Modulen auf 1
```

Die Sortierung löscht zuerst vorhandene Sortieranweisungen (*SortFields*) um dann eine neue zu erstellen. Die Prozedur benutzt eine Funktion zur Bestimmung vorhandener Items in der Kollektion. Das Ergebnis der Auswertung zeigt Abb. 2.25.

Abb. 2.25 Beispiel-Ergebnis der Konsolidierung

Produkteil	Anzahl	Std
A1	11	6
A2	1	4,3
B1	6	6
B2	2	4,4
B3	3	14
C1	29	20
C2	3	12
D1	13	9
D2	33	2,6
E1	24	5,9

2.5.3 Dokument-Manager

In diesem Anwendungsbeispiel (Abb. 2.26) wird eine Excel-Mappe als Dokument-Manager verwendet. Zunächst wird ein Verzeichnis vorgegeben, ab dem dann in einem Synchronisationslauf alle Unterverzeichnisse und Dateien verwaltet werden.

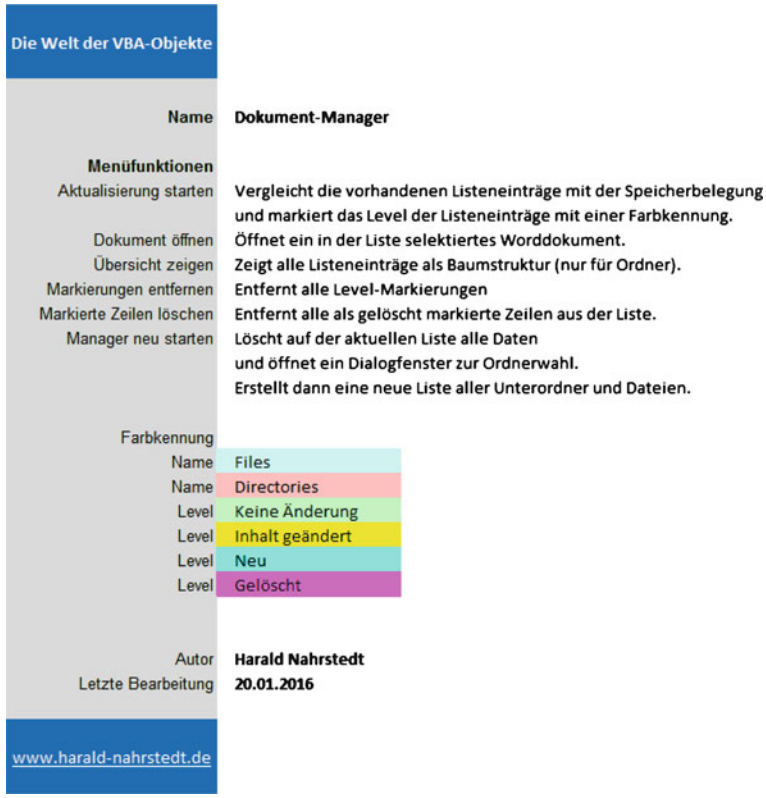


Abb. 2.26 Das Cover des Dokument-Managers

Der Vorteil gegenüber einem normalen Explorer liegt darin, dass der Dokument-Manager Änderungen zwischen zwei Synchronisierungsläufen grafisch darstellt und außerdem eigene Kommentare zu Ordnern und Dateien erlaubt.

Im ersten Schritt wird ein Arbeitsblatt aktiviert, auf dem die neue Ordnerstruktur dargestellt werden soll. Das kann sowohl ein neues als auch ein bereits genutztes Arbeitsblatt sein. Mit der Zustimmung durch ein Dialogfeld werden die Inhalte des aktiven Arbeitsblattes gelöscht. In einem Dialogfeld wird dann das oberste Verzeichnis der Ordnerstruktur (Root) gewählt. Danach erfolgt die Auswertung (Abb. 2.27).

	A	B	C	D	E	F
1		Root	I:\01_Eigene Daten\01_Technik\01_Bücher\10_Die Welt der V	Generiert am	15-12-2015 12:30	
2				Aktualisiert am	15-12-2015 12:30	
3	T	Level	Name	Description	Date	Size
4	F	1	01-01_Modellbasierte Entwicklung.docx		14-12-2015 09:55	284822
5	F	1	01-02_Objekte und Objektlisten.docx		14-12-2015 10:21	128260
6	F	1	01-03_Formulare und Steuerelemente.docx		14-12-2015 10:06	212569
7	F	1	01-04_Klassen und Bibliotheken.docx		14-12-2015 10:13	150703
8	F	1	01-05_Shapes und Formen.docx		14-12-2015 10:24	189833
9	F	1	02-01_Excel Anwendungen.docx		14-12-2015 11:44	72367
10	F	1	02-02_Excel Arbeitsmappen.docx		14-12-2015 11:45	176872
11	F	1	02-03_Excel Blätter.docx		14-12-2015 11:45	81833
12	F	1	02-04_Excel Zellen und ihre Sammlungen.docx		14-12-2015 13:42	155022
13	F	1	02-05_Excel-Interaktionen.docx		14-12-2015 18:43	137552
14	F	1	03-01_Word Anwendungen.docx		09-08-2015 09:51	118964

Abb. 2.27 Beispiel von Listeneinträgen

Die Spalte Description dient zur Eingabe eigener Kommentare. In zeitlichen Abständen erfolgen nun Aktualisierungen. Das Ergebnis des Vergleichs zwischen Listeneintrag und Speicherbelegung wird in Spalte *Level* durch Farbmarken eingetragen (Abb. 2.26). Diese können bei Bedarf, spätestens jedoch bei einer neuen Aktualisierung, gelöscht werden. Als gelöscht markierte Einträge können auch in der Liste gelöscht werden.

Innerhalb der Arbeitsmappe des Dokument-Managers können beliebig viele Arbeitsblätter mit unterschiedlichen Roots verwaltet werden. So lassen sich auch Teilbereiche eines Projekts übersichtlich kontrollieren.

Die Codezeilen gebe ich hier aus Platzgründen nicht wieder. Das Beispiel ist als Download von meiner Webseite (Übersicht) abrufbar.

Word-Objekte

3

Word ist die weltweit bedeutendste Textverarbeitung zur Erstellung und Verwaltung von Dokumentationen. Sie besitzt ebenso wie Excel eine Objekt-Bibliothek und muss für andere Anwendung zur frühen Bindung unter Verweise in der VBA-Entwicklungsumgebung ausgewählt werden.

3.1 Word-Anwendungen

Das Anwendungs-Objekt *Application* ist das oberste Objekt der Word-Objekt-Hierarchie (Abb. 3.1). Genau wie bei Excel ist auch die Word-Anwendung eng mit dem Dokument verbunden, welches das eigentliche Nutzungs-Objekt ist. So werden beide, wenn sie existieren, beim Start auch aufgerufen.

H. Nahrstedt, *Die Welt der VBA-Objekte*, DOI 10.1007/978-3-658-13891-2_3

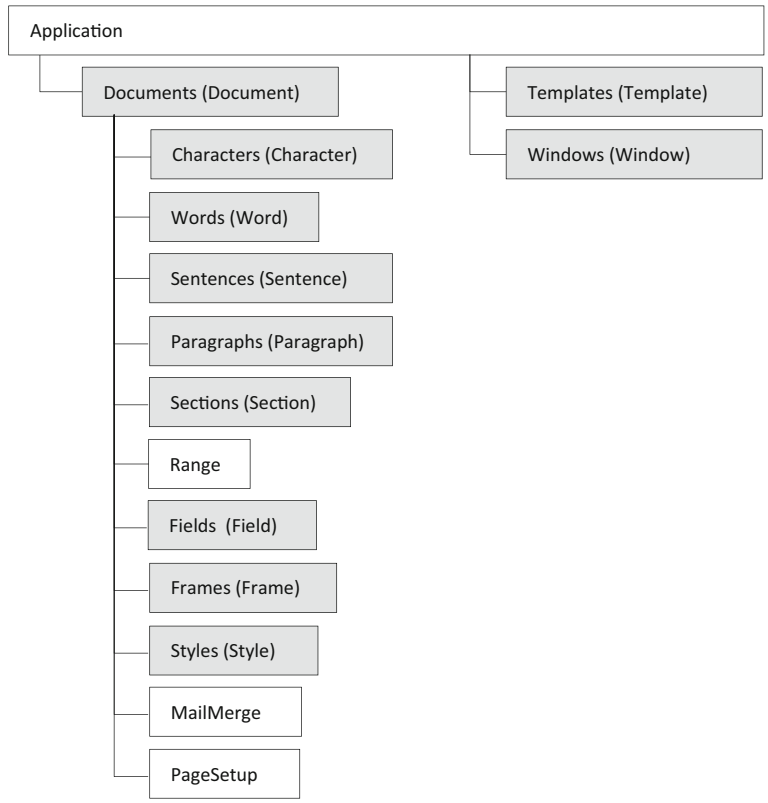


Abb. 3.1 Die wichtigsten Unterobjekte und Objektlisten (*grau*) des Application-Objekts

Im Objektkatalog befindet sich eine ausführliche Darstellung der Word-Objektmodell-referenz. Mit ihr kann durch die Dokumentation navigiert werden. Ebenso gibt es eine Aufstellung aller Elemente des Application-Objekts und die der nachfolgenden Objekte. Auch die VBA-Hilfe nennt alle Elemente des Word Application-Objekts.

3.1.1 Eigenschaften

Das Anwendungs-Objekt von Word verhält sich anders als das Anwendungs-Objekt von Excel. Mit jedem Aufruf einer Word-Datei bekommt jedes Dokument ein eigenes Fenster. Trotzdem gehören sie alle zur Anwendung. Nur mit gedrückter Umschalttaste und einem Doppelklick auf das Word-Symbol wird eine neue Anwendung zusätzlich geöffnet. Die Prozedur (Code 3.1), die wir schon von Excel kennen, schaltet mit der Eigenschaft *Visible* gleichzeitig alle Dokumente auf unsichtbar und dann wieder auf sichtbar.

Code 3.1 Die Eigenschaft Visible der Word-Anwendung wirkt sich auf alle geöffneten Dokumente aus

```
Sub ApplicationVisible()
   Application.Visible = False
   MsgBox "Anwendung nicht sichtbar! Weiter mit OK."
   Application.Visible = True
End Sub
```

Von allen Eigenschaften, die eine Word-Anwendung besitzt, werden im Code 3.2 einige ausgegeben. Im Objektkatalog finden sich noch viele andere, für die sich eine ausführliche Betrachtung lohnt.

Code 3.2 Die Prozedur liefert Informationen zur Anwendung

```
Sub ApplicationInfos()
   Dim sInfo As String

   With Application
      sInfo = "Version: " & .Version & vbCrLf
      sInfo = sInfo & "Betriebssystem: " & .System.OperatingSystem & _
              vbCrLf
      sInfo = sInfo & "SystemPfad: " & .Path & vbCrLf
      sInfo = sInfo & "Aktiver Drucker: " & .ActivePrinter & vbCrLf
      sInfo = sInfo & "Benutzer Name: " & .UserName & vbCrLf
      MsgBox sInfo, vbInformation + vbOKOnly, "Systeminfo"
   End With
End Sub
```

Gerade im Umgang mit viel Text ist die Autokorrektur-Eigenschaft der Word-Anwendung ein wichtiges Instrument. Dazu in Code 3.3 einige Codezeilen.

Code 3.3 Die Prozedur setzt Autokorrektur-Eigenschaften der Anwendung

```
Sub DoAutoCorrect()
   Dim appWord As Application

   Set appWord = CreateObject("Word.Application")
   appWord.Visible = True
   With appWord.AutoCorrect
      'erster Buchstabe eines Wochentages großschreiben
      .CorrectDays = True
      'Korrektureintrag in der aktiven Anwendung:
      .Entries.Add Name:="hate", Value:="hatte"
   End With
End Sub
```

Die Eigenschaft *Caption* ist auch bei dieser Anwendung gegeben. So kann die Überschrift des Anwendungsfensters gelesen und beschrieben werden.

```
'Syntax
   Application.Caption

'Beispiele:
   sText = appWord.Caption
   appWord.Caption = "TEST"
```

Die Eigenschaft StatusBar der Anwendung steht für den Text in der Statuszeile des Anwendungsfensters. Auch sie kann wie Caption gelesen und beschrieben werden.

```
'Syntax
   Application.StatusBar

'Beispiel:
   appWord.StatusBar = "Datei wurde gespeichert!"
```

3.1.2 Methoden

Die Instanziierung einer Word-Anwendung ist wenig hilfreich, wenn nicht gleichzeitig eine Instanziierung eines Word-Dokuments damit verbunden ist (Code 3.4). Wie ein Arbeitsblatt in Excel ist das Dokument die Arbeitsplattform in Word.

Code 3.4 Die Prozedur öffnet ein neues Word-Dokument

```
Sub OpenNewApplication()
   Dim appWord As Application
   Dim docText As Document

   Set appWord = CreateObject("Word.Application")
   appWord.Visible = True
   Set docText = appWord.Documents.Add
   MsgBox docText.Name
   docText.Close
   appWord.Quit
   Set docText = Nothing
   Set appWord = Nothing
End Sub
```

Während die Methode *Close* das Dokument schließt, beendet die Methode *Quit* die Word-Anwendung. Ein neu geöffnetes Dokument kann mit der Methode *Save* gespeichert werden.

```
'Syntax
   Save(Noprompt,OriginalFormat)

'Beispiel:
   docText.Save NoPrompt:=True
   'True = alle speichern, False = nur bei Änderung
```

Es gibt auch die Eigenschaft *Saved*, die auf *True* steht, wenn keine weitere Änderung im Dokument stattfand. Die Anweisung (Code 3.5) speichert das Dokument wenn eine Änderung seit der letzten Speicherung vorliegt.

Code 3.5 Die Anweisung speichert ein Dokument nach einer Änderung

```
'Beispiel:
   If docText.Saved = False Then _
      docText.Save
```

Da ein neu geöffnetes Dokument noch keinen Namen hat, kann mit der Methode *SaveAs* auch Pfad und Name der Datei mit angegeben werden.

```
'Syntax
   SaveAs (FileName [, FileFormat][, LockComments][, Password])

'Beispiel:
   docText.SaveAs2 FileName:= "C:\Temp\Ablage.docx"
```

Die Druckausgabe einer Anwendung erfolgt mit der Methode *PrintOut*.

```
'Syntax
   Application.PrintOut

'Beispiele:
   'druckt die ersten beiden Seiten
   appWord.PrintOut Range:=wdPrintFromTo, From:= "1", To:= "2"
   'druckt die Kommentare im Dokument
   appWord.PrintOut Item:=wdPrintComments
   'druckt 6 Seiten auf einem Blatt
   appWord.PrintOut PrintZoomRow:=2, PrintZoomColumn:=3
   'druckt das Dokument mit 75% verringerter Größe
   appWord.PrintOut Range:=wdPrintFromTo, From:= "1", To:= "2"
```

Eine bereits bestehende Word-Datei kann mit der Methode *GetObject* instanziiert werden (Code 3.6).

Code 3.6 Die Prozedur öffnet ein bestehendes Word-Dokument

```
Sub OpenExistApplication()
   Dim appWord As Application
   Dim docText As Document
   Dim sFile   As String

   sFile = "C:\Temp\Ablage.docx"
   If Not Dir(sFile) = "" Then
      Set docText = GetObject(sFile)
      docText.Activate
      Set appWord = docText.Application

      MsgBox appWord.Name

      docText.Close
      Set docText = Nothing
      Set appWord = Nothing
   End If
End Sub
```

Eine sehr wichtige Methode ist *MergeDocuments*. Sie erlaubt den Vergleich von Dokumenten und gibt eine Zusammenfassung in einem weiteren Dokument aus (Code 3.7). Ein Ausgabebeispiel zeigt Abb. 3.2.

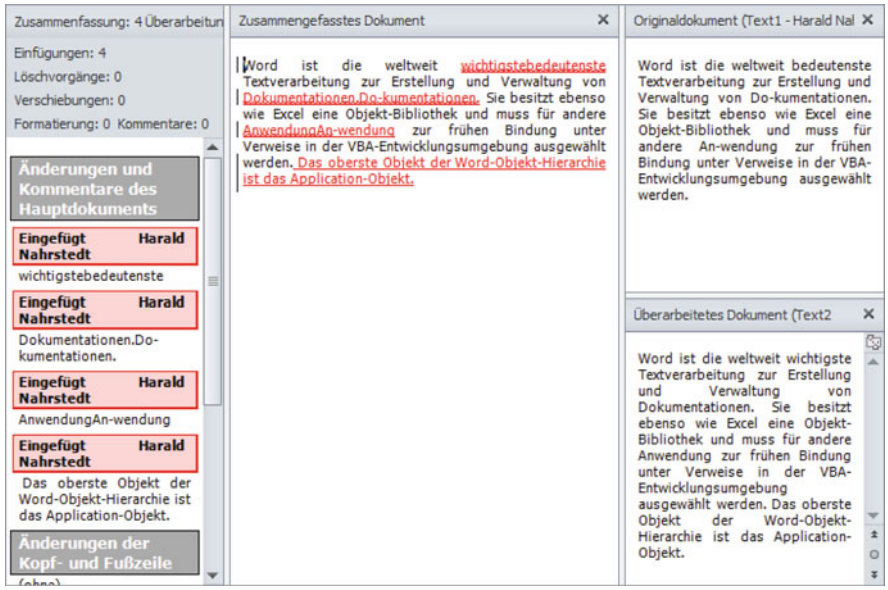


Abb. 3.2 Ausgabebeispiel

Code 3.7 Die Prozedur vergleicht zwei Word-Dokumente

```
Sub DoMergeDocuments()
   Dim docText1 As Document
   Dim docText2 As Document

   Set docText1 = GetObject("C:\Temp\Text1.docx")
   Set docText2 = GetObject("C:\Temp\Text2.docx")
   Application.MergeDocuments _
      OriginalDocument:=docText1, _
      RevisedDocument:=docText2, _
      Destination:=wdCompareDestinationNew
End Sub
```

Zum Abschluss soll noch die Methode *Move* an zwei kleinen Helferchen gezeigt werden. Die erste Prozedur blendet für kleine Zwischenrechnungen den Windows-Rechner ein und die zweite Methode Windows-Notepad für kurze Notizen (Code 3.8). *Move* positioniert sie jeweils an eine vorgegeben Position.

Code 3.8 Zwei kleine Helferchen

```
Sub UseCalc()
   Shell ("Calc.exe")
   With Tasks("Rechner")
       .WindowState = wdWindowStateNormal
       .Move Top:=50, Left:=50
   End With
End Sub

Sub UseNotiz()
   Shell ("Notepad.exe")
   With Tasks("Unbenannt - Editor")
       .WindowState = wdWindowStateNormal
       .Move Top:=50, Left:=50
       .Height = 200
       .Width = 200
   End With
End Sub
```

Die Tasknamen lassen sich in einer *For-Each*-Schleife leicht finden, falls weitere Helferchen gewünscht sind.

3.1.3 Ereignisse

Auch bei der Word-Anwendung müssen Ereignisse wieder über den Umweg einer Objektvariablen kreiert werden. Im Code-Fenster von *ThisDocument* werden Definitionen und Prozeduren eingestellt (Code 3.9).

Code 3.9 Instanziierung einer Anwendungs-Objektvariablen

```
Public WithEvents appWord As Application

Private Sub Document_Open()
   If appWord Is Nothing Then
      Set appWord = ThisDocument.Application
   End If
End Sub
Private Sub appWord_DocumentBeforeClose _
   (ByVal Doc As Document, Cancel As Boolean)
   If Doc.Saved = False Then
      MsgBox "Dokument wurde geändert und sollte gespeicheret werden!"
   Else
      MsgBox "Dokument wurde nicht geändert!"
   End If
End Sub
```

Die Instanziierung ist ja bereits von Excel bekannt. Interessant ist die Abfrage mit *Is Nothing*, ob eine Objektvariable für Anwendungs-Ereignisse bereits existiert, denn sie darf ja nur einmal im Code vorkommen.

Eine praktische Anwendung der Ereignisprozedur *appWord_DocumentBeforeClose* ist der Eintrag eines Zeitstempels in ein Dokument bevor es geschlossen wird (Code 3.10).

Code 3.10 Ereignisprozedur für Dokument-Zeitstempel

```
Public WithEvents appWord As Application

Private Sub appWord_DocumentBeforeClose( _
   ByVal Doc As Document, Cancel As Boolean)
   With Doc.Paragraphs(1).Range
      If InStr(.Text, "Letzte Änderung:") = 0 Then
         .InsertParagraphBefore
         Selection.SetRange Start:=0, End:=0
      Else
         Selection.SetRange Start:=0, End:=.End - 1
      End If
   End With
```

```
   Selection.Range.Text = "Letzte Änderung: " & _
      Date & ":" & Format(Time, "Short Time")
End Sub

Private Sub Document_Open()
   If appWord Is Nothing Then
      Set appWord = ThisDocument.Application
   End If
End Sub
```

Die Ereignisprozedur prüft, ob sich im ersten Absatz bereits ein Eintrag *Letzte Änderung:* befindet. Ist das nicht der Fall, erfolgt ein solcher Eintrag. Danach wird der vorhandene Eintrag, ob neu oder alt, markiert und durch einen neuen Zeitstempel ersetzt (Abb. 3.3).

Letzte Änderung: 16.12.2015:09:00

Abb. 3.3 Beispiel eines Zeitstempels

3.2 Word-Dokumente

Das Word-Objekt *Document* ist ein Element der Objektliste *Documents* und kann folglich dort unter seinem Index oder aus einer *For-Each-Next*-Schleife angesprochen werden.

```
'Syntax:
Documents.(Item(IndexNr oder Name))

'Beispiele:
Documents.Item(1)
Documents.Item("C:\Temp\Protokoll.docx")
```

In der VBA-Hilfe befinden sich unter *Elemente des Document-Objekts* eine Aufstellung aller Eigenschaften, Methoden und Ereignisse des *Workbooks*-Objekts.

3.2.1 Eigenschaften

Neben der Auswahl eines Dokuments aus der Objektliste gibt es auch hier ein besonderes Dokument, in dem sich der Programmcode befindet und mit *ThisDocument* angesprochen wird. Ähnlich dem *ThisWorkbook*-Objekt in Excel. In der Prozedur (Code 3.11) wird der volle Name des aktiven Dokuments ausgegeben (Abb. 3.4). Während die Eigenschaft *Name* nur den Namen des Dokuments liefert, wird mit *FullName* auch der Pfad mit angegeben.

Abb. 3.4 Ausgabe des aktiven Dokumentnamens

Code 3.11 Die Prozedur gibt den vollen Namen des zugehörigen Dokuments aus

```
Sub ActiveDocumentName()
   Dim objAktiv As Document

   Set objAktiv = ThisDocument
   MsgBox objAktiv.FullName
   Set objAktiv = Nothing
End Sub
```

Zum aktiven Dokument gehört ein Fenster, dessen Darstellung mit der Eigenschaft *ActiveWindow* gesetzt werden kann (Code 3.12).

Code 3.12 Die Prozedur verändert Position und Maße des aktiven Dokument-Fensters

```
Sub SetDocumentWindow()
   Dim objDok As Document
   Set objDok = ThisDocument
   With objDok.ActiveWindow
      .Left = 50
      .Top = 50
      .Width = 400
      .Height = 400
   End With
   Set objDok = Nothing
End Sub
```

Neben einer Vielzahl von Eigenschaften die sich in der Objektbibliothek befinden, ist das *Password* eine der wichtigen. Oft muss ein Dokument vor Veränderungen geschützt werden. Doch seit dem ersten Auftauchen von Computerviren wissen wir, einen wirklichen Schutz gibt es nicht. Die Prozedur (Code 3.13) legt für das aktive Dokument ein Passwort fest und schließt die Anwendung, so dass mit dem nächsten Start das Passwort angegeben werden muss.

Code 3.13 Die Prozedur vergibt ein Passwort für das aktive Dokument

```
Sub SetDocumentPassword()
   Dim objDok As Document
   Dim sPassw As String
   sPassw = "Word"
   Set objDok = ThisDocument
   With objDok
      '.Password = sPassw
      .Password = ""
      .Close
   End With
   Set objDok = Nothing
End Sub
```

3.2.2 Methoden

Ein neues Dokument lässt sich mit der Methode *CreateObject* der VBA.Interaction-Klasse erzeugen. Eine saubere Art der Programmierung beginnt mit der Anwendung (Code 3.14). Auch dies wurde in Excel bereits in ähnlicher Weise gezeigt.

Code 3.14 Die Prozedur erstellt ein neues Dokument

```
Sub OpenNewApplication()
   Dim appWord As Application
   Dim docText As Document

   Set appWord = CreateObject("Word.Application")
   appWord.Visible = True
   Set docText = appWord.Documents.Add

   MsgBox docText.Name

   docText.Close
   appWord.Quit
   Set docText = Nothing
   Set appWord = Nothing
End Sub
```

Das Öffnen eines bereits vorhandenen Dokuments kann mit der *GetObject*-Methode durchgeführt werden (Code 3.15).

Code 3.15 Die Prozedur öffnet ein vorhandenes Dokument

```
Sub OpenExistApplication()
   Dim appWord As Application
   Dim docText As Document
   Dim sFile   As String

   sFile = "C:\Temp\Ablage.docx"
   If Not Dir(sFile) = "" Then
      Set docText = GetObject(sFile)
      docText.Activate
      Set appWord = docText.Application

      MsgBox appWord.Name

      docText.Close
      Set docText = Nothing
      Set appWord = Nothing
   End If
End Sub
```

Da alle geöffneten Dokumente zur Objektliste *Documents* gehören, ist eine vereinfachte Version über die Methode *Open* der Objektliste möglich (Code 3.16).

Code 3.16 Die Prozedur öffnet ein vorhandenes Dokument über die Open Methode der Dokumentenliste

```
Sub OpenDocument()
   Documents.Open FileName:="C:\Temp\Ablage.docx"
End Sub
```

In den vorherigen Prozeduren wurde bereits die Methode *Close* zum Schließen eines Dokuments und die Methode *Quit* zum Schließen der Anwendung eingesetzt. Soll ein Dokument vorher gespeichert werden, dann sind die Methoden *Save* bzw. *SaveAs* auch hier vorhanden (Code 3.17).

Code 3.17 Die Prozedur erinnert ans Speichern wenn das Dokument geändert wurde

```
Sub ActiveDocumentSave()
   Dim objDoku As Document

   Set objDoku = ThisDocument
   If Not objDoku.Saved Then
      MsgBox "Bitte Dokument speichern!"
   End If
   objDoku.Close
   Set objDoku = Nothing
End Sub
```

Die Prozedur (Code 3.18) speichert ein neu erstelltes Dokument mit Pfad- und Namens-Angabe mithilfe der Methode *SaveAs*.

Code 3.18 Die Prozedur speichert ein Dokument mit der SaveAs Methode

```
Sub DocumentSaveAs()
   Dim objDoku    As Document
   Dim sPfad      As String
   Dim sName      As String

   Set objDoku = Application.Documents.Add
   sPfad = "C:\Temp\"
   sName = "Test"
   objDoku.SaveAs FileName:=sPfad & sName
   objDoku.Close
   Set objDoku = Nothing
End Sub
```

Nicht selten müssen Änderungen, die an einer Kopie des Original-Dokuments vorgenommen werden, wieder ins Original zurückkopiert werden. Die Methode *Merge* erfüllt genau diese Aufgabe. Im Beispiel (Code 3.19) wird die Kopie von Text1 als Text2 mit seinen Änderungen wieder zurück nach Text1 geführt.

Code 3.19 Die Prozedur führt Änderungen in Text2, der Kopie von Text1, und Text1 zusammen

```
Sub MergeDocuments()
   Dim appWord As Application
   Dim docText As Document
   Dim sFile1  As String
   Dim sFile2  As String

   sFile1 = "C:\Temp\Text1.docx"
   sFile2 = "C:\Temp\Text2.docx"
   If Not (Dir(sFile1) = "" And _
      Dir(sFile2) = "") Then
      Set docText = GetObject(sFile1)
      docText.Activate
      docText.Merge FileName:=sFile2

      Set docText = Nothing
      Set appWord = Nothing
   End If
End Sub
```

Das Ergebnis (Abb. 3.5) entspricht dem Ergebnis der Methode *MergeDocuments* des Word-Objekts *Application*. Beide Texte werden in ein weiteres Dokument zusammengeführt.

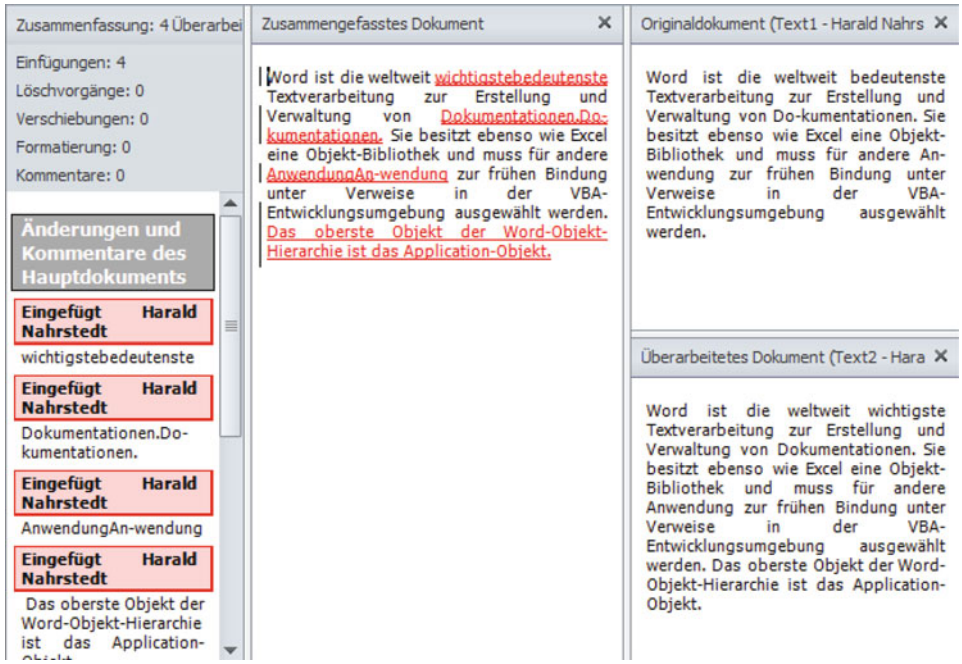


Abb. 3.5 Die Oberfläche der Merge-Methode

3.2.3 Ereignisse

Jedes Word-Dokument verfügt unter dem Namen *ThisDocument* über ein Objekt mit Ereignissen (Events). Zu denen gehören auch *Document_Open* und *Document_Close*. Das erste Ereignis wird beim Öffnen des Dokuments aufgerufen und das zweite bevor das Dokument geschlossen wird. In Excel heißt das Ereignis korrekter *_BeforeClose*. Damit sind wir auch in Word in der Lage eigene CommandBar-Elemente und Controls zu installieren (Code 3.20).

Code 3.20 Template für Menüeinträge unter Word

```
Const sMenuName As String = "WordMenü"

Private Sub Document_Open()
   InitMenu
End Sub
```

```
Private Sub Document_Close()
   RemoveMenu
End Sub

Private Sub InitMenu()
   Dim objMenuBar       As CommandBar
   Dim objMenuGroup     As CommandBarControl
   Dim objMenuButton    As CommandBarControl

   RemoveMenu

   Set objMenuBar = Application.CommandBars.Add _
      (sMenuName, msoBarTop)
   objMenuBar.Visible = True

   Set objMenuGroup = objMenuBar.Controls.Add _
      (Type:=msoControlPopup, Temporary:=False)
   With objMenuGroup
      .Caption = sMenuName
      .Tag = sMenuName
      .TooltipText = "Aktion wählen ..."
   End With

   Set objMenuButton = objMenuGroup.Controls.Add _
      (Type:=msoControlButton, Temporary:=True)
   With objMenuButton
      .BeginGroup = False
      .Caption = "DokumentName"
      .FaceId = 59
      .OnAction = "ActiveDocumentName"
      .Style = msoButtonIconAndCaption
      .TooltipText = "DokumentName anzeigen"
      .Tag = "Dokumentname"
   End With

   Set objMenuBar = Nothing
   Set objMenuGroup = Nothing
   Set objMenuButton = Nothing
End Sub

Private Sub RemoveMenu()
   Dim objMenuBar       As CommandBar
   Dim objMenuControl   As CommandBarControl
```

```
   For Each objMenuBar In Application.CommandBars
      For Each objMenuControl In objMenuBar.Controls
         If objMenuControl.Tag = sMenuName Then
            objMenuControl.Delete
         End If
      Next
      If objMenuBar.Name = sMenuName Then
         objMenuBar.Delete
      End If
   Next

   Set objMenuBar = Nothing
   Set objMenuControl = Nothing
End Sub
```

3.2.4 Dokument-Eigenschaften

Auch hier unterscheiden wir zwischen den *BuiltInDocumentProperties*, die mit der Erstellung eines Dokuments angelegt werden, und den *CustomDocumentProperties*, die jederzeit von den Nutzern angelegt, geändert und auch gelöscht werden können. Zuerst werden die *BuiltInDocumentProperties* gelesen (Code 3.21) und beschrieben (Code 3.22).

Anhang 3, Tab. A3.1 zeigt eine Übersicht aller *BuiltInDocumentProperties.*

Code 3.21 Die Prozedur liest BuiltInDocumentProperties des aktiven Dokuments

```
Sub ReadDocumentProperties()
   Dim prpDoc  As Variant
   Dim sHead   As String
   Dim sText   As String

   Set prpDoc = ActiveDocument.BuiltInDocumentProperties
   sHead = "Built In Document Properties: "
   sText = "Titel des Dokuments: " & prpDoc("Title") & vbCrLf
   sText = sText & "Thema des Dokuments: " & prpDoc("Subject") & _
           vbCrLf
   sText = sText & "Autor: " & prpDoc("Author") & vbCrLf
   sText = sText & "Manager: " & prpDoc("Manager") & vbCrLf
   sText = sText & "Firma: " & prpDoc("Company") & vbCrLf
   sText = sText & "Kategorie: " & prpDoc("Category") & vbCrLf
   sText = sText & "Stichwörter: " & prpDoc("KeyWords") & vbCrLf
   sText = sText & "Kommentar: " & prpDoc("Comments") & vbCrLf
   sText = sText & "Hyperlinkbasis: " & prpDoc("Hyperlink base")
   MsgBox sText, vbInformation, sHead
   Set prpDoc = Nothing
End Sub
```

Code 3.22 Die Prozedur beschreibt BuiltInDocumentProperties des aktiven Dokuments

```
Sub WriteDocumentProperties()
   Dim prpDoc

   Set prpDoc = ActiveDocument.BuiltInDocumentProperties
   prpDoc("Title") = "Titel des Dokuments"
   prpDoc("Subject") = "Thema des Dokuments"
   prpDoc("Author") = "Autor"
   prpDoc("Manager") = "Manager"
   prpDoc("Company") = "Firma"
   prpDoc("Category") = "Kategorie"
   prpDoc("KeyWords") = "Stichwörter"
   prpDoc("Comments") = "Kommentar"
   prpDoc("Hyperlink base") = "Hyperlinkbasis"
   Set prpDoc = Nothing
End Sub
```

Das Lesen und Beschreiben von *CustomDocumentProperties* geschieht in gleicher Weise wie bei den *BuiltInDocumentProperties*. Die Prozedur (Code 3.23) erstellt einen benutzerdefinierten Eintrag im aktiven Dokument. Die Prozedur (Code 3.24) löscht einen benutzerdefinierten Eintrag im aktiven Dokument.

Code 3.23 Die Prozedur erstellt einen benutzerdefinierten Eintrag im aktiven Dokument

```
Sub CreateCustomProperty()
   Dim sName      As String
   Dim sInhalt    As String

   sName = InputBox("Eigenschaftsname = ")
   sInhalt = InputBox("Inhalt von " & sName & " = ")
   On Error Resume Next
   ThisDocument.CustomDocumentProperties.Add _
      Name:=sName, LinkToContent:=False, _
      Type:=msoPropertyTypeString, Value:=sInhalt
   If Err.Number = 0 Then
      MsgBox sName & " mit Inhalt" & vbLf & _
         sInhalt & " erstellt!", vbOKOnly & vbInformation
   Else
      MsgBox sName & " nicht erstellt!", vbOKOnly & vbCritical
   End If
   On Error GoTo 0
End Sub
```

Code 3.24 Die Prozedur löscht einen benutzerdefinierten Eintrag im aktiven Dokument

```
Sub DeleteCustomProperty()
   Dim sName      As String

   sName = InputBox("Dokumentname = ", "LÖSCHVORGANG")
   On Error Resume Next
   ThisDocument.CustomDocumentProperties(sName).Delete
   If Err.Number = 0 Then
      MsgBox sName & " gelöscht!", vbOKOnly & vbInformation
   Else
      MsgBox sName & " nicht gelöscht!", vbOKOnly & vbCritical
   End If
   On Error GoTo 0
End Sub
```

Diese Prozeduren werden am häufigsten verwendet.

3.3 Word-Zeichen und ihre Sammlungen

3.3.1 Zeichen

So wie die *Cells* die kleinsten Elemente in Excel sind, sind es die Zeichen-Objekte *Characters* in Word. Es gibt kein einzelnes *Character*-Objekt in Word, sondern nur die Auflistung von Zeichen als Objektliste *Characters*.

```
'Syntax:
   Document.Characters [(Item)]

'Beispiele:
   Thisdocument.Characters(4)     'liefert den 4. Buchstaben
   Thisdocument.Characters.Count  'liefert die Anzahl Zeichen im
                                  'Dokument
   Thisdocument.Characters.First  'liefert erstes Zeichen als
                                  'Range-Objekt
   Thisdocument.Characters.Last   'liefert letztes Zeichen als
                                  'Range-Objekt
```

In der VBA-Hilfe befinden sich unter *Elemente des Characters-Objekts* eine Aufstellung aller Eigenschaften und der einzigen Methode *Item* der *Character*-Objektliste.

Während die Objektliste *Characters* nur die Methode *Item* besitzt, die ein Range-Objekt liefert, sind es auch nur sechs Eigenschaften die der Objektliste zur Verfügung stehen.

Erst mit der Anhäufung von Zeichen zu weiteren Word-Objekten kommen auch weitere Eigenschaften und Methoden hinzu. Abb. 3.6 zeigt die Objektstrukturen aus mengentheoretischer Sicht als Obermenge bzw. Teilmenge. Dabei kann ein Zeichen auch ein Objekt aus *Words*, *Sentences*, *Paragraphs*, *Sections* oder *Ranges* sein.

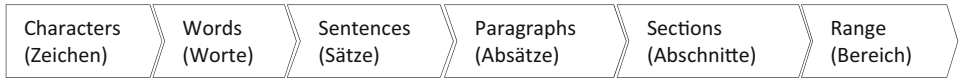


Abb. 3.6 Die Hierarchie der Zeichen-Objekte

3.3.2 Textbereiche

Das Bereichs-Objekt *Range* in Excel unterscheidet sich erheblich vom Bereichs-Objekt *Range* in Word. In Word versteht man unter Bereich eine zusammenhängende Folge von Zeichen, einschließlich der oft nicht sichtbaren Steuerzeichen.

```
'Syntax:
Document.Range [(Start:=Zahl, End:=Zahl)]

'Beispiel:
Thisdocument.Range (Start:=0, End:=10)
```

In der VBA-Hilfe stehen die Elemente des Word Range-Objekts.

Viele Word-Objekte verfügen über die Eigenschaft *Range*. Die Prozedur (Code 3.25) öffnet ein Dokument und weist seinen Inhalt einem Range-Objekt zu. Dadurch, dass die Parameter *Start* und *End* nicht angegeben werden, wird der gesamte Inhalt zugewiesen.

Code 3.25 Die Prozedur nutzt die Eigenschaft Characters eines Range Objekts

```
Sub DocuRange()
   Dim objDoc  As Document
   Dim rngTxt  As Range
   Dim sText   As String

   Set objDoc = Documents.Open("C:\Temp\Text1.docx")
   Set rngTxt = objDoc.Range
   With rngTxt
      sText = "Der Text in " & objDoc.Name & vbCrLf & _
         "beginnt mit dem Buchstaben " & .Characters(1) & vbCrLf & _
         "und besitzt " & .Characters.Count & " Zeichen."
      MsgBox sText
   End With
```

```
   objDoc.Close
   Set rngTxt = Nothing
   Set objDoc = Nothing
End Sub
```

Die Eigenschaft *Count* gehört grundsätzlich zu einer Objektliste und liefert in diesem Fall die Anzahl der Zeichen (Abb. 3.7).

Abb. 3.7 Ausgabe mit der Prozedur

Über die Eigenschaft *Text* des Bereichs-Objekts kann der Inhalt des Dokuments einer Textvariablen zugewiesen und so auf einfache Weise geändert werden (Code 3.26).

Code 3.26 Die Prozedur verändert den Inhalt eines Dokuments über sein Range-Objekt

```
Sub ChangeDocumentText()
   Dim objDoc  As Document
   Dim rngNew  As Range
   Dim sText   As String

   Set objDoc = Documents.Open("C:\Temp\Text1.docx")
   sText = objDoc.Range.Text
   sText = "Text vor Inhalt!" & vbCrLf & _
      sText & "Text nach Inhalt!"
   objDoc.Range.Text = sText
   objDoc.Close
   Set rngNew = Nothing
   Set objDoc = Nothing
End Sub
```

In der Prozedur (Code 3.27) wird im Range-Objekt des Dokumentinhalts als Unterobjekt ein Bereich definiert, der sich vor dem eigentlichen Text befindet, dessen Text gleich dem Nulltext entspricht. Erreicht wird dies dadurch, dass die Parameter *Start* und *End* auf Null gesetzt werden.

Code 3.27 Die Prozedur erzeugt einen neuen Bereich vor dem vorhandenen Dokumentinhalt

```
Sub InsertFirstRange()
   Dim objDoc  As Document
   Dim rngNew  As Range

   Set objDoc = Documents.Open("C:\Temp\Text1.docx")
   Set rngNew = objDoc.Range(Start:=0, End:=0)
   rngNew.Text = "Text vor dem Inhalt!"
   objDoc.Close
   Set rngNew = Nothing
   Set objDoc = Nothing
End Sub
```

Genauso lässt sich der Inhalt ein leeres Range-Objekt definieren (Code 3.28). Dabei muss beachtet werden, dass das letzte Zeichen ein Steuerzeichen ist, daher *objDoc.Range.End − 1*. Der Wert hätte sich auch aus der Zeichenanzahl des Dokuments − 1 bestimmen lassen.

Code 3.28 Die Prozedur erzeugt einen neuen Bereich nach dem vorhandenen Dokumentinhalt

```
Sub InsertLastRange()
   Dim objDoc  As Document
   Dim rngNew  As Range
   Dim lLast   As Long

   Set objDoc = Documents.Open("C:\Temp\Text1.docx")
   lLast = objDoc.Range.End - 1
   Set rngNew = objDoc.Range(Start:=lLast, End:=lLast)
   rngNew.Text = vbCrLf & "Text nach dem Inhalt!"
   objDoc.Close
   Set rngNew = Nothing
   Set objDoc = Nothing
End Sub
```

Range-Objekte und natürlich die darin definierten Range-Objekte können mit unterschiedlichen Eigenschaftswerten belegt werden (Code 3.29).

Code 3.29 Die Prozedur verändert die Zeichen-Eigenschaften vom 13. bis zum 50. Zeichen

```
Sub ChangeRange()
   Dim objDoc  As Document
   Dim rngTxt  As Range
```

```
   Set objDoc = Documents.Open("C:\Temp\Text1.docx")
   Set rngTxt = objDoc.Range(Start:=13, End:=50)
   With rngTxt
      .Bold = True
      .Italic = True
      With .Font
         .Name = "Times"
         .Size = 12
      End With
   End With
   objDoc.Close
   Set rngTxt = Nothing
   Set objDoc = Nothing
End Sub
```

3.3.3 Abschnitte

Ein Abschnitts-Objekt *Section* ist ein Objekt aus der Objektliste *Sections*. Über ihren Index kann auf die einzelnen Abschnitte zugegriffen werden. Unter einem Abschnitt ist der Bereich von Textbeginn bis zum ersten Umbruch, zwischen zwei Umbrüchen, bzw. vom letzten Umbruch bis zum Textende zu verstehen. Zu den Eigenschaften gehören auch Seiteneinstellungen, so wie Kopf- und Fußzeilen (Code 3.30).

Code 3.30 Die Prozedur setzt die Eigenschaften des ersten Abschnitts

```
Sub SectionAttributes()
   Dim objDoc  As Document
   Dim secDoc  As Section

   Set objDoc = Documents.Open("C:\Temp\Text1.docx")
   Set secDoc = objDoc.Sections(1)
   With secDoc
      With .PageSetup
      'setze Seitenränder
         .LeftMargin = InchesToPoints(0.5)
         .RightMargin = InchesToPoints(0.3)
      End With
      'Kopf- und Fusszeile der ersten Seite
      .Headers(wdHeaderFooterPrimary). _
         Range.Text = "Kopftext"
      .Footers(wdHeaderFooterPrimary). _
         Range.Text = "Fusstext"
      With .Range.Font
         .Name = "Times"
```

```
         .Size = 12
      End With
   End With
   objDoc.Close
   Set secDoc = Nothing
   Set objDoc = Nothing
End Sub
```

Über das Unterobjekt *Range* eines Abschnitt-Objekts können wieder die bereits besprochen Eigenschaften eines Range-Objektes angesprochen werden. Mit der Methode *Add* lassen sich neue Abschnitte erzeugen (Code 3.31).

Code 3.31 Die Prozedur fügt vor den Dokumentinhalt einen Seitenumbruch ein

```
Sub CreateSection()
   Dim objDoc  As Document
   Dim rngTxt  As Range

   Set objDoc = Documents.Open("C:\Temp\Text1.docx")
   Set rngTxt = objDoc.Range
   objDoc.Sections.Add Range:=rngTxt
   objDoc.Close
   Set rngTxt = Nothing
   Set objDoc = Nothing
End Sub
```

Die Methode *Add* verfügt über zwei Parameter. Wird ein Bereich angegeben, dann erfolgt der Umbruch vor diesem Bereich. Wird kein Bereich angegeben, dann erfolgt der Umbruch am Dokumentende. Unter Start wird die Umbruchsart mit einer *wdSectionStart*-Konstanten angegeben. Fehlt diese Angabe, dann erfolgt ein Seitenumbruch.

```
'Syntax:
   Sections.Add [(Range:=Bereich, Start:=wdSectionStart Konstante)]

'Beispiel:
   Thisdocument.Sections.Add
```

In der VBA-Hilfe unter dem Begriff *WdSectionStart-Enumeration* sind alle Konstanten aufgeführt.

3.3.4 Absätze

Das Absatz-Objekt *Paragraph* bilden einen zusammenhängenden Text aus Zeichen, Worten und Sätzen. In einem größeren Text werden Absätze durch das Steuerzeichen *vbCrLf*

(Carriage Return Line Feed) getrennt. Auch Absätze sind eine Aufzählung und werden ebenfalls durch einen Index, in diesem Fall die Absatznummer identifiziert.

Die Prozedur (Code 3.32) ermittelt alle Absätze in einem Dokument und zeigt den ersten Buchstaben in einem Dialogfenster.

```
'Syntax:
   Paragraphs (Index)

'Beispiel:
   Thisdocument.Paragraphs(3) 'dritter Absatz des aktuellen Dokuments
```

Code 3.32 Die Prozedur liest alle Absätze eines Dokuments und zeigt den ersten Buchstaben

```
Sub ReadAllParagraphs()
   Dim objDoc  As Document
   Dim parDoc  As Paragraph

   Set objDoc = Documents.Open("C:\Temp\Text1.docx")
   For Each parDoc In objDoc.Paragraphs
      MsgBox parDoc.Range.Characters(1)
   Next
   objDoc.Close
   Set parDoc = Nothing
   Set objDoc = Nothing
End Sub
```

Absätze verfügen wie alle Zeichen-Sammlungen über ähnliche Eigenschaften und Methoden. Dazu kommen dann gestalterische Elemente (Code 3.33).

Code 3.33 Die Prozedur zentriert den ersten Absatz eines Dokuments

```
Sub SetParagraphs()
   Dim objDoc  As Document
   Dim parDoc  As Paragraph

   Set objDoc = Documents.Open("C:\Temp\Text1.docx")
   With objDoc.Paragraphs(1)
   'Textausrichtung
      .Alignment = wdAlignParagraphCenter
   'Abstände mit Zeilenangaben
      .SpaceAfter = LinesToPoints(2)
      .SpaceBefore = LinesToPoints(1)
   End With
```

```
   objDoc.Close
   Set parDoc = Nothing
   Set objDoc = Nothing
End Sub
```

Textergänzungen und Absätze lassen sich mit den Methoden *InsertBefore* und *InsertAfter* gestalten (Code 3.34).

Code 3.34 Die Prozedur erzeugt neue Absätze

```
Sub InsertParagraphs()
   Dim objDoc  As Document
   Dim parDoc  As Paragraph

   Set objDoc = Documents.Open("C:\Temp\Text1.docx")
   objDoc.Paragraphs(1).Range.InsertAfter _
      vbCrLf & "Dieser Satz wurde mit InsertAfter erstellt!"
   objDoc.Paragraphs(2).Range.InsertBefore _
      "Dieser Satz wurde mit InsertBefore erstellt!" & vbCrLf

   objDoc.Close
   Set parDoc = Nothing
   Set objDoc = Nothing
End Sub
```

Einen neuen Absatz erzeugt die Methode *TypeParagraph* und die Methode *InsertBreak* erzeugt einen Seitenwechsel mit Absatzwechsel.

3.3.5 Sätze

Satz-Objekte *Sentences* bilden ein Unterobjekt von Absätzen, Abschnitten und Bereichen. Die Prozedur (Code 3.35) liest alle Sätze des aktuellen Dokuments und gibt den ersten Buchstaben aus.

Code 3.35 Die Prozedur liest alle Sätze eines Dokuments

```
Sub ReadAllSentences()
   Dim objDoc  As Document
   Dim senDoc  As Range

   Set objDoc = Documents.Open("C:\Temp\Text1.docx")
   For Each senDoc In objDoc.Sentences
      MsgBox senDoc.Characters(1)
   Next
```

```
   MsgBox "Anzahl Sätze: " & objDoc.Sentences.Count
   objDoc.Close
   Set senDoc = Nothing
   Set objDoc = Nothing
End Sub
```

3.3.6 Worte

Die kleinste sinnvolle Zusammenfassung von Zeichen ist das Wort-Objekt *Word*. Worte werden ebenso als Objektliste *Words* einer Sammlung verwaltet. Die Prozedur (Code 3.36) gibt Informationen über die Wörter des aktuellen Dokuments aus.

Code 3.36 Die Prozedur schreibt Informationen über Words ins Direktfenster

```
Sub ReadAllWords()
   Dim objDoc  As Document
   Dim wrdDoc  As Range

   Set objDoc = Documents.Open("C:\Temp\Text1.docx")
   For Each wrdDoc In objDoc.Words
      Debug.Print wrdDoc.Characters(1);
   Next
   Debug.Print "Anzahl Wörter: " & objDoc.Words.Count
   Debug.Print "Letztes Wort: " & objDoc.Words.First
   Debug.Print "Letztes Wort: " & objDoc.Words(objDoc.Words.Count - 2)
   objDoc.Close
   Set wrdDoc = Nothing
   Set objDoc = Nothing
End Sub
```

Da das letzte Steuerzeichen und der letzte Punkt auch als Wörter mitgezählt werden, bestimmt sich der Index des wirklich letzten Wortes aus (Count − 2). Auch bei Wörtern finden sich viele Eigenschaften und Methoden aus dem Bereich wieder. Interessant beim Einfügen von Wörtern ist, dass die vorhandenen Wörter als letztes Zeichen das Leerzeichen besitzen (Code 3.37).

Code 3.37 Die Prozedur ändert die Wortdarstellung und fügt neue Worte ein

```
Sub ChangeWords()
   Dim objDoc  As Document
   Dim rngDoc  As Range

   Set objDoc = Documents.Open("C:\Temp\Text1.docx")
   Set rngDoc = objDoc.Range
```

```
   With rngDoc.Words(6)
      .Bold = True
      With .Font
         .Name = "Times"
         .Size = 12
         .Italic = True
      End With
      .InsertAfter "und Textverwaltung "
   End With
   objDoc.Close
   Set rngDoc = Nothing
   Set objDoc = Nothing
End Sub
```

3.3.7 Textmarkierungen

Auch die Word-Objektstruktur verfügt über einen markierten Bereich, das Auswahl-Objekt *Selection*, mit ähnlichen Eigenschaften und Methoden wie das *Range*-Objekt. Die Prozedur (Code 3.38) markiert erst das erste Wort des Dokuments und setzt bei dem Auswahl-Objekt die Eigenschaft *Bold* auf *True* (Fettschrift). Anschließend wird die Auswahl auf den ganzen Inhalt (*WholeStory*) erweitert. Danach wird der ganze Text auf Schrägschrift (Italic) gesetzt.

Code 3.38 Die Prozedur arbeitet mit dem Selection-Objekt

```
Sub CreateSelection()
   Dim objDoc  As Document
   Dim selDoc  As Selection

   Set objDoc = Documents.Open("C:\Temp\Text1.docx")
   Set selDoc = objDoc.Words(1)
   With selDoc
      .Bold = True
      .WholeStory
      .Italic = True
   End With
   objDoc.Close
   Set selDoc = Nothing
   Set objDoc = Nothing
End Sub
```

Nun folgt eine Prozedur (Code 3.39), die zuerst den Inhalt markiert, dann die Markierung an das Ende des Inhalts setzt und von dort zwei Zeilen rückwärts geht. Dies kann im Debug-Mode gut überprüft werden.

Code 3.39 Arbeiten mit dem Selection-Objekt

```
Sub SelectionRange()
   Dim objDoc  As Document

   Set objDoc = Documents.Open("C:\Temp\Text1.docx")
   objDoc.Range.Select
   With Selection
      .EndOf Unit:=wdStory, Extend:=wdMove
      .HomeKey Unit:=wdLine, Extend:=wdExtend
      .MoveUp Unit:=wdLine, Count:=2, Extend:=wdExtend
   End With
   objDoc.Close
   Set objDoc = Nothing
End Sub
```

Eine der wichtigsten Eigenschaften des Auswahl-Objekts ist *Text*. Mit dieser Eigenschaft kann der Inhalt des markierten Bereichs an eine Textvariable zur weiteren Verarbeitung (Code 3.40) übergeben werden.

Code 3.40 Arbeiten mit der Text-Eigenschaft

```
Sub ReadSelectionText()
   Dim objDoc  As Document
   Dim sText   As String

   Set objDoc = Documents.Open("C:\Temp\Text1.docx")
   sText = objDoc.Range.Text
   sText = Left(sText, 50)
   objDoc.Close
   Set objDoc = Nothing
End Sub
```

Aber es kann auch über die Eigenschaft *Text* des Auswahl-Objekts ein neuer Text hinzugefügt werden (Code 3.41).

Code 3.41 Die Text-Eigenschaft des Selection-Objekts

```
Sub NewText()
   Dim iCount As Integer
   Documents.Add
   For iCount = 1 To 6
      Selection.Text = "Zeile " & Str(iCount) & vbCrLf
      Selection.MoveDown Unit:=wdParagraph, Count:=1
   Next iCount
End Sub
```

Ja sogar mit der Methode *Calculate* kann eine Formel im markierten Text ausgewertet werden (Code 3.42). Ein Beispiel ist in Abb. 3.8 zu sehen.

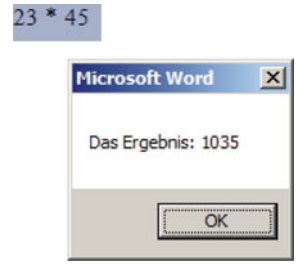


Abb. 3.8 Ausgabe einer Beispiel-Kalkulation

Code 3.42 Die Prozedur berechnet die markierte Formel im Text

```
Sub CalcSelection()
   Dim objDoc As Document
   Set objDoc = New Document
   With objDoc.Range
      .Text = "23 * 45"
      .Select
   End With
   MsgBox "Das Ergebnis: " & Selection.Calculate
End Sub
```

Die Prozedur (Code 3.43) markiert das erste Wort des Textinhalts. Erweitert dann die Markierung auf den Absatz und kopiert den so markierten Text in ein neues Dokument.

Code 3.43 Die Prozedur kopiert markierten Text in ein neues Dokument

```
Sub CopyPasteSeletion()
   Dim objDoc  As Document

   Set objDoc = Documents.Open("C:\Temp\Text1.docx")
   objDoc.Words(1).Select
   With objDoc.ActiveWindow.Selection
      .EndOf Unit:=wdParagraph, Extend:=wdExtend
      If Selection.Type = wdSelectionNormal Then
         .Copy
         Documents.Add.Content.Paste
      End If
   End With
   objDoc.Close
   Set objDoc = Nothing
End Sub
```

Zur Auflösung einer Markierung kann die Methode *Collapse* verwendet werden.

```
'Syntax:
   Selection.Collapse [Direction]

'Beispiel:
   Selection.Collapse Direction:=wdCollapseEnd
```

Für den Parameter *Direction* können die Konstanten *wdCollapseEnd* oder *wdCollapse Start* angegeben werden (Code 3.44). Informationen liefert die VBA-Hilfe.

Code 3.44 Die Prozedur zeigt die Wirkung der Methode Collapse

```
Sub CollapseSelection()
   Dim objDoc  As Document

   Set objDoc = Documents.Open("C:\Temp\Text1.docx")
   objDoc.Range.Select
   Selection.Collapse Direction:=wdCollapseEnd
   objDoc.Close
   Set objDoc = Nothing
End Sub
```

3.3.8 Tabellen

Ein Tabellen-Objekt *Table* in Word ist ein Objekt im Dokument mit einer bestimmten Anzahl Zeilen und Spalten. Tabellen werden in der Objektliste *Tables* verwaltet. Die Prozedur (Code 3.45) löscht alle Tabellen im aktiven Dokument.

Code 3.45 Die Prozedur löscht alle Tabellen des aktiven Dokuments

```
Sub DeleteAllTables()
   Dim tblDaten   As Table
   For Each tblDaten In ActiveDocument.Tables
      tblDaten.Delete
   Next
   Set tblDaten = Nothing
End Sub
```

Der Tabellen-Objektliste einer Texteinheit kann mit der *Add*-Methode eine neue Tabelle hinzugefügt werden. Im Beispiel (Code 3.46) wird die Tabelle am Anfang des Dokuments erzeugt und besitzt 4 Zeilen und 3 Spalten. Die automatische Anpassung an das Dokument wird aktiviert. Das Format der Tabelle ist das der *Formatvorlage1*. Danach werden alle

Zellen der Tabelle mit Pseudozufallszahlen gefüllt. Am Ende erfolgt eine Autoanpassung der Spaltenbreiten an den Inhalt.

Code 3.46 Die Prozedur erzeugt eine neue Tabelle mit Zufallszahlen

```
Sub CreateTable()
   Dim rngDoc     As Range
   Dim tblDaten   As Table
   Dim lRow       As Long
   Dim lCol       As Long
   Dim dZahl      As Double

   Randomize
   dZahl = Timer
   Set rngDoc = ActiveDocument.Range(Start:=0, End:=0)
   Set tblDaten = ActiveDocument.Tables.Add _
      (Range:=rngDoc, NumRows:=4, NumColumns:=3, _
      DefaultTableBehavior:=wdWord9TableBehavior)
   With tblDaten
      .Style = "Formatvorlage1"
      For lRow = 1 To 4
         For lCol = 1 To 3
            '.Cell(lRow, lCol).Range.InsertAfter Rnd(dZahl)
            .Cell(lRow, lCol).Range = Rnd(dZahl)
         Next lCol
      Next lRow
      .Columns.AutoFit
   End With
   Set tblDaten = Nothing
   Set rngDoc = Nothing
End Sub
```

Die Prozedur (Code 3.47) verändert verschiedene Eigenschaften einer Tabelle und ihrer Unterobjekte.

Vordefinierte Tabellen-Formate können in der VBA-Hilfe unter dem Begriff *WdTable-Format-Enumeration* aufgerufen werden.

Code 3.47 Die Prozedur setzt einige Eigenschaften der ersten Tabelle

```
Sub FormatTable()
   Dim rngDoc     As Range
   Dim tblDaten   As Table

   Set rngDoc = ActiveDocument.Range(Start:=0, End:=0)
   Set tblDaten = rngDoc.Tables(1)
   With tblDaten
```

```
      .AutoFormat Format:=wdTableFormatElegant
      With .Range.Font
         .Name = "Arial"
         .Size = 10
      End With
      .Rows(1).Borders.OutsideLineStyle = wdLineStyleTriple
      .Rows(1).Range.Font.Bold = True
      .Columns(2).Shading.BackgroundPatternColorIndex = wdGray25
   End With
   Set tblDaten = Nothing
   Set rngDoc = Nothing
End Sub
```

In der Prozedur (Code 3.47) wurde die entsprechende Tabelle unter ihrer Indexnummer angesprochen. Doch nicht immer ist die bekannt. Hilfreich ist die Verwendung der Eigenschaft *Title*. Die Prozedur (Code 3.48) erzeugt eine bereits bekannte Tabelle und sucht sie dann in der Objektliste zum Löschen.

Code 3.48 Die Prozedur erzeugt und löscht eine Tabelle mit Titel

```
Sub FindAndDeleteTable()
   Dim rngDoc     As Range
   Dim tblDaten   As Table
   Dim lRow       As Long
   Dim lCol       As Long
   Dim dZahl      As Double

'erzeugt Tabelle
   Randomize
   dZahl = Timer
   Set rngDoc = ActiveDocument.Range
   Set tblDaten = ActiveDocument.Tables.Add(rngDoc, 4, 3)
   With tblDaten
      .Title = "ZufallsZahlen"
      .Style = "Formatvorlage1"
      For lRow = 1 To 4
         For lCol = 1 To 3
            .Cell(lRow, lCol).Range = Rnd(dZahl)
         Next lCol
      Next lRow
      .Columns.AutoFit
   End With
'finde Tabelle zum löschen
   For Each tblDaten In rngDoc.Tables
      If tblDaten.Title = "ZufallsZahlen" Then
```

```
            tblDaten.Delete
         End If
      Next

      Set tblDaten = Nothing
      Set rngDoc = Nothing
End Sub
```

Zur Veränderung einer Tabelle können die Methoden *Add* und *Delete* auch auf Zeilen und Spalten angewendet werden. Die Prozedur (Code 3.49) fügt in der ersten Tabelle des aktiven Dokuments vor die dritte Spalte eine neue Spalte ein. Danach wird die zweite Zeile gelöscht.

Code 3.49 Die Prozedur fügt eine Spalte ein und löscht eine Zeile

```
Sub ChangeTableFormat()
   Dim rngDoc     As Range
   Dim tblDaten   As Table

   Set rngDoc = ActiveDocument.Range
   Set tblDaten = ActiveDocument.Tables(1)
   tblDaten.Columns.Add _
      BeforeColumn:=tblDaten.Columns(3)
   tblDaten.Rows(2).Delete
End Sub
```

3.3.9 ABC-Analyse

Die ABC-Analyse ist eine betriebswirtschaftliche Methode zur Einteilung von Objekten in drei Kategorien. Sie dient zur Planungs- und Entscheidungsfindung. Die Objekte werden durch relevante Kenngrößen beschrieben und danach sortiert. Das wichtigste Element steht an erster Stelle (Element der Kategorie A) und das unwichtigste Element steht an letzter Stelle (Element der Kategorie C). Alle Elemente werden nun einer der drei Kategorien zugeordnet.

A – sehr wichtig
B – weniger wichtig
C – unwichtig

Die Grenzen zwischen den Kategorien werden nach Anwendungsfall und Zielsetzung festgelegt (Tab. 3.1). Sie orientieren sich meist an den Grundlagen des sogenannten Pareto-Prinzips, auch 80/20-Regel genannt.

Tab. 3.1 Typische Grenzwerte für eine ABC-Analyse

Kategorie	Mengenanteil [%]	Wertanteil [%]
A	10–20	65–85
B	20–50	15–35
C	50–80	10–20

Die ABC-Analyse bringt nur dann gute Ergebnisse, wenn sich die Merkmalswerte wie Mengen oder Beträge deutlich unterscheiden. Oft liegen viele sehr unterschiedliche Merkmale vor, so dass erst mit Hilfe der ABC-Analyse eine Kategorisierung möglich ist. Die Flexibilität der ABC-Analyse zeigt sich in den vielen Anwendungsgebieten. So wird sie beim Zeitmanagement, im Projektmanagement, bei der Lagerplanung, beim Marketing, bei der Betriebsanalyse, bei der Standortbestimmung, bei der Materialflussplanung, bei der Qualitätssicherung, etc. eingesetzt, um nur einige zu nennen.

Der Vorteile der ABC-Analyse ist, dass Wesentliches vom Unwesentlichen getrennt wird. Sie erlaubt die Identifizierung von Ansatzpunkten zur Verbesserung mit vertretbarem Aufwand. Unwirtschaftliche Anstrengungen werden vermieden. Sie liefert eine übersichtliche und grafische Darstellung der Ergebnisse.

Aber auch die Nachteile sollen erwähnt werden. So sind drei Klassen sehr grob für eine Klassifizierung. Die Festlegung der Grenzwerte beruht auf Erfahrung und ist daher willkürlich. Die ABC-Analyse bietet nur die Aufnahme des Istzustandes.

Unser Modell besteht aus einem Word-Dokument mit einer Tabelle von zwanzig Kunden mit Bestellmengen und Umsatzwerten. Eine weitere Tabelle enthält die Grenzwerte der Kategorien (Abb. 3.9). Eine Analyse soll ein ABC-Diagramm und eine Lorenzkurve ausgeben.

Abb. 3.9 Beispiel-Tabellen (Ausschnitt aus der Umsatztabelle)

Kategorie	Mengenanteile [%]	Wertanteile [%]
A	15	70
B	25	20
C	60	10

Firma	Bestellmenge [Stück]	Umsatz [Euro]
Becker	92	121800
Meyer	56	60400
Schulz	33	21700
Müller	45	35200
Schuster	38	17300

Den Algorithmus gibt der Pseudocode (Code 3.50) wieder.

Code 3.50 Pseudocode zur ABC-Analyse

```
Lösche vorhandene Diagramme im Dokument
Stelle Excel-Anwendung in den Hintergrund
Kopiere Dokument-Tabellen in Excel-Tabellen
Sortiere Kunden-Umsätze nach Umsatz
Bestimme Gesamtumsatz
For alle Kunden
   Kumuliere Umsatz
   Setze kumulierten Umsatz in Spalte A, B oder C
   Setze kumulierten Anteil in Spalte A, B oder C
Next
Erstelle ABC-Diagramm
Kopiere Diagramm über die Zwischenablage
Erstelle Lorenz-Diagramm
Kopiere Diagramm über die Zwischenablage
Schließe Excel-Anwendung
```

Die Analyse der Daten wir in der Excel-Anwendung durchgeführt. Das Ergebnis sind zwei Diagramme, die automatisch ins Dokument übernommen werden. Abb. 3.10 zeigt das ABC-Diagramm und Abb. 3.11 das Lorenzdiagramm. Beide Diagramme sind sich ähnlich, lediglich ihre Ordinaten weisen unterschiedliche Werte auf.

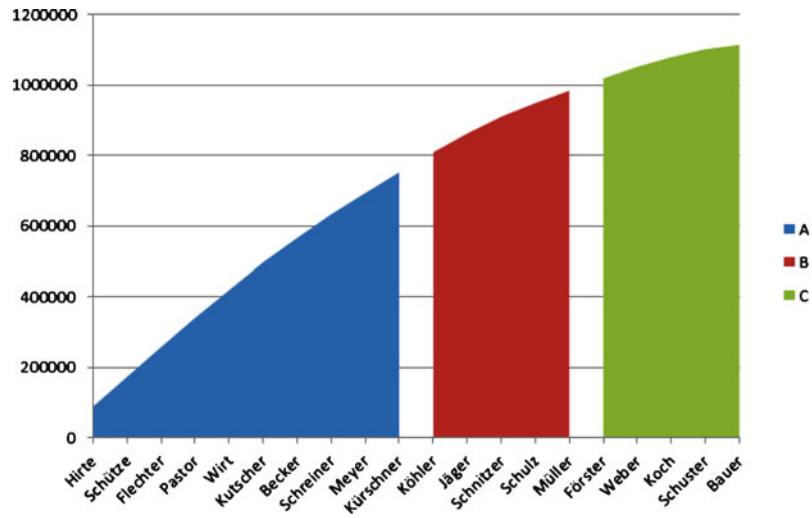


Abb. 3.10 ABC-Diagramm zum Beispiel

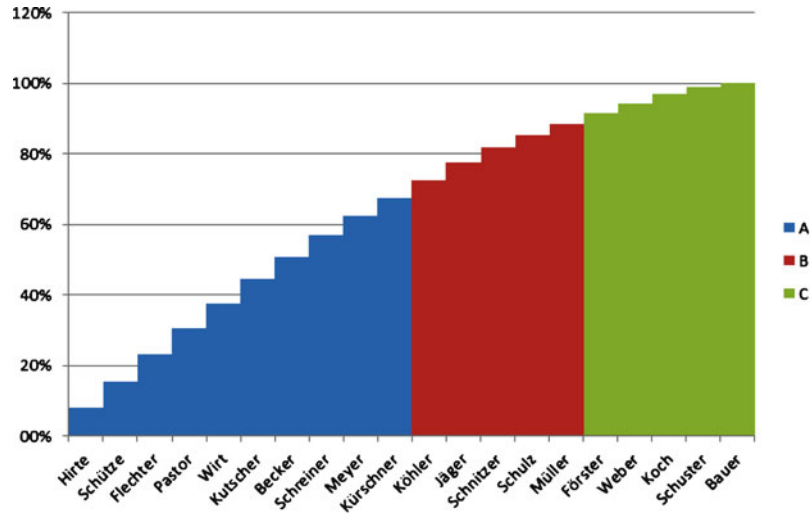


Abb. 3.11 Lorenz-Diagramm zum Beispiel

3.4 Word-Texthilfen

3.4.1 Textpositionen

Um eine einzelne Textmarke im Dokument zu positionieren, müssen wir zuvor Methoden kennenlernen, die den Cursor im Dokument bewegen. Betrachten wir die Methode *Goto*, die sowohl auf ein Dokument, einen Bereich oder einen markierten Bereich angewendet werden kann.

```
'Syntax:
   Selection.GoTo [What:=Element, Which:=Richtung, _
      Count:=Zähler, Name:=Name]

'Beispiel:
   Selection.Goto What:=wdGoToHeading
```

Der Parameter *What* (Anhang 3, Tab. A3.2) beschreibt das Objekt der Bewegung, während der Parameter *Which* (Anhang 3, Tab. A3.3) die Richtung vorgibt. Für beide gibt es in den Tabellen Konstante, die alle mit der Präfix *wdGoTo* beginnen. Die Werte sind für den Fall vorgegeben, dass die Konstanten in der Programmierung bei Objekten mit später Bindung eingesetzt werden.

Die Prozedur (Code 3.51) demonstriert die Anwendung der Methode *GoTo*.

Code 3.51 Cursorbewegungen

```
Sub GoToCursor()
   Dim objDoc     As Document

   Set objDoc = Documents.Open("C:\Temp\Text1.docx")
   objDoc.Range.Select 'markiert den gesamten Textinhalt
   'setzt Cursor an den Anfang der ersten Linie
   Selection.GoTo what:=wdGoToLine, which:=wdGoToFirst
   'setzt Cursor zwei Zeilen weiter
   Selection.GoTo what:=wdGoToLine, which:=wdGoToNext, Count:=2
   'setzt Cursor eine Zeile zurück
   Selection.GoTo what:=wdGoToLine, which:=wdGoToPrevious

   objDoc.Close
   Set objDoc = Nothing
End Sub
```

Eine weitere Möglichkeit bieten die Methoden *HomeKey* und *EndKey*. Sie stehen für die Tasten *Pos1* und *Ende* und ihre Anwendung auf eine Texteinheit. Die *Unit*-Parameter stehen im Anhang 3, Tab. A3.4.

```
'Syntax:
   Selection.HomeKey [Unit:=Texteinheit, Extend]
   Selection.EndKey [Unit:=Texteinheit, Extend]

'Beispiel:
   Selection.HomeKey Unit:=wdStory
```

Der Parameter *Extend* kann auf die Werte *wdMove* oder *wdExtend* gesetzt werden. Mit *wdMove* wird lediglich der Cursor gesetzt, ohne dass eine Markierung erfolgt. Mit *wdExtend* erfolgt auch eine Markierung (Code 3.52).

Code 3.52 Tastenbewegungen

```
Sub KeyPosCursor()
   Dim objDoc     As Document

   Set objDoc = Documents.Open("C:\Temp\Text1.docx")
   objDoc.Range.Select 'markiert den gesamten Textinhalt
   'setzt Cursor an den Anfang des Textes
   Selection.HomeKey Unit:=wdStory, Extend:=wdMove
   'setzt Cursor ans Ende der ersten Zeile
   Selection.EndKey Unit:=wdLine, Extend:=wdMove
   'setzt Cursor ans Ende des Textes
   Selection.EndKey Unit, Count, Extend]
```

```
   objDoc.Close
   Set objDoc = Nothing
End Sub
```

Der Cursor lässt sich auch von seiner Position aus nach oben, unten, links oder rechts verschieben mit den Methoden *MoveUp*, *MoveDown*, *MoveLeft* und *MoveRight*.

```
'Syntax:
   Selection.MoveX [Unit:=Einheit, Extend:=Extend]

'Beispiel:
   Selection.MoveDown Unit:=wdLine
```

Die Unit-Parameter für die Methoden *MoveUp* und *MoveDown* stehen im Anhang 3, Tab. A3.5. Für die Methoden *MoveLeft* und *MoveRight* stehen sie im Anhang 3, Tab. A3.6.

Die Prozedur (Code 3.53) demonstriert eine Verschiebung mit und ohne Markierung. Die umfangreiche Kommentierung beschreibt die Wirkung der Anwendungen.

Code 3.53 Verschiebung des Cursors ohne und mit Markierung

```
Sub MoveCursor()
   Dim objDoc     As Document

   Set objDoc = Documents.Open("C:\Temp\Text1.docx")
   objDoc.Range.Select 'markiert den gesamten Textinhalt
   'setzt Cursor an den Anfang der ersten Linie
   Selection.GoTo what:=wdGoToLine, which:=wdGoToFirst
   'verschiebt den Cursor an den Anfang des dritten Wortes
   Selection.MoveRight Unit:=wdWord, Count:=3, Extend:=wdMove
   'Verschiebt den Cursor zwei Zeilen nach unten
   'damit ist auch der Bereich markiert
   Selection.MoveDown Unit:=wdLine, Count:=2, Extend:=wdExtend
   'setzt Cursor zwei Worte zurück
   'damit ändert sich auch der markierte Bereich
   Selection.MoveLeft Unit:=wdWord, Count:=1, Extend:=wdExtend
   objDoc.Close
   Set objDoc = Nothing
End Sub
```

Weitere Methoden sind *StartOf* und *EndOf*, die sich immer auf den Anfang oder das Ende einer Texteinheit beziehen.

```
'Syntax:
   Selection.StartOf [Unit:=Texteinheit, Extend]
   Selection.EndOf [Unit:=Texteinheit, Extend]
```

```
'Beispiel:
   Selection.StartOf Unit:=wdStory
```

Die Prozedur (Code 3.54) zeigt deren Anwendung. Die Parameter sind oben bereits beschrieben.

Code 3.54 Setzen des Cursors mit StartOf und EndOf

```
Sub StartEndCursor()
   Dim objDoc     As Document

   Set objDoc = Documents.Open("C:\Temp\Text1.docx")
   'setzt den Cursor an den Anfang der Story
   Selection.StartOf Unit:=wdStory, Extend:=wdMove
   'setzt den Cursor an das Ende des ersten Satzes
   'und markiert den ersten Satz
   Selection.EndOf Unit:=wdSentence, Extend:=wdExtend

   objDoc.Close
   Set objDoc = Nothing
End Sub
```

3.4.2 Textmarken

Neben einem markierten Text können auch Textmarken-Objekte *Bookmarks* zur Markierung einer Textposition verwendet werden. Eine Textmarke kann dabei nur eine Position oder auch ein markierter Bereich sein. Die Prozedur (Code 3.55) ordnet einem Range-Objekt den Dokument-Inhalt zu. Markiert danach das erste Wort, das in meinem Übungstext *Word* lautet. Erstellt eine Textmarke mit dem Namen *Ersetze* für den markierten Bereich und weist danach der Eigenschaft *Text* des Range-Objekts der Textmarke einen anderen Text zu.

Code 3.55 Textänderung an einer Textmarke

```
Sub CreateChangeBookmark()
   Dim objDoc     As Document
   Dim rngDoc     As Range

   Set objDoc = Documents.Open("C:\Temp\Text1.docx")
   Set rngDoc = objDoc.Range
   rngDoc.Words(1).Select
   With objDoc
      .Bookmarks.Add Name:="Ersetze", Range:=Selection.Range
      .Bookmarks("Ersetze").Range.Text = _
```

```
         "MSWord (ein eingetragenes Warenzeichen von Microsoft) "
      .Close
   End With
   Set rngDoc = Nothing
   Set objDoc = Nothing
End Sub
```

Anders verläuft es bei der Prozedur (Code 3.56). Da wird zunächst im gleichen Text eine Textmarke an eine bestimmte Position im Text kreiert und dann an dieser Stelle weiterer Text eingefügt.

Code 3.56 Texteinfügung an einer Textmarke

```
Sub CreateInsertBookmark()
   Dim objDoc     As Document
   Dim rngDoc     As Range

   Set objDoc = Documents.Open("C:\Temp\Text1.docx")
   objDoc.Range.Select
   With Selection
      .StartOf Unit:=wdStory
      .MoveRight Unit:=wdSentence, Count:=1, Extend:=wdMove
      .Bookmarks.Add Name:="Ersetze", Range:=Selection.Range
      .Bookmarks("Ersetze").Range.Text = _
         "Aber Word kann noch viel mehr! "
   End With
   objDoc.Close
   Set rngDoc = Nothing
   Set objDoc = Nothing
End Sub
```

Textmarken können über die Objektliste angesprochen und auch gelöscht werden (Code 3.57).

Code 3.57 Die Prozedur liest alle Textmarken im Dokument und löscht eine bestimmte

```
Sub DeleteBookmark()
   Dim objDoc     As Document
   Dim bmkDoc     As Bookmark

   Set objDoc = Documents.Open("C:\Temp\Text1.docx")
   For Each bmkDoc In objDoc.Bookmarks
      If bmkDoc.Name = "Ersetze" Then
         bmkDoc.Delete
      End If
```

```
   Next
   objDoc.Close
   Set bmkDoc = Nothing
   Set objDoc = Nothing
End Sub
```

3.4.3 Textseiten

Mit Hilfe der Methode *PageSetup* lassen sich Eigenschaften einer Textseite formatieren (Code 3.58). Manche Eigenschaften verwenden als Werte *Point*-Angaben. Die sind nicht immer verständlich und darum bietet VBA einige Umrechnungsmethoden an (Anhang 3, Tab. A3.7).

Code 3.58 Die Prozedur setzt Seitenränder und Absatzabstände

```
Sub SetPageParameter()
   Dim objDoc As Document

   Set objDoc = Documents.Open("C:\Temp\Text1.docx")
   'Seitenränder
   With objDoc.PageSetup
      .LeftMargin = CentimetersToPoints(5)
      .RightMargin = CentimetersToPoints(3)
      .TopMargin = CentimetersToPoints(2)
      .BottomMargin = CentimetersToPoints(3)
   End With
   'Absatzabstände
   With objDoc.Paragraphs
      .SpaceBefore = LinesToPoints(1)
      .SpaceAfter = LinesToPoints(2)
   End With
   objDoc.Close
   Set objDoc = Nothing
End Sub
```

3.4.4 Fensterteilung

Ein Fensterausschnitt-Objekt *Pane* ist ein Objekt, das als Objektliste *Panes* vertreten ist und die Aufteilung der Textdarstellung in einem Fenster verwaltet. Fensterausschnitte werden über das aktive Fenster eines aktiven Dokuments angesprochen und mit der Methode *Add* erzeugt (Code 3.59).

Code 3.59 Die Prozedur erzeugt eine Trennlinie und löscht sie wieder

```
Sub ActivePanes()
   Dim objDoc As Document

   Set objDoc = Documents.Open("C:\Temp\Text1.docx")
   'erzeugt eine Trennlinie bei 25% vertikal
   objDoc.ActiveWindow.Panes.Add SplitVertical:=25
   'löscht die Trennlinie
   objDoc.ActiveWindow.ActivePane.Close

   objDoc.Close
   Set objDoc = Nothing
End Sub
```

Aber auch weitere Fensteranteile lassen sich aufrufen. So erzeugt die Prozedur (Code 3.60) einen Fensterausschnitt für vorhandene Kommentare. Auch Ausschnitte für den Kopf- und Fußbereich sind aufrufbar.

Code 3.60 Die Prozedur erstellt einen Fensterausschnitt für vorhandene Kommentare

```
Sub CommentPane()
   Dim objDoc As Document
   Set objDoc = Documents.Open("C:\Temp\Text1.docx")
   objDoc.ActiveWindow.View.Type = wdNormalView
   If objDoc.Comments.Count >= 1 Then
      objDoc.ActiveWindow.View.SplitSpecial = wdPaneComments
      objDoc.ActiveWindow.ActivePane.Close
   End If
   objDoc.Close
   Set objDoc = Nothing
End Sub
```

3.4.5 Tabulatoren

Die Tabulatoren-Objekte *TabStops* sind jedem hinlänglich bekannt. Sie werden in Word auf dem Zeilenlineal vermerkt, so dass wir zur Kontrolle im aktiven Fenster die Eigenschaft *DisplayRulers* auf *True* setzen müssen. Mit der Methode *ClearAll* können alle vorhandenen TabStops auf der Ebene der Absätze gelöscht werden (Code 3.61). Danach werden zwei neue Tabulatoren definiert.

Code 3.61 Die Prozedur zeigt den Umgang mit Tabulatoren

```
Sub SetTabStops()
   Dim objDoc As Document
   Set objDoc = Documents.Open("C:\Temp\Text1.docx")
   'blendet Lineale ein
   objDoc.ActiveWindow.ActivePane.DisplayRulers = True
   'löscht vorhandene Tabs
   objDoc.Paragraphs.TabStops.ClearAll
   'legt zwei neue Tabs an
   With objDoc.Paragraphs.TabStops
      .Add Position:=CentimetersToPoints(6), _
         Alignment:=wdAlignTabLeft, _
         Leader:=wdTabLeaderDots
      .Add Position:=CentimetersToPoints(8), _
         Alignment:=wdAlignTabCenter
   End With
   objDoc.Close
   Set objDoc = Nothing
End Sub
```

Eine Übersicht der verschiedenen Tabulatorarten zeigt Anhang 3, Tab. A3.8.

3.4.6 Sonderzeichen

Mit der Methode InsertSymbol können Sonderzeichen im Text eingefügt werden. Allerdings muss das Zeichen bekannt sein, d. h. in welcher Schriftart es welchen Zeichencode besitzt. Im Menüband unter *Einfügen / Symbol* öffnet sich ein Dialogfenster (Abb. 3.12) zur Bestimmung.

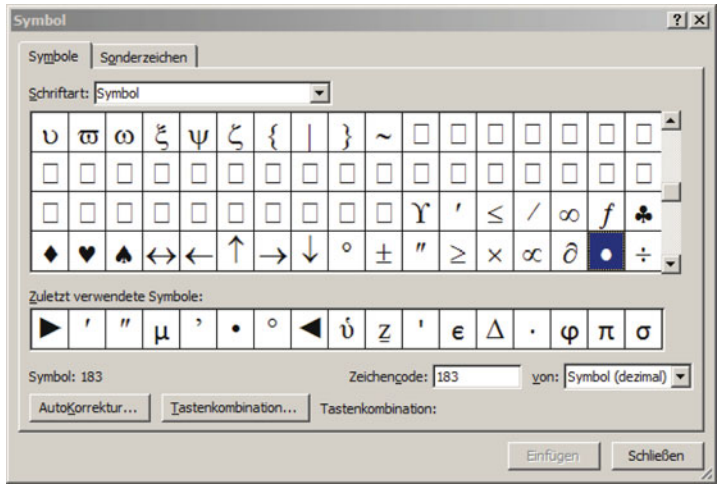


Abb. 3.12 Dialogfenster zum Einfügen von Symbolen

Die Prozedur (Code 3.62) zeigt den Umgang mit Sonderzeichen.

Code 3.62 Die Prozedur fügt vor dem Text ein Sonderzeichen ein

```
Sub SymbolInsert()
   Dim objDoc As Document

   Set objDoc = Documents.Open("C:\Temp\Text1.docx")
   objDoc.Range.Select 'markiert den gesamten Textinhalt
   'setzt Cursor an den Anfang der ersten Linie
   Selection.GoTo What:=wdGoToLine, Which:=wdGoToFirst
   'fügt das Symbol 183 ein
   Selection.InsertSymbol _
      characterNumber:=183, Font:="Symbol", Unicode:=False
   objDoc.Close
   Set objDoc = Nothing
End Sub
```

3.4.7 Listenformate

Für Texteinheiten, wie *Range*, *Selection*, etc., lassen sich Aufzählzeichen über die Eigenschaft *Document.Content.ListFormat* einfügen. In der Prozedur (Code 3.63) wird zuerst der Inhalt des Dokuments mit Standardaufzählzeichen versehen. Danach bekommt der Inhalt Nummerierungszeichen und letztlich ein Aufzählzeichen aus einer Galerie von Möglichkeiten.

Code 3.63 Inhaltslistenformate

```
Sub ContentListFormats()
   Dim objDoc As Document

   Set objDoc = Documents.Open("C:\Temp\Text1.docx")
   With objDoc
      .Content.ListFormat.ApplyBulletDefault
      .Content.ListFormat.ApplyNumberDefault
      .Content.ListFormat.ApplyListTemplate _
         ListTemplate:=ListGalleries(wdBulletGallery). _
         ListTemplates(3)
   End With
   objDoc.Close
   Set objDoc = Nothing
End Sub
```

3.4.8 Standardschreiben

Der tägliche Büroalltag verlangt oft Schreiben, die einen immer wiederkehrenden gleichen oder ähnlichen Inhalt besitzen. Ein klassischer Fall für Textmarken. Als Beispiel ist ein Schreiben dargestellt (Abb. 3.13), in dem Textmarken vergeben sind. Beim Öffnen des Dokuments soll sich ein Dialogfenster automatisch öffnen mit allen Textmarken in einer Liste. Durch Anklicken der Textmarke in der Liste wird deren Text ins rechte Textfeld gesetzt und markiert. Dadurch kann der Text direkt überschrieben werden. Mit der Schaltfläche *Speichern* wird der Text dann direkt ins Schreiben übernommen.

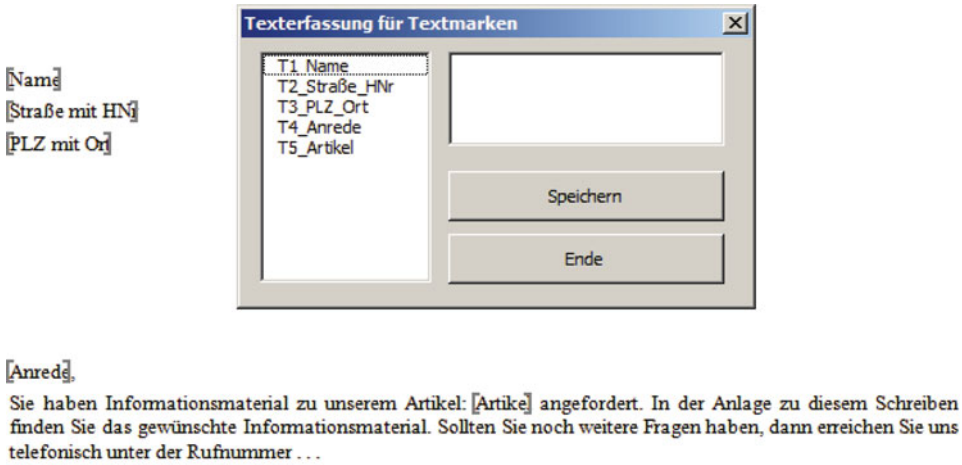


Abb. 3.13 Beispielschreiben mit Dialogfenster

Gestartet wird das Dialogfenster mit der Ereignisprozedur *Document_Open* im Codefenster von *Thisdocument*.

Code 3.64 Startprozedur im Codefenster von ThisDocument

```
Private Sub Document_Open()
   Load frmTextmarken
   frmTextmarken.Show
End Sub
```

Die Gestalt des Formulars *frmTextmarken* ist im Abb. 3.13 zu sehen. Auf der Oberfläche befinden sich die Schaltflächen *cmdExit* und *cmdSave*. Ein Listenfeld *lbxMarken* und eine Textbox *tbxText*.

Mit dem Aufruf des Formulars wird zuerst die Initialisierungs-Prozedur *UserForm_Initialize* ausgeführt, die alle im Dokument vorhandenen Textmarken ins Listenfeld schreibt (Code 3.65). Die anderen Prozeduren sind Click-Ereignisse je nach Wahl der Schaltfläche.

Code 3.65 Der Code des Formulars frmTextmarken

```
Private Sub cmdExit_Click()
   Unload Me
End Sub

Private Sub cmdSave_Click()
   Dim objDoc     As Document
   Dim bmkText    As Bookmark
   Dim sText As String

   Set objDoc = ThisDocument
   sText = lbxMarken.List(lbxMarken.ListIndex)
   Set bmkText = objDoc.Bookmarks(sText)
   bmkText.Range.Text = tbxText
   objDoc.Application.ScreenRefresh
   Set bmkText =Nothing
   Set objDoc = Nothing
End Sub

Private Sub lbxMarken_Click()
   Dim objDoc     As Document
   Dim bmkText    As Bookmark
   Dim sText      As String

   Set objDoc = ThisDocument
   sText = lbxMarken.List(lbxMarken.ListIndex)
   Set bmkText = objDoc.Bookmarks(sText)
   tbxText = bmkText.Range.Text
   tbxText.SetFocus
   tbxText.SelStart = 0
   tbxText.SelLength = Len(tbxText)
   Set bmkText =Nothing
   Set objDoc = Nothing
End Sub

Private Sub UserForm_Initialize()
   Dim objDoc     As Document
   Dim bmkText    As Bookmark
```

```
   Set objDoc = ThisDocument
   lbxMarken.Clear
   For Each bmkText In objDoc.Bookmarks
      lbxMarken.AddItem bmkText.Name
   Next
   Set objDoc = Nothing
End Sub
```

In einem weiteren Modul *modTextmarken* befinden sich zwei Hilfsprozeduren, die nicht erforderlich aber hilfreich sind (Code 3.66). Die Prozedur *Start* startet das Formular. Die zweite Prozedur *DeleteAllBookmarks* löscht alle vorhandenen Textmarken für eine Neuorganisation.

Code 3.66 Hilfsprozeduren

```
Public Sub Start()
   Load frmTextmarken
   frmTextmarken.Show
End Sub

Private Sub DeleteAllBookmarks()
   Dim objDoc     As Document
   Dim bmkText    As Bookmark

   Set objDoc = ThisDocument
   For Each bmkText In objDoc.Bookmarks
      bmkText.Delete
   Next
   Set objDoc = Nothing
End Sub
```

3.5 Word-Textvorlagen

In Word wird zwischen *Formatvorlagen* und *Dokumentvorlagen* unterschieden. Die Formatvorlagen werden je nach Anwendung in verschiedene Formatvorlagetypen eingeteilt (Anhang 3, Tab. A3.9). Eine Dokumentvorlage kann mehrere Formatvorlagen enthalten und ist neben anderen Einstellungen auch eine Sammlung von Formatvorlagen.

3.5.1 Formatvorlagen

Die Objektliste *Styles* enthält alle Formatvorlagen eines Dokuments. Die Prozedur (Code 3.67) gibt alle vorhandenen Formatvorlagen des zugehörigen Dokuments im Direktfenster aus. Es könnte auch statt *ThisDocument* mit *ActiveDocument* ein ausgewähltes Dokument angesprochen werden.

Code 3.67 Die Prozedur gibt alle vorhandenen Formatvorlagen aus

```
Sub ShowAllStyles()
   Dim docActive  As Document
   Dim objStyle   As Style

   Set docActive = ThisDocument
   On Error Resume Next
   For Each objStyle In docActive.Styles
      With objStyle
         Debug.Print "------------------------------"
         Debug.Print "Style:  " & .BaseStyle
         Debug.Print "Name:   " & .NameLocal
         Debug.Print "Update: " & .AutomaticallyUpdate
         Debug.Print "Font:   " & .Font
         Debug.Print "InUse:  " & .InUse
         Debug.Print "Parag:  " & .ParagraphFormat
      End With
   Next
   Set docActive = Nothing
End Sub
```

Im Anhang 3, Tab. A3.10 werden einige wichtige Eigenschaften einer Formatvorlage kurz beschrieben. Da einige Formatvorlagen nicht alle genannten Eigenschaften besitzen, steht in der Prozedur die Anweisung *On Error Resume Next*. So wird zur Vermeidung einer Fehlermeldung die nächste Anweisung ausgeführt. Eine mögliche Ausgabe zeigt Abb. 3.14.

```
------------------------------
Style:  Standard
Name:   Beschriftung
Update: Falsch
InUse:  Wahr
------------------------------
Style:  Absatz-Standardschriftart
Name:   BesuchterHyperlink
InUse:  Wahr
------------------------------
Style:  Standard
Name:   Blocktext
Update: Falsch
InUse:  Wahr
------------------------------
Style:  Absatz-Standardschriftart
Name:   Buchtitel
InUse:  Wahr
```

Abb. 3.14 Ausgabe der Beispiel-Formatvorlage-Eigenschaften

Auch vordefinierte Formatvorlagen können per Wertzuweisung geändert werden. Die Prozedur (Code 3.68) verwendet Eigenschaften die sich gut in der Objektbibliothek recherchieren lassen.

Code 3.68 Die Prozedur setzt die Eigenschaften des Standardformats

```
Sub SetNormalStyle()
   'wdStyleNormal = -1
   With ThisDocument.Styles(wdStyleNormal).Font
      .Name = "Times New Roman"
      .Size = 12
      .Bold = False
      .Italic = False
      .Underline = wdUnderlineNone 'Wert = 0
      .UnderlineColor = wdColorAutomatic 'Wert = -16777216
      .StrikeThrough = False
      .DoubleStrikeThrough = False
      .Outline = False
      .Emboss = False
      .Shadow = False
      .Hidden = False
      .SmallCaps = False
      .AllCaps = False
      .Color = wdColorAutomatic
      .Engrave = False
      .Superscript = False
      .Subscript = False
      .Spacing = 0
      .Scaling = 100
      .Position = 0
      .Kerning = 0
      .Animation = wdAnimationNone 'Wert = 0
      .Ligatures = wdLigaturesNone 'Wert = 0
      .NumberSpacing = wdNumberSpacingDefault 'Wert = 0
      .NumberForm = wdNumberFormDefault 'Wert = 0
      .StylisticSet = wdStylisticSetDefault 'Wert = 0
      .ContextualAlternates = 0
   End With
End Sub
```

Mit der Eigenschaft *LanguageID* ist es möglich, alle Formatvorlagen eines Dokuments einer Sprache anzupassen. Mitunter gibt es verschiedene Nutzer eines Dokuments aus unterschiedlichen Ländern. Dann ist eine Vereinheitlichung wichtig, da auch die Silben-

trennung von der eingestellten Sprache abhängig ist. So wie in der Prozedur (Code 3.69) alle vorhandenen Formatvorlagen des Dokuments auf Deutsch (*wdGerman*) umgestellt werden, kann dies auch mit einer anderen Sprache geschehen. *Styles* vom Typ 1 oder 2 sind Formatvorlagen für Zeichen und Absätze und somit sprachrelevant.

Code 3.69 Umschaltung auf deutsche Namen für Zeichen- und Absatz-Formatvorlagen

```
Sub SwitchLanguageID()
   Dim docActive  As Document
   Dim objStyle   As Style

   Set docActive = ThisDocument
   For Each objStyle In docActive.Styles
      If objStyle.Type < 3 Then
         objStyle.LanguageID = wdGerman
      End If
   Next
   Set docActive = Nothing
End Sub
```

Mit der Methode *Add* des Style-Objekts lassen sich neue Formatvorlagen erzeugen. Im Beispiel (Code 3.70) wird eine Formatvorlage *Demo* erstellt. Zuvor sollte ein Textbereich markiert werden, da die neue Formatvorlage auf die Auswahl direkt angewendet wird.

Code 3.70 Die Prozedur erstellt eine Formatvorlage und wendet sie an

```
Sub CreateStyle()
   Dim objStyle As Style

   Set objStyle = ThisDocument.Styles.Add _
      (Name:="Demo", Type:=wdStyleTypeCharacter)
   With objStyle.Font
      .Name = "Arial"
      .Size = 9
      .Italic = True
   End With
   Selection.Range.Style = "Demo"
   Set objStyle = Nothing
End Sub
```

Die verschiedenen Typen stehen in der Objektbibliothek unter *WdStyleType-Enumeration*.

Die Anwendung einer Formatvorlage auf ein Textelement zeigt das Beispiel (Code 3.71). Zuerst wird dem Textelement eine Objektvariablen zugewiesen und da-

nach der Objektvariablen eine Formatvorlage. Konkret wird den ersten zwei Absätzen die Standard-Formatvorlage *Normal* zugewiesen.

Code 3.71 Die Prozedur weist den ersten beiden Absätzen das Normal-Format zu

```
Sub SetStyle()
   Dim rngText As Range

   Set rngText = ThisDocument.Range( _
      Start:=ActiveDocument.Paragraphs(1).Range.Start, _
      End:=ActiveDocument.Paragraphs(2).Range.End)
   rngText.Style = wdStyleNormal
   Set rngText = Nothing
End Sub
```

Auch der Weg über ein *Selection*-Objekt ist hier möglich. Im Beispiel (Code 3.72) erhält der erste Absatz in der Auswahl die Kopfformatvorlage *Heading1*.

Code 3.72 Die Prozedur weist einem Auswahlbereich eine Formatvorlage zu

```
Sub SetSelectionStyle()
   Dim rngText As Range

   Set rngText = ThisDocument.Range
   rngText.Select
   Selection.Paragraphs(1).Style = wdStyleHeading1
   Set rngText = Nothing
End Sub
```

Die Methode *Delete* löscht eine Formatvorlage, sofern es sich nicht um eine Standard-Formatvorlage handelt. Mit der Eigenschaft *BuiltIn* kann dies überprüft werden. Die Prozedur (Code 3.73) entfernt alle nicht verwendeten Formatvorlagen.

Code 3.73 Die Prozedur löscht alle unbenutzten Nicht-Standard-Formatvorlagen

```
Sub DeleteNotUsedStyles()
   Dim styTemp As Style

   For Each styTemp In ThisDocument.Styles
      If styTemp.InUse = False And _
         styTemp.BuiltIn = False Then
         styTemp.Delete
      End If
   Next
End Sub
```

3.5.2 Listenvorlagen

Word verfügt über eine Objektliste *ListTemplates*, die mit der Installation vordefinierte Listenformate verwaltet. Sie werden im Dialogfeld *Nummerierung und Aufzählungszeichen* angezeigt. Die Prozedur (Code 3.74) liest alle Elemente der Auflistung und gibt mit dem Status an, ob eine oder mehrere Ebenen existieren.

Code 3.74 Die Prozedur liest alle Elemente der Objektliste ListTemplates

```
Sub ReadAllListTemplates()
   Dim objList As ListTemplate

   For Each objList In ThisDocument.ListTemplates
      Debug.Print "*" & objList.OutlineNumbered
   Next
End Sub
```

Wie bei allen Auflistungen lassen sich auch hier mit der Methode *Add* neue Elemente der Listenvorlagen erzeugen.

Formatvorlagen können mit einer Listenvorlage verknüpft werden. Dies wird mit der Methode *LinkToListTemplate* erreicht.

```
'Syntax:
   Ausdruck.LinkToListTemplate(ListTemplate[, ListLevelNumber])

'Beispiel:
   Thisdocument.LinkToListTemplate ListTemplate:=ltlTemp
```

Der Parameter *ListTemplate* ist erforderlich und gibt die Listenvorlage an, mit der die Formatvorlage verknüpft werden soll. Optional ist der Parameter *ListLevelNumber*, der die Listenebene angibt. Wird er nicht angegeben, dann wird die Ebene der Formatvorlage gewählt.

Im Beispiel (Code 3.75) wird eine neue Listenvorlage erstellt. Danach werden die Formatvorlagen für die Überschriften 1 bis 6 mit den Ebenen 1 bis 6 verknüpft. Die neue Listenvorlage wird anschließend dem Dokument zugewiesen. Alle mit einer Formatvorlage für die Überschriften formatierten Abschnitte nehmen die Nummerierung aus der Listenvorlage an.

Code 3.75 Die Prozedur erzeugt eine neue Listenvorlage

```
Sub CreateAndUseListTemplate()
   Dim docTemp    As Document
   Dim ltlTemp    As ListTemplate
   Dim iLoop      As Integer
```

```
   Set docTemp = ThisDocument
   Set ltlTemp = docTemp.ListTemplates.Add _
      (OutlineNumbered:=True)
   For iLoop = 1 To 6
      With ltlTemp.ListLevels(iLoop)
         .NumberStyle = wdListNumberStyleArabic
         .NumberPosition = InchesToPoints(0.25 * (iLoop - 1))
         .TextPosition = InchesToPoints(0.25 * iLoop)
         .NumberFormat = "%" & iLoop & "."
      End With
      With docTemp.Styles("Überschrift " & Trim(Str(iLoop)))
         .LinkToListTemplate ListTemplate:=ltlTemp
      End With
   Next iLoop
   docTemp.Content.ListFormat.ApplyListTemplate _
      ListTemplate:=ltlTemp
   Set ltlTemp = Nothing
   Set docTemp = Nothing
End Sub
```

3.5.3 Dokumentvorlagen

Eine Dokumentvorlage ist ein besonderer Dokumenttyp, der Text, Formatvorlagen, Prozeduren, Funktionen und weitere Einstellungen zusammenfassend speichert. Sie dient als Grundlage zur Erstellung neuer Dokumente und spart viel Zeit. Ihre ganze Wirksamkeit erhalten Dokumentvorlagen erst mit ihren Formatvorlagen und Prozeduren. Mit der Installation von Word wird automatisch die Dokumentvorlage *Normal.dotm* verwendet.

Mit der Erstellung eines neuen Dokuments durch die Methode *Add*, kann als Parameter eine eigene Dokumentvorlage vorgegeben werden (Code 3.76).

Code 3.76 Die Prozedur erzeugt ein neues Dokument mit der Dokumentvorlage Fachbuch.dotm

```
Sub CreateDocWithTemp()
   Dim objDoku    As Document
   Dim sVorlage   As String
   sVorlage = "C:\Templates\Fachbuch.dotm"
   Set objDoku = Application.Documents.Add(sVorlage)
   Stop
   objDoku.Close
   Set objDoku = Nothing
End Sub
```

Word verwaltet unter Optionen wichtige Einstellungen zur Nutzung. In VBA existiert dazu das Wahlobjekt *Option*, das zahlreiche Eigenschaften für diese Werte besitzt, so auch für Verzeichnisse besonderer Speicherorte, wie das für Dokumentvorlagen. Die Eigenschaft heißt *DefaultFilePath* und wird über einen Parameter angesprochen. Im Anhang 3, Tab. A3.11 sind alle möglichen Parameter aufgeführt.

Der Parameter *wdUserTemplatesPath* liefert dann auch das Verzeichnis, in dem alle Dokumentvorlagen stehen sollten (Code 3.77). Das erspart langes Suchen.

Code 3.77 Die Prozedur nutzt eine Dokumentvorlage aus dem Standardverzeichnis

```
Sub CreateDocWithTempPath()
   Dim objDoku    As Document
   Dim sVorlage   As String
   Dim sTempPath  As String

   sVorlage = "\Fachbuch.dotm"
   sTempPath = Options.DefaultFilePath(wdUserTemplatesPath)
   Set objDoku = Application.Documents.Add(sTempPath & sVorlage)
   Stop
   objDoku.Close
   Set objDoku = Nothing
End Sub
```

In Netzwerken können andere Verzeichnisse die Aufgabe der Template-Sammlung übernehmen. Es besteht auch die Möglichkeit, einem bestehenden Dokument eine neue Dokumentvorlage zuzuordnen. Im Beispiel (Code 3.78) wird erst ein Dokument mit der Standardvorlage erzeugt und danach gegen eine andere Dokumentvorlage mit der Methode *AttachedTemplate* ausgetauscht.

Code 3.78 Die Prozedur weist einem Dokument eine andere Dokumentvorlage zu

```
Sub DocAttachedTemp()
   Dim objDoku    As Document
   Dim sVorlage   As String
   Dim sTempPath  As String

   sVorlage = "\Fachbuch.dotm"
   sTempPath = Options.DefaultFilePath(wdUserTemplatesPath)
   Set objDoku = Application.Documents.Add
   Stop
   objDoku.AttachedTemplate = sTempPath & sVorlage
   Stop
   objDoku.Close
   Set objDoku = Nothing
End Sub
```

Mit der Methode *OpenAsDocument* kann eine Dokumentvorlage als Dokument geöffnet und deren Formatvorlagen bearbeitet werden. Die Prozedur (Code 3.79) öffnet die Dokumentvorlage des aktiven Dokuments als Dokument und schließt sie wieder ohne sie erneut zu speichern.

Code 3.79 Die Prozedur öffnet die aktive Dokumentvorlage als Dokument

```
Sub OpenTemplateAsDoc()
   Dim objDoc As Document

   Set objDoc = ThisDocument.AttachedTemplate.OpenAsDocument
   With objDoc
      If .Content.Text <> Chr(13) Then
         MsgBox "Template is not empty"
      Else
         MsgBox "Template is empty"
      End If
      .Close SaveChanges:=wdDoNotSaveChanges
   End With
   Set objDoc = Nothing
End Sub
```

Die Prozedur (Code 3.80) öffnet erneut die aktive Dokumentvorlage und ändert die Formatvorlage in *Überschrift1* um sie mit dieser Änderung dann zu speichern.

Code 3.80 Die Prozedur ändert eine Formatvorlage in der aktiven Dokumentvorlage

```
Sub ChangeTemplateAsDoc()
   Dim objDoc As Document

   Set objDoc = ThisDocument.AttachedTemplate.OpenAsDocument
   With objDoc
      With .Styles(wdStyleHeading1).Font
         .Name = "Times NewRoman"
         .Size = 18
         .Bold = True
         .Italic = False
      End With
      .Close SaveChanges:=wdSaveChanges
   End With
   Set objDoc = Nothing
End Sub
```

Die Prozedur (Code 3.81) erstellt ein Backup von der aktiven Dokumentvorlage.

Code 3.81 Die Prozedur erstellt ein Backup von der aktiven Dokumentvorlage

```
Sub TempBackUp()
   Dim objDoc As Document

   Set objDoc = ThisDocument.AttachedTemplate.OpenAsDocument
   With objDoc
      .SaveAs2 FileName:="C:\Temp\BackUp.dotm"
      .Close savechanges:=wdDoNotSaveChanges
   End With
   Set objDoc = Nothing
End Sub
```

Mit der Methode *OrganizerCopy* können Formatvorlagen zwischen Dokumenten und Dokumentvorlagen kopiert werden. Dabei müssen die Dateien mit ihren Pfadangaben als Text vorliegen (Code 3.82).

Code 3.82 Die Prozedur übertragt die Formatvorlage DemoStyle an eine Dokumentvorlage

```
Sub CopyHeading2ToTemplate()
   Dim objStyle As Style

   For Each objStyle In ThisDocument.Styles
      If objStyle = "DemoStyle" Then
         Application.OrganizerCopy _
            Source:="C:\Temp\Demo.docm", _
            Destination:="C:\Templates\BackUp.dotm", _
            Name:="DemoStyle", _
            Object:=wdOrganizerObjectStyles
      End If
   Next
End Sub
```

Mit der Methode *UpdateStyles* werden die Formatvorlagen aus der Dokumentvorlage in das Dokument übertragen. Vorhandene Formatvorlagen im Dokument werden überschrieben. Die Prozedur (Code 3.83) öffnet ein neues Dokument mit einer eigenen Dokumentvorlage. Die Formatvorlage *Überschrift1* in der Dokumentvorlage wird dann geändert und gespeichert. Mit der Methode *UpdateStyles* werden dann die Formatvorlagen in der Dokumentvorlage ins Dokument übertragen.

Code 3.83 Die Prozedur überträgt Formatvorlage aus der Dokumentvorlage ins Dokument

```
Sub UpdateHeading()
   Dim objDoc     As Document
   Dim objTmp     As Document
   Dim sVorlage   As String
   Dim sTempPath  As String

   sVorlage = "\Fachbuch.dotm"
   sTempPath = Options.DefaultFilePath(wdUserTemplatesPath)
   Set objDoc = Application.Documents.Add(sTempPath & sVorlage)
   Stop
   Set objTmp = objDoc.AttachedTemplate.OpenAsDocument
   With objTmp
      With .Styles(wdStyleHeading1).Font
         .Name = "Arial Black"
         .Size = 18
         .Bold = True
         .Italic = False
      End With
      .Close Savechanges:=wdSaveChanges
   End With
   Stop
   objDoc.UpdateStyles
   objDoc.Close Savechanges:=wdDoNotSaveChanges
   Set objDoc = Nothing
End Sub
```

3.5.4 Globale Dokumentvorlagen

Globale Dokumentvorlagen werden in der Regel nicht wie eine Dokumentvorlage verwendet. Sie besitzen weder Text noch Formatvorlagen. Dafür eignen sie sich für Prozeduren und Funktionen. Um sie nicht mit den normalen Dokumentvorlagen zu verwechseln, sollten sie in einem gesonderten Verzeichnis liegen.

Wie eine globale Dokumentvorlage im Dokument als *AddIn* installiert wird, um danach eine Prozedur im AddIn aufzurufen, zeigt die Prozedur (Code 3.84). Es ist die Ereignis-Prozedur *Document_Open*, die beim Öffnen des Dokuments aufgerufen wird.

Code 3.84 Die Ereignis-Prozedur installiert eine globale Dokumentvorlage

```
Private Sub Document_Open()
   Dim sAddIn As String

   sAddIn = "C:\GlobalTemplates\" & _
      "03-05-03_GlobaleDokumentvorlage.dotm"
   AddIns(sAddIn).Installed = True          'installiert AddIn
   Application.Run MacroName:="GlobalOpen"  'startet AddIn-Prozedur
End Sub
```

Eine globale Dokumentvorlage kann zunächst als Dokument mit Prozeduren und Funktionen erstellt werden. Dann sollte das Dokument als Dateityp **.dotm* in ein gesondertes Verzeichnis gespeichert werden. Zur späteren Bearbeitung kann mit einem Klick der rechten Maustaste auf den Dateieintrag und der Auswahl *Öffnen*, die Dokumentvorlage wieder zur Bearbeitung geöffnet werden (Abb. 3.15).

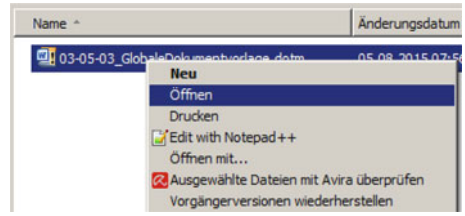


Abb. 3.15 Öffnen einer Dokumentvorlage mit dem Kontextmenü zur Bearbeitung

Die globale Dokumentvorlage erhält im Codefenster von *ThisDocument* die Konstruktion eines Menüband-Eintrags, wie bereits schon gezeigt (Code 3.85).

Code 3.85 Codezeilen im Codefenster des Thisdocument-Objekts der globalem Dokumentvorlage

```
Option Explicit

Public objDocu       As Document
Const sMenuName      As String = "DokumentInfo"

Public Sub GlobalOpen()
   Set objDocu = ActiveDocument
   InitMenu
End Sub

Private Sub GlobalClose()
   Set objDocu = Nothing
   RemoveMenu
End Sub
```

```
Private Sub InitMenu()
   Dim objMenuBar       As CommandBar
   Dim objMenuGroup     As CommandBarControl
   Dim objMenuButton    As CommandBarControl

   Call RemoveMenu
   Set objMenuBar = Application.CommandBars.Add _
      (sMenuName, msoBarTop)
   objMenuBar.Visible = True

   Set objMenuGroup = objMenuBar.Controls.Add _
      (Type:=msoControlPopup, Temporary:=False)
   With objMenuGroup
      .Caption = sMenuName
      .Tag = sMenuName
      .TooltipText = "Aktion wählen ..."
   End With

   Set objMenuButton = objMenuGroup.Controls.Add _
      (Type:=msoControlButton, Temporary:=True)
   With objMenuButton
      .BeginGroup = False
      .Caption = "DokInfo zeigen"
      .FaceId = 231
      .OnAction = "StartInfo"
      .Style = msoButtonIconAndCaption
      .TooltipText = "DokumentInfo zeigen"
      .Tag = "DokumentInfo zeigen"
   End With

   Set objMenuButton = objMenuGroup.Controls.Add _
      (Type:=msoControlButton, Temporary:=True)
   With objMenuButton
      .BeginGroup = False
      .Caption = "DokInfo entfernen"
      .FaceId = 232
      .OnAction = "GlobalClose"
      .Style = msoButtonIconAndCaption
      .TooltipText = "DokumentInfo entfernen"
      .Tag = "DokumentInfo entfernen"
   End With

   Set objMenuBar = Nothing
   Set objMenuGroup = Nothing
   Set objMenuButton = Nothing
End Sub
```

```
Private Sub StartInfo()
   Load frmDocInfo
   Set frmDocInfo.objDoku = objDocu 'Übergibt Dokument als Objekt
   frmDocInfo.Show
End Sub

Private Sub RemoveMenu()
   Dim objMenuBar       As CommandBar
   Dim objMenuControl   As CommandBarControl

   For Each objMenuBar In Application.CommandBars
      For Each objMenuControl In objMenuBar.Controls
         If objMenuControl.Tag = sMenuName Then
            objMenuControl.Delete
         End If
      Next
      If objMenuBar.Name = sMenuName Then
         objMenuBar.Delete
      End If
   Next

   Set objMenuBar = Nothing
   Set objMenuControl = Nothing
End Sub
```

Zusätzlich wird eine globale Objektvariable für das Dokument definiert, denn die globale Dokumentvorlage nutzt ein Formular *frmDocInfo* zur Ausgabe der Auswertung (Abb. 3.16). Die Prozedur *StartInfo* erstellt zunächst das Formular im Arbeitsspeicher, übergibt danach das Dokument-Objekt und zeigt anschließend das Formular. Das Formular enthält lediglich drei Textboxen, in denen die Anzahl der Zeichen, der Sätze und der Absätze im Dokument ausgegeben wird. Die Schaltfläche *Exit* gibt den Focus an das Dokument zurück.

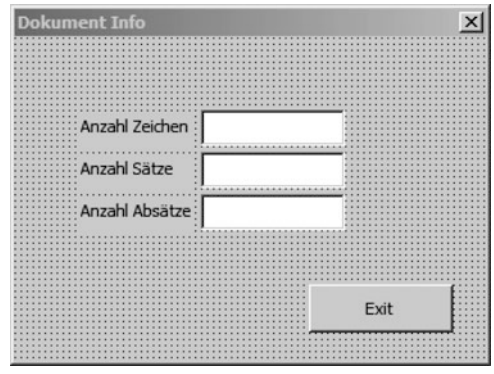


Abb. 3.16 Formular zur Auswertung in der globalen Dokumentvorlage

Mit der Initialisierung des Formulars werden die Zähler der Textelemente an die entsprechende Textbox übergeben (Code 3.86).

Code 3.86 Codezeilen im Formular der globalen Dokumentvorlage

```
Public objDoku As Word.Document

Private Sub cmdExit_Click()
   Unload Me
End Sub

Private Sub UserForm_Initialize()
   With ThisDocument.objDoku.Range
      tbxZeichen = .Characters.Count
      tbxSaetze = .Sentences.Count
      tbxAbsaetze = .Paragraphs.Count
   End With
End Sub
```

3.6 Word-Interaktionen

An ein paar Beispielen soll die Zusammenarbeit von Objekten aus Excel und Word gezeigt werden.

3.6.1 Word-Dokumente in Excel verwalten

Bei dieser Anwendung werden viele Dokumente erstellt, die den gleichen Aufbau besitzen. Ein gutes Beispiel sind Anforderungen für ein Projekt. Zu den Regeln eines Anforderungsmanagements (Requirement-Engineering) gehören

- Identifizierbarkeit,
- Eindeutigkeit,
- Atomarität,
- Nachweisbarkeit,
- Vollständigkeit,
- Testbarkeit,

um die wichtigsten zu nennen. Ebenso werden Formulierungsmuster wie in Abb. 3.17 genutzt.

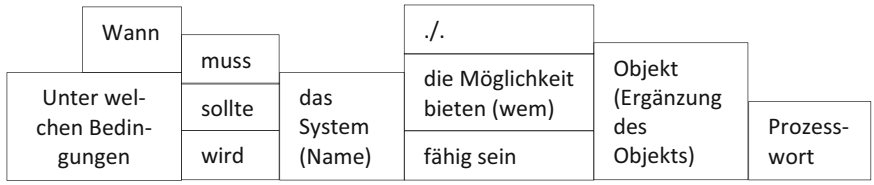


Abb. 3.17 Formulierungsmuster für Anforderungen

Doch uns interessieren weniger die Inhalte als das Modell. Gehen wir davon aus, dass zur Atomarität für jede Anforderung ein Dokument gehört. Die erste Seite eines jeden Dokuments enthält dann die wichtigen Daten in einer Tabelle. Dazu gehört auch eine eindeutige ID.

Ebenso wollen wir einen Status verwenden, der als Minimalausstattung die Attribute

- Offen,
- InArbeit,
- Fertig,
- Annuliert

besitzt. Die Dokumente sollen aus der Excel-Verwaltung gestartet werden. Mit dem Schließen werden dann die wichtigsten Daten zurück in die Verwaltung übertragen. Zum Modell gehört neben der Excel-Anwendung für die Verwaltung eine Dokumentvorlage für die Anforderungsdokumentation, die zur Erstellung der Dokumente Verwendung findet (Abb. 3.18).

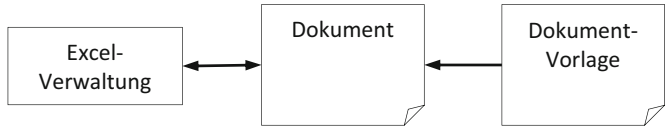


Abb. 3.18 Modell der Objekte zur Dokumenten-Verwaltung

3.6.1.1 Der Aufbau der Dokumentvorlage

Der erste Schritt ist der Entwurf der Dokumentvorlage, bzw. der Front-Tabelle. Sie soll alle wichtigen Daten verwalten. Tab. 3.2 zeigt die verwendete Form und enthält nur wenige Daten. Dadurch gewinnen wir an Übersichtlichkeit.

Tab. 3.2 Front-Tabelle des Anforderungs-Dokuments

Anforderung-ID	Dient gleichzeitig als Dateiname
Titel	Kurzbeschreibung der Anforderung
Status	{Offen, In Arbeit, Fertig, Annuliert}
Erstellt am	Datum in der Form TT.MM.JJJJ
Letzte Bearbeitung	Datum in der Form TT.MM.JJJJ
Sachbearbeiter	Namenskürzel XY in Optionen

Die zweite Seite des Dokuments enthält seine Entwicklungsgeschichte (History) (Tab. 3.3).

Tab. 3.3 History des Dokuments

Revision	**Änderung**	**Autor**	**Datum**

Die dritte Seite beginnt mit der Struktur des Inhalts. Auch die sollte in allen Dokumenten gleich sein (Abb. 3.19).

1 Einführung
1.1 Abkürzungen
1.2 Referenzliste
2 Beschreibung
3 Anhang

Abb. 3.19 Inhaltsvorgabe in der Dokumentvorlage

3.6.1.2 Die Funktionalität der Excel-Verwaltung

Die Excel-Verwaltung besteht aus zwei Arbeitsblättern, einem Cover und der Verwaltungsliste. Das Cover enthält alle Daten die einmalig zur Anwendung benötigt werden, wie den Pfad der Dokumentation, den Pfad der Dokumentvorlage, den Versionsstand, das letzte Bearbeitungsdatum, den Autor und weitere Daten (Abb. 3.20). Mit anderen Dokumentvorlagen lassen sich verschiedene Verwaltungssysteme aufbauen.

Abb. 3.20 Aufbau des Covers

Das Arbeitsblatt Verwaltung gibt den Inhalt der Front-Tabelle des Dokuments wieder (Abb. 3.21).

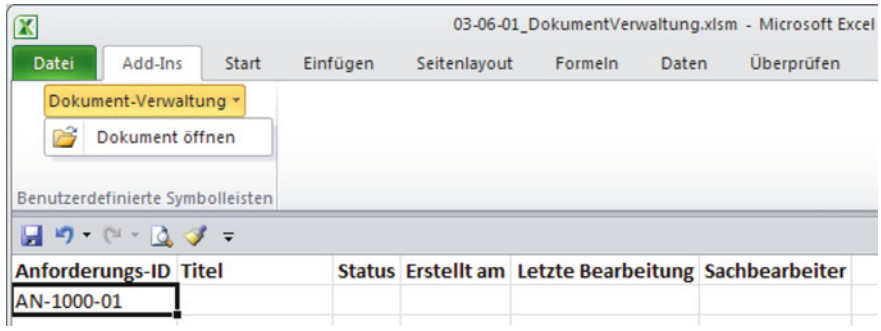


Abb. 3.21 Aufbau des Arbeitsblattes Verwaltung mit erstem Eintrag und Menüfunktion

Über den Menüeintrag *Dokument öffnen*, den als Code zu zeigen ich mir hier schenke, wird die Prozedur *DocumentOpen* (Code 3.87) in einem Codemodul der Excel-Arbeitsmappe aufgerufen.

Code 3.87 Die Prozedur öffnet das in der Verwaltung ausgewählte Dokument

```
Private Sub DocumentOpen()
   Dim appWord    As Word.Application
   Dim docTmp     As Word.Document
   Dim wshTmp     As Worksheet
   Dim wshCover   As Worksheet
   Dim sDocName   As String
   Dim sPfad      As String
   Dim vResult    As Variant
   Dim sDocVor    As String

'Test
   Set wshTmp = ActiveSheet
   If Not wshTmp.Name = "Verwaltung" Or _
      Not ActiveCell.Column = 1 Or _
      Selection.Cells.Text = "" Then
      MsgBox "Bitte Anforderungs-ID wählen!"
      Set wshTmp = Nothing
      Exit Sub
   End If

'Aufruf
   sDocName = Selection.Cells.Text
   sPfad = ThisWorkbook.Worksheets("Cover").Cells(7, 3)
   sDocVor = ThisWorkbook.Worksheets("Cover").Cells(8, 3)
   If Dir(sPfad & "\" & sDocName & ".docx") = "" Then
      vResult = MsgBox("Anforderung nicht vorhanden! " & vbLf & _
      "Neu erstellen?", vbInformation + vbYesNo, "WICHTIGER HINWEIS")
      If vResult = vbNo Then
         Exit Sub
         Set wshTmp = Nothing
      Else 'Dokument neu erstellen
         Set appWord = CreateObject("Word.Application")
         appWord.Visible = True
         Set docTmp = appWord.Documents.Add(Template:=sDocVor)
         docTmp.SaveAs2 Filename:=sPfad & "\" & sDocName & ".docx"
         With docTmp.Tables(1) 'Übergabe an die Front-Tabelle
            .Cell(1, 2).Range.Text = sDocName
            .Cell(2, 2).Range.Text = wshTmp.Cells(ActiveCell.Row, 2)
            .Cell(3, 2).Range.Text = "Offen"
            .Cell(4, 2).Range.Text = Format(Date, "DD.MM.YYYY")
            .Cell(5, 2).Range.Text = Format(Date, "DD.MM.YYYY")
            .Cell(6, 2).Range.Text = Application.UserName
         End With
```

```
         With wshTmp 'Eintragungen in die Verwaltung
            .Cells(ActiveCell.Row, 3) = "Offen"
            .Cells(ActiveCell.Row, 4) = Format(Date, "DD.MM.YYYY")
            .Cells(ActiveCell.Row, 5) = Format(Date, "DD.MM.YYYY")
            .Cells(ActiveCell.Row, 6) = Application.UserName
         End With
         Set docTmp.objtmp = wshTmp       'Übergabe Worksheet-Objekt
         docTmp.lActRow = ActiveCell.Row  'Übergabe aktive Zeile
      End If
   Else 'Dokument besteht bereits
      Set appWord = CreateObject("Word.Application")
      appWord.Visible = True
      Set docTmp = appWord.Documents.Open _
         (Filename:=sPfad & "\" & sDocName & ".docx")
      With docTmp.Tables(1)  'Übergabe an die Front-Tabelle
         .Cell(5, 2).Range.Text = Format(Date, "DD.MM.YYYY")
         .Cell(6, 2).Range.Text = Application.UserName
      End With
      With wshTmp 'Eintragungen in die Verwaltung
         .Cells(ActiveCell.Row, 5) = Format(Date, "DD.MM.YYYY")
         .Cells(ActiveCell.Row, 6) = Application.UserName
      End With
      Set docTmp.objtmp = wshTmp       'Übergabe Worksheet-Objekt
      docTmp.lActRow = ActiveCell.Row  'Übergabe aktive Zeile
   End If
   Set docTmp = Nothing
   Set appWord = Nothing
End Sub
```

3.6.1.3 Die Funktionalität der Dokumentvorlage

In der Dokumentvorlage ist eine Reaktion auf das Ereignis *Schließen* vorgesehen. Zunächst wird das Dokument grundsätzlich noch einmal gespeichert. Danach erfolgt die Übergabe der Daten aus der Front-Tabelle des Dokuments an die Verwaltung. Da bei der Übergabe Steuerzeichen mit im Text stehen, wird eine *Replace*-Funktion davor geschaltet (Code 3.88). Sie entfernt das Return-Zeichen (ASCII-Character 13).

Code 3.88 Die Ereignis-Prozedur und globale Variable in der Dokumentvorlage

```
Public objTmp     As Excel.Worksheet
Public lActRow    As Long

Private Sub Document_Close()
   Dim sText
   ActiveDocument.Save
   If lActRow > 0 Then
```

```
      sText = ActiveDocument.Tables(1).Cell(2, 2)
      objTmp.Cells(lActRow, 2) = ReplaceChar(sText)
      sText = ActiveDocument.Tables(1).Cell(3, 2)
      objTmp.Cells(lActRow, 3) = ReplaceChar(sText)
      sText = ActiveDocument.Tables(1).Cell(5, 2)
      objTmp.Cells(lActRow, 5) = ReplaceChar(sText)
      sText = ActiveDocument.Tables(1).Cell(2, 2)
      objTmp.Cells(lActRow, 6) = Application.UserName
   End If
   Set objTmp = Nothing
End Sub

Function ReplaceChar(sText As Variant) As String
   sText = Replace(sText, Chr(7), "")
   ReplaceChar = Replace(sText, Chr(13), "")
End Function
```

3.6.1.4 Datenanalyse

Mit den in der Verwaltung vorhandenen Werten lassen sich verschiedene Auswertungen durchführen. Dazu gehört zum Beispiel eine Statistik über den Status der Anforderungen (Abb. 3.22). Aufgerufen wird die Prozedur (Code 3.89) wieder über einen Menüeintrag.

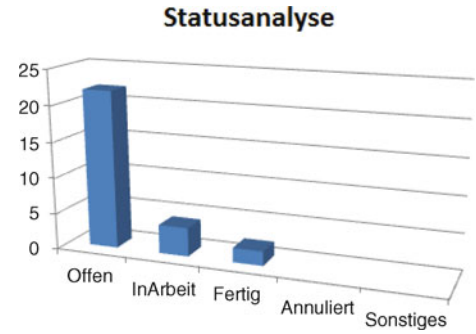


Abb. 3.22 Ergebnis der Statusanalyse

Code 3.89 Die Prozedur erstellt eine Statusanalyse

```
Private Sub CreateStatus()
   Dim wshCover    As Worksheet
   Dim wshDaten    As Worksheet
   Dim chtDaten    As Variant
   Dim diaStatus   As Shape
   Dim lRow        As Long
   Dim lMax        As Long
   Dim iIndex      As Integer
   Dim rngDaten    As Range
```

```
   Dim lSumme(5)  As Long
   Dim sStatus    As String

   Set wshDaten = Worksheets("Verwaltung")
   Set wshCover = Worksheets("Cover")
   For Each chtDaten In wshDaten.ChartObjects
      chtDaten.Delete
   Next
   wshCover.Range("C18:C22").Clear
   lMax = wshDaten.UsedRange.Rows.Count
   For lRow = 2 To lMax
      sStatus = wshDaten.Cells(lRow, 3)
      sStatus = Replace(sStatus, Chr(13), "")
      Select Case sStatus
      Case "Offen"
         iIndex = 18
      Case "InArbeit"
         iIndex = 19
      Case "Fertig"
         iIndex = 20
      Case "Annuliert"
         iIndex = 21
      Case Else
         iIndex = 22
      End Select
      wshCover.Cells(iIndex, 3) = _
         wshCover.Cells(iIndex, 3) + 1
   Next
   Set diaStatus = wshDaten.Shapes.AddChart
   Set rngDaten = wshCover.Range("B18:C22")
   With diaStatus
      .Name = "StatusAnalyse"
      With .Chart
         .ChartType = xl3DColumnStacked
         .HasTitle = True
         .ChartTitle.Text = "Statusanalyse"
         .Legend.Delete
         .SetSourceData Source:=rngDaten, PlotBy:=xlColumns
      End With
   End With
   Set diaStatus = Nothing
End Sub
```

3.6.2 Arbeitsblätter mit Word ausgeben

So wie Excel erste Wahl für Berechnungen und Visualisierungen ist, ist es Word für Dokumentationen und Ausgaben. Auch wenn Excel inzwischen einige Druckvarianten besitzt, kann Word doch mehr bieten. In diesem Beispiel sollen sechs Excel-Arbeitsblätter auf einem DIN A4 Blatt mit Word ausgegeben werden.

Die Lösung bietet sich in zwei Schritten an. Zunächst werden die ausgewählten Blätter in ein Word-Dokument kopiert, um dann im zweiten Schritt gedruckt zu werden. Die Auswahl der Arbeitsblätter kann auf zwei Wegen ausgeführt werden. Zuerst wird das Register des ersten Arbeitsblattes markiert.

Wenn danach die Umschalttaste gedrückt und gehalten wird, kann mit der Maus das Register des letzten Arbeitsblattes ausgewählt werden. Alle Arbeitsblätter zwischen diesen werden dann ebenfalls markiert.

Wenn nach der Auswahl des ersten Arbeitsblattes die Steuerungstaste gedrückt und gehalten wird, können weitere Arbeitsblätter durch Anklicken mit der Maus ausgewählt und durch erneutes Klicken auch wieder abgewählt werden. Über das Kontextmenü (rechte Maustaste) können alle Gruppierungen auch wieder aufgehoben werden.

Nachdem so alle zu druckenden Arbeitsblätter ausgewählt sind, wird die Prozedur zur Druckausgabe (Code 3.90) aufgerufen. Das kann über einen Menüeintrag oder als Prozeduraufruf geschehen.

Code 3.90 Die Prozedur druckt maximal sechs Arbeitsblätter mit der Word-Druckfunktion

```
Sub WordPrint()
   Dim wshTab      As Worksheet
   Dim appWord     As Word.Application
   Dim docWord     As Word.Document
   Dim bNext       As Boolean
   Dim vResult     As Variant

'kopiere und zeige markierte Arbeitsblätter
   Set appWord = CreateObject("Word.Application")
   appWord.Visible = True
   Set docWord = appWord.Documents.Add
   appWord.Caption = "DRUCKAUSGABE"
   docWord.PageSetup.Orientation = wdOrientLandscape
```

```
   bNext = False
   For Each wshTab In ActiveWindow.SelectedSheets
      If bNext = True Then
         appWord.Selection.InsertBreak Type:=wdPageBreak
      Else
         bNext = True
      End If
      wshTab.UsedRange.Copy
      docWord.Activate
      appWord.Selection.PasteExcelTable _
         LinkedToExcel:=False, _
         WordFormatting:=False, _
         RTF:=True
   Next
   appWord.ActiveWindow.ActivePane.View.Zoom.Percentage = 40

'Druckausgabe
   vResult = MsgBox("Druck starten?", _
      vbInformation + vbYesNo, "DRUCKABFRAGE")
   If vResult = vbYes Then
      With appWord
         .PrintOut _
            Filename:="", _
            Item:=wdPrintDocumentWithMarkup, _
            Copies:=1, _
            PageType:=wdPrintAllPages, _
            Collate:=True, _
            Background:=True, _
            PrintToFile:=False, _
            PrintZoomRow:=3, _
            PrintZoomColumn:=2, _
            PrintZoomPaperWidth:=0, _
            PrintZoomPaperHeight:=0
         MsgBox "Warten auf die Druckausgabe!"
      End With
   End If

'Anwendung schließen
   With appWord
      .ActiveDocument.Close SaveChanges:=False
      .Quit SaveChanges:=False
   End With
   Set appWord = Nothing
   Set docWord = Nothing
End Sub
```

Beim Start wird zuerst ein Worddokument erstellt mit den ausgewählten Tabellen auf jeder Seite. Das Dokument wird mit dem Zoomfaktor 40 % angezeigt, so dass eine gute Übersicht gegeben ist (Abb. 3.23). Als Ausrichtung habe ich Querformat gewählt, was wohl den meisten Excel-Tabellen entspricht.

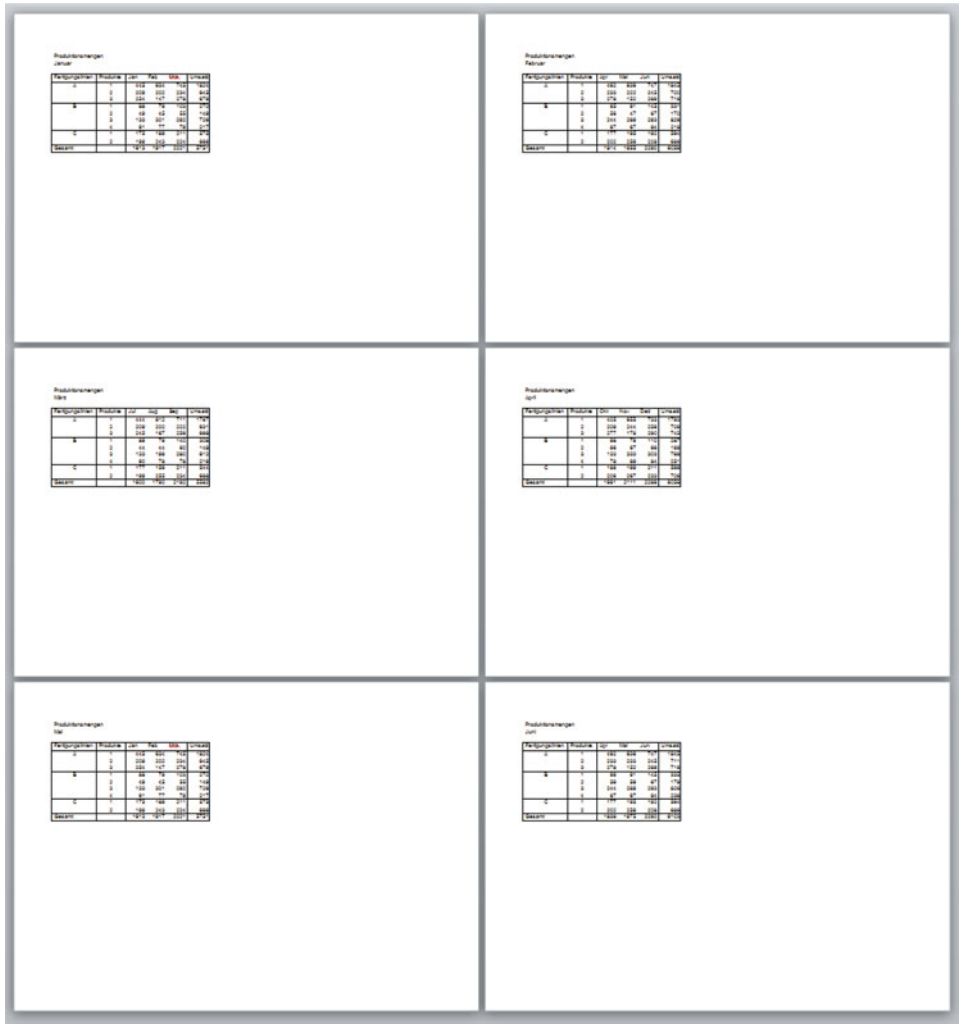


Abb. 3.23 Anzeige der für den Druck ausgewählten Arbeitsblätter

Mit der Anzeige erfolgt die Abfrage zum Druckstart. Entspricht die Darstellung nicht der gewünschten Form*, kann hier der Druck noch abgebrochen werden. Zwischen Druckstart und Dokument schließen* ist ein so geringer Zeitraum, dass der Druck nicht gestartet wird. Daher ist noch einmal eine Textbox vorgesehen, deren Schaltfläche *OK* nach dem *Druckstart* dann das *Schließen* des Word-Dokuments genehmigt.

3.6.3 Word-Tabellen in Excel auswerten

In diesem Beispiel werden Word-Tabellen in Excel ausgewertet und visualisiert. Ausgang ist ein Dokument mit mehreren Tabellen. Die betreffende Tabelle wird im Dokument markiert. Zur Auswertung wird die Tabelle nach Excel übertragen und es werden Summe, Mittelwert, Maximum und Minimum bestimmt. Außerdem wird ein Diagramm erstellt und in der Zwischenablage abgelegt. Vor dem Start wird noch der Diagrammtyp (Balken- oder Kreis-Diagramm) bestimmt. Nach der Auswertung kann das Diagramm aus der Zwischenablage an eine beliebige Stelle ins Dokument kopiert werden. Die Anzahl der Tabellen im Dokument ist beliebig. Ihr Aufbau besteht aus zwei Spalten. In der ersten Spalte befindet sich ein Name und in der zweiten Spalte der dazugehörige Wert. Die Tabellen haben Spaltenüberschriften.

Zum Aufruf des Formulars (Abb. 3.24) wird in einem Code-Modul die Prozedur (Code 3.91) verwendet.

Abb. 3.24 Startformular

Code 3.91 Startprozedur für das verwendet Formular

```
Private Sub ErstelleDiagramm()
   Load frmDiagramm
   frmDiagramm.Show
End Sub
```

Die Codezeilen des Formulars gehören zu den zwei Schaltflächen und der Prozedur zur Diagrammerstellung (Code 3.92).

Code 3.92 Die Prozeduren des Start-Formulars

```
Private Sub cmdAbbruch_Click()
   Unload Me
End Sub

Private Sub cmdOK_Click()
   DiagrammErstellen
   Unload Me
End Sub
```

```
Sub DiagrammErstellen()
   Dim appExcel As Object
   Dim wshDaten As Object
   Dim lMax As Long
   Dim sText As String
   Dim rngDaten As Variant

'Abfrage, ob die Tabelle markiert ist
   If Selection.Information(wdWithInTable) = True Then
      Selection.Tables(1).Select
      Selection.Copy
      frmDiagramm.Hide
      Set appExcel = Excel.Application
      With appExcel
         .Visible = True
         .Workbooks.Add
         Set wshDaten = .ActiveSheet
         .ActiveSheet.Paste
      'Diagramm
         .Charts.Add
         With ActiveChart
            If frmDiagramm.optSäulendiagramm = True Then
               .ChartType = xl3DColumn
            ElseIf frmDiagramm.optKreisdiagramm = True Then
               .ChartType = xl3DPie
            End If
            .SetSourceData appExcel.Worksheets(1).UsedRange
            .Location xlLocationAsNewSheet
            .ChartArea.Copy
         End With
      'Auswertung
         lMax = wshDaten.UsedRange.Rows.Count
         sText = "B2:B" & Trim(Str(lMax))
         Set rngDaten = wshDaten.Range(sText)
         With appExcel.WorksheetFunction
            sText = "Maximum = " & .Max(rngDaten)
            sText = sText & vbLf & "Minimum = " & .Min(rngDaten)
            sText = sText & vbLf & "Summe = " & .Sum(rngDaten)
            sText = sText & vbLf & "Mittelwert = " & .Average(rngDaten)
         End With
         .ActiveWorkbook.Close False
         .Quit
      End With
```

```
      Set appExcel = Nothing
      MsgBox sText & vbLf & vbLf & _
         "Das Diagramm wurde in die Zwischenablage kopiert!", _
         vbInformation
   Else
      MsgBox "Bitte den Cursor in eine Tabelle setzen!", _
      vbInformation
      frmDiagramm.Hide
   End If
End Sub

Private Sub UserForm_Initialize()
   optSäulendiagramm.Value = True
End Sub
```

PowerPoint-Objekte

4

Täglich werden Millionen Präsentationen mithilfe von PowerPoint durchgeführt. Längst gibt es im Internet Beispiele und Tipps für die richtige Gestaltung. Das Anwendungsgebiet für PowerPoint ist vielfältig, angefangen von Referaten über Präsentationen bis hin zum Report. Weniger bekannt ist, dass auch PowerPoint über eine VBA-Entwicklungsumgebung verfügt.

4.1 PowerPoint-Anwendungen

Das Anwendungs-Objekt *Application* ist auch hier das oberste Objekt (Abb. 4.1). Einige Objekte sind bereits aus den anderen Anwendungen bekannt. Das wichtigste Objekt ist das Folien-Objekt *Slide*, das hier als Objektliste *Slides* auftritt. Auch das Präsentations-Objekt *Presentation* wird als Objektliste verwaltet und ebenso die Objektliste *SlideShowWindows* aller aktiven Bildschirmpräsentationen.

H. Nahrstedt, *Die Welt der VBA-Objekte*, DOI 10.1007/978-3-658-13891-2_4

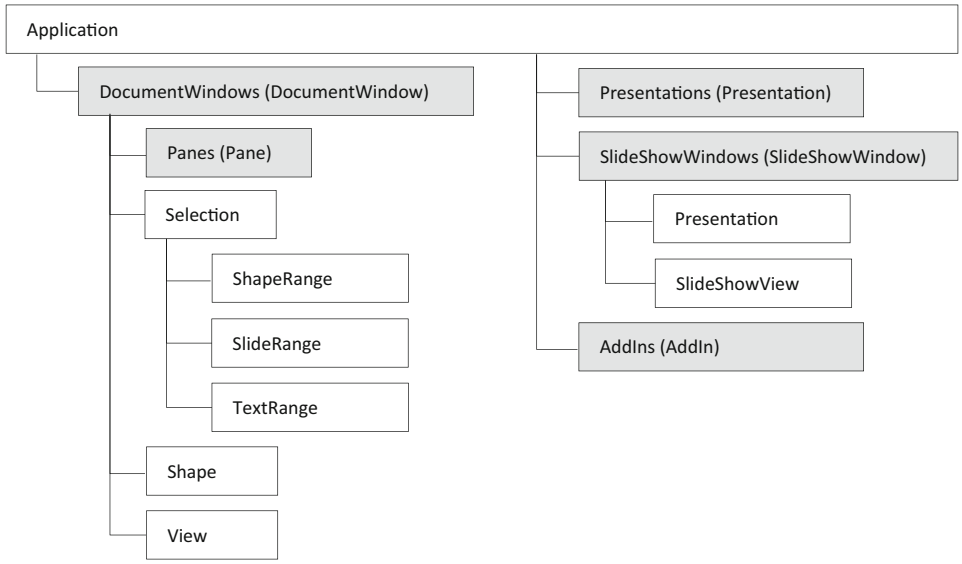


Abb. 4.1 Die wichtigsten Unterobjekte und Objektlisten (*grau*) des Application-Objekts

Im Objektkatalog befindet sich eine ausführliche Darstellung der PowerPoint-Objektmodellreferenz. Mit ihr kann durch die Dokumentation navigiert werden. Ebenso gibt es eine Aufstellung aller Elemente des *Application*-Objekts und die der nachfolgenden Objekte.

4.1.1 Eigenschaften

Die Eigenschaft *WindowState* steuert die Größe des Bildschirms.

```
'Syntax
   Application.WindowState = ppWindowKonstante
'Beispiel:
   AppPPoint.WindowState = ppWindowMaximized 'Vollbildschirm
```

Die Konstanten der Eigenschaft *WindowState* sind in Anhang 4, Tab. A4.1 aufgeführt.

Die Prozedur (Code 4.1) ermittelt einige Systemdaten über die Eigenschaften des Anwendungs-Objekts.

Code 4.1 Die Prozedur liefert PowerPoint-Systeminformationen

```
Sub PPInfo()
   Dim appPpt As Application
   Dim sText As String
```

```
   Set appPpt = Application
   With appPpt
      sText = "System: " & .OperatingSystem & vbCrLf
      sText = sText & "Version: " & .Version & vbCrLf
      sText = sText & "Pfad: " & .Path & vbCrLf
      sText = sText & "Name: " & .Name & vbCrLf
      sText = sText & "BildschirmStatus: " & .WindowState & vbCrLf
      sText = sText & "Präsentation: " & _
         .ActivePresentation.FullName & vbCrLf
      sText = sText & "Drucker: " & .ActivePrinter & vbCrLf
      sText = sText & "Titel: " & .Caption
   End With

   MsgBox sText, vbInformation + vbOKOnly, "INFORMATIONEN"
   Set appPpt = Nothing
End Sub
```

Die Eigenschaft *Caption* repräsentiert die Titelzeile des Fensters und kann durch eine Wertzuweisung gesetzt werden. Die Positionierung des Anwendungsfensters ist über *ActiveWindow* möglich (Code 4.2).

Code 4.2 Die Prozedur verändert Darstellung und Titel des Anwendungsfensters

```
Sub PPTitelWindow()
   Dim appPpt As Application

   Set appPpt = Application
   With appPpt
      With .ActiveWindow
         .Left = 50
         .Top = 50
         .Width = 400
         .Height = 400
      End With
      .Caption = "DEMO"
   End With
   Set appPpt = Nothing
End Sub
```

4.1.2 Methoden

Eine PowerPoint-Anwendung startet beim Aufruf mit einer Präsentation als Unterobjekt, ähnlich wie die Excel-Anwendung mit einer Arbeitsmappe startet. Daher gibt es hier wenige Methoden, die im nachfolgenden Beispiel zusammengefasst sind.

Die Methode *Activate* aktiviert die PP-Anwendung. Die Methode *Help* ruft die PP-Hilfe auf, die auch mit der F1-Taste startet. Mit der Methode *Run* können weitere Prozeduren gestartet werden. Letztlich schließt die Methode *Quit* die Anwendung (Code 4.3).

Code 4.3 Die wichtigsten Methoden der PP-Anwendung

```
Sub PPCreate()
   Dim appPpt As Application

   Set appPpt = CreateObject("PowerPoint.Application")
   appPpt.Activate
   appPpt.WindowState = ppWindowNormal
   appPpt.Help              'startet F1 Hilfe
   appPpt.Run "PPInfo"      'führt die Prozedur PPInfo aus
   appPpt.Quit
   Set appPpt = Nothing
End Sub
```

4.1.3 Ereignisse

Wie bei allen anderen Anwendungen auch, gibt es keinen Code-Container für die Ereignisse der PP-Anwendung. Auch hier muss man also wieder den Weg über eine Objektvariable gehen und mit *WithEvents* die Ereignisse von der Anwendung erben. Da PP weder ein Objekt *Workbook* noch ein Objekt *Document* besitzt, nutzen wir diesmal ein Klassenmodul. Es erhält die nachfolgende Anweisung und bekommt den Namen *clsEvents* (Code 4.4).

Code 4.4 Ereigniszuordnung im Klassenmodul clsEvent

```
Public WithEvents appEvent As Application
```

In einem Codemodul stehen die nachfolgenden Anweisungen (Code 4.5).

Code 4.5 Initialisierung und Terminierung der Anwendungs-Objektvariablen

```
Dim PPObject As New clsEvents

Sub Auto_Open()
   Set PPObject.appEvent = Application
End Sub

Sub Auto_Close()
   Set PPObject.appEvent = Nothing
End Sub
```

Im Klassenmodul finden wir nun in der Objektauswahl das Objekt *appEvent* und in der Ereignisauswahl die dazugehörigen Events (Abb. 4.2).

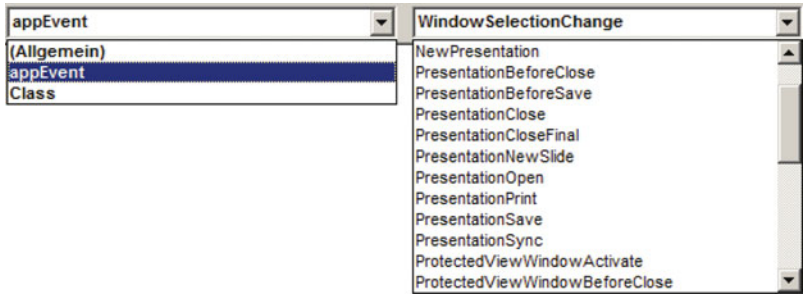


Abb. 4.2 Auswahl in der Entwicklungsumgebung

Viele dieser Ereignisse beziehen sich auf Präsentationen, die wir im nächsten Kapitel behandeln.

4.1.4 Auto_Open und Auto_Close

Die Ereignisse *Open* und *Close*, bzw. *BeforeClose* suchen wir im Objekt der Anwendung vergeblich. Sie sind wohl von den Entwicklern vergessen worden. Über den Umweg eines *Add-Ins* gelangen wir dennoch zu unseren Prozeduren.

Im ersten Schritt erstellen wir in einer leeren PP-Anwendung ein Codemodul mit den bekannten Prozeduren (Code 4.6). Zum Test erhalten sie lediglich Bildschirmausgaben.

Code 4.6 Auto-Testprozeduren

```
Sub Auto_Open()
   MsgBox "Auto_Open"
End Sub

Sub Auto_Close()
   MsgBox "Auto_Close"
End Sub
```

Diese Anwendung wird danach zuerst einmal gespeichert, um für spätere Änderungen zur Verfügung zu stehen. Danach wird die gleiche Anwendung als Dateityp PowerPoint-Add-In (*.ppam) unter dem Namen *C:\Temp\PP_AutoTest.ppam* für dieses Beispiel gespeichert. Dazu bitte ein anderes Verzeichnis nutzen.

Um nun das *AddIn* einzubinden muss in einer normalen PP unter *Datei* und *Optionen* die Gruppe *Add-Ins* gewählt werden (Abb. 4.3).

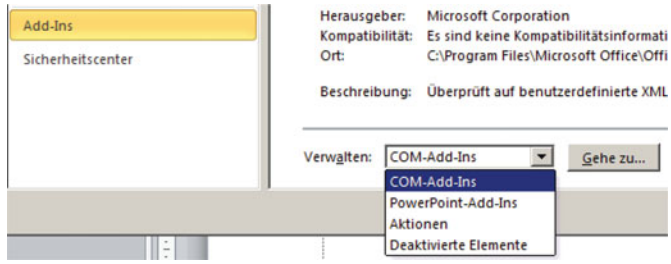
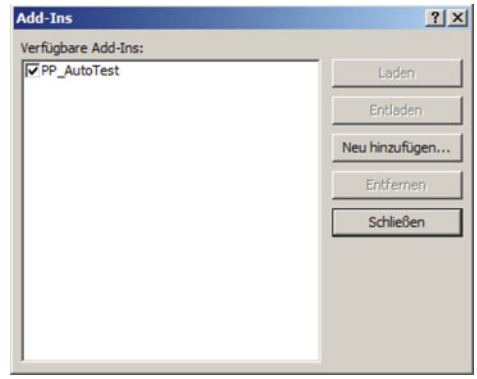


Abb. 4.3 Auswahl der Add-Ins unter Optionen

In der Liste *Verwalten* werden die *PowerPoint-Add-Ins* gewählt. Mit *Gehe zu* ... öffnet sich das Dialogfenster Add-Ins (Abb. 4.4). Unter *Neu hinzufügen* ... kann dann das von uns erstellt Add-In *PP_AutoTest* eingetragen werden.

Abb. 4.4 Eintrag des Add-Ins im Dialogfenster



Mit jedem Öffnen und Schließen einer PP-Anwendung sehen wir dann die vorgesehenen Meldungen, aber auch bei jeder. Denn dies ist der Nachteil, dass ein eingetragenes Add-In so lange wirksam bleibt, bis es wieder ausgetragen ist. Nach dieser Anschauung kann dann das Add-In auch wieder ausgetragen werden. Später werden wir eine sinnvolle Anwendung besprechen.

4.2 PowerPoint-Präsentationen

Ähnlich wie in Excel die Anwendung Arbeitsmappen beherbergt, in denen sich wiederum Arbeitsblätter befinden die die eigentliche Arbeitsplattform darstellen, so beherbergt die PowerPoint-Anwendung die Präsentations-Objektliste *Presentations*, in denen sich die Folien-Objekte *Slides* befinden, die hier die Arbeitsplattform bilden. Die richtige Übersetzung von *Slides* müsste Dias oder Lichtbilder lauten, doch es hat sich der Begriff *Folie* durchgesetzt, wohl in Anlehnung an die alten Projektionsgeräte.

```
'Syntax:
   Presentations.(Item(IndexNr oder Name))

'Beispiel:
   Presentations.Item(1)
```

Im Objektkatalog sind unter *Elemente des Präsentation-Objekts* alle Eigenschaften und Methoden aufgeführt.

4.2.1 Eigenschaften

Neben der Auswahl einer Präsentation aus der Objektliste gibt es auch hier eine besondere Präsentation, die sich auf die aktive Präsentation bezieht und mit *ActivePresentation* angesprochen wird. In der Prozedur (Code 4.7) wird der volle Name des aktiven Dokuments ausgegeben (Abb. 4.5). Während die Eigenschaft *Name* nur den Namen des Dokuments liefert, wird mit *FullName* auch der Pfad mit angegeben.

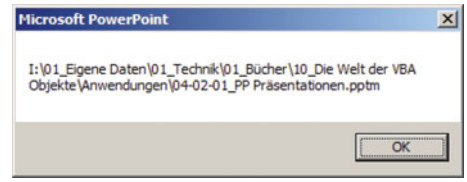


Abb. 4.5 Ausgabe des aktiven Präsentationsnamens

Code 4.7 Die Prozedur gibt den vollen Namen der aktiven Präsentation aus

```
Sub ActivePresentationName()
   Dim objAktiv As Presentation

   Set objAktiv = ActivePresentation
   MsgBox objAktiv.FullName
   Set objAktiv = Nothing
End Sub
```

4.2.2 Methoden

Da Präsentationen in einer Objektliste verwaltet werden liegt es nahe, dass auch hier mit der Methode *Add* neue Präsentationen erstellt werden (Code 4.8). Eine neu erstellte Präsentation wird mit der Methode *SaveAs* gespeichert. Dabei wird auch der Name festgelegt.

Code 4.8 Die Prozedur erstellt und speichert eine neue Präsentation

```
Sub CreateNewPresentation()
   Dim pppNew As Presentation

   Set pppNew = Application.Presentations.Add
   Stop
   pppNew.Close
   Set pppNew = Nothing
End Sub
```

Vorhandene Präsentationen werden mit der Methode *Open* geladen und können mit der Methode *Save* oder *SaveAs* wieder gespeichert werden. Die Prozedur (Code 4.9) öffnet eine vorhandene Präsentation auch zum Ändern, verändert die Fensterdarstellung und speichert sie wieder, einmal unter ihrem Namen und ein zweites Mal als Kopie in eine andere Datei.

Code 4.9 Die Prozedur öffnet und speichert eine vorhandene Präsentation

```
Sub OpenPresentation()
   Dim pppDemo As Presentation

   Set pppDemo = Application.Presentations.Open _
      (FileName:="C:\Temp\Demo.pptm", ReadOnly:=msoFalse)
   With pppDemo
      With .Application.ActiveWindow
         .Left = 100
         .Top = 100
      End With
      .Save
      .SaveAs FileName:="C:\Temp\Demo2.pptm"
   End With
   pppDemo.Close
   Set pppDemo = Nothing
End Sub
```

Würde in der Methode *Open* als Parameter *ReadOnly:=msoTrue* angegeben, dann könnte die Methode *Save* nicht ausgeführt werden.

Für die Methode *Open* sind die im Anhang 4, Tab. A4.2 aufgeführten Parameter zulässig.

4.2.3 Ereignisse

Präsentationen selbst verfügen über keine Ereignisprozeduren. Doch bereits unter dem Anwendungs-Objekt haben wir einige Ereignisprozeduren zur Präsentation kennen gelernt. Wir könnten also wieder eine Klasse einrichten und hätten die Events.

An dieser Stelle möchte ich aber zeigen, dass auch ein Formular (UserForm) Klasseneigenschaften besitzt. Wir erstellen ein neues Formular mit dem Namen *clsEvents* (Abb. 4.6).

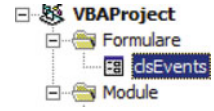


Abb. 4.6 Neues Formular um Projekt-Explorer

Mit der Anweisung

```
Public WithEvents appEvent As Application
```

im Formular ist der erste Schritt getan. In einem Codemodul kann dann wie gehabt eine Objektvariable instanziiert und terminiert werden (Code 4.10). Mit der Instanziierung stehen im Formular die Objektvariable und ihre Ereignisprozeduren zur Auswahl.

Code 4.10 Instanziierung und Terminierung einer Objektvariablen mit Events – Version 1

```
Dim PPObject As New clsEvents

Sub Auto_Open()
   Set PPObject.appEvent = Application
End Sub

Sub Auto_Close()
   Set PPObject.appEvent = Nothing
End Sub
```

Doch wir gehen noch einen Schritt weiter und laden das Formular bei der Instanziierung und entfernen es wieder mit der Terminierung (Code 4.11). Die Angabe *vbModal* = *True* gestattet die weitere Nutzung der Präsentation bei geöffnetem Formular.

Code 4.11 Instanziierung und Terminierung einer Objektvariablen mit Events – Version 2

```
Dim PPObject As New clsEvents

Sub Auto_Open()
   Set PPObject.appEvent = Application
   Load clsEvents
   clsEvents.Show vbModal = True
End Sub
```

```
Sub Auto_Close()
   Set PPObject.appEvent = Nothing
   Unload clsEvents
End Sub
```

Das Formular erhält eine Textbox *tbxEvent* mit der Eigenschaft *Multiline* = *True* und eine Schaltfläche *cmdExit* zum Löschen des Formular (Abb. 4.7).

Abb. 4.7 Formular für Event-anwendungen

Auch der Code im Formular bleibt übersichtlich (Code 4.12). Ich gebe zu, dass ich für das Auffinden der Objektkette im Objektkatalog zum Übertragen des Kommentars der aktuellen Folie an die Textbox einige Zeit benötig habe.

Code 4.12 Prozeduren im Formular zur Ausgabe des Folienkommentars

```
Public WithEvents appEvent As Application

Private Sub appEvent_SlideShowNextSlide(ByVal Wn As SlideShowWindow)
   Dim sText As String
   sText = Wn.View.Slide.NotesPage.Shapes. _
      Placeholders(2).TextFrame.TextRange.Text
   clsEvents.tbxEvent.Text = sText
End Sub

Private Sub cmdExit_Click()
   Auto_Close
End Sub

Private Sub UserForm_Activate()
   Me.Repaint
End Sub
```

Die Anwendung dieses Beispiels ist nur sinnvoll, wenn mindestens zwei Bildschirme oder die Kombination Bildschirm und Beamer vorliegen, denn während der Präsentation wird der Folienkommentar ins Formular übertragen (Abb. 4.8). Die Größe des Formulars und die Schriftgröße können den eigenen Bedürfnissen angepasst werden.

Abb. 4.8 Bildschirmaufteilung der Anwendung

4.2.4 Vorlagen

Ähnlich wie Vorlagen für ein Dokument gibt es auch Vorlagen für eine Präsentation. Es gibt jedoch keine Standardversion und alle Vorlagen müssen selbst organisiert werden. Vorlagen werden aus einer normalen Präsentation unter den Dateitypen *potx* bzw. mit Prozeduren unter *potm* gespeichert. Die Zuweisung wird mit der Methode *ApplyTemplate* durchgeführt.

Die Prozedur (Code 4.13) weist einer neuen Präsentation die Vorlage Technik zu.

Code 4.13 Die Prozedur erstellt eine neue Präsentation mit Vorlage

```
Sub NewPresentationTemplate()
   Dim pppNew As Presentation

   Set pppNew = Application.Presentations.Add
   pppNew.ApplyTemplate FileName:="C:\Temp\Technik.potm"
   Set pppNew = Nothing
End Sub
```

Da sich in der neuen Präsentation noch keine Folie befindet, ist die Vorlage nicht sichtbar, sondern erst mit dem Anlegen der ersten Folie. Die Prozedur (Code 4.14) weist einer vorhandenen Präsentation eine andere Vorlage zu. Bei dieser Vorlage wurden alle Design- und Textelemente aus dem Folienmaster entfernt und die Präsentation unter *Blank.potx* gespeichert, so dass sich eine leere Vorlage ergibt.

Code 4.14 Die Prozedur weist einer vorhandenen Präsentation eine leere Vorlage zu

```
Sub NormalPresentationTemplate()
   Dim pppNew As Presentation

   Set pppNew = ActivePresentation
   pppNew.ApplyTemplate FileName:="c:\Temp\Blank.potx"
   Set pppNew = Nothing
End Sub
```

Eine leere Vorlage hat den Vorteil, dass bei der Erstellung der Inhalt im Vordergrund steht. Daher sollte sie in keiner Vorlagensammlung fehlen.

4.3 PowerPoint-Folien

Die Folien-Objektliste *Slides* bildet die Arbeitsplattform von PowerPoint. Entsprechend besitzt dieses Objekt viele Eigenschaften und Methoden und ihre Möglichkeiten können leicht verwirren. Folien sind in Präsentationen eingebettet und werden aus ihnen angewendet.

```
'Syntax:
   Slides.(Item(IndexNr oder Name))

'Beispiel:
   Slides.Item(1) 'Erste Folie
```

Im Objektkatalog unter *Elemente des Slide-Objekts* sind alle Eigenschaften, Methode und Ereignisse aufgeführt.

4.3.1 Eigenschaften und Design

4.3.1.1 Layout

In Präsentationen sind Farben und Formen ein wichtiges Thema. Daher verfügt das Objekt über einige wichtige Eigenschaften zu diesem Thema. Eine davon ist *Layout* und im Anhang 4, Tab. A4.3 sind einige wichtige aufgeführt, alle können im Qbjektkatalog unter *Layout* aufgerufen werden. Die Prozedur (Code 4.15) setzt einige Folien-Eigenschaften.

Code 4.15 Die Prozedur setzt einige Eigenschaften der ersten Folie

```
Sub SetSlidesAttributes()
   Dim objSlide As Slide
```

```
   Set objSlide = ActivePresentation.Slides(1)
   With objSlide
      .ApplyTemplate "C:\temp\Blank.potx"
      .Layout = ppLayoutTitle
      .BackgroundStyle = msoBackgroundStylePreset1
   End With
   Set objSlide = Nothing
End Sub
```

Es kann zur Auswahl von Folien auch ein Folienbereich über das Range-Objekt angegeben werden (Code 4.16).

Code 4.16 Die Prozedur setzt den Hintergrund der Folien 2 bis 4

```
Sub SetSlidesRangeAttributs()
   Dim objSlides As SlideRange

   Set objSlides = ActivePresentation.Slides.Range(Array(2, 3, 4))
   With objSlides
      .FollowMasterBackground = msoFalse
      .Background.Fill.ForeColor.RGB = RGB(200, 255, 200) 'hellgrün
      .Layout = ppLayoutComparison
   End With
   Set objSlides = Nothing
End Sub
```

4.3.1.2 Platzhalter

In der letzten Prozedur wurden mit dem Layouttyp *ppLayoutComparison* neben der Bilddarstellung auch Text- und Grafikbereiche installiert, die vom Anwender durch Eingabe oder Auswahl gefüllt werden können. Sie fungieren somit als Platzhalter-Objekt *Placeholder* für Text, Diagramme, Tabellen, Organigramme oder andere Objekte, die ihrerseits als Formen verwaltet werden. Platzhalter bilden somit eine Untermenge der Formen.

Platzhalter werden als Objektliste *Placeholders* verwaltet und können über ihren Index ausgewählt werden (Code 4.17).

Code 4.17 Die Prozedur zeigt alle Platzhalterin Folie 2 im Direktfenster

```
Sub ReadAllPlaceholder()
   Dim objShapes As Shape

   For Each objShapes _
      In ActivePresentation.Slides(2).Shapes.Placeholders
      Debug.Print objShapes.Name
   Next
End Sub
```

Die Prozedur (Code 4.18) fügt eine neue Folie mit einem Textlayout an Position 2 ein und füllt die Platzhalter mit Text.

Code 4.18 Die Prozedur erzeugt eine zweite Folie als Inhaltsangabe

```
Sub TestPlace()
   Dim objShapes As Shapes

   Set objShapes = ActivePresentation.Slides.Add _
      (2, ppLayoutText).Shapes
   With objShapes
   'Titel
      .Title.TextFrame.TextRange.Text = "Inhaltsangabe"
   'alternativ
      .Placeholders(1).TextFrame.TextRange.Text = "Inhaltsangabe"
   'Untertitel
      .Placeholders(2).TextFrame.TextRange.Text = _
         "Item 1" & vbCrLf & "Item 2"
   End With
   Set objShapes = Nothing
End Sub
```

4.3.1.3 Hintergrund

Das Hintergrund-Objekt *Background* ist das Unterobjekt für den Folienhintergrund. Soll der Hintergrund nur für eine Folie festgelegt werden, dann muss die Übernahme aus der Masterfolie über die Eigenschaft *FollowMasterBackround* für diese Folie ausgeschaltet werden.

Die Methode *Fill* füllt den Hintergrund mit der angegebenen Farbe. Über die Eigenschaft *PresetGradient* der Methode *Fill* können auch Farbverläufe (Anhang 4, Tab. A4.4) erstellt werden.

Folien können ebenso über ihre Namen angesprochen werden. Die Prozedur (Code 4.19) vergibt zuerst Namen um sie dann anzusprechen.

Code 4.19 Die Prozedur vergibt Foliennamen und nutzt diese anschließend

```
Sub SetSlidesRangeAttributs2()
   Dim objSlides As SlideRange

   With ActivePresentation
      .Slides(2).Name = "Koala"
      .Slides(3).Name = "Pinguine"
      .Slides(4).Name = "Tulpen"
   End With
```

```
    Set objSlides = ActivePresentation.Slides. _
       Range(Array("Koala", "Tulpen"))
    With objSlides
       .FollowMasterBackground = msoFalse
       .Background.Fill.PresetGradient _
          msoGradientHorizontal, 1, msoGradientLateSunset
    End With
    Set objSlides = Nothing
End Sub
```

4.3.1.4 Designvorlagen

Neben den Entwurfsvorlagen gibt es eine Vielzahl von Designvorlage-Objekten *ApplyThemes*, die mit der Installation von PowerPoint zur Verfügung stehen. Sie befinden sich im Verzeichnis

C:\Program Files\Microsoft Office\Document Themes 14

und sind als Dateityp *.thmx gespeichert. Mit einem einfachen Anwendungsbeispiel wollen wir diese Dateien aus einer Liste auswählen und ihr Design in der Anwendung studieren. Dazu erstellen wir ein Formular *frmDesign*, das über eine Startprozedur mit *vbModal* = *True* geladen wird (Abb. 4.9).

Abb. 4.9 Formular auf einer Folie mit Design Trek

Mit der Initialisierung des Formulars wird eine Listbox mit den Dateinamen unter dem Themenverzeichnis geladen. Durch Anklicken eines Eintrags in der Listbox wird das Designthema auf die Folien der Präsentation angewendet (Code 4.20).

Code 4.20 Codezeilen des Anwendungsformulars

```
Const sTHEMEPATH As String = _
   "C:\Program Files\Microsoft Office\Document Themes 14"

Private Sub cmdExit_Click()
   Unload Me
End Sub

Private Sub lbxDesign_Click()
   Dim pppDesign  As Presentation
   Dim sSelect    As String

   Set pppDesign = ActivePresentation
   sSelect = lbxDesign.List(lbxDesign.ListIndex)
   pppDesign.ApplyTheme (sTHEMEPATH & "\" & sSelect)
End Sub

Private Sub UserForm_Initialize()
   Dim fsoObject  As Object
   Dim objFolder  As Object
   Dim objFiles   As Object
   Dim objFile    As Object

   Set fsoObject = CreateObject("Scripting.FileSystemObject")
   Set objFolder = fsoObject.getfolder(sTHEMEPATH)
   Set objFiles = objFolder.Files

   For Each objFile In objFiles
      If InStr(objFile, "") > 0 Then
         lbxDesign.AddItem objFile.Name
      End If
   Next
   Set fsoObject = Nothing
   Set objFolder = Nothing
   Set objFiles = Nothing
End Sub
```

4.3.1.5 Farbschemen

Die Methode *ApplyThemeColorScheme* wendet ein Farbschema auf die vorgegebene Präsentation an. Auch hier bietet PowerPoint bereits mit der Installation eine Vielzahl Schemen. Das Formular wird erweitert um eine zweite Liste *lbxColor* (Abb. 4.10).

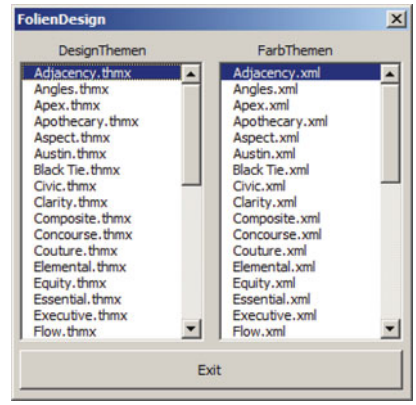


Abb. 4.10 Formular Version 2

Bei der Programmierung muss beachtet werden, dass mit der Initialisierung des Formulars und der Zuweisung Null an den Listenindex auch das Event *Click* ausgelöst wird. Dadurch bekommt die erste Liste zwar einen Index, die zweite aber erst später und es folgt eine Fehlermeldung. Mit der Abfrage, dass beide Indizes mindestens Null sein müssen, wird diesem Fehler vorgebeugt (Code 4.21).

Code 4.21 Codezeilen des Anwendungsformulars Version 2

```
Const sTHEMEPATH As String = _
   "C:\Program Files\Microsoft Office\Document Themes 14"

Const sCOLORPATH As String = _
   "C:\Program Files\Microsoft Office\Document Themes 14\Theme Colors"

Private Sub cmdExit_Click()
   Unload Me
End Sub

Private Sub lbxColor_Click()
   If lbxDesign.ListIndex >= 0 And _
      lbxColor.ListIndex >= 0 Then
      SetDesign
   End If
End Sub
```

```
Private Sub lbxDesign_Click()
   If lbxDesign.ListIndex >= 0 And _
      lbxColor.ListIndex >= 0 Then
      SetDesign
   End If
End Sub

Private Sub SetDesign()
   Dim pppDesign  As Presentation
   Dim sDesign    As String
   Dim sColor     As String
   Dim iCount     As Integer

   Set pppDesign = ActivePresentation
   sDesign = lbxDesign.List(lbxDesign.ListIndex)
   pppDesign.ApplyTheme (sTHEMEPATH & "\" & sDesign)
   sColor = lbxColor.List(lbxColor.ListIndex)
   With pppDesign
      For iCount = 1 To .Slides.Count
         .Slides(iCount).ApplyThemeColorScheme (sCOLORPATH & "\" & _
           sColor)
      Next iCount
   End With
End Sub

Private Sub UserForm_Initialize()
   Dim fsoObject  As Object
   Dim objFolder  As Object
   Dim objFiles   As Object
   Dim objFile    As Object

   Set fsoObject = CreateObject("Scripting.FileSystemObject")

   Set objFolder = fsoObject.getfolder(sTHEMEPATH)
   Set objFiles = objFolder.Files
   lbxDesign.Clear
   For Each objFile In objFiles
      If InStr(objFile, "") > 0 Then
         lbxDesign.AddItem objFile.Name
      End If
   Next
   lbxDesign.ListIndex = 0
   Set objFiles = Nothing
   Set objFolder = Nothing
```

```
   Set objFolder = fsoObject.getfolder(sCOLORPATH)
   Set objFiles = objFolder.Files
   lbxColor.Clear
   For Each objFile In objFiles
      If InStr(objFile, "") > 0 Then
         lbxColor.AddItem objFile.Name
      End If
   Next
   lbxColor.ListIndex = 0
   Set objFiles = Nothing
   Set objFolder = Nothing

   Set fsoObject = Nothing
End Sub
```

4.3.1.6 Hintergrundstil

Ein Hintergrundstil-Objekt *BackgroundStyle* ist ein Gestaltungsbeispiel für Folien. Powerpoint kennt zwölf verschiedene Stile *msoBackgroundStyle* und zusätzlich die Möglichkeiten ohne Stil und Stilkombinationen. Im Objektkatalog stehen alle Möglichkeiten.

Wir erweitern die Design-Anwendung dadurch, dass das Formular eine weitere Liste zur Auswahl der Hintergrundstile erhält (Abb. 4.11). Die neue Listbox hat den Namen *lbxGround* und entsprechend erweitert sich der Code im Formular (Code 4.22). Mit dem Anklicken eines Eintrags in einem der beiden Listenfelder erfolgt die Anwendung. In der aktuellen Liste kann auch mit den Pfeiltasten nach oben und unten eine Auswahl erfolgen.

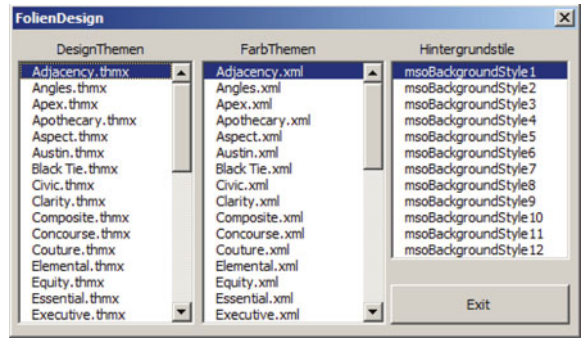


Abb. 4.11 Formular Version 3

Code 4.22 Codezeilen des Anwendungsformulars Version 3

```
Const sTHEMEPATH As String = _
   "C:\Program Files\Microsoft Office\Document Themes 14"
Const sCOLORPATH As String = _
   "C:\Program Files\Microsoft Office\Document Themes 14\Theme Colors"
```

```
Private Sub cmdExit_Click()
   Unload Me
End Sub

Private Sub lbxColor_Click()
   If lbxDesign.ListIndex >= 0 And _
      lbxColor.ListIndex >= 0 And _
      lbxGround.ListIndex >= 0 Then
      SetDesign
   End If
End Sub

Private Sub lbxDesign_Click()
   If lbxDesign.ListIndex >= 0 And _
      lbxColor.ListIndex >= 0 And _
      lbxGround.ListIndex >= 0 Then
      SetDesign
   End If
End Sub

Private Sub lbxGround_Click()
   If lbxDesign.ListIndex >= 0 And _
      lbxColor.ListIndex >= 0 And _
      lbxGround.ListIndex >= 0 Then
      SetDesign
   End If
End Sub

Private Sub SetDesign()
   Dim pppDesign As Presentation
   Dim objSlide   As Slide
   Dim sDesign    As String
   Dim sColor     As String
   Dim iCount     As Integer

   Set pppDesign = ActivePresentation
   sDesign = lbxDesign.List(lbxDesign.ListIndex)
   pppDesign.ApplyTheme (sTHEMEPATH & "\" & sDesign)
   pppDesign.Slides.Range.BackgroundStyle = lbxGround.ListIndex + 1
   sColor = lbxColor.List(lbxColor.ListIndex)
   With pppDesign
      For iCount = 1 To .Slides.Count
         .Slides(iCount).ApplyThemeColorScheme (sCOLORPATH & "\" & _
           sColor)
      Next iCount
```

```
   End With
   Set pppDesign = Nothing
End Sub

Private Sub UserForm_Initialize()
   Dim fsoObject  As Object
   Dim objFolder  As Object
   Dim objFiles   As Object
   Dim objFile    As Object
   Dim iCount     As Integer

   Set fsoObject = CreateObject("Scripting.FileSystemObject")
   Set objFolder = fsoObject.getfolder(sTHEMEPATH)
   Set objFiles = objFolder.Files
   lbxDesign.Clear
   For Each objFile In objFiles
      If InStr(objFile, "") > 0 Then
         lbxDesign.AddItem objFile.Name
      End If
   Next
   lbxDesign.ListIndex = 0
   Set objFiles = Nothing
   Set objFolder = Nothing
   Set objFolder = fsoObject.getfolder(sCOLORPATH)
   Set objFiles = objFolder.Files
   lbxColor.Clear
   For Each objFile In objFiles
      If InStr(objFile, "") > 0 Then
         lbxColor.AddItem objFile.Name
      End If
   Next
   lbxColor.ListIndex = 0
   Set objFiles = Nothing
   Set objFolder = Nothing
   Set fsoObject = Nothing
   With lbxGround
      .Clear
      For iCount = 1 To 12
         .AddItem "msoBackgroundStyle" & Trim(Str(iCount))
      Next iCount
      .ListIndex = 0
   End With
End Sub
```

4.3.1.7 Add-In Design

Da diese Anwendung immer gut zur Auswahl eines Designs verwendet werden kann, wollen wir sie als *Add-In* erstellen und mit einem Menüeintrag versehen, der mit *Auto_Open* installiert und mit *Auto_Clos*e wieder entfernt wird (Code 4.23).

Code 4.23 Menüaufruf im Add-In

```
Const sMenuName As String = "FolienDesign"

Sub Auto_Open()
   InitMenu
End Sub

Sub Auto_Close()
   RemoveMenu
End Sub

Private Sub InitMenu()
   Dim objMenuBar       As CommandBar
   Dim objMenuGroup     As CommandBarControl
   Dim objMenuButton    As CommandBarControl

   Call RemoveMenu
   Set objMenuBar = Application.CommandBars.Add _
      (sMenuName, msoBarTop)
   objMenuBar.Visible = True

   Set objMenuGroup = objMenuBar.Controls.Add _
      (Type:=msoControlPopup, Temporary:=False)
   With objMenuGroup
      .Caption = sMenuName
      .Tag = sMenuName
      .TooltipText = "Aktion wählen ..."
   End With

   Set objMenuButton = objMenuGroup.Controls.Add _
      (Type:=msoControlButton)
   With objMenuButton
      .BeginGroup = False
      .Caption = "DesignAuswahl"
      .FaceId = 417
      .OnAction = "DesignStart"
      .Style = msoButtonIconAndCaption
      .TooltipText = "DesignAuswahl"
      .Tag = "DesignAuswahl"
   End With
```

```
   Set objMenuBar = Nothing
   Set objMenuGroup = Nothing
   Set objMenuButton = Nothing
End Sub

Private Sub RemoveMenu()
   Dim objMenuBar       As CommandBar
   Dim objMenuControl   As CommandBarControl

   For Each objMenuBar In Application.CommandBars
      For Each objMenuControl In objMenuBar.Controls
         If objMenuControl.Tag = sMenuName Then
            objMenuControl.Delete
         End If
      Next
      If objMenuBar.Name = sMenuName Then
         objMenuBar.Delete
      End If
   Next
   Set objMenuBar = Nothing
   Set objMenuControl = Nothing
End Sub

Sub DesignStart()
   Load frmDesign
   frmDesign.Show vbModal = True
End Sub
```

4.3.2 Methoden

4.3.2.1 Add-Methoden

Folien werden als Objektliste verwaltet. Zur Erstellung einer neuen Folie wird die Methode *AddSlide* mit den Parametern *Index* und *Layout* angegeben. Dabei gibt der *Index* die Position der Folie in der Präsentation wieder.

```
'Syntax:
   Slides.AddSlide (Index, Layout)

'Beispiel:
   Slides.AddSlide (5, pptLayout)
```

Der Index darf nicht höher als die Anzahl Folien + 1 sein, damit die Folie in der Liste oder an deren Ende platziert wird (Code 4.24).

Code 4.24 Die Prozedur erstellt eine neue Präsentation mit der ersten Folie

```
Sub CreateNewSlide()
   Dim appPpt     As Application
   Dim pppDemo    As Presentation
   Dim objSlide   As Slide

   Set appPpt = CreateObject("PowerPoint.Application")
   appPpt.Activate
   appPpt.WindowState = ppWindowNormal
   Set pppDemo = appPpt.Presentations.Add
   With pppDemo
      .Slides.AddSlide 1, _
      .Designs(1).SlideMaster.CustomLayouts(1)
   End With
   'pppDemo.Close
   Set objSlide = Nothing
   Set pppDemo = Nothing
   Set appPpt = Nothing
End Sub
```

Wie schon bei den Platzhaltern erwähnt, werden die Elemente auf einer Folie in der Objektliste *Shapes* verwaltet. Beginnen wir mit dem Titel. Wird der Titel der ersten Folie gelöscht, dann erzeugt Prozedur (Code 4.25) einen neuen. Die Eigenschaft *HasTitle* des Shapes von *Folie1* wird zuvor abgefragt, bevor mit der Methode *AddTitle* ein neuer Titel erzeugt wird.

Code 4.25 Die Prozedur erzeugt einen neuen Titel auf der vorgegebenen Folie

```
Sub SetTitle()
   Dim pppDemo As Presentation
   Set pppDemo = ActivePresentation
   With pppDemo.Slides(1)
      If Not .Shapes.HasTitle Then
         .Shapes.AddTitle.TextFrame.TextRange.Text = _
         "PowerPoint Anwendungen"
      End If
   End With
   Set pppDemo = Nothing
End Sub
```

Mit der Methode *AddPicture* wird ein Bild einer Folie hinzugefügt (Code 4.26).

Code 4.26 Die Prozedur fügt der Folie 2 ein Bild hinzu

```
Sub SetPicture()
   Dim pppDemo As Presentation
```

```
   Set pppDemo = ActivePresentation
   With pppDemo.Slides(2)
      .Shapes.AddPicture FileName:="C:\Temp\Chrysantheme.jpg", _
      LinkToFile:=msoTrue, _
      SaveWithDocument:=msoTrue, _
      Left:=200, Top:=200, Width:=200, Height:=200
   End With
   Set pppDemo = Nothing
End Sub
```

Mit der Methode *AddMediaObject* wird ein Medien-Objekt einer Folie hinzugefügt (Code 4.27).

Code 4.27 Die Prozedur fügt der Folie 2 ein Videoclip hinzu

```
Sub SetMedia()
   Dim pppDemo As Presentation
   Set pppDemo = ActivePresentation
   With pppDemo.Slides(2)
      .Shapes.AddMediaObject FileName:="C:\Temp\Example.avi", _
      Left:=200, Top:=200, Width:=200, Height:=200
   End With
   Set pppDemo = Nothing
End Sub
```

Mit der Methode *AddTextEffect* wird ein WordArt-Objekt einer Folie hinzugefügt. Die Prozedur (Code 4.28) zeigt alle dreißig Effektformen.

Code 4.28 Die Prozedur fügt der Folie 2 alle 30 WordArt-Effektformen hinzu

```
Sub SetWordArt()
   Dim pppDemo As Presentation
   Dim iLoop   As Integer
   Set pppDemo = ActivePresentation
   With pppDemo.Slides(2)
      For iLoop = 0 To 29
         .Shapes.AddTextEffect _
         PresetTextEffect:=iLoop, _
         Text:="Demo", FontName:="Arial Black", FontSize:=14, _
         FontBold:=False, FontItalic:=msoFalse, _
         Left:=200, Top:=16 * iLoop
      Next iLoop
   End With
   Set pppDemo = Nothing
End Sub
```

Weitere Add-Methoden lassen sich im Objektkatalog finden und testen.

4.3.2.2 Folienanordnung

Neben der Schreibweise *Slides(i)* gibt es noch die Form *Slides.Item(i).* Beide bezeichnen ein Folienobjekt an i-ter Stelle der Objektliste. Durch Einfügen und Löschen wird die Position einer Folie verändert.

In der Prozedur (Code 4.29) wird eine bestimmte Folie mit der Methode *Delete* entfernt.

Code 4.29 Die Prozedur löscht die Folie 2 in der aktuellen Präsentation

```
Sub SlideDelete()
   Dim pppDemo    As Presentation
   Dim objSlide   As Slide
   Set pppDemo = ActivePresentation
   Set objSlide = pppDemo.Slides.Item(2)
   objSlide.Delete
   Set objSlide = Nothing
   Set pppDemo = Nothing
End Sub
```

Mit der Methode *Copy* können Folien in die Zwischenablage gespeichert und mit der Methode *Paste* aus dieser in die aktuelle Präsentation wieder eingefügt werden (Code 4.30).

Code 4.30 Die Prozedur fügt eine Kopie der Folie 2 an Position 4 ein

```
Sub SlideCopyPaste()
   Dim pppDemo    As Presentation
   Dim objSlide   As Slide

   Set pppDemo = ActivePresentation
   Set objSlide = pppDemo.Slides.Item(2)
   objSlide.Copy
   pppDemo.Slides.Paste Index:=4

   Set objSlide = Nothing
   Set pppDemo = Nothing
End Sub
```

Mit der Methode *Range* können eine beliebige Anzahl von Formen oder Folien angesprochen werden. Das wurde zuvor schon in Kombination mit der *Array*-Funktion gezeigt. Die Prozedur (Code 4.31) entfernt die Folien 2 und 4, um sie danach in der neuen Konstellation an Position 3 zusammen wieder einzufügen.

Code 4.31 Die Prozedur verschiebt Folien an eine neue Position

```
Sub SlideCut()
   Dim pppDemo    As Presentation

   Set pppDemo = ActivePresentation
   pppDemo.Slides.Range(Array(2, 4)).Cut
   pppDemo.Slides.Paste 3
   Set pppDemo = Nothing
End Sub
```

Die Methode *InsertFromFile* erlaubt den Import von Folien aus einer anderen Präsentation.

```
Print Syntax:
   ActivePresentation.InsertFromFile(Filename, Index, SlideStart, _
     SlideEnd)

Print Beispiel:
   ActivePresentation.InsertFromFile "C:\Temp\Demo.pptx", 3, 2, 4
```

Die Prozedur (Code 4.32) fügt die Folien von Position 2 bis 4 aus einer externen Präsentation an die Position 3 der aktuellen Präsentation ein.

Code 4.32 Die Prozedur import Folien aus einer externen Präsentation

```
Sub SlidesImport()
   Dim pppDemo    As Presentation

   Set pppDemo = ActivePresentation
   pppDemo.Slides.InsertFromFile "C:\Temp\Demo.pptm", 3, 2, 4
   Set pppDemo = Nothing
End Sub
```

4.3.2.3 Präsentation ausführen

Präsentationen sind für die Vorführung auf einem Bildschirm oder Beamer gedacht. Ihre Ausführung startet die Methode *Run*. Mit der Methode *SlideShowSettings* wird die Ausführung eingestellt.

Die Prozedur (Code 4.33) öffnet eine Präsentation aus einem Verzeichnis heraus. Mit der Einstellung *WithWindows:=msoFalse* bleibt sie im Hintergrund unsichtbar. Die Folien sind durchlaufend.

Code 4.33 Die Prozedur startet eine Präsentation aus einem Verzeichnis heraus

```
Sub SlidesOpenShow()
   Dim pppDemo As Presentation

   Set pppDemo = Presentations.Open( _
      FileName:="C:\Temp\Demo.pptm", _
      WithWindow:=msoFalse)
   With pppDemo.SlideShowSettings
      .RangeType = ppShowNamedSlideShow
      .ShowType = ppShowTypeSpeaker
      .Run
   End With
   Set pppDemo = Nothing
End Sub
```

Für die Eigenschaft *ShowType* gibt es drei Möglichkeiten, die im Anhang 4, Tab. A4.5 stehen.

In der Prozedur (Code 4.34) wird die aktuelle Präsentation in einem eigenen Fenster gestartet und es werden die Folien 2 bis 5 wiedergegeben. Zusammen mit der Methode *Run* wird die Darstellung des Fensters gesetzt.

Code 4.34 Die Prozedur startet einen Teil die aktuelle Präsentation in einem Fenster

```
Sub SlideShowWindow()
   Dim pppDemo As Presentation

   Set pppDemo = ActivePresentation
   With pppDemo.SlideShowSettings
      .RangeType = ppShowSlideRange
      .StartingSlide = 2
      .EndingSlide = 5
      .ShowType = ppShowTypeWindow
      With .Run
         .Left = 100
         .Top = 100
         .Width = 400
         .Height = 300
      End With
   End With
   Set pppDemo = Nothing
End Sub
```

Mit der Eigenschaft *SlideShowTransition* wird der Übergang zwischen den Folien verwaltet. Die Prozedur (Code 4.35) setzt zuerst für alle Folien den Übergang auf 5 Sekunden mit einem Akustiksignal. Der Folienwechsel erfolgt von oben rechts nach unten links. Die Eigenschaft *AdvanceMode* hat nach Anhang 4, Tab. A4.6 drei mögliche Werte.

Hat die Eigenschaft *LoopUntilStopped* den Wert *msoTrue*, startet die Präsentation erneut bis die Taste *ESC* gedrückt wird. Die Eigenschaft *PointerType* legt den Zeigertyp fest, der bei der Präsentation verwendet wird. Mögliche Werte zeigt Anhang 4, Tab. A4.7.

Code 4.35 Die Prozedur setzt Übergänge und führt die Präsentation aus

```
Sub SlideShowWithTransition()
   Dim pppDemo As Presentation
   Dim lLoop   As Long

'setze Übergänge
   Set pppDemo = ActivePresentation
   For lLoop = 1 To pppDemo.Slides.Count
      With ActivePresentation.Slides(lLoop).SlideShowTransition
         .Speed = ppTransitionSpeedFast
         .EntryEffect = ppEffectStripsDownLeft
         .SoundEffect.ImportFromFile "C:\Temp\Gong.wav"
         .AdvanceOnTime = True
         .AdvanceTime = 5
      End With
   Next lLoop
   pppDemo.SlideShowSettings.AdvanceMode = _
         ppSlideShowUseSlideTimings

'starte Show
   With pppDemo.SlideShowSettings
      .RangeType = ppShowSlideRange
      .StartingSlide = 1
      .EndingSlide = pppDemo.Slides.Count
      .ShowType = ppShowTypeWindow
      .LoopUntilStopped = msoTrue
      With .Run
         .Left = 100
         .Top = 100
         .Width = 400
         .Height = 300
         With .View
            .PointerType = ppSlideShowPointerArrow
            .PointerColor.RGB = RGB(255, 0, 0)
         End With
      End With
   End With
   Set pppDemo = Nothing
End Sub
```

4.3.2.4 Folien drucken

Die Druckausgabe von Präsentationen oder Folien wird mit der Methode *PrintOut* durchgeführt.

```
Print Syntax:
   Presentation.PrintOut(From, To, PrintToFile, Copies, Collate)

Print Beispiel:
   ActivePresentation.PrintOut From:=1, To:=3, Copies:=2, _
     Collate:=msoFalse
```

Die Eigenschaft *FitToPage* = *msoTrue* passt die Foliengröße der Seitengröße an. Die Eigenschaft *FrameSlides* des *PrintOption*-Objekts gibt der Folie im Druck einen Rahmen. Mit der Eigenschaft *PrintColorType* wird die Farbe des Drucks festgelegt. Mögliche Parameter zeigt im Anhang 4, Tab. A4.8.

Mit dem Eigenschaftswert *PrintFontsAsGraphic* = *msoTrue* werden Schriften als Grafik gedruckt, vorausgesetzt der Drucker lässt dies zu. Wird die Eigenschaft *PrintHiddenSlides* = *msoTrue* gesetzt, dann werden auch ausgeblendete Folien mit gedruckt. Die Eigenschaft *PrintBackground* legt fest, ob der Druck im Hintergrund erfolgen soll, ohne Angabe ist Standard. Die Form des Drucks bestimmt die Eigenschaft *OutputType*. Deren mögliche Parameter zeigt Anhang 4, Tab. A4.9.

Die Prozedur (Code 4.36) druckt die gesamte Präsentation in eine Datei.

Code 4.36 Die Prozedur druckt die aktive Präsentation

```
Sub PrintOutSlides()
   Dim pppDemo As Presentation

   Set pppDemo = ActivePresentation
   With pppDemo.PrintOptions
      .FitToPage = msoTrue
      .FrameSlides = msoTrue
      .PrintColorType = ppPrintColor
      .PrintFontsAsGraphics = msoTrue
      .OutputType = ppPrintOutputSixSlideHandouts
      .PrintHiddenSlides = msoTrue
      .PrintInBackground = msoTrue
   End With
   pppDemo.PrintOut _
      From:=1, To:=pppDemo.Slides.Count, _
      PrintToFile:="C:\Temp\PrintOut", _
      Copies:=1, Collate:=msoFalse
   Set pppDemo = Nothing
End Sub
```

4.4 PowerPoint-Master

Die Master-Objekte bilden die Basis für die Gestaltung aller Objekte in PowerPoint, also für Folien, Titel, Handzettel, Notizen. Entsprechend der Grundeinstellung liegt ein *Master* vor und kann über das Objekt abgefragt werden, wie zum Beispiel der *Folien Master* für Folien.

```
'Syntax:
   Presentation.SlideMaster.Attribut

'Beispiel:
   ActivePresentation.SlideMaster.Background.Fill
```

4.4.1 Folien-Master

Das Folien-Master-Objekt *SlideMaster* ist ein Unterobjekt des Präsentations-Objekts und wird darüber angesprochen. Er besitzt die Design-Eigenschaften und -Methoden die wir bereits von den Folien kennen, nur als gesamte Vorgabe für den Aufbau einer Präsentation. Natürlich können dann einzelne Folien (oder alle) immer noch eigene Designwerte zugewiesen bekommen (Code 4.37).

Code 4.37 Die Prozedur setzt Eigenschaften des Folien-Masters

```
Sub SlideMasterDesign()
   Dim pppDemo As Presentation

   Set pppDemo = ActivePresentation
   With pppDemo.SlideMaster
      .ApplyTheme THEMENAME
      With .Background.Fill
         .PresetGradient Style:=msoGradientDiagonalUp, _
            Variant:=1, PresetGradientType:=msoGradientGold
         .PresetTextured PresetTexture:=msoTextureGranite
      End With
   End With
   Set pppDemo = Nothing
End Sub
```

4.4.2 Titel-Master

Das Titel-Master-Objekt *TitleMaster* verfügt neben den Design-Elementen auch über Objekte wie Kopf- und Fußzeilen. Mit der Eigenschaft *HasTitleMaster* kann die Existenz eines Titel-Masters abgefragt werden.

Die Prozedur (Code 4.38) prüft die Existenz des Titel-Masters. Ist er nicht vorhanden, dann wird er hinzugefügt.

Code 4.38 Die Prozedur prüft und setzt den Titel-Master

```
Sub SetTitleMaster()
   Dim pppDemo As Presentation

   Set pppDemo = ActivePresentation
   With pppDemo
      If Not .HasTitleMaster Then .AddTitleMaster
   End With
   Set pppDemo = Nothing
End Sub
```

Die Prozedur (Code 4.39) zeigt die Handhabung von Textbereichen im Titel-Master.

Code 4.39 Die Prozedur schreibt einen Fußtext in den Titel-Master

```
Sub SetFooterText()
   Dim pppDemo As Presentation

   Set pppDemo = ActivePresentation
   With pppDemo
      If .HasTitleMaster Then
         .TitleMaster.HeadersFooters _
         .Footer.Text = "Fußtext"
      End If
   End With
   Set pppDemo = Nothing
End Sub
```

4.4.3 Handzettel-Master

Für das Handzettel-Master-Objekt *HandoutMaster* gibt es ebenso Eigenschaften zum Design. Die Prozedur (Code 4.40) setzt Farben für den Hintergrund und fügt ein Bild aus einem Verzeichnis formatfüllend ein.

Code 4.40 Die Prozedur setzt den Hintergrund des Handzettel-Masters

```
Sub SetHandoutMaster()
   Dim pppDemo As Presentation

   Set pppDemo = ActivePresentation
   With pppDemo.HandoutMaster.background
      .Fill.ForeColor.RGB = RGB(255, 255, 255)
      .Fill.BackColor.SchemeColor = ppAccent1
      .Fill.UserPicture "C:\Temp\Logo.jpg"
   End With
   Set pppDemo = Nothing
End Sub
```

4.4.4 Notizen-Master

Auch für das Notizen-Master-Objekt *NotesMaster* sind im Objektkatalog Design-Elemente vorgegeben. Interessanter ist jedoch die Arbeit mit den Notizen direkt. Es gibt oft den Fall, dass der Inhalt der Folien als Kopie hier abgelegt wird, um ihn dann weiter zu kommentieren. Die Prozedur (Code 4.41) übernimmt diesen Vorgang und nutzt das *CommandBarControl*-Objekt mit der Bezeichnung *Notizenlayout...*, identifizierbar mit der Id=700, zur Bestimmung der Übernahme von Bild und/oder Text (Abb. 4.12). Damit dies schrittweise für jede Folie passiert, wird ein Formular *frmControl* (Abb. 4.13) mit Schaltflächen genutzt (Code 4.42).

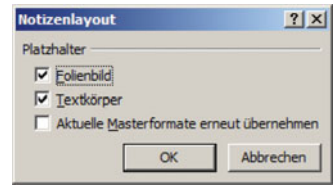


Abb. 4.12 Dialogfeld Notizenlayout

Abb. 4.13 Formular zur Steuerung

Code 4.41 Prozedur kopiert Folieninhalte in Notizseiten

```
Sub ApplyMasterToNotes()
   Dim pppDemo    As Presentation
   Dim ctlFind    As CommandBarControl
   Dim objSlide   As Slide

   Set ctlFind = CommandBars.FindControl(Id:=700)
   Debug.Print ctlFind.Caption
   If (ctlFind Is Nothing) Then
      MsgBox "Layout Control nicht vorhanden!"
      Exit Sub
   End If

   ActiveWindow.ViewType = ppViewNotesPage
   Set pppDemo = ActivePresentation
   For Each objSlide In pppDemo.Slides
      ActiveWindow.View.GotoSlide (objSlide.SlideIndex)
      Do
         ctlFind.Execute
         Load frmControl
         frmControl.Show vbModal
      Loop While iSwitch = 2
   Next
End Sub
```

Code 4.42 Ereignisprozeduren der Schaltflächen im Formular

```
Private Sub cmdExit_Click()
   End
End Sub

Private Sub cmdOk_Click()
   iSwitch = 1
   Unload Me
End Sub

Private Sub cmdReturn_Click()
   iSwitch = 2
   Unload Me
End Sub
```

4.5 PowerPoint-Interaktionen

An zwei Beispielen wird das Zusammenspiel von Powerpoint mit Excel und Word erklärt.

4.5.1 Foliengenerierung aus Excel

Wie alle Beispiele in diesem Buch liefere ich kein fertiges Konzept, sondern ich gebe die ersten Schritte vor, die dann der Leser weiterentwickeln kann, wenn er denn möchte. Gerade dieses Beispiel eignet sich besonders für eigene Übungen.

In einer Excelmappe sind Jahresumsätze aufgeführt, die bereits auch als Chart vorliegen (Abb. 4.14).

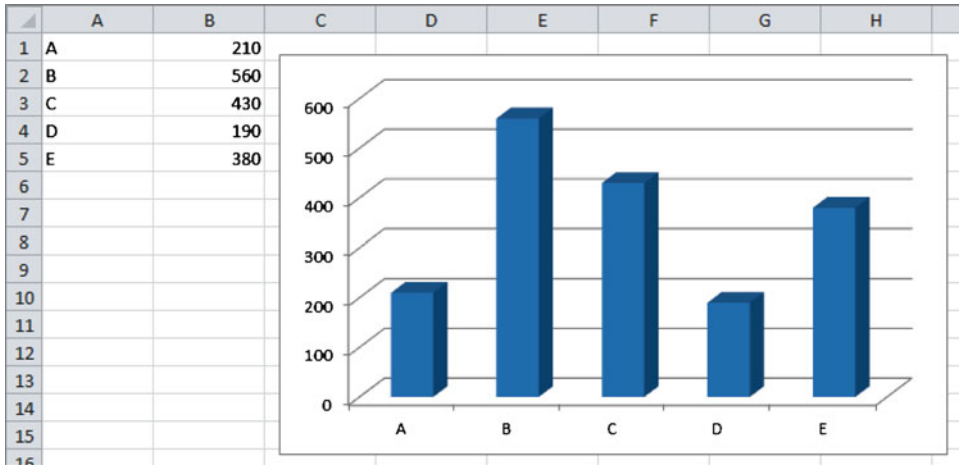


Abb. 4.14 Umsatzbeispiel

Aus der Excel-Anwendung heraus soll eine Folie mit der Umsatzgrafik erstellt werden. Der Registername des Excel-Arbeitsblattes dient als Überschrift in der Folie. Wenn es noch keine Präsentation gibt, dann soll sie neu erstellt werden.

Die Prozedur (Code 4.43) erstellt eine Folie immer am Ende der Präsentation, kopiert das Diagramm mit der Methode *Copy* in die Zwischenablage und fügt es dann mit der Methode *Paste* in die Folie ein. Voraussetzung dabei ist, dass das jeweilige Arbeitsblatt mit dem Diagramm aktiviert ist. So lassen sich verschiedene Arbeitsblätter mit unterschiedlichen Diagrammen Schritt für Schritt in die Präsentation übertragen. Wer mag, kann die Erstellung des Diagramms mit einbinden. Ebenso kann die Darstellung des Diagramms auf der Folie noch abgeändert werden. Als Layout wurde die sechste Layoutform im SlideMaster (Abb. 4.15) ausgewählt. So lassen sich auch noch andere Layoutformen mit unterschiedlichen *Placeholder*-Objekten ansprechen.

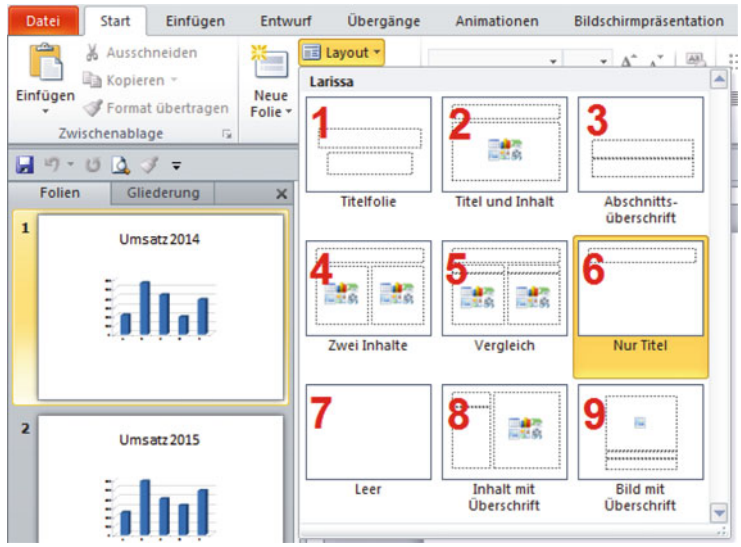


Abb. 4.15 Layoutformen im SlideMaster

Code 4.43 Die Prozedur erstellt aus Excel heraus PowerPoint-Folien

```
Sub ChartToPowerPoint()
   Dim exlBook     As Excel.Workbook
   Dim exlSheet    As Excel.Worksheet
   Dim pptApp      As PowerPoint.Application
   Dim pptShow     As PowerPoint.Presentation
   Dim pptSlide    As PowerPoint.Slide
   Dim sPath       As String

   sPath = "C:\Temp\Umsatz.pptx"

'PP instanziieren
   Set pptApp = CreateObject("Powerpoint.Application")
   With pptApp
      .Visible = msoTrue
      .WindowState = ppWindowNormal
      .Activate
   'PPP öffnen oder erstellen
      If sPath = "" Then
         Set pptShow = .Presentations.Add
      Else
```

```
         If Dir(sPath) = "" Then
            Set pptShow = .Presentations.Add
         Else
            Set pptShow = .Presentations.Open(sPath, msoFalse)
         End If
      End If
   End With

'Folie als letzte Folie erstellen und Text zuweisen
   Set pptSlide = pptShow.Slides.AddSlide( _
      pptShow.Slides.Count + 1, pptShow.SlideMaster.CustomLayouts(6))
   pptSlide.Select

'Chart in die Zwischenablage kopieren
   Set exlBook = ThisWorkbook
   Set exlSheet = exlBook.ActiveSheet
   pptSlide.Shapes(1).TextFrame.TextRange = exlSheet.Name
   exlSheet.ChartObjects(1).Copy

'Chart aus der Zwischenablage einfügen
   pptShow.Application.ActiveWindow.View.Paste

'PPP speichern und schließen
   pptShow.SaveAs (sPath)
   pptApp.Quit
   Set exlBook = Nothing
   Set exlSheet = Nothing
   Set pptShow = Nothing
   Set pptApp = Nothing
End Sub
```

Eine weitere Prozedur ist hilfreich, wenn es um die Layoutgestaltung geht. Sie steht in einem Codemodul der aktuellen Präsentation. Die Prozedur (Code 4.44) liest in einer vorhandenen Präsentation die Folien und gibt deren Namen und Layout im Direktfenster aus, Ebenso die auf den Folien vorhandenen *Placeholder*-Objekte.

Code 4.44 Die Prozedur gibt die Attribute vorhandener Folien im Direktfenster aus

```
Sub ReadAllPlaceholder()
   Dim pptSlide   As Slide
   Dim pptShape   As Shape
```

```
   For Each pptSlide In ActivePresentation.Slides
      With pptSlide
         Debug.Print .Name, .Layout, .CustomLayout.Name
         For Each pptShape In .Shapes.Placeholders
            Debug.Print "-", pptShape.Name
         Next
      End With
   Next
End Sub
```

4.5.2 Präsentation mit Word und Excel erstellen

In diesem Beispiel wird die Bearbeitung einer neuen Präsentation über Formulare gesteuert und die Anbindung an Word und Excel erlaubt die Übernahme von Text und Diagrammen. Im ersten Schritt wird ein Menü-Formular (Abb. 4.16) z. B. mit *ALT-F8* durch die Prozedur *StarteAnwendung* (Code 4.45) aufgerufen.

Abb. 4.16 Menü-Formular und Vorlage der Präsentation

Code 4.45 Start des Menü-Formulars in einem Codemodul

```
Sub StarteAnwendung()
    frmHauptmenue.Show
End Sub
```

Das Menü-Formular beinhaltet vier Schaltflächen (Code 4.46).

Code 4.46 Die Ereignisprozeduren des Menü-Formulars

```
Private Sub cmdBeenden_Click()
   Unload Me
End Sub

Private Sub cmdDiagrammFolie_Click()
   frmDiagramm.Show
End Sub

Private Sub cmdTextFolie_Click()
   frmText.Show
End Sub

Private Sub cmdTitelFolie_Click()
   frmTitelfolie.Show
End Sub
```

Zur Bearbeitung der Titelfolie, die im Template bereits vorhanden ist, können weitere Folien angehängt werden. Es wird zwischen Folien mit Text und solchen mit einem Diagramm unterschieden. Die Anwendung wird mit dem Ereignis *cmdBeenden_Click* geschlossen.

Die Bearbeitung beginnt mit der Titelfolie und dem Aufruf des Bearbeitungsformulars (Abb. 4.17) für die Titelfolie. Es übernimmt den vorhandenen Untertext aus der Folie und speichert mit Beenden die neue Eingabe zurück in die Folie (Code 4.47).

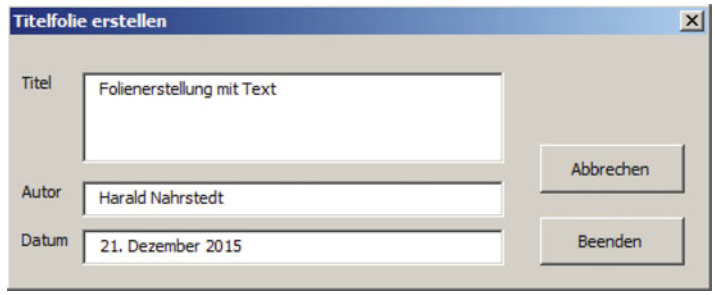


Abb. 4.17 Bearbeitungsformular für die Titelfolie

Code 4.47 Die Ereignisprozeduren des Titelfolien-Bearbeitungsformulars

```
Private Sub UserForm_Activate()
  Me.tbxAutor = ActivePresentation.BuiltInDocumentProperties("Author")
  Me.tbxDatum = Format(Date, "dd. mmmm yyyy")
  Me.tbxTitel.SetFocus
End Sub
```

```
Private Sub cmdErstellen_Click()
  With ActivePresentation.Slides(1)
     .Shapes(1).TextFrame.TextRange.Text = Me.tbxTitel
     .Shapes(2).TextFrame.TextRange.Text = _
        Me.tbxAutor & ", " & Me.tbxDatum
  End With
  Me.Hide
End Sub

Private Sub cmdAbbrechen_Click()
  Me.Hide
End Sub
```

Das Formular für eine Folie mit Text (Abb. 4.18) übernimmt einen markierten Text aus dem aktuellen Word-Dokument (Code 4.48).

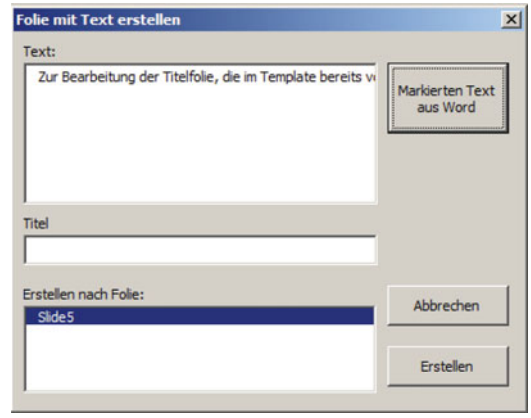


Abb. 4.18 Bearbeitungsformular für eine neue Folie mit Text

Code 4.48 Die Ereignisprozeduren des Formulars für eine neue Folie mit Text

```
Sub cmdAbbrechen_Click()
  Me.Hide
End Sub

Private Sub cmdErstellen_Click()
  Dim pptNew As PowerPoint.Slide

  Set pptNew = ActivePresentation.Slides.Add _
    (Me.lbxSlides.ListIndex + 2, ppLayoutText)
  With pptNew
    .Shapes(1).TextFrame.TextRange.Text = Me.tbxTitle
    .Shapes(2).TextFrame.TextRange.Text = Me.tbxText
```

```
    .Shapes(2).AnimationSettings.TextUnitEffect = ppAnimateByParagraph
    .Shapes(2).AnimationSettings.EntryEffect = ppEffectFlyFromBottom
  End With
  pptNew.Select
  UserForm_Activate
  Set pptNew = Nothing
End Sub

Private Sub cmdTextAusWord_Click()
  Dim wrdApp As Object

  On Error GoTo ErrorHandling
  Set wrdApp = GetObject(, "Word.Application")
  If wrdApp.Selection.Type = wdSelectionIP Then
    MsgBox "Bitte markieren Sie den gewünschten Text!", vbExclamation
  End If
  Me.tbxText = wrdApp.Selection.Text
  Set wrdApp = Nothing
  Exit Sub
ErrorHandling:
  If Err = 429 Then
    If MsgBox("Word ist noch nicht aktiv! " & _
       "Soll es gestartet werden?", vbQuestion + vbYesNo) = vbYes Then
       Set wrdApp = CreateObject("Word.Application")
       wrdApp.Visible = True
       Set wrdApp = Nothing
    End If
  End If
End Sub

Private Sub UserForm_Activate()
  Dim pptSLide As PowerPoint.Slide
  Me.tbxText = ""
  Me.tbxTitle = ""
  Me.lbxSlides.Clear
  For Each pptSLide In ActivePresentation.Slides
    Me.lbxSlides.AddItem pptSLide.Name
  Next
  If Me.lbxSlides.ListCount > 0 Then
    Me.lbxSlides.Selected(Me.lbxSlides.ListCount - 1) = True
  End If
End Sub
```

Ein Titeltext ist frei eingebbar. Sind bereits mehrere Folien erstellt worden, dann kann auch die Position der neuen Folie bestimmt werden.

Ebenso gibt es ein Formular für eine neue Folie mit einem Diagramm (Abb. 4.19) und die entsprechenden Ereignisprozeduren (Code 4.49).

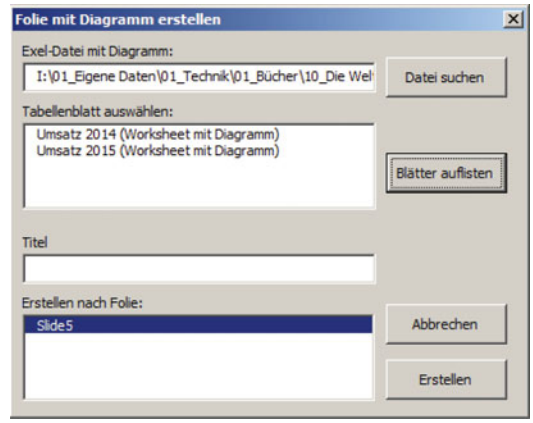


Abb. 4.19 Bearbeitungsformular für eine neue Folie mit Diagramm

Code 4.49 Die Ereignisprozeduren des Formulars für eine neue Folie mit Diagramm

```
Private Sub cmdAbbrechen_Click()
   Me.Hide
End Sub

Private Sub cmdSheets_Click()
   Dim exlBook    As Excel.Workbook
   Dim exlTab     As Object
   Dim sText      As String

'Excel öffnen wenn möglich
   If Dir(Me.tbxPfad) = "" Then
      MsgBox "Die angegeben Datei existiert nicht!", vbCritical
      Exit Sub
   End If
   Set exlBook = Excel.Application.Workbooks.Open(Me.tbxPfad)
'Liste aller Sheets füllen
   Me.lbxSheets.Clear
   For Each exlTab In exlBook.Sheets
      sText = exlTab.Name
      If TypeName(exlTab) = "Worksheet" Then
         If exlTab.ChartObjects.Count > 0 Then
            sText = sText & " (Worksheet mit Diagramm)"
         Else
            sText = sText & " (Worksheet)"
         End If
```

```
      Else
          sText = sText & " (Sheet)"
      End If
      Me.lbxSheets.AddItem sText
   Next
   Set exlTab = Nothing
   Set exlBook = Nothing
End Sub

Private Sub cmdErstellen_Click()
   Dim exlBook    As Excel.Workbook
   Dim exlSheet   As Object
   Dim pptSLide   As Slide
   Dim lLeft      As Long
   Dim lTop       As Long
   Dim lWidth     As Long
   Dim lHeight    As Long
   Dim iDiaNr     As Integer
   Dim sText      As String

'Excel öffnen wenn möglich
   If Dir(Me.tbxPfad) = "" Then
      MsgBox "Die angegeben Datei existiert nicht!", vbCritical
      Exit Sub
   End If
   Set exlBook = Excel.Application.Workbooks.Open(Me.tbxPfad)

'Folie erstellen und füllen
   Set pptSLide = ActivePresentation.Slides.Add _
      (Me.lbxSlides.ListIndex + 2, ppLayoutChart)
   With pptSLide
      .Shapes(1).TextFrame.TextRange.Text = Me.tbxTitle
      .Select

   'Maße des zweiten Placeholders merken und ihn löschen
      lLeft = .Shapes(2).Left
      lTop = .Shapes(2).Top
      lWidth = .Shapes(2).Width
      lHeight = .Shapes(2).Height
      .Shapes(2).Delete

   'Diagramm bestimmen
      Set exlSheet = exlBook.Sheets(Me.lbxSheets.ListIndex + 1)
      exlSheet.Activate
```

```
      If exlSheet.Type = xlWorksheet Then
         iDiaNr = 1
         If exlSheet.ChartObjects.Count > 1 Then
            sText = "Das Tabellenblatt enthält " & _
             Str(exlSheet.ChartObjects.Count) & " Diagramme." & vbCrLf
            sText = sText & "Bitte geben Sie die Nummer " & _
               "des gewünschten Diagramms ein (1 bis " & _
               exlSheet.ChartObjects.Count & "):"
            iDiaNr = InputBox(sText, "Diagramm auswählen", 1)
            If (iDiaNr <= 0) Or _
               (iDiaNr > exlSheet.ChartObjects.Count) Then
               Exit Sub
            End If
         End If
         exlSheet.ChartObjects(iDiaNr).Select
         ActiveChart.ChartArea.Copy
      Else
         exlSheet.ChartArea.Copy
      End If
      .Shapes.Paste
      .Shapes(2).Left = lLeft
      .Shapes(2).Top = lTop
      .Shapes(2).Width = lWidth
      .Shapes(2).Height = lHeight
   End With
   exlBook.Close
   UserForm_Activate
   Set pptSLide = Nothing
   Set exlSheet = Nothing
   Set exlBook = Nothing
End Sub

Private Sub cmdSuchen_Click()
   Dim fdlSearch As FileDialog
   Set fdlSearch = Application.FileDialog(msoFileDialogFilePicker)
   With fdlSearch
      .Filters.Add "Excel-Dateien", "*.xlsx", 1
      .Filters.Add "Excel-Dateien", "*.xlsm", 2
      .Filters.Add "Alle Dateien", "*.*", 3
      .InitialView = msoFileDialogViewDetails
      .Title = "Excel-Workbook zum Diagramm-Import wählen!"
      .Show
      If .SelectedItems.Count > 0 Then
          Me.tbxPfad = .SelectedItems(1)
      End If
```

```
   End With
   Set fdlSearch = Nothing
End Sub

Private Sub UserForm_Activate()
   Dim pptSLide As Slide
   Me.lbxSlides.Clear
   For Each pptSLide In ActivePresentation.Slides
      Me.lbxSlides.AddItem pptSLide.Name
   Next
   If Me.lbxSlides.ListCount > 0 Then
      Me.lbxSlides.Selected(Me.lbxSlides.ListCount - 1) = True
   End If
   Me.lbxSheets.Clear
End Sub

Private Sub UserForm_Deactivate()
   On Error Resume Next
   MappeXl.Close SaveChanges:=False
   Set MappeXl = Nothing
End Sub
```

Outlook-Objekte

5

Outlook ist ein ideales Programm zur Selbstorganisation und zum Arbeiten im Team. Ausgestattet mit einem ewigen Kalender erlaubt es die Verwaltung von Terminen, Kontakten und Aufgaben. Es ermöglicht den Zugriff auf E-Mail-Konten und damit den schriftlichen und bildlichen Austausch von Informationen.

5.1 Outlook-Anwendungen

Das Anwendungs-Objekt *Application* ist auch hier das oberste Objekt (Abb. 5.1). Es bildet für die Anwendungsbereiche den Container für deren Objektlisten. Daher gibt es wenige Eigenschaften und mehr Methoden zu betrachten.

Das Outlook-Fenster lässt sich mehrfach öffnen, so wie bei anderen Anwendungen auch. Doch im Projektexplorer ist immer nur eine Anwendung zu finden.

Die drei wichtigsten Unterobjekte bzw. Objektlisten sind *NameSpace*, *Explorers* und *Inspectors*. Die Objektlisten *Explorers* und *Inspectors* verwalten Objekte der Oberfläche, wobei *Explorers* die Verzeichniselemente und *Inspectors* die Formularelemente beinhalten, die aktiv genutzt werden. Über *NameSpace* kann ein direkter Zugriff auch auf Objekte erfolgen, die nicht in der Oberfläche liegen.

H. Nahrstedt, *Die Welt der VBA-Objekte*, DOI 10.1007/978-3-658-13891-2_5

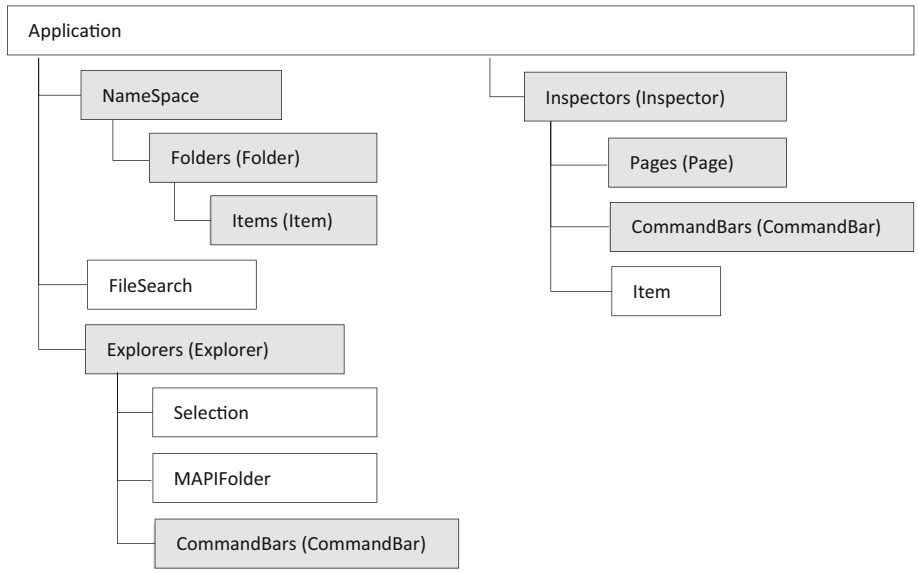


Abb. 5.1 Die wichtigsten Unterobjekte und Objektlisten (*grau*) des Application-Objekts

5.1.1 Eigenschaften

Die wenigen Eigenschaften der Anwendung sind oft schreibgeschützt und bieten nur Abfragemöglichkeiten (Code 5.1).

Code 5.1 Die Prozedur liefert einige Einstellungswerte

```
Sub OutlookInfo()
   With Application
      Debug.Print .DefaultProfileName
      Debug.Print .Explorers.Count
      Debug.Print .Inspectors.Count
      Debug.Print .Name
      Debug.Print .ProductCode
      Debug.Print .Version
   End With
End Sub
```

5.1.2 Methoden

In Outlook gibt es drei Methoden, um die aktiven Objekt anzusprechen. Die aktive Ordnerübersicht kann mit der Methode *ActiveExplorer* ausgewertet werden. Zum Testen der Prozedur (Code 5.2) sollte ein Mailordner wie zum Beispiel Posteingang geöffnet und einige Einträge markiert sein.

Code 5.2 Die Prozedur nennt alle Absender der markierten Mails

```
Sub OutlookActiveExplorer()
   Dim olkExplor  As Explorer
   Dim olkSelect  As Selection
   Dim sText      As String
   Dim iLoop      As Integer

   sText = "Selektierte Einträge von: "
   Set olkExplor = Application.ActiveExplorer
   Set olkSelect = olkExplor.Selection
   For iLoop = 1 To olkSelect.Count
      sText = sText & vbCrLf & _
         olkSelect.Item(iLoop).SenderName
   Next iLoop
   Debug.Print sText
   Set olkExplor = Nothing
   Set olkSelect = Nothing
End Sub
```

Zum aktiven Formular bietet die Methode *ActiveInspector* den Zugang. Für das Testen der Prozedur (Code 5.3) muss vor dem Start eine Mail geöffnet sein. Die Prozedur gibt deren Bodytext im Direktfenster aus und schließt die Mail.

Code 5.3 Die Prozedur gibt den Bodytext einer geöffneten Mail aus

```
Sub OutlookItemClose()
   Dim olkInspec  As Inspector
   Dim olkItem    As MailItem

   Set olkInspec = Application.ActiveInspector
   Set olkItem = olkInspec.CurrentItem
   Debug.Print olkItem.Body
   olkItem.Close olSave
   Set olkInspec = Nothing
   Set olkItem = Nothing
End Sub
```

Das aktive Fenster *ActiveWindow* ist die dritte Methode eine Darstellung anzusprechen. Die Prozedur (Code 5.4) prüft, ob das aktuelle Fenster ein Formular vom Type *Inspector* ist. In diesem Fall wird es als Vollbild dargestellt, falls es nicht bereits diesen Status besitzt.

Code 5.4 Die Prozedur schaltet ein Formular im Fenster auf Vollbild

```
Sub OutlookActiveWindow()
   Dim olkWindow  As Object

   Set olkWindow = Application.ActiveWindow
   Debug.Print TypeName(olkWindow)
   If TypeName(olkWindow) = "Inspector" Then
      olkWindow.WindowState = olMaximized
   End If
   Set olkWindow = Nothing
End Sub
```

Die Methode *AdvancedSearch* ist eine Möglichkeit, ein beliebiges Element in Outlook zu suchen. Die im Objektkatalog angegebene Prozedur ist leider fehlerhaft. Ein Anwendungsbeispiel folgt unter Ereignisse. Die Methode *CreateItem* erlaubt das Anlegen neuer Elemente. Zur Auswahl stehen die im Anhang 5, Tab. A5.1 aufgeführten *Items*.

Dazu drei einfache Beispiele. Die Prozedur (Code 5.5) erzeugt ein E-Mail-Formular und setzt einige Eigenschaften auf Werte. Für die Eigenschaft *HTMLBody* werden Sprechelemente von *HTML* verwendet.

Code 5.5 Die Prozedur erzeugt ein E-Mail-Formular

```
Sub CreateHTMLMail()
   Dim appOut  As Application
   Dim objMail As MailItem

   Set appOut = Application
   Set objMail = appOut.CreateItem(olMailItem)
   With objMail
      .BodyFormat = olFormatHTML
      .HTMLBody = "<HTML><H2>Überschrift</H2>" & _
         "<BODY>Der Mailtext wird in HTML dargestellt!</BODY></HTML>"
      .Display
      .To = "Mailadresse"
      .CC = "Kopie an Mailadresse"
      .Subject = "TEST"
   End With
   Set objMail = Nothing
   Set appOut = Nothing
End Sub
```

Die Prozedur (Code 5.6) erzeugt ein Aufgabenformular und setzt ebenfalls einige Eigenschaften.

Code 5.6 Die Prozedur erzeugt eine neue Aufgabe

```
Sub CreateTask()
   Dim appOut  As Application
   Dim objTask As TaskItem

   Set appOut = Application
   Set objTask = appOut.CreateItem(olTaskItem)
   With objTask
      .Subject = "Aufgabentitel"
      .DueDate = "30.10.2015"
      .Role = "Name" 'zuständig ist ...
      .Save
   End With
   Set objTask = Nothing
   Set appOut = Nothing
End Sub
```

Die Prozedur (Code 5.7) erzeugt einen neuen Meeting-Termin.

Code 5.7 Die Prozedur erzeugt einen neuen Meeting-Termin

```
Sub CreateDate()
   Dim appOut     As Application
   Dim objDate    As AppointmentItem
   Dim objReqAtt  As Recipient 'erforderliche Teilnehmer
   Dim objOptAtt  As Recipient 'optionale Teilnehmer
   Dim objResAtt  As Recipient 'Ressourcen

   Set appOut = Application
   Set objDate = appOut.CreateItem(olAppointmentItem)
   With objDate
      .MeetingStatus = olMeeting
      .Subject = "Status Meeting"
      .Location = "Raum 123"
      .Start = "10.30.2015 13:30"
      .Duration = 60
      Set objReqAtt = objDate.Recipients.Add("Hans Müller")
      objReqAtt.Type = olRequired
      Set objOptAtt = objDate.Recipients.Add("Gerd Schmidt")
      objReqAtt.Type = olOptional
      .Display
      .Send
```

```
   End With
   Set objReqAtt = Nothing
   Set objOptAtt = Nothing
   Set objResAtt = Nothing
   Set objDate = Nothing
   Set appOut = Nothing
End Sub
```

Die Methode *Quit* schließt die Outlook-Anwendung.

5.1.3 Ereignisse

Ereignis-Prozeduren im Anwendungs-Objekt von Outlook lassen sich über das Klassenmodul *ThisOutlookSession* anwenden. Im Codefenster ist das Objekt *Application* anwählbar und daneben befinden sich seine Ereignisse.

Das erste davon heißt *AdvancedSearchComplete* und ist bei der Methode *AdvancedSearch* sehr hilfreich. Mitunter brauchen Auswertungen in Outlook ihre Zeit und ein Suchvorgang ist so eine Methode. Im Beispiel (Code 5.8) wird ein Suchvorgang mit einer Prozedur gestartet ohne das Ergebnis abzuwarten. Mit der oben genannten Ereignisprozedur liegt dann das Ergebnis vor.

Code 5.8 Codezeilen in ThisOudlookSession für einen Suchvorgang

```
Private Sub Application_AdvancedSearchComplete(ByVal SearchObject _
  As Search)
   Debug.Print SearchObject.Tag, "endet um " & Time
   Debug.Print SearchObject.Filter
   Debug.Print "Anzahl Items: " & SearchObject.Results.Count
End Sub

Public Sub GetSearch(objStore As Store, sSearch As String)
   Dim objFolder  As Folder
   Dim objSearch  As Search
   Dim sFind      As String
   Set objFolder = objStore.GetRootFolder
   sFind = """urn:schemas:httpmail:subject""" & _
      " LIKE '%" & sSearch & "%' OR " & _
      """urn:schemas:httpmail:textdescription""" & _
      " LIKE '%" & sSearch & "%'"
```

```
   Debug.Print sFind
   Set objSearch = Application.AdvancedSearch( _
      Scope:=GetScope(objStore), _
      Filter:=sFind, _
      SearchSubFolders:=True, _
      Tag:=objStore.FilePath & " - " & sSearch)
   Debug.Print objSearch.Tag, "startet um " & Time
   Set objFolder = Nothing
   Set objSearch = Nothing
End Sub

Function GetScope(objStore As Store) As String
   Dim objRoot    As Folder
   Dim objFolder  As Folder
   Dim sScope     As String
   Set objRoot = objStore.GetRootFolder
   For Each objFolder In objRoot.Folders
      sScope = sScope & ",'" & objFolder.FolderPath & "'"
   Next
   GetScope = Mid(sScope, 2)
   Set objRoot = Nothing
End Function

Sub StartSearch()
   Dim appOut     As Application
   Dim sSearch    As String
   sSearch = "springer"
   Set appOut = Application
   GetSearch appOut.Session.DefaultStore, sSearch
   Set appOut = Nothing
End Sub
```

Die Startprozedur *StartSearch* benutzt das Unterobjekt *Store* der Outlook Anwendung. Bei dem Objekt *Store* handelt es sich um eine Datei auf dem lokalen Computer. Über dieses Objekt ist der Zugriff auf alle Ordner und Suchordner der aktuellen Sitzung *Session* möglich, wie in der Prozedur *GetScope*.

So ausgerüstet startet der Suchvorgang über *AdvancedSearch* in *GetSearch*.

Die Ereignisse *Startup* und *Quit* entsprechen den Ereignissen *Auto_Open* und *Auto_Close* anderer Anwendungen. In ihnen lassen sich die Anweisungen für ein eigenes Menü installieren. Beispiele liegen ja bereits vor.

Sollen im Klassenmodul *ThisOutlookSession* Ereignisprozeduren untergeordneter Objekte genutzt werden, so geht dies wieder über die Definition einer Objektvariablen mit Events. Zum Beispiel definiert die Anweisung

```
Private WithEvents objExplorer As Explorer
```

eine Objektvariable vom Typ Explorer und mit der Anweisung

```
Set objExplorer = Application.ActiveExplorer
```

in einer Prozedur stehen auch die Ereignisse des Explorer-Objekts zur Verfügung.

5.2 Outlook NameSpace

Die Aufgabe des Objekts *Namespace* ist es, Strukturen von Ordnern und Verzeichnissen zu verwalten.

5.2.1 Eigenschaften

Über das Anwendungs-Objekt *Application* gibt es zwei Möglichkeiten auf ein *NameSpace* Objekt zuzugreifen, über die Eigenschaft *Session* oder die Methode *GetNameSpace*.

Die Prozedur (Code 5.9) benutzt die Eigenschaft *Session* um Informationen über alle angemeldeten Konten zu geben. Da die Datenmenge oft mehr ist als das Direktfenster fassen kann, werden sie in eine Textdatei geschrieben und anschließend mit dem Texteditor *Notepad* angezeigt. Die Anweisung *On Error Resume Next* sorgt dafür, dass es keine Fehlermeldung gibt, wenn eine Eigenschaft nicht gefunden wird.

Code 5.9 Die Prozedur liefert Informationen zu aktuellen Konten

```
Public Sub ReadSessionAccounts()
   Dim objShell         As Object
   Dim fsoDat           As Object
   Dim objDatei         As Object
   Dim sTextFile        As String
   Dim objSession       As NameSpace
   Dim Report           As String
   Dim objAccounts      As Accounts
   Dim objAccount       As Account
   Dim objCategory  As Category

   Set objShell = CreateObject("WScript.Shell")
   Set fsoDat = CreateObject("Scripting.FileSystemObject")
```

```
sTextFile = "C:\Temp\Text.txt"
Set objDatei = fsoDat.CreateTextFile(sTextFile, True)

Set objSession = Application.Session
Set objAccounts = objSession.Accounts

On Error Resume Next
For Each objAccount In objAccounts
   With objAccount
      objDatei.Write "Account:" & vbCrLf
      objDatei.Write "   Type: " & .AccountType & vbCrLf
      objDatei.Write "   Auto Discover Connection Mode: " & _
         .AutoDiscoverConnectionMode & vbCrLf
      objDatei.Write "   Class: " & .Class & vbCrLf
      objDatei.Write "   Display Name: " & .DisplayName & vbCrLf
      objDatei.Write "   Exchange Connection Mode: " & _
         .ExchangeConnectionMode & vbCrLf
      objDatei.Write "   Exchange Mailbox Server Name: " & _
         .ExchangeMailboxServerName & vbCrLf
      objDatei.Write "   Exchange Mailbox Server Version: " & _
         .ExchangeMailboxServerVersion & vbCrLf
      objDatei.Write "   SMTP Address: " & .SmtpAddress & vbCrLf
      objDatei.Write "   User Name: " & .UserName & vbCrLf
      With .CurrentUser
         objDatei.Write "Current User:" & vbCrLf
         objDatei.Write "   Type: " & .Type & vbCrLf
         objDatei.Write "   Name: " & .Name & vbCrLf
         objDatei.Write "   Index: " & .Index & vbCrLf
         objDatei.Write "   Address: " & .Address & vbCrLf
         objDatei.Write "   Display Type: " & .DisplayType & vbCrLf
         objDatei.Write "   Auto Response: " & .AutoResponse & _
           vbCrLf
         objDatei.Write "   Meeting Response Status: " & _
            .MeetingResponseStatus & vbCrLf
         objDatei.Write "   Resolved: " & .Resolved & vbCrLf
         objDatei.Write "   Sendable: " & .Sendable & vbCrLf
         objDatei.Write "   Tracking Status: " & .TrackingStatus _
           & vbCrLf
         objDatei.Write "   Tracking Status Time: " & _
            .TrackingStatusTime & vbCrLf
      End With
      If .DeliveryStore.Categories.Count > 0 Then
         For Each objCategory In .DeliveryStore.Categories
            objDatei.Write "Delivery Store Category:" & vbCrLf
            With objCategory
```

```
                objDatei.Write "   Name: " & .Name & vbCrLf
                objDatei.Write "   ID: " & .CategoryID & vbCrLf
                objDatei.Write "   Shortcut Key: " & .ShortcutKey _
                  & vbCrLf
                objDatei.Write "   Border Color: " & _
                   .CategoryBorderColor & vbCrLf
                objDatei.Write "   Color: " & .Color & vbCrLf
                objDatei.Write "   Gradient Bottom Color: " & _
                   .CategoryGradientBottomColor & vbCrLf
                objDatei.Write "   Gradient Top Color: " & _
                   .CategoryGradientTopColor & vbCrLf
             End With
          Next
       End If
       objDatei.Write "Delivery Store:" & vbCrLf
       With .DeliveryStore
          objDatei.Write "   Class: " & .Class & vbCrLf
          objDatei.Write "   Store ID: " & .StoreID & vbCrLf
          objDatei.Write "   Exchange Store Typ: " & _
             .ExchangeStoreType & vbCrLf
          objDatei.Write "   File Path :" & .FilePath & vbCrLf
          objDatei.Write "   Is Cached Exchange: " & _
             .IsCachedExchange & vbCrLf
          objDatei.Write "   Is Conversation Enabled: " & _
             .IsConversationEnabled & vbCrLf
          objDatei.Write "   Is Data File Store: " & _
             .IsDataFileStore & vbCrLf
          objDatei.Write "   Is Instant Search Enabled: " & _
             .IsInstantSearchEnabled & vbCrLf
          objDatei.Write "   Is Open: " & .IsOpen & vbCrLf
       End With
       objDatei.Write "Auto Discover Xml: " & .AutoDiscoverXml & _
         vbCrLf
       objDatei.Write "----------------------------------------" _
         & vbCrLf
    End With
 Next
 objDatei.Close
 objShell.Run "notepad.exe c:\Temp\Text.txt"
 Set objAccounts = Nothing
 Set objSession = Nothing
 Set objDatei = Nothing
 Set fsoDat = Nothing
 Set objShell = Nothing
End Sub
```

5.2.2 Methoden

Statt der Eigenschaft *Application.Session* kann auch die Methode *GetNameSpace* angewendet werden. Beide Wege

```
Dim objSession As NameSpace
Set objSession = Application.GetNamespace("MAPI")
Set objSession = Application.Session
```

liefern ein *NameSpace* Objekt. Zur Bestätigung liefert die Prozedur (Code 5.10) über *GetNameSpace* die Eigenschaft *DisplayName* der aktuellen Konten.

Code 5.10 Die Prozedur liefert alle Konten mit DisplayNames

```
Sub ReadAccountsDisplayNames()
   Dim objSession    As NameSpace
   Dim objAccounts   As Accounts
   Dim objAccount    As Account

   Set objSession = Application.GetNamespace("MAPI")
   Set objAccounts = objSession.Accounts
   For Each objAccount In objAccounts
      Debug.Print objAccount.DisplayName
   Next
   Set objSession = Nothing
   Set objAccounts = Nothing
End Sub
```

Das *NameSpace* Objekt selbst verfügt über einige *Get*-Methoden die im Anhang 5, Tab. A5.2 aufgeführt sind.

Die Prozedur (Code 5.11) liefert als einfaches Beispiel den Standard-Kalendernamen.

Code 5.11 Verwendung des Unterobjekts Folder von NameSpace

```
Sub ReadNameSpaceFolderName()
   Dim objSession    As NameSpace
   Dim objFolder     As Folder

   Set objSession = Application.GetNamespace("MAPI")
   Set objFolder = objSession.GetDefaultFolder(olFolderCalendar)
   Debug.Print objFolder.Name
   Set objFolder = Nothing
   Set objSession = Nothing
End Sub
```

Die Prozedur (Code 5.12) liefert den Namen des *Root-Items* aller Folder. Es ist nicht verwunderlich, dass der Name *Outlook* lautet.

Code 5.12 Die Prozedur liefert den Namen des Root-Folders

```
Sub ReadRootItemName()
   Dim objSession    As NameSpace
   Dim objFolder     As Folder

   Set objSession = Application.GetNamespace("MAPI")
   Set objFolder = objSession.Folders.Item(1)
   Debug.Print objFolder.Name
   Set objFolder = Nothing
   Set objSession = Nothing
End Sub
```

Nach dem Objekt-Modell von Outlook ist das Unterobjekt von *NameSpace* das Ablage-Objekt *Store* und dessen Unterobjekt das Ordner-Objekt *Folder*. Beide Unterobjekte liegen als Objektliste vor. Mit der Prozedur (Code 5.13) werden alle Ordner entsprechend der aufgezeigten Hierarchie gelesen. Die Startprozedur *ReadFolderInStores* benutzt zur Auswertung der Unterordner die rekursive Prozedur *ReadSubFolder*.

Code 5.13 Die Prozedur liest alle Folder der aktiven Sitzung von Outlook

```
Sub ReadFolderInStores()
   Dim objStores     As Stores
   Dim objStore      As Store
   Dim objRoot       As Folder

   Set objStores = Application.Session.Stores
   For Each objStore In objStores
      Set objRoot = objStore.GetRootFolder
      Debug.Print "--------------------------------"
      Debug.Print "DisplayName:"; objStore.DisplayName
      Debug.Print "RootFolder: "; objRoot.FolderPath
      ReadSubFolders objRoot
      Set objRoot = Nothing
   Next
   Set objStores = Nothing
End Sub

Private Sub ReadSubFolders(ByVal objRoot As Folder)
   Dim objFolders    As folders
   Dim objFolder     As Folder
   Static bLoop      As Boolean
```

```
   If bLoop = True Then
      Set objFolders = objRoot.folders
      If objFolders.Count > 0 Then
         For Each objFolder In objFolders
            Debug.Print objFolder.FolderPath
            ReadSubFolders objFolder
         Next
      End If
      Set objFolders = Nothing
   End If
   bLoop = True
End Sub
```

In der rekursiven Prozedur sorgt eine statische Variable vom Typ *Boolean* dafür, dass das *RootFolder* nicht mehrfach ausgewertet wird.

5.2.3 Ereignisse

Das Objekt *NameSpace* wartet mit nur zwei Ereignissen auf. Zur Anschauung wird das Ereignis *OptionsPagesAdd* in der Prozedur (Code 5.14) angewendet. Dieses Ereignis tritt immer auf, wenn im Kontextmenü eines Ordners die Auswahl *Eigenschaften* gewählt wird. Die Funktionalität ist somit leicht zu testen.

Code 5.14 Codezeilen im Klassenmodul ThisOutlookSession

```
Dim WithEvents objSession As NameSpace

Private Sub Application_Startup()
   Set objSession = Session
End Sub

Private Sub objSession_OptionsPagesAdd _
   (ByVal Pages As PropertyPages, ByVal Folder As MAPIFolder)
   Dim sText As String
   sText = "PageClass: " & Pages.Parent & vbCrLf
   sText = sText & "FolderPath: " & Folder.FolderPath
   MsgBox sText, vbOKOnly, "OptionPagesAdd"
End Sub
```

5.3 Outlook-Explorer

Das Outlook-Objekt *Explorer* ist ein Fenster, in dem der Inhalt eines Ordners dargestellt wird.

5.3.1 Eigenschaften

Über die Eigenschaft *CurrentFolder* ist der Zugriff auf den Ordner möglich, den das Fenster wiedergibt. Sind mehrere Outlook-Fenster mit unterschiedlichen Ansichten geöffnet, dann ergeben sich auch mehrere Ordner (Code 5.15).

Code 5.15 Die Prozedur zeigt alle aktiven Ordner

```
Sub ReadAllExplorers()
   Dim objExplorers As Explorers
   Dim objExplorer  As Explorer

   Set objExplorers = Application.Explorers
   For Each objExplorer In objExplorers
      Debug.Print objExplorer.CurrentFolder.Name
   Next
   Set objExplorers = Nothing
End Sub
```

Die Eigenschaft *CurrentView* ist eine Wissenschaft für sich, denn der Filter wird über SQL-Anweisungen bzw. DASL-Abfragen gesteuert. Viele Informationen dazu liefert der Objektkatalog. Die Prozeduren (Code 5.16) liefern eine Filteranwendung auf eine Maillliste.

Code 5.16 Ein- und Ausschalt-Prozedur für einen Maillisten-Filter

```
'Filter einschalten
Private Sub CurrentViewFilter()
    Dim objView As View

    Set objView = Application.ActiveExplorer.CurrentView
    objView.Filter = _
      "%last7days(""urn:schemas:httpmail:datereceived"")%"
    objView.Save
    objView.Apply
    Set objView = Nothing
End Sub
```

```
'Filter ausschalten:
Private Sub ResetCurrentView()
   Dim objView As View

   Set objView = Application.ActiveExplorer.CurrentView
   objView.Reset
   objView.Apply
   Set objView = Nothing
End Sub
```

Mögliche Syntax-Beispiele für den Filter zeigt im Anhang 5, Tab. A5.3.

Neben der Eigenschaft *Filter* sind auch verschiedene Design-Eigenschaften ansprechbar. Die Prozedur (Code 5.17) setzt einige Eigenschaften der Kalenderdarstellung, vorausgesetzt sie ist aktiv.

Code 5.17 Die Prozedur setzt die Kalenderdarstellung auf eigene Werte

```
Sub SetCalendarView()
   Dim objView As View

   Set objView = Application.ActiveExplorer.CurrentView
   If objView.ViewType = olCalendarView Then
      With objView
         .CalendarViewMode = olCalendarViewMultiDay
         .DaysInMultiDayMode = 14
         .DayWeekFont.Name = "Verdana"
         .DayWeekFont.Size = 8
         .DayWeekTimeFont.Name = "Verdana"
         .DayWeekTimeFont.Size = 16
         .Save
         .Apply
      End With
   End If
   Set objView = Nothing
End Sub
```

Mögliche Konstante für den *CalendarViewMode* zeigt im Anhang 5, Tab. A5.4.

5.3.2 Methoden

Die Methode *Search* führt eine Sofortsuche auf dem Ordner des aktiven Explorers durch. Der Vorgang entspricht dem Suchstart auf der Benutzeroberfläche. Mit der Methode *ClearSearch* wird die Darstellung wieder zurückgesetzt (Code 5.18).

Code 5.18 Die Prozedur startet eine Sofortsuche im aktiven Explorer

```
Sub SearchInActiceExplorer()
   Dim objExplorer   As Explorer
   Dim sSearch       As String

   Set objExplorer = Application.ActiveExplorer
   sSearch = InputBox("Suchbegriff: ", "Sofortsuche")
   objExplorer.Search sSearch, olSearchScopeCurrentFolder
   MsgBox "Weiter mit OK", vbOKOnly
   objExplorer.ClearSearch
   Set objExplorer = Nothing
End Sub
```

Die Methoden *Pane* und *IsPaneVisible* verwalten die Darstellung des aktiven Explorers (Code 5.19).

Code 5.19 Die Prozedur schaltet die Darstellung des aktiven Explorers ein und aus

```
Sub ToggleViewExplorer()
   Dim objExplorer As Explorer

   Set objExplorer = Application.ActiveExplorer
   objExplorer.ShowPane olPreview, _
        Not objExplorer.IsPaneVisible(olPreview)
   Set objExplorer = Nothing
End Sub
```

Über die Eigenschaft *Selection* können Elemente im aktiven Explorer ausgewählt werden, die die Methode *RemoveFromSelection* wieder aufhebt. Die Prozedur (Code 5.20) setzt das Löschdatum der markierten Maileinträge auf den nächsten Tag und hebt die Markierung auf.

Code 5.20 Die Prozedur setzt das Löschdatum markierter Mails auf den nächsten Tag

```
Public Sub SetExpiryTime()
   Dim objExplorer   As Explorer
   Dim objSelection  As Selection
   Dim objSelect     As MailItem
   Dim dTime         As Date

   Set objExplorer = Application.ActiveExplorer
   Set objSelection = objExplorer.Selection
   If objSelection.Count = 0 Then
      MsgBox "Keine Mails markiert!"
```

```
   Else
      dTime = Now + 1
      For Each objSelect In objSelection
         With objSelect
            Debug.Print .Subject, .ReceivedTime
            .ExpiryTime = dTime
            objExplorer.RemoveFromSelection objSelect
         End With
      Next
   End If
   Set objSelection = Nothing
   Set objExplorer = Nothing
End Sub
```

5.3.3 Ereignisse

Die Ereignisse des aktiven Explorers können über eine Objektvariable mit Vererbung der Events im Klassenmodul *ThisOutlookSession* abgefragt werden. Die Prozedur (Code 5.21) liefert zum Event *SelectionChange* den Namen des aktiven Ordners.

Code 5.21 Codezeilen im Klassenmodul ThisOutlookSession für Explorer-Ereignisse

```
Public WithEvents objExplorer As Explorer

Private Sub Application_Startup()
   Set objExplorer = Application.ActiveExplorer
End Sub

Private Sub objExplorer_SelectionChange()
   Debug.Print objExplorer.CurrentFolder.Name
End Sub
```

Der aktive Explorer ist es auch, der das Menüband besitzt und so kann über ihn auch das Unterobjekt *CommandBar* angesprochen werden (Code 5.22). Aufgerufen wird die eigene Menü-Prozedur aus der Ereignisprozedur *Application_Startup*.

Code 5.22 Die Prozedur erzeugt einen eigenen Menüeintrag

```
Sub InitMenu()
   Dim objMenuBar    As CommandBar
   Dim objMenuHead   As CommandBarControl
   Dim objMenuGroup  As CommandBarControl
   Dim objMenuButton As CommandBarControl
   Dim Control
```

```
    'RemoveMenu

    Set objMenuBar = Application.ActiveExplorer.CommandBars("Tools")
'HeadList
    Set objMenuHead = objMenuBar.Controls.Add _
        (Type:=msoControlPopup, Temporary:=True)
    With objMenuHead
        .Caption = "Outlook Tools"
        .TooltipText = "Gruppe wählen ..."
    End With
'SubList
    Set objMenuGroup = objMenuHead.Controls.Add _
        (Type:=msoControlPopup, Temporary:=True)
    With objMenuGroup
        .Caption = "Kalender"
        .TooltipText = "Aktion wählen ..."
    End With
'Button
    Set objMenuButton = objMenuGroup.Controls.Add _
        (Type:=msoControlButton, Temporary:=True)
    With objMenuButton
        .BeginGroup = True
        .Caption = "14 Tage Ansicht"
        .FaceId = 59
        .OnAction = "CalendarView14Days" 'eigene Prozedur
        .Style = msoButtonIconAndCaption
        .TooltipText = "14 Tage Ansicht"
    End With
End Sub
```

Eine *Remove*-Prozedur wie in anderen Anwendungen ist hier nicht erforderlich, da hier die Anweisung *Temporary:=True* ebenfalls wirksam ist.

5.4 Outlook-Inspector

Die Objektliste *Inspectors* verwaltet alle geöffneten Formulare des aktiven *Explorers*.

5.4.1 Eigenschaften

Die Eigenschaft *Count* ist neben dem Zugriff auf das Anwendungs-Objekt eine der wenigen Eigenschaften. Sind einige Formulare wie z. B. E-Mails geöffnet, dann liefert die Prozedur (Code 5.23) deren Kopfzeilen.

Code 5.23 Die Prozedur zeigt die Kopfzeilen der geöffneten Formulare

```
Private Sub CountInspectors()
   Dim objInspect As Inspectors
   Dim iLoop      As Integer
   Dim iCount     As Integer

   Set objInspect = Inspectors
   iCount = objInspect.Count
   If iCount > 0 Then
      For iLoop = 1 To iCount
         Debug.Print objInspect.Item(iLoop).Caption
       Next iLoop
   Else
       MsgBox "Keine Formulare geöffnet."
   End If
   Set objInspect = Nothing
End Sub
```

Über das Objekt *ActiveInspector* mit der Eigenschaft *CurrentItem* erschließt sich der Inhalt eines Formulars (Code 5.24).

Code 5.24 Die Prozedur gibt den Inhalt einer Mail aus

```
Private Sub ReadMailItem()
   Dim objInspect As Inspector
   Dim objMail    As MailItem
   Set objInspect = Application.ActiveInspector
   If Not TypeName(objInspect) = "Nothing" Then
      Set objMail = objInspect.CurrentItem
      With objMail
         Debug.Print "Betreff: " & .Subject
         Debug.Print "An: " & .To
         Debug.Print "Copy an: &.CC"
         Debug.Print "Text: " & .Body
         Debug.Print "Kategorie: " & .Categories
         Debug.Print "Sender: " & .Sender
      End With
      Set objMail = Nothing
   End If
   Set objInspect = Nothing
End Sub
```

Um dieses Beispiel erfolgreich zu testen, muss eine Mail geöffnet sein.

Formulare besitzen neben Eigenschaften auch *Controls* (Steuerelemente). Sie werden über die Eigenschaft *ModifiedFormPages* angesprochen. Diese besitzt eine Objektliste *Controls*.

5.4.2 Methoden

Auch die Anzahl der Methoden ist sehr begrenzt. Die Methode *Add* erzeugt ein neues Formular, dessen Typ durch einen Parameter bestimmt wird. Anhang 5, Tab. A5.5 zeigt eine Übersicht.

Die Prozedur (Code 5.25) erzeugt ein neues Formular vom Typ *Posteingang* und stellt darin die erste Mail des Ordners ein. Erst mit der Methode *Display* wird das Formular auch dargestellt. Dies erinnert an die Formular-Objekte mit den Methoden *Load* und *Show*.

Code 5.25 Die Anwendung der Methoden Add und Display

```
Private Sub CreateFormular()
   Dim objInspect As Inspector
   Dim sMailNr    As String

   sMailNr = 1
   Set objInspect = Inspectors.Add( _
      Session.GetDefaultFolder(olFolderInbox).Items(sMailNr))
   objInspect.Display
   Set objInspect = Nothing
End Sub
```

Mit der Methode *Close* kann ein Formular geschlossen werden. Die Prozedur (Code 5.26) schließt das aktive Formular ohne es zu speichern.

Code 5.26 Die Prozedur schließt das aktive Formular

```
Private Sub CloseFormular()
   Dim objInspect As Inspector

   Set objInspect = Application.ActiveInspector
   objInspect.Close olDiscard
   Set objInspect = Nothing
End Sub
```

5.4.3 Ereignisse

Wie schon gezeigt, können Ereignisse von Objekten über Objektvariable und der Anweisung *WithEvents* im Klassenmodul *ThisOutlookSession* angelegt werden.

Die Prozedur (Code 5.27) sendet eine Abfrage, wenn ein geöffnetes Formular in seiner Größe verändert werden soll. Die Antwort *Ja* setzt das Formular in einen Ziehmodus, die Antwort *Nein* beendet die Prozedur ohne Veränderung. Genutzt wird dazu der Parameter *Cancel* der Ereignisprozedur. Damit die Ereignisprozedur aktiv wird, muss die Objektvariable instanziiert werden. Diese Aufgabe übernimmt die Prozedur *InspectorInstanziierung*

Code 5.27 Die Codezeilen instanziieren eine Ereignisprozedur zur Formularänderung

```
Public WithEvents objInspector As Inspector

Public Sub InspectorInstanziierung()
   Set objInspector = Application.ActiveInspector
End Sub

Private Sub objInspector_BeforeSize(Cancel As Boolean)
   Dim lReturn As Long

   lReturn = MsgBox("Veränderung zulassen? ", vbYesNo)
   If lReturn = vbYes Then
      Cancel = False
   Else
      Cancel = True
   End If
End Sub
```

Verliert ein Formular seinen Aktivstatus, so kann dies über das Ereignis *Deactivate* abgefangen werden. Wird z. B. ein Mailformular geschlossen, dann liefert die Prozedur (Code 5.28) die entsprechende Betreff-Zeile.

Code 5.28 Die Prozedur schließt das aktive Formular

```
Private Sub objInspector_Deactivate()
   Debug.Print objInspector.Caption
End Sub
```

Ändert sich die Seite des aktiven Formulars, dann tritt das Ereignis *PageChange* auf (Code 5.29).

Code 5.29 Die Ereignisprozedur liefert den Namen der aktiven Seite

```
Private Sub objInspector_PageChange(ActivePageName As String)
   Debug.Print ActivePageName
End Sub
```

5.5 Outlook-Einzelobjekte

Die Element-Objekte *Items* sind die eigentlichen Arbeitsobjekte in Outlook. Es kann sich dabei um Mails, Termine, Kontakte, Aufgaben und andere Objekte handeln. Wir haben sie bereits genutzt, doch hier soll noch einmal auf die wichtigsten eingegangen werden.

5.5.1 Mail-Objekte

Das Mail-Objekt *MailItem* verfügt über eine Vielzahl von Eigenschaften, Methoden und Ereignissen. Sie sind im Objektkatalog hinreichend erklärt. Erzeugt wird ein Mail-Objekt mit der Methode *CreateItem* und ist ein Formular. Entsprechend muss es mit der Methode *Display* sichtbar werden und die Methode *Send* sendet die Mailnachricht (Code 5.30).

Code 5.30 Die Prozedur erstellt und (sendet) eine Mailnachricht

```
Sub CreateMailItem()
   Dim objItem As MailItem

   On Error Resume Next
   Set objItem = Application.CreateItem(olMailItem)
   With objItem
      .Subject = "Testbeispiel MailItem"
      .To = "EmpfängerMailAdresse"
      .CC = "KopieAnMailAdressen"
      .Attachments.Add = "C:\Temp\Test.txt"
      .Body = "Hier steht die Mitteilung!"
      .Categories = "Geschäftlich"
      .Importance = olImportanceNormal
      .Display
      '.Send
   End With
   Set objItem = Nothing
End Sub
```

Der Zugriff auf ein MailItem kann über das *NameSpace* Objekt erfolgten. Die Prozedur (Code 5.31) liest die neunte Mail im Posteingang.

Code 5.31 Die Prozedur liest die neunte Mail im Posteingang

```
Sub DisplayMail()
   Dim objSpace   As NameSpace
   Dim objItem    As MAPIFolder
   Dim objFolder  As Folder
   Set objSpace = Application.GetNamespace("MAPI")
```

```
   Set objFolder = objSpace.GetDefaultFolder(olFolderInbox)
   objFolder.Display
   Set objItem = objFolder.Items(9)
   objItem.Display
   Set objSpace = Nothing
   Set objFolder = Nothing
End Sub
```

Und es können auch vereinfachte Suchläufe programmiert werden. Die Prozedur (Code 5.32) sucht im Ordner *Posteingang* nach Mails die im Betreff das *Suchwort* haben.

Code 5.32 Die Prozedur sucht in allen Mails im Posteingang nach einem Wort in Betreff

```
Sub SearchInSubjects()
   Dim objSpace   As NameSpace
   Dim objItem    As MailItem
   Dim objFolder  As Folder
   Dim sSearch    As String

   sSearch = "Suchwort"
   Set objSpace = Application.GetNamespace("MAPI")
   Set objFolder = objSpace.GetDefaultFolder(olFolderInbox)
   For Each objItem In objFolder.Items
      With objItem
         If InStr(.Subject, sSearch) > 0 Then
            Debug.Print .ReceivedTime, .Subject
         End If
      End With
   Next
   Set objSpace = Nothing
   Set objFolder = Nothing
End Sub
```

Im Anwendungs-Objekt gibt es das Ereignis *NewMail*, das auf neue Mails reagiert. Die Prozedur (Code 5.33) ermittelt die erste ungelesene Mail.

Code 5.33 Die Ereignisprozedur reagiert auf neue Mails

```
'Muss im Klassenmodul ThisOutlookSesson stehen
Private Sub Application_NewMail()
   Dim objFolder    As MAPIFolder
   Dim objItem      As MailItem
```

```
   Set objFolder = Session.GetDefaultFolder(olFolderInbox)
   objFolder.Items.Sort "[ReceivedTime]", False
   Set objItem = objFolder.Items.GetFirst
   Debug.Print objItem.ReceivedTime, objItem.Subject
   Set objItem = Nothing
   Set objFolder = Nothing
End Sub
```

Die Prozedur (Code 5.34) ermittelt alle *MailItems* eines ausgewählten Ordners über das *PickFolder*-Element. Mit einer *For-Each-Next*-Schleife können alle *Items* eines Ordners auch bearbeitet werden.

Code 5.34 Die Prozedur liest alle Items eines ausgewählten Ordners

```
Private Sub ReadSelectFolder()
   Dim objSpace   As NameSpace
   Dim objFolder  As MAPIFolder
   Dim objItem    As MailItem

   Set objSpace = GetNamespace("MAPI")
   Set objFolder = objSpace.PickFolder
   For Each objItem In objFolder.Items
      Debug.Print objItem.ReceivedTime, objItem.Subject
   Next
End Sub
```

Eine nützliche Anwendung des Ereignisses *AttachmentAdd* ist die Überprüfung der Mailgröße. Das Ereignis tritt auf, wenn einer Mail ein Anhang hinzugefügt wird. Die Codezeilen im Klassenmodul *ThisOutlookSession* (Code 5.35) überprüfen, ob die Mail größer als 500.000 Byte ist.

Code 5.35 Codezeilen überprüfen die Größe einer Mail

```
Private WithEvents eveMail As MailItem

Sub AttachmentAdd()
   Dim objAttachments   As Attachments
   Dim objAttachment    As Attachment

   Set eveMail = Application.CreateItem(olMailItem)
   eveMail.Subject = "Mail mit Anhang"
   Set objAttachments = eveMail.Attachments
   Set objAttachment = objAttachments.Add("C:\Temp\Example.avi", _
     olByValue)
   eveMail.Display
```

```
   Set objAttachment = Nothing
   Set objAttachments = Nothing
   Set eveMail = Nothing
End Sub

Private Sub eveMail_AttachmentAdd(ByVal newAttachment As Attachment)
   If newAttachment.Type = olByValue Then
      eveMail.Save
      If eveMail.Size > 500000 Then
         MsgBox "Warnung: Mailgröße ist " & eveMail.Size & _
         " Bytes!", vbCritical
      End If
   End If
End Sub
```

5.5.2 Termin-Objekte

Das Termin-Objekt *AppointmentItem* verwaltet einen Termin, einen Serientermin oder eine Besprechung. Ein Termin wird ebenfalls durch die Methode *CreateItem* erstellt (Code 5.36 und Code 5.37).

Code 5.36 Die Prozedur erzeugt einen Kalendereintrag

```
Sub CreateAppointmentItem()
   Dim objItem As AppointmentItem

   Set objItem = Application.CreateItem(olAppointmentItem)
   With objItem
      .Subject = "Abgabetermin"
      .Body = "Manuskript vorlegen!"
      .Duration = 120
      .Importance = olImportanceHigh
      .Location = "Office"
      .Categories = "Autor"
      .ReminderMinutesBeforeStart = 60
      .Start = "30.03.2016"
      .Display
      .Save
   End With
   Set objItem = Nothing
End Sub
```

Code 5.37 Die Prozedur erzeugt eine Meetingmitteilung

```
Sub CreateConferenceItem()
   Dim objItem       As AppointmentItem
   Dim objRequired   As Recipient
   Dim objOptional   As Recipient
   Dim objResource   As Recipient

   Set objItem = Application.CreateItem(olAppointmentItem)
   With objItem
      .MeetingStatus = olMeeting
      .Subject = "Status Meeting"
      .Location = "Konferenzraum 3"
      .Start = #9/27/2015 1:30:00 PM#
      .Duration = 90
      Set objRequired = .Recipients.Add("Lisa Müller")
      objRequired.Type = olRequired
      Set objOptional = .Recipients.Add("Kevin Costner")
      objOptional.Type = olOptional
      Set objResource = .Recipients.Add("Konferenzraum 3")
      objResource.Type = olResource
      .Display
      .Save
      .Send
   End With
   Set objItem = Nothing
End Sub
```

Das Ereignis *PropertyChange* kann genutzt werden, um Ereignisse in einem Termin abzufangen. Die Prozedur (Code 5.38) verhindert das Abschalten der Erinnerung von Terminen mit dem Betreff *Status Meeting*. Auch hier muss wieder eine Objektvariable im Klassenmodul *ThisOutlookSession* definiert und instanziiert werden.

Code 5.38 Die Prozedur verhindert das Abschalten der Erinnerung

```
Private WithEvents eveAppointment As AppointmentItem

Sub InitializeAppointmentEvents()
   Set eveAppointment = Application.GetNamespace("MAPI"). _
      GetDefaultFolder(olFolderCalendar).Items("Status Meeting")
End Sub

Private Sub eveAppointment_PropertyChange(ByVal Name As String)
   Select Case Name
   Case "ReminderSet"
      MsgBox "Sie können die Erinnerung nicht entfernen!"
```

```
      eveAppointment.ReminderSet = True
   End Select
End Sub
```

5.5.3 Kontakt-Objekte

Das Kontakt-Objekt *ContactItem* verwaltet Kontaktdaten und Kontaktgruppen. Die Prozedur (Code 5.39) erzeugt ein neues Objekt und verwendet einige wichtige Eigenschaften und Methoden.

Code 5.39 Die Prozedur erzeugt einen neuen Kontakteintrag

```
Sub CreateContactItem()
   Dim objItem As ContactItem

   Set objItem = Application.CreateItem(olContactItem)
   With objItem
      .FirstName = "Lisa"
      .LastName = "Müller"
      .CompanyName = "Strategy AG"
      .MailingAddressStreet = "Uferstraße 1"
      .MailingAddressPostalCode = "12345"
      .MailingAddressCity = "Irgendwo"
      .CompanyMainTelephoneNumber = "01234 567"
      .HomeTelephoneNumber = "01234 678"
      .MobileTelephoneNumber = "012121212"
      .Email1Address = "lisa.mueller@company.com"
      .Categories = "Personal"
      .Display
      .ShowCategoriesDialog
      .ShowBusinessCardEditor
      .Save
   End With
   Set objItem = Nothing
End Sub
```

Über die Ereignisse *NewInspector* und *PropertyChange* lässt sich bestimmen, welche Eigenschaft im Formular eine Änderung erfährt.

Mit der Prozedur *InitializeInspectors* (Code 5.40) wird die Ereignisprozedur *NewInspector* aktiviert, die ihrerseits dann die Objektvariable *objContact* instanziiert, so dass auch das Ereignis *PropertyChange* aktiv ist. Ausgegeben wird der Name der Eigenschaft, die im Formular geändert wurde.

Code 5.40 Die Codezeilen in ThisOutlookSession reagieren auf Änderungen im neuen Formular

```
Private WithEvents eveInspectors    As Inspectors
Private WithEvents eveContact       As ContactItem

Sub InitializeInspectorsEvents()
   Set eveInspectors = Application.Inspectors
End Sub

Private Sub eveInspectors_NewInspector(ByVal Inspector As Inspector)
  If TypeOf Inspector.CurrentItem Is ContactItem Then
    Set eveContact = Inspector.CurrentItem
  End If
End Sub

Private Sub eveContact_PropertyChange(ByVal Name As String)
   Debug.Print Name
End Sub
```

5.5.4 Aufgaben-Objekte

Das Aufgaben-Objekt *TaskItem* verwaltet Aufgaben, die zu einem bestimmten Zeitpunkt erledigt werden müssen. Die Prozedur (Code 5.41) erzeugt ein neues Item und verwendet einige wichtige Eigenschaften. Die Aufgabe startet in 30 Tagen und muss in ca. 6 Monaten erledigt sein. Sie ist bereits zu 40 % gelöst. Gleichzeitig wird das Dialogfenster zur Auswahl der Kategorie geöffnet.

Code 5.41 Die Prozedur erzeugt eine neue Aufgabe

```
Sub CreateTaskItem()
   Dim objItem As TaskItem

   Set objItem = Application.CreateItem(olTaskItem)
   With objItem
      .Subject = "Anforderung"
      .Body = "Beschreibung der Anforderung."
      .Complete = False
      .StartDate = Now + 30
      .DueDate = Now + 180
      .Importance = olImportanceLow
      .Owner = "Lisa Müller"
      .PercentComplete = 40
      .Status = olTaskWaiting
```

```
      .Display
      .ShowCategoriesDialog
   End With
   Set objItem = Nothing
End Sub
```

Mit der Methode *Send* lassen sich Statusreports von der geöffneten Aufgabe per Mail versenden (Code 5.42).

Code 5.42 Die Prozedur sendet einen Statusreport aus der geöffneten Aufgabe

```
Sub SendStatusReport()
   Dim objTask      As TaskItem
   Dim objInspector As Inspector
   Dim objReport    As MailItem

   Set objInspector = Application.ActiveInspector
   If Not TypeName(objInspector) = "Nothing" Then
      If TypeName(objInspector.CurrentItem) = "TaskItem" Then
         Set objTask = objInspector.CurrentItem
         Set objReport = objTask.StatusReport
         With objReport
            .Subject = objTask.Subject
            .To = objTask.Owner
            .CC = "Hans Schmidt"
            .Display
            .Send
         End With
         Set objReport = Nothing
         Set objTask = Nothing
       Else
           MsgBox "Keine Aufgabe geöffnet!"
       End If
   Else
       MsgBox "Keine Aufgabeordner geöffnet!"
   End If
   Set objInspector = Nothing
End Sub
```

Um mehrere Items gleichzeitig zu bearbeiten, kann das *Selection* Objekt eingesetzt werden. Voraussetzung ist, dass die betreffenden Einträge im Ordner vorher markiert werden. Die Prozedur (Code 5.43) setzt die Priorität markierter Aufgaben auf *High*.

Code 5.43 Die Prozedur setzt die Priorität markierter Aufgaben auf High

```
Public Sub ChangeTasksPriority()
   Dim objTask        As TaskItem
   Dim objSelection   As Selection
   Dim objFolder      As MAPIFolder
   Dim iLoop          As Integer

   Set objFolder = ActiveExplorer.CurrentFolder
   If objFolder.DefaultItemType = olTaskItem Then
      Set objSelection = ActiveExplorer.Selection
      For iLoop = 1 To objSelection.Count
         Set objTask = objSelection(iLoop)
         objTask.Importance = olImportanceHigh
         objTask.Save
      Next
      Set objSelection = Nothing
   End If
   Set objFolder = Nothing
End Sub
```

Mit dem Ereignis *CustomAction* lassen sich Aktionen in der Aufgabe programmtechnisch begleiten (Code 5.44).

Code 5.44 Die Ereignisse reagieren auf das Öffnen und Beschreiben einer Aufgabe

```
Private WithEvents eveInspectors    As Inspectors
Private WithEvents eveTask          As TaskItem

Sub InitializeTaskEvents()
   Set eveTask = Application.CreateItem(olTaskItem)
   With eveTask
      .Subject = "Task zum Test"
      .Display
      .Save
   End With
   Set eveTask = Nothing
End Sub

Private Sub eveTask_CustomAction( _
  ByVal Action As Object, ByVal Response As Object, Cancel As Boolean)
  Debug.Print Action.Name
End Sub
```

5.5.5 Journal-Objekte

Das Journal-Objekt *Journal* zeichnet Aktionen in Outlook auf, die in Bezug zu einem Kontakt stehen. Der Eintrag erfolgt auf einer Zeitskala (Abb. 5.2). So ist es möglich, die benötigte Zeit für eine Aktion festzuhalten. Es lassen sich Mails oder andere Anwendungsdokumente aus Excel und Word verfolgen. Ebenso können Aktionen außerhalb des PCs dokumentiert werden, wie Telefonate und Gespräche.

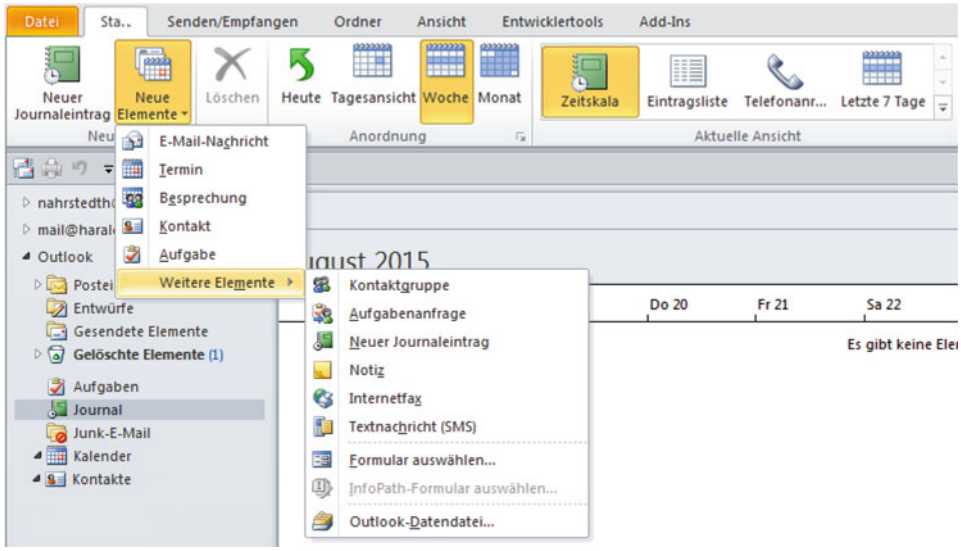


Abb. 5.2 Auswahl neuer Elemente

Mithilfe der Zeitskala lassen sich Aktionen sehr gut zeitlich einordnen. Die Prozedur (Code 5.45) schaltet auf die Journalansicht um.

Code 5.45 Die Prozedur schaltet auf die Journalansicht um

```
Sub GotoJournalFolder()
    Set Application.ActiveExplorer.CurrentFolder = _
    Session.GetDefaultFolder(olFolderJournal)
End Sub
```

Ein Objekt *JournalItem* wird über die Methode *CreateItem* erzeugt (Code 5.46). Die Methode *StartTimer* startet die Zeitmessung für den Journaleintrag und ermöglicht damit die Kontrolle der Zeitmessfunktion. Entsprechend stoppt die Methode *StopTimer* die Zeitmessung.

Code 5.46 Die Prozedur erzeugt einen neuen Journaleintrag

```
Sub CreateJournalItem()
   Dim objJournal As JournalItem

   Set objJournal = Application.CreateItem(olJournalItem)
   With objJournal
      .Subject = "Projekt Status"
      .Body = "Status Report Diagramm"
      .Companies = "Muster GmbH"
      .Type = "Besprechung"
      .Importance = olImportanceNormal
      .Duration = 30
      .Display
      .StartTimer
      .ShowCategoriesDialog
   End With
   Set objJournal = Nothing
End Sub
```

Da eine Aktion oft mit einem Kontakt verbunden ist, lässt sich ein Journaleintrag direkt aus einem ContactItem heraus erzeugen (Code 5.47). Dazu muss der Kontakteintrag markiert sein.

Code 5.47 Die Prozedur erzeugt über einen markierten Kontakt einen Journaleintrag

```
Sub CreateJournalItemForContact()
   Dim objJournal As JournalItem
   Dim objContact As ContactItem

   If ActiveExplorer.Selection.Count > 0 Then
      If TypeName(ActiveExplorer.Selection.Item(1)) = _
         "ContactItem" Then
         Set objContact = ActiveExplorer.Selection.Item(1)
         Set objJournal = Application.CreateItem(olJournalItem)
         With objJournal
            .Subject = objContact.Subject
            .Body = objContact.MailingAddress
            .Companies = objContact.Title & vbLf & _
               objContact.CompanyName & vbLf & _
               objContact.FullName
            .ContactNames = objContact.FullName
            .Categories = objContact.Categories
            .Display
            .StartTimer
```

```
         End With
         Set objJournal = Nothing
         Set objContact = Nothing
      Else
         MsgBox "Kein Kontakt markiert!"
      End If
   Else
      MsgBox "Kein Ordner markiert!"
   End If
End Sub
```

Ebenso kann ein Journaleintrag von einer Mail aus erzeugt werden (Code 5.48). Dazu muss dann die Mail markiert sein.

Code 5.48 Die Prozedur erzeugt über einen markierten Emaileintrag einen Journaleintrag

```
Sub CreateJournalItemForMail()
   Dim objJournal As JournalItem
   Dim objMail    As MailItem

   If ActiveExplorer.Selection.Count > 0 Then
      If TypeName(ActiveExplorer.Selection.Item(1)) = _
         "MailItem" Then
         Set objMail = ActiveExplorer.Selection.Item(1)
         Set objJournal = Application.CreateItem(olJournalItem)
         With objJournal
            .Subject = objMail.Subject
            .Body = objMail.Body
            .ContactNames = objMail.SenderName
            .Companies = objMail.Companies
            .Categories = objMail.Categories
            .Type = "Email"
            .Display
            .StartTimer
         End With
         Set objJournal = Nothing
         Set objMail = Nothing
      Else
         MsgBox "Keine Email markiert!"
      End If
   Else
      MsgBox "Kein Ordner markiert!"
   End If
End Sub
```

Als Ereignisprozedur soll das Ereignis *AfterWrite* betrachtet werden (Code 5.49). Es tritt auf, nachdem der Journaleintrag gespeichert wurde.

Code 5.49 Die Codezeilen reagieren auf einen gespeicherten Journaleintrag

```
Private WithEvents eveJournal As JournalItem

Sub InitializeJournalEvents()
   Set eveJournal = Application.CreateItem(olJournalItem)
   With eveJournal
      .Subject = "Item zum Test"
      .Display
      .Save
      .StartTimer
   End With
   Set eveJournal = Nothing
End Sub

Private Sub eveJournal_Write(Cancel As Boolean)
   Debug.Print eveJournal.Subject
End Sub
```

5.5.6 Notiz-Objekte

Das Notiz-Objekt *NoteItem* verwaltet Notizen auf Textbasis, in denen sich zahlreiche Informationen auf bequeme Art und Weise erfassen lassen. Als kleine farbige Notizzettel können sie in Ordnern oder auf dem Desktop abgelegt werden.

Die Prozedur (Code 5.50) schaltet auf den Ordner Notizen um.

Code 5.50 Die Prozedur schaltet auf den Ordner Notizen um

```
Sub GotoNoteFolder()
    Set Application.ActiveExplorer.CurrentFolder = _
    Session.GetDefaultFolder(olFolderNotes)
End Sub
```

Die Prozedur (Code 5.51) erzeugt eine neue Notiz und löscht sie nach einem Stopp wieder.

Code 5.51 Die Prozedur erzeugt eine Notiz

```
Sub CreateNoteItem()
   Dim objNote As NoteItem
   Dim lnghwnd As Long
   Dim lngFlags As Long
   Dim lngFunktionswert As Long
```

```
    Set objNote = Application.CreateItem(olNoteItem)
    With objNote
        .Body = "Hier steht der Notiztext."
        .Top = 100
        .Left = 100
        .Height = 200
        .Width = 300
        .Color = olGreen
        .Display
        Stop
        .Delete
    End With
    Set objNote = Nothing
End Sub
```

Das *NoteItem* verfügt über keine Ereignisse.

5.5.7 Verteiler-Objekte

Das Verteiler-Objekt *DistListItem* stellt eine Verteilerliste zu einem Ordner vom Typ *Kontakte* dar. Eine solche Liste wird verwendet, um jedem Kontakt in dieser Liste eine Nachricht zu senden.

Die Prozedur (Code 5.52) erstellt eine neue Verteilerliste mit der Methode *CreateItem*.

Code 5.52 Die Prozedur erzeugt eine neue Verteilerliste

```
Sub CreateDistList()
    Dim objList As DistListItem

    Set objList = Application.CreateItem(olDistributionListItem)
    objList.Display
    Set objList = Nothing
End Sub
```

Kategorien sind ein gutes Ordnungsschema für Kontakte. Oft muss eine Email an Kontakte der gleichen Kategorie gesendet werden. Dann hilft eine Verteilerliste, in der alle Kontakte einer Kategorie vorhanden sind.

Die Prozedur (Code 5.53) erzeugt eine neue Verteilerliste mit Elementen aus dem Standard-Kontaktordner die einer Kategorie zugeordnet sind. Die Verteilerliste erhält den Namen der Kategorie, die zuvor eingegeben wird.

Code 5.53 Die Prozedur erzeugt eine Verteilerliste mit allen Kontakten einer Kategorie

```
Sub CreateCateList()
   Dim objSpace    As NameSpace
   Dim objFolder   As Folder
   Dim objList     As DistListItem
   Dim sCategory   As String
   Dim sName       As String
   Dim iLoop       As Integer
   Dim objRecpnt   As Recipient

   sCategory = InputBox("Kathegorie = ")
   Set objList = Application.CreateItem(olDistributionListItem)
   With objList
      .Subject = sCategory
      .Display
   End With
   Set objSpace = Application.GetNamespace("MAPI")
   Set objFolder = objSpace.GetDefaultFolder(olFolderContacts)
   objFolder.Display
   For iLoop = 1 To objFolder.Items.Count
      If objFolder.Items(iLoop).Categories = sCategory Then
         sName = objFolder.Items(iLoop).Email1Address
         Set objRecpnt = Application.Session.CreateRecipient(sName)
         objRecpnt.Resolve
         If objRecpnt.Resolved Then
            objList.AddMember objRecpnt
         End If
         Set objRecpnt = Nothing
      End If
   Next iLoop
   objList.Close (olSave)
   Set objFolder = Nothing
   Set objSpace = Nothing
   Set objList = Nothing
End Sub
```

Nicht immer ist es der Standard-Kontaktordner und einfacher ist es, einen Kontakt in einem Ordner als Muster zu markieren. Diese Aufgabe erfüllt die Prozedur (Code 5.54). Mit der Methode *Move* wird die Verteilerliste dann in das ausgewählte Verzeichnis verschoben.

Code 5.54 Die Prozedur erzeugt eine Verteilerliste in einem beliebigen Kontaktverzeichnis

```
Sub CreateCurrentList()
   Dim objExplorer   As Explorer
   Dim objSelection  As Selection
   Dim objItem       As ContactItem
   Dim objFolder     As Folder
   Dim objList       As DistListItem
   Dim objRecpnt     As Recipient
   Dim sCategory     As String
   Dim sName         As String
   Dim iLoop         As Integer

'liest Kategorie von markiertem Kontakt
   Set objExplorer = Application.ActiveExplorer
   Set objSelection = objExplorer.Selection
   Set objItem = objSelection.Item(1)
   sCategory = objItem.Categories

'sucht Kontakte im aktiven Ordner mit gleicher Kategorie
   Set objFolder = objExplorer.CurrentFolder
   Set objList = Application.CreateItem(olDistributionListItem)
   With objList
      .Subject = sCategory
      .Display
   End With
   For iLoop = 1 To objFolder.Items.Count
      If objFolder.Items(iLoop).Categories = sCategory Then
         sName = objFolder.Items(iLoop).Email1Address
         Set objRecpnt = Application.Session.CreateRecipient(sName)
         objRecpnt.Resolve
         If objRecpnt.Resolved Then
            objList.AddMember objRecpnt
         End If
         Set objRecpnt = Nothing
      End If
   Next iLoop
   objList.Move objFolder
   Set objExplorer = Nothing
   Set objSelection = Nothing
   Set objItem = Nothing
   Set objFolder = Nothing
   Set objList = Nothing
End Sub
```

Wie eine vorhandene Verteilerliste gelesen wird zeigt Prozedur (Code 5.55).

Code 5.55 Die Prozedur öffnet eine Verteilerliste

```
Sub OpenDistList()
   Dim objSpace      As NameSpace
   Dim objFolder     As Folder
   Dim objListFolder As Folder
   Dim objList       As DistListItem

   Set objSpace = Application.GetNamespace("MAPI")
   Set objFolder = objSpace.GetDefaultFolder(olFolderContacts)
   Set objListFolder = objFolder.folders("<Foldername mit Liste>")
   Set objList = objListFolder.Items("<Listenname>")
   objList.Display
   Set objList = Nothing
   Set objListFolder = Nothing
   Set objFolder = Nothing
   Set objSpace = Nothing
End Sub
```

Es gibt noch weitere Items wie

- DocumentItem,
- MeetingItem,
- MobileItem,
- PostItem,
- RemoteItem,
- ReportItem,
- SharingItem,
- SimpleItems,
- StorageItem,

deren Erkundung ich dem Leser überlasse. Hilfreich ist wie immer der Objektkatalog.

5.6 Outlook-Interaktionen

Zu Outlook existieren eine Menge kleiner „Helferchen“. Von der automatischen E-Mail-Gestaltung, über Verteilerlisten und Kategoriezuordnungen bis hin zu Kalenderdarstellungen. Doch hier interessiert uns das Zusammenspiel von Office-Anwendungen.

5.6.1 Umsätze reporten

Ein Word-Dokument ist Ausgang einer typischen Office-Alltagssituation. In diesem Dokument existiert eine Tabelle mit Umsatzdaten (Abb. 5.3).

Abb. 5.3 Einfache Umsatztabelle in einem Word-Dokument

Monat	Umsatz
Jan	450
Feb	612
Mrz	388
Apr	345
Mai	218
Jun	367
Jul	455
Aug	444
Sep	412
Okt	398
Nov	455
Dez	420

Diese Daten sollen in einem Report vorgestellt werden. Dazu ist mindestens eine PowerPoint-Folie erforderlich und der Report soll in 14 Tagen stattfinden.

Wir erinnern uns, dass Excel die Anwendung mit großer Funktionalität für Berechnungen und Analysen ist, bis hin zur Visualisierung mit Diagrammen. Folglich übertragen wir die Daten der Umsatztabelle von Word nach Excel, erstellen damit ein Balkendiagramm oder Kreisdiagramm und legen es in die Zwischenablage. Von dort kann es dann in das Word-Dokument übernommen werden. Bei der Gelegenheit nutzen wir eine Excelfunktion zur Bestimmung des Mittelwertes. Dieser Wert überschreitet in einer globalen Variablen die Anwendungsgrenzen. Die Prozedur (Code 5.56) wird nur dann aktiv, wenn die Tabelle im Word-Dokument zuvor markiert wurde.

Code 5.56 Die Prozedur erstellt in Excel ein Diagramm und nutzt eine Worksheet-Funktion

```
Sub DiagrammErstellen()
   Dim docTest    As Word.Document
   Dim appExcel   As Excel.Application

   Set docTest = ThisDocument

'Abfrage, ob die Tabelle markiert ist
   docTest.Tables(1).Select
   Selection.Copy
   Set appExcel = Excel.Application
   With appExcel
      .Workbooks.Add
      .ActiveSheet.Paste
      .Charts.Add
```

```
    With ActiveChart
       .ChartType = xl3DColumn
       .SetSourceData appExcel.Worksheets(1).UsedRange
       .Location xlLocationAsNewSheet
       .ChartArea.Copy
    End With
    dMittelWert = .WorksheetFunction.Average(.Worksheets(1).UsedRange)
      .ActiveWorkbook.Close False
      .Quit
   End With
   Set appExcel = Nothing
End Sub
```

Diese Aktion läuft im Hintergrund und ist somit für den Anwender nicht sichtbar. Nach der Ausführung kann das Diagramm an eine beliebige Stelle im Dokument manuell eingefügt werden (Abb. 5.4).

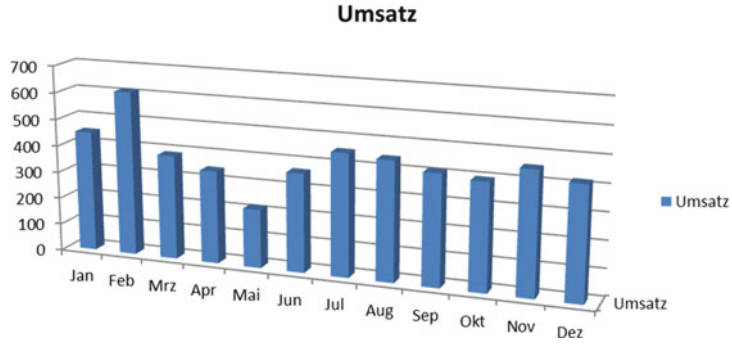


Abb. 5.4 Umsatzdiagramm aus der Zwischenablage

Mit der Prozedur (Code 5.57) wird eine PowerPoint-Anwendung geöffnet und darin eine Titelfolie und eine leere Folie erstellt. Die Titelfolie erhält einen Titeltext und die leere Folie das Diagramm aus der Zwischenablage. Die zweite Folie erhält auch ein Kommentarfeld, in das der errechnete Mittelwert eingetragen wird. Die Präsentation (Abb. 5.5) bleibt über das Prozedurende hinaus geöffnet und kann so weiter bearbeitet und gespeichert werden.

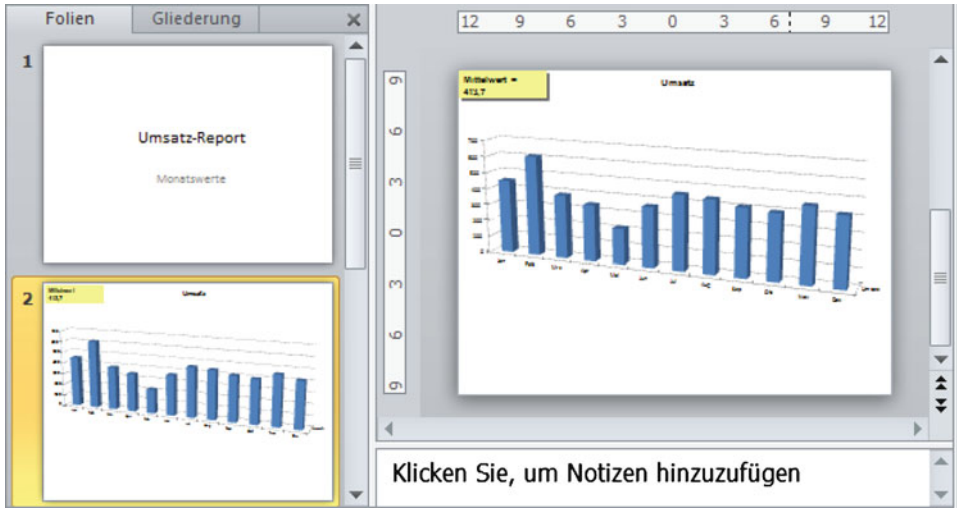


Abb. 5.5 Erstellte Präsentations-Vorlage

Code 5.57 Die Prozedur erstellt zwei PowerPoint-Folien für einen Umsatz-Report

```
Sub FolieErstellen()
   Dim pptDemo    As PowerPoint.Application
   Dim pptShow    As PowerPoint.Presentation
   Dim pptFolie   As PowerPoint.Slide

   Set pptDemo = CreateObject("PowerPoint.Application")
   With pptDemo
      .Visible = True
      .Presentations.Add
   End With
   Set pptShow = pptDemo.ActivePresentation
   Set pptFolie = pptShow.Slides.Add(1, ppLayoutTitle)
   With pptFolie
      .Shapes(1).TextFrame.TextRange.Text = "Umsatz-Report"
      .Shapes(2).TextFrame.TextRange.Text = "Monatswerte"
   End With
```

```
    Set pptFolie = Nothing
    Set pptFolie = pptShow.Slides.Add(2, ppLayoutBlank)
    With pptFolie
       .Shapes.PasteSpecial (ppPasteDefault)
       .Shapes(1).Chart.Legend.Delete
       .Shapes.AddComment
       .Shapes(2).TextFrame.TextRange.Text = _
          "Mittelwert = " & Round(dMittelWert, 1)
    End With
    Set pptFolie = Nothing

    Set pptFolie = Nothing
    Set pptShow = Nothing
    Set pptDemo = Nothing
End Sub
```

Der letzte Schritt besteht in der Festlegung des Report-Termins. Er wird vom aktuellen Datum um 14 Tage in die Zukunft festgelegt (Abb. 5.6). Für die Präsentation ist eine Dauer von zwei Stunden vorgesehen. Auch hier erfolgt im *Bodytext* die Angabe des Mittelwertes. Zum Abschluss wird die Erledigung der Aufgabe gemeldet (Code 5.58).

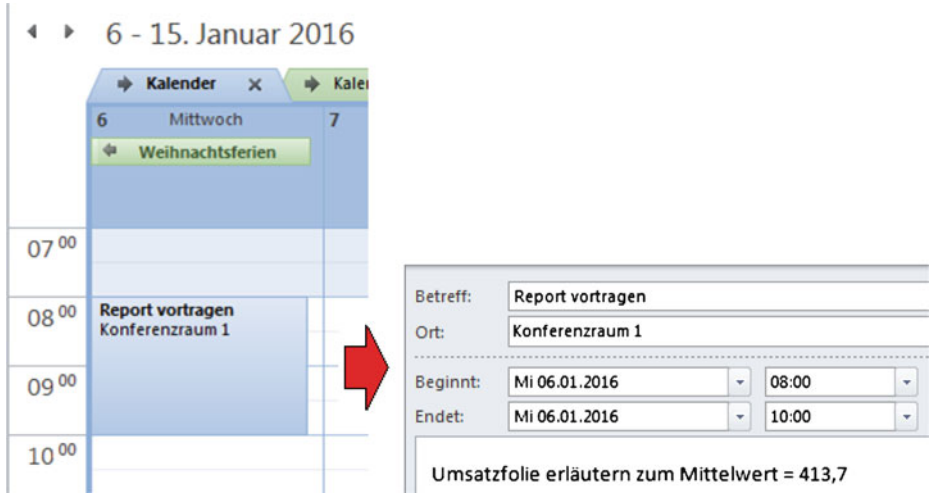


Abb. 5.6 Termineintrag im Kalender

Code 5.58 Die Prozedur trägt einen Termin im Kalender ein

```
Sub TerminErstellen()
   Dim outDemo     As Outlook.Application
   Dim objTermin   As Outlook.AppointmentItem
   Dim vTag        As Variant
```

```
    Set outDemo = CreateObject("Outlook.Application")
    Set objTermin = outDemo.createItem(1)

    vTag = Date + 14
    With objTermin
       .Start = vTag & " 08:00"          'Eintrag
       .Subject = "Report vortragen"
       .body = "Umsatzfolie erläutern zum Mittelwert = " & _
         Round(dMittelWert, 1)
       .Location = "Konferenzraum 1"
       .Duration = "120"                 'Dauer in Minuten
       .ReminderMinutesBeforeStart = 10 'Erinnerung in Minuten
       .ReminderPlaySound = True         'Erinnerung mit Sound
       .ReminderSet = True               'Erinnerung wiederholen
       .Save
    End With

    Set objTermin = Nothing
    Set outDemo = Nothing
End Sub
```

Alle drei Prozeduren lassen sich im Outlook-Objekt *ThisDocument* zu einer Aufgabe zusammenfassen (Code 5.59).

Code 5.59 Zusammenfassung aller Teilaufgaben

```
Private Sub cmdStart_Click()
    DiagrammErstellen
    FolieErstellen
    TerminErstellen
    MsgBox "Aufgaben ausgeführt!"
End Sub
```

5.6.2 Mails mit einer Excel-Tabelle senden

Aus einer Excel-Tabelle (Abb. 5.7) sollen E-Mails versendet werden (Code 5.60). Mit diesem Mail-Explorer lassen sich rudimentär Mitteilungen generieren. Der Entwurf kann auch noch weiter ausgebaut werden.

	A	B	C
1	**Empfänger**	**Betreff**	**Text**
2	muster(at)test.com	Geburtstag	Zum 20. die besten Wünsche
3	muster(at)test.com	Meeting	Treffen uns um 18:00 am 1. des nächsten Monats!
4	muster(at)test.com	Urlaub	Habe für 3 Personen gebucht.
5	muster(at)test.com	Urlaub	Habe für 3 Personen gebucht.
6	muster(at)test.com	Kochen	Was gibt es zum Essen?
7	muster(at)test.com	Einkaufen	Kannst du noch Käse kaufen?
8	muster(at)test.com	Meeting	Treffen uns um 18:00 am 1. des nächsten Monats!
9	muster(at)test.com	Geburtstag	Zum 30. die besten Wünsche!
10	muster(at)test.com	Geburtstag	Zum 40. die besten Wünsche!

Abb. 5.7 Beispiel einer Mailliste

Code 5.60 Die Prozedur sendet E-Mails aus einer Excel-Arbeitsliste

```
Sub MailExplorer()
   Dim wshList    As Worksheet
   Dim appOut     As Object
   Dim outMail    As Object
   Dim lRow       As Long
   Dim lRowMax    As Long

   Set wshList = ThisWorkbook.ActiveSheet
   lRowMax = wshList.UsedRange.Rows.Count

'Sendeschleife
   For lRow = 2 To lRowMax
      Set appOut = CreateObject("Outlook.Application")
      Set outMail = appOut.CreateItem(0)
      With outMail
      'Empfänger
         .To = wshList.Cells(lRow, 1)
      'Betreff
         .Subject = wshList.Cells(lRow, 2)
      'Text ohne Formatierung , max 1024 Zeichen
         .Body = wshList.Cells(lRow, 3)
      'Mail anzeigen (auskommentieren?)
         .Display
      'Postausgang
         .Send
      End With
      Set outMail = Nothing
      Set appOut = Nothing
      'Sendepause
      Application.Wait (Now + TimeValue("0:00:05"))
    Next lRow
    Set wshList = Nothing
End Sub
```

Die Funktion *Wait* erzeugt zwischen den Mails eine 5-Sekunden-Pause. Dadurch kann Outlook die einzelnen Mails ohne Zeitprobleme verarbeiten. Da immer das aktuelle Arbeitsblatt in Excel verwendet wird, können mehrere unterschiedliche Maillisten in einer Arbeitsmappe stehen.

5.6.3 Word-Dokument als Anlage versenden

Word-Dokumente lassen sich direkt als Mail versenden. Dabei wird aber das Mail-Dialogfenster *modal* geöffnet, so dass alle anderen Fenster blockiert sind. Es können weder Termine aus Kalendern abgefragt noch Daten aus Mails kopiert werden. Die Prozedur (Code 5.61) umgeht diese Einschränkung. Zuvor wird eine Aufforderung zur Speicherung des Dokuments bearbeitet. Danach wird das Mail-Dialogfeld geöffnet (Abb. 5.8).

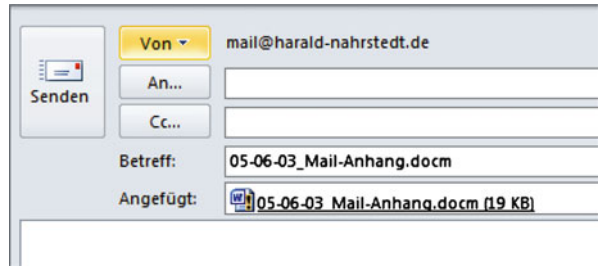


Abb. 5.8 E-Mail-Dialogfenster

Code 5.61 Die Prozedur erstellt einen Mail-Dialog mit dem Dokument im Anhang

```
Sub AttachToMail()
   Dim appOut    As Object
   Dim outMail   As Object
   Dim outAtt    As Object
   Dim sName     As String
   Dim objDlg    As Office.FileDialog

   If ActiveDocument.Saved Then
      sName = ActiveDocument.Fullname
   Else
      Set objDlg = Application.FileDialog(msoFileDialogSaveAs)
      objDlg.Show
      If objDlg.SelectedItems.Count Then
         sName = objDlg.SelectedItems(1)
         ActiveDocument.SaveAs sName
      Else
         Exit Sub
      End If
      Set objDlg = Nothing
   End If
```

```
    Set appOut = GetObject(, "outlook.application")
    Set outMail = appOut.CreateItem(0)
    Set outAtt = outMail.Attachments.Add(sName)
    outMail.Subject = outAtt.DisplayName
    outMail.Display

    Set outAtt = Nothing
    Set outMail = Nothing
    Set appOut = Nothing
End Sub
```

Mit einer Ergänzung weiterer Eigenschaften und der Methode *Send* kann die E-Mail direkt versendet werden.

Access-Objekte

6

Das Problem im Umgang mit Objekten in Access-Datenbanken ist nicht die Suche nach dem geeigneten Objekt, sondern vielmehr die Frage welche Methode angewendet werden soll. Denn bei der Entwicklung von Access wurden neue Wege eingeschlagen und die alten Strukturen sind geblieben. So gibt es mehrere Möglichkeiten für Lösungen, die von den Objektbibliotheken der Entwicklungen abhängen.

6.1 Access-Objektbibliotheken

Access besitzt im Gegensatz zu den meisten anderen Office-Anwendungen gleich **drei** Objekt-Bibliotheken.

6.1.1 Microsoft Access Objekt-Bibliothek

Die *Microsoft Access Object Library* besitzt als oberstes Objekt das Anwendungs-Objekt *Application* und danach folgen weitere Unterobjekte, ähnlich der Struktur der Objekt-Bibliotheken der anderen Office-Anwendungen (Abb. 6.1).

Die Bibliothek kann unter *Extras* in *Verweise* unter dem Namen *Microsoft Access 14.0 Object Library* eingebunden werden. Ihre Position im System-Verzeichnis lautet:

C:\Program Files\Microsoft Office\Office14\MSACC.OLB.

H. Nahrstedt, *Die Welt der VBA-Objekte*, DOI 10.1007/978-3-658-13891-2_6

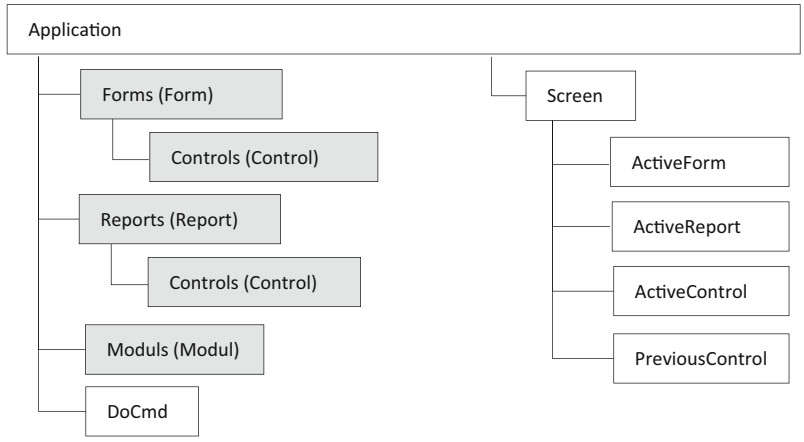


Abb. 6.1 Die wichtigsten Unterobjekte und Objektlisten des Application-Objektmodells

6.1.2 Data Access Objekt-Bibliothek

Diese Bibliothek beschreibt *Data Access*-Objekte (DAO), die dem Objekt *DBEngine* untergeordnet sind (Abb. 6.2).

Während das *Application*-Objekt für die Benutzeroberfläche zuständig ist, ermöglicht das *DBEngine*-Objekt den direkten Zugriff auf die *Microsoft Jet Database Engine.* Durch die Teilung der Bibliotheken muss nicht der gesamte Objektumfang geladen werden, nur um auf Datenbanken zuzugreifen. Das ist auch für Zugriffe aus Anwendungen wie Word und Excel sinnvoll. Auch Programmiersprachen wie *Visual Bas*ic und *Visual C++* können die DAO-Bibliothek nutzen.

Die Bibliothek kann unter *Extras* in *Verweise* unter dem Namen *Microsoft Office 14.0 Access database engine Object Library* eingebunden werden. Ihre Position im System-Verzeichnis lautet:

C:\Program Files\Common Files\microsoft shared\OFFICE14\ACEDAO.DLL.

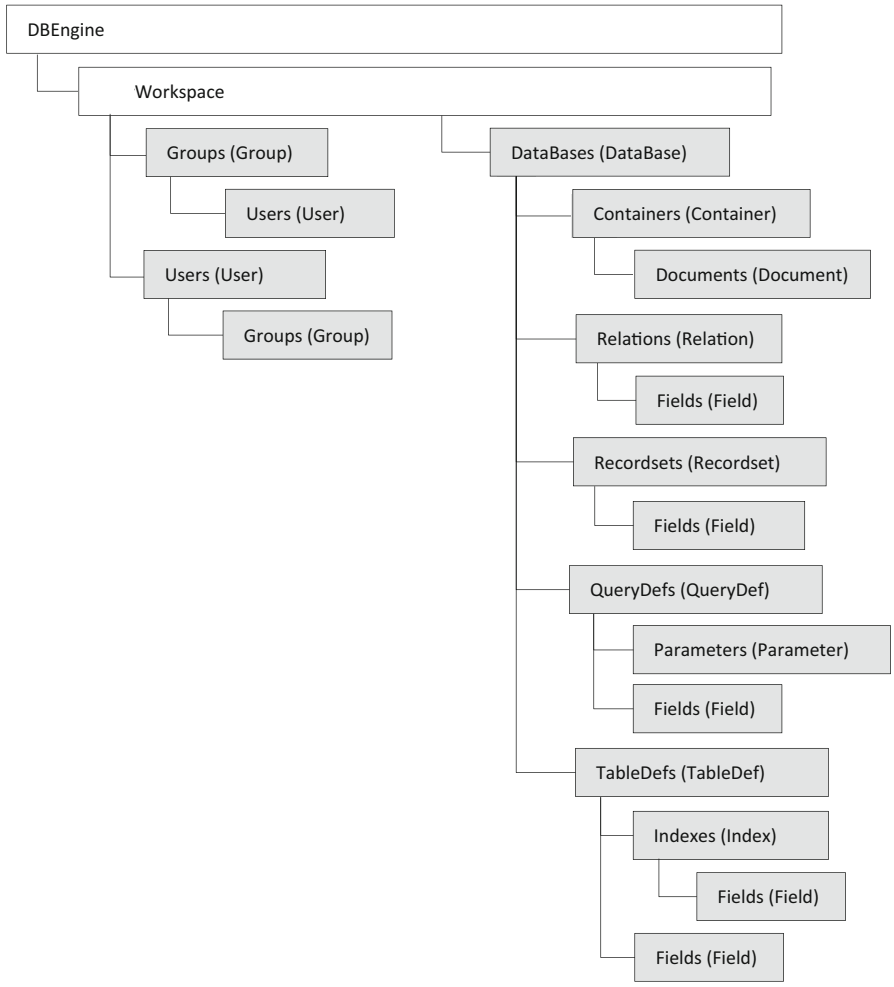


Abb. 6.2 Die wichtigsten Unterobjekte und Objektlisten (*grau*) des DBEngine-Objektmodells

6.1.3 ActiveX Data Objekt-Bibliothek

Diese Bibliothek (Abb. 6.3) beschreibt *ActiveX Data* Objekte (ADO). Sie sollte ursprünglich die DAO-Bibliothek ablösen. Da aber die DAO-Objekte einfacher zu programmieren sind, werden sie immer noch gerne eingesetzt. Die ADOs stellen eine Schnittstelle zur Datenbank OLEDB bereit. Damit können nicht nur relationale Datenbankobjekte, sondern jede Art von Objektsammlung gehandhabt werden.

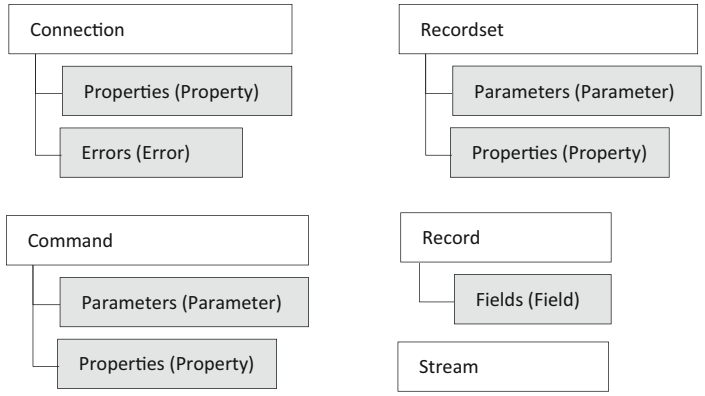


Abb. 6.3 Die wichtigsten Unterobjekte und Objektlisten (*grau*) des ADO-Objektmodells

Aus der Darstellung ist ersichtlich, dass das ADO-Objektmodell aus mehreren unabhängigen Objektgruppen besteht. Die Bibliothek kann unter *Extras* in *Verweise* unter dem Namen *Microsoft ActiveX Data Object 6.1 Library* eingebunden werden. Ihre Position im System-Verzeichnis lautet:

C:\Program Files\Common Files\System\ado\msado15.dll.

Unter *Verweise* befinden sich mehrere gleichnamige Bibliotheken mit unterschiedlichen Versionsnummern. Es sind die Vorgänger älterer Versionen.

6.1.4 SQL Datenbanksprache

Eigentlich ist die SQL-Sprache ein Thema für ein eigenes Buch und es gibt einige davon. Dennoch komme ich hier nicht umhin, ein paar Grundbegriffe zu erläutern. Wer mit einer Access-Datenbank arbeitet kommt eigentlich ohne SQL aus, denn die Entwicklung wird durch grafische Benutzeroberflächen (sogenannte GUI = graphical user interface) erleichtert. Tatsache ist jedoch, dass viele Aktionen auf SQL-Befehle zurückgeführt werden und die GUI nur die notwendig minimalste Menge an SQL-Befehlen darstellt.

Neben dem SQL-Standard gibt es in Access eigene Erweiterungen. Dies muss derjenige beachten, der sich mit SQL beschäftigen will und den Zugriff auf andere Datenbanken plant. Nachfolgend einige Anwendungsbeispiele.

Der Zugriff auf eine Auswahl von Daten in einer Tabelle kann mit der Auswahlabfrage *SELECT* erfolgen.

```
'Syntax einer SQL-Abfrage:
SELECT Felder FROM Tabellen [WHERE Bedingung][ORDER BY Sortierfeld]

'Beispiel:
'Erstellt als Auswahl die Felder Region, Kunde, Umsatz
'aus der Verkaufsliste
'filtert diese nach der Region West
'und sortiert sie absteigend
SELECT Region, Kunde, Umsatz FROM Verkaufsliste
WHERE Region='West' ORDER BY Umsatz DESC
```

Zu beachten ist, dass unter *FROM* mehrere Tabellen durch Komma getrennt angegeben werden können. Auch Mengenoperationen auf ihnen sind möglich.

Parameterabfragen sind noch flexibler, da sie durch die Angabe von Parametern gesteuert werden.

```
'Syntax:
PARAMETERS Parameter1 Datentyp1 [Parameter2 Datentyp2] ...;
SQL-Abfrage

'Beispiel:
PARAMETERS [Bitte Region wählen] Text;
SELECT * FROM Verkaufsliste
```

Anhang 6, Tab. A6.1 enthält die Datentypen in SQL.

Sollen Duplikate vermieden werden, dann wird der *SELECT* Anweisung ein *DISTINCT* hinzugefügt.

```
'Syntax:
SELECT DISTINCT Datenfeld FROM Tabellenname ORDER BY Datenfeld

'Beispiel:
SELECT DISTINCT Region FROM Verkaufsliste ORDER BY Region='West'
```

Die *SELECT* Anweisung in Verbindung mit der Angabe *TOP* liefert Extremwerte.

```
'Syntax:
SELECT TOP n Datenfelder FROM Tabellen ORDER BY Datenfelder

'Beispiel: Liefert die 10 größten Umsatzwerte
SELECT TO 10 Umsatz FROM Verkaufsliste ORDER BY Umsatz DESC
```

Die *SELECT* Anweisung lässt sich auch in Kombination mit Aggregatfunktionen verwenden. *Feldname* ist der Name des einzigen Feldes, in dem das Ergebnis stehen soll.

```
'Syntax:
'SELECT Funktion AS Feldname FROM Tabelle

'Beispiel: Bestimmt den Gesamtumsatz
SELECT SUM AS GesamtUmsatz FROM Verkaufsliste
```

Mit der zusätzlichen Angabe *SELECT BY* können Gruppierungen erzeugt werden.

```
'Syntax:
SELECT Felder FROM Tabellen [WHERE Bedingungen]
[GROUP BY Felder] [HAVING Bedingungen]
[ORDER BY Felder]

'Beispiel: Bestimmt den Gesamtumsatz je Kunde
SELECT [Kunde], SUM [Umsatz] AS UmsatzSumme
FROM Verkaufsliste
GROUP BY [Kunde]
```

Für Aktualisierungsabfragen lautet die Anweisung *UPDATE*.

```
'Syntax:
UPDATE Tabelle SET Feld1 = Wert1[, Feld2 = Wert2] ...
[WHERE Bedingungen]

'Beispiel: Führt eine Namenänderung durch
UPDATE Verkaufsliste SET Kunde = Ross & Sohn oHG
WHERE Kunde = 'Ross oHG'
```

Einfügeabfragen werden mit der Anweisung *INSERT INTO* durchgeführt.

```
'Syntax:
INSERT INTO ZielTabelle [(Feld1[, Feld2[, ...]])
SELECT-Abfrage
```

```
'Beispiel:
INSERT INTO Verkaufsliste (Region, Verkäufer, Kunde,
Produktkategorie, Umsatz, Auftragsnummer, Auftragseingang)
VALUES ('West', 'Weber Assmann GmbH', 'Zugfedern',
'1.823', 'A2012086', '12.02.2010')
```

Tabellenerstellungsabfragen fügen Teile einer oder mehrerer Tabellen zu einer neuen Tabelle zusammen.

```
'Syntax:
SELECT [(Feld1[, Feld2[, ...]]) INTO NeueTabelle
FROM Quelltabellen
[WHERE Bedingungen]
[ORDER BY SortierFelder]

'Beispiel:
SELECT Produktkategorie, Kunde, Umsatz INTO Produktumsätze
FROM Verkaufsliste
WHERE Region = 'West'
```

Für Löschabfragen gibt es die Anweisung *DELETE*. Dabei werden immer ganze Datensätze gelöscht.

```
'Syntax:
DELETE Tabelle oder * FROM Tabelle [WHERE Bedingung]

'Beispiel: Löscht alle Datensätze der Region West
DELETE * FROM Verkaufsliste WHERE Region = 'West'
```

Eine *UNION*-Abfrage kombiniert Felder aus unterschiedlichen Tabellen zu einem Ergebnis.

```
'Syntax:
SELECT Abfrage1 UNION [ALL] SELECT Abfrage2 [UNION [ALL]] ...

'Beispiel: Kunden sind auch Lieferanten - Umsatzvergleich
SELECT Kunde, Umsatz FROM Verkaufsliste UNION
SELECT Kunde, Umsatz FROM Lieferliste UNION
ORDER BY Kunde
```

6.2 Microsoft Access Objekt-Bibliothek

Jede Office-Anwendung besitzt eine Bibliothek in deren Namen auch die Version angegeben ist. So lautet der Bibliotheksname für die Access-Anwendung in Office Version 2010 „Microsoft Access 14.0 Object Library“. Dieses Kapitel befasst sich mit ihren Objekten und deren Eigenschaften, Methoden und Ereignissen.

6.2.1 Access-Anwendungen

Das Anwendungs-Objekt *Application* ist auch hier das oberste Objekt (Bild). Da es für die Benutzeroberfläche zuständig ist, enthalten Objektlisten wie *Forms*, *Reports* und *Modules* nur bereits geöffnete Objekte. Die Prozedur (Code 6.1) öffnet ein Formular und wählt darin den dritten Datensatz. Ohne zuvor das Formular zu öffnen wäre der Zugriff über die Anwendung nicht möglich.

Code 6.1 Die Prozedur wählt den dritten Datensatz einer Tabelle

```
Sub AppDoCommand()
   Dim appAccess As Application

   Set appAccess = Application
   With appAccess.DoCmd
      .OpenForm "Verkaufsliste", acLayout
      .GoToRecord acDataForm, "Verkaufsliste", acGoTo, 3
   End With
   Set appAccess = Nothing
End Sub
```

Das Objekt der Darstellung wird über die Anweisung *Open* der Methode *DoCmd* ausgewählt. Anhang 6, Tab. A6.2 zeigt eine Übersicht aller Anweisungen zu Objektdarstellungen.

Die Ansicht eines Formulars wird über Konstante gesteuert (Anhang 6, Tab. A6.3). Ein markiertes Steuerfeld-Objekt *Control* im geöffneten Formular bestimmt die Prozedur (Code 6.2).

Code 6.2 Die Prozedur ermittelt das aktive Steuerfeld im geöffneten Formular

```
Sub AktiveControl()
   Dim objForm     As Form
   Dim objControl  As Control

   Set objForm = Screen.ActiveForm 'aktives Formular
   Set objControl = Screen.ActiveControl 'aktives Control
```

```
   Debug.Print objControl.Name & _
      " im aktiven Formular " & _
      objForm.Name & " markiert."
   Set objControl = Nothing
   Set objForm = Nothing
End Sub
```

Die Prozedur (Code 6.3) öffnete eine neue Access-Anwendung und erstellt darin eine neue Personalliste mithilfe einer SQL-Anweisung.

Code 6.3 Die Prozedur erstellt eine neue Access-Datenbank mit einer leeren Tabelle

```
Sub NewApplication()
   Dim appAccess  As Application
   Dim objData    As Database
   Dim sFile      As String

   sFile = "C:\Temp\Test.accdb"
   If Not Dir(sFile) = "" Then
      Kill sFile
   End If
   Set appAccess = CreateObject("Access.Application")
   With appAccess
      .Visible = True
      .NewCurrentDatabase (sFile)
      Set objData = appAccess.CurrentDb
      objData.Execute "CREATE TABLE Personal " _
        & "(Vorname CHAR, Nachname CHAR, " _
        & "Geburtsdatum DATETIME, " _
        & "CONSTRAINT MyTableConstraint UNIQUE " _
        & "(Vorname, Nachname, Geburtsdatum));"
      .Quit acQuitPrompt
   End With
   Set objData = Nothing
   Set appAccess = Nothing
End Sub
```

Die Auswahl einer vorhandenen Datenbank kann sehr komfortabel mit der Methode *FileDialog* gestaltet werden. Die Prozedur (Code 6.4) gibt zwei Dateigruppen über den Filter vor. Dabei werden die Access-Datenbanken bevorzugt (FilterIndex = 2). Nach der Auswahl wird die Datenbank in einer neuen Anwendung geladen und ebenso eine vorhandene Tabelle.

Code 6.4 Die Prozedur öffnet eine ausgewählte Datenbank und eine Tabelle darin

```
Sub OpenExistDB()
'  Referenz erforderlich: Microsoft Office 14.0 Object Library
   Dim objDialog  As Office.FileDialog
   Dim appAccess  As Application
   Dim objData    As Database
   Dim sFile      As String

   Set objDialog = Application.FileDialog(msoFileDialogFilePicker)
   With objDialog
      .Title = "Bitte wählen ..."""
      .Filters.Clear
      .Filters.Add "Alle Dateien", "*.*"
      .Filters.Add "Access Datenbanken", "*.accdb"
      .FilterIndex = 2
      If .Show = True Then
         sFile = .SelectedItems(1)
         Set appAccess = CreateObject("Access.Application")
         appAccess.Visible = True
         appAccess.OpenCurrentDatabase (sFile)
         appAccess.DoCmd.OpenTable "Personal", acViewNormal
      Else
         MsgBox "Keine Auswahl!"
      End If
   End With
   Stop
   Set objDialog = Nothing
End Sub
```

6.2.2 DoCmd

Die Eigenschaft *DoCmd* des Anwendungs-Objekts bringt eine Vielzahl Methoden für Aktionen in der Datenbank mit. Die Prozedur (Code 6.5) öffnet das Formular *Verkaufsliste* und filtert die Datensätze nach der *Region West*.

Code 6.5 Die Prozedur zeigt eine gefilterte Liste

```
Sub DoOpenForm()
   With Application.DoCmd
      .OpenForm "Verkaufsliste"
      .SetFilter WhereCondition:="[Region] Like ""West"""
   End With
End Sub
```

In Anhang 6, Tab. A6.4 sind die wichtigsten Methoden des DoCmd-Objekts aufgeführt.

Die Prozedur (Code 6.6) schließt das aktive Formular mit der Methode *Close*. Zuvor sorgt die Methode *RunCommand* mit der Konstanten *acCmdSaveRecord* dafür, dass alle Änderungen gespeichert werden ohne danach zu fragen. Ist kein Formular geöffnet, dann wird die Fehlerbeschreibung ausgegeben.

Code 6.6 Die Prozedur schließt ein aktives Formular mit Speicherung

```
Sub DoCloseForm()
   On Error GoTo ErrorHandler
   DoCmd.RunCommand acCmdSaveRecord
   DoCmd.Close
   Exit Sub
ErrorHandler:
   MsgBox Err.Description
End Sub
```

In der Objektbibliothek sind unter *AcCommand-Enumeration* 677 verschiedene Werte aufgeführt und *acCmdSaveRecord* ist einer davon. Wie mächtig die Methoden sind zeigt die Prozedur (Code 6.7). Eine Abfrage wird mit dem Objekt *DoCmd* und der Methode *OutputTo* in eine Excel-Arbeitsmappe exportiert.

Code 6.7 Die Prozedur exportiert eine Abfrage in eine Excel-Arbeitsmappe

```
Sub DoExportQuery()
   With Application.DoCmd
      .OpenQuery "Umsatzliste Region", acViewNormal
      .OutputTo acOutputQuery, "Umsatzliste Region", _
         "ExcelWorkbook(*.xlsx)", _
         "C:\Temp\UmsatzRegion.xlsx", True, "", 0, _
         acExportQualityScreen
      .Close
   End With
   DoCmd.Close
End Sub
```

In den nächsten Kapiteln werden die Unterobjekte in der Access-Datenbank betrachtet. Insbesondere die Frage nach den Objektlisten und deren Zugriff. Auch dies wird wieder mit der *For-Each-Next* Schleife durchgeführt.

6.2.3 Tabellen

Das Objekt *Table* kennzeichnet eine Tabelle, wie sie bereits vorher schon erzeugt wurde. Die Bibliothek kennt jedoch kein *Table*-Objekt, sondern eine Objektliste *AllTables*. Ein Tabellen-Objekt kann als *AccessObject* definiert werden. Es verfügt über keine Ereignisse.

Die Prozedur (Code 6.8) liest alle Tabellen in der geöffneten Datenbank.

Code 6.8 Die Prozedur liest alle Tabellen (auch die Systemtabellen MSys...)

```
Sub ReadAllTable()
   Dim objTable As AccessObject

   For Each objTable In Application.CurrentData.AllTables
      Debug.Print objTable.Name
   Next
End Sub
```

Über die Eigenschaft *CurrentData* des Anwendungs-Objekts lassen sich alle Objektlisten (Anhang 6, Tab. A6.5) ansprechen.

Über die Eigenschaft *CurrentProject* des Anwendungs-Objekts lassen sich die anderen Objektlisten (Anhang 6, Tab. A6.6) ansprechen.

6.2.4 Formulare

Wie zuvor beschrieben können Formulare über die Objektliste *AllForms* angesprochen werden (Code 6.9).

Code 6.9 Die Prozedur liest alle Formulare der aktiven Datenbank

```
Sub ReadAllForms()
   Dim objForm    As AccessObject

   For Each objForm In Application.CurrentProject.AllForms
      Debug.Print objForm.FullName
   Next
End Sub
```

Für einzelne Formulare ist das *DoCmd*-Objekt ein guter Ansatz (Code 6.10).

Code 6.10 Die Prozedur öffnet und schließt ein Formular

```
Sub OpenCloseForm()
   Dim appAccess As Application

'öffnen
   Set appAccess = Application
   appAccess.DoCmd.OpenForm FormName:="Verkaufsliste"
   Stop
```

```
'schließen
   appAccess.DoCmd.Close acForm, "Verkaufsliste"
   Set appAccess = Nothing
End Sub
```

6.2.5 Berichte

Die Berichte-Objekte *Reports* können über die Objektliste *AllReports* aufgerufen werden (Code 6.11).

Code 6.11 Die Prozedur schreibt alle Reportnamen ins Direktfenster

```
Sub ReadAllReports()
   Dim objReport As AccessObject

   For Each objReport In Application.CurrentProject.AllReports
      Debug.Print objReport.FullName
   Next
End Sub
```

Mit der Methode *OpenReport* kann ein Bericht ausgegeben werden (Code 6.12).

Code 6.12 Die Prozedur druckt den angegebenen Report

```
Sub OpenReport()
   Dim appAccess As Application

   Set appAccess = Application
   With appAccess.DoCmd
      .OpenReport ReportName:="Umsatzliste Region"
   End With
   Set appAccess = Nothing
End Sub
```

6.2.6 Funktionen

Ähnlich den Funktionen *Minimalwert*, *Maximalwert*, etc. in Excel gibt es auch in Access ein paar wichtige Funktionen (Anhang 6, Tab. A6.7). Dabei handelt es sich um Positions- und Grenzwertbestimmungen. Sie gelten für alle Tabellen und Formulare der aktuellen Anwendung (Code 6.13).

Code 6.13 Die Prozedur liefert den Umsatz im ersten Datensatz der Verkaufsliste

```
Sub GoToFirst()
   Dim dUmsatz As Double
   dUmsatz = DFirst("Umsatz", "Verkaufsliste")
   Debug.Print dUmsatz
End Sub
```

Die Funktionen verfügen über Kriterienparameter, so dass auch eine Selektion möglich ist. Die Prozedur (Code 6.14) bestimmt den Kunden und den Umsatz einer bestimmten ID.

Code 6.14 Die Prozedur bestimmt den Kunden und den Umsatz von ID = 199

```
Sub SearchUmsatz()
   Dim vSearch As Variant
   vSearch = DLookup("Kunde", "Verkaufsliste", "[ID]=199")
   Debug.Print "Kunde: " & vSearch
   vSearch = DLookup("Umsatz", "Verkaufsliste", "[ID]=199")
   Debug.Print "Umsatz: " & vSearch
End Sub
```

Mehrere Kriterien lassen sich mit AND und OR verknüpfen. Die Prozedur (Code 6.15) bestimmt den Gesamtumsatz einer Region und Artikelart.

Code 6.15 Die Prozedur bestimmt den Gesamtumsatz der Region Ost an Wälzlagern

```
Sub UmsatzSummeWest()
   Dim dUmsatz As Double
   dUmsatz = DSum("[Umsatz]", "Verkaufsliste", _
      "[Region]='Ost' AND [Produktkategorie]='Wälzlager'")
   Debug.Print dUmsatz
End Sub
```

6.3 Data Access Objekt-Bibliothek

In Abschn. 6.1 wurde bereits das Objektmodell dieser Datenbank vorgestellt. Darin ist *DBEngine* das oberste Objekt.

6.3.1 DBEngine-Objekt

Über die Eigenschaft *DBEngine* des Anwendungs-Objekts kann auf das Objekt *DBEngine* zugegriffen werden. Die Prozedur (Code 6.16) liest alle Eigenschaften des *DBEngine*-Objekts der aktuellen Datenbank, die einen Wert besitzen.

Code 6.16 Die Prozedur zeigt alle lesbaren Eigenschaftswerte

```
Sub ReadEngineProperties()
   Dim objApp  As Application
   Dim iLoop   As Integer

   On Error Resume Next
   Set objApp = Application
   For iLoop = 0 To objApp.DBEngine.Properties.Count - 1
      With objApp.DBEngine.Properties(iLoop)
         Debug.Print .Name, .Value
      End With
   Next iLoop
   Set objApp = Nothing
End Sub
```

Das Objekt *DBEngine* verfügt über eine eigene Fehlerauflistung. Nicht zu verwechseln mit dem *Err*-Objekt von VBA. Die DAO-Fehlerliste *Errors* ist ein eigener Speicher und kann mehrere Fehler gleichzeitig speichern, da Operationen auf Datenbanken auch mehrere Fehler gleichzeitig erzeugen können. Gelöscht werden die Fehlereinträge erst mit der nächsten fehlerhaften Operation. Die Prozedur (Code 6.17) versucht eine nicht vorhandene Datenbank zu öffnen.

Code 6.17 Die Prozedur demonstriert eine DAO-Fehlerbehandlung

```
Sub CreateError()
   Dim objTest    As Database
   Dim sText      As String
   Dim iError     As Integer

   On Error GoTo ErrorHandler
'Versuch eine nicht vorhandene Datenbank zu öffnen
   Set objTest = DBEngine.Workspaces(0). _
      OpenDatabase("C:\Temp\Test.accdb")
   Exit Sub

ErrorHandler:
   For iError = 0 To DBEngine.Errors.Count - 1
      With DBEngine.Errors(iError)
         sText = "Fehlertext: " & .Description
         sText = sText & vbLf & "Fehlernummer: " & .Number
         sText = sText & vbLf & "Fehlerort: " & .Source
         sText = sText & vbLf & "Hilfstext: " & .HelpContext
         MsgBox sText, vbCritical, "FEHLERMELDUNG NR. " & _
            Str(iError)
      End With
   Next iError
End Sub
```

6.3.2 Workspace und Database

Das Objekt *Workspace* ist das direkte Unterobjekt des Objekts *DBEngine*. Ist eine Access-Datenbank geöffnet, dann enthält sie automatisch ein Objekt *Workspace* mit dem Index *Null*. Ein Objekt *Workspace* ist die Sitzung eines Anwenders in der mehrere Datenbanken geöffnet werden können. Mit der Methode *CreateWorkspace* lassen sich neue *Workspaces* erzeugen.

Das Objekt *Database* repräsentiert eine geöffnete Access Datenbank und ist somit ein Unterobjekt eines Objekts *Workspace*. Die Prozedur (Code 6.18) öffnet eine vorhandene Datenbank und gibt die Namen aller Tabellen im Direktfenster aus. Dabei sind auch die Systemtabellen (MSys...).

Code 6.18 Die Prozedur gibt alle Tabellen einer Access-Datenbank aus

```
Sub ListTables()
   Dim objData    As Database
   Dim lCount     As Long
   Dim lLoop      As Long
   Dim sPath      As String

   sPath = "C:\Temp\Test.accdb"
   Set objData = DBEngine.Workspaces(0).OpenDatabase(sPath)
   With objData
      lCount = .TableDefs.Count 'Anzahl Tabellen
      For lLoop = 0 To lCount - 1
         Debug.Print .TableDefs(lLoop).Name
      Next
      .Close
   End With
End Sub
```

Die Prozedur (Code 6.19) erzeugt ein weiteres Objekt *Workspace* und öffnet dazu eine Datenbank.

Code 6.19 Die Prozedur erzeugt einen neuen Workspace mit einem Database-Objekt

```
Sub OpenExterneDB()
   Dim objSpace   As Workspace
   Dim objData    As Database

   Set objSpace = CreateWorkspace( _
      Name:="Test", UserName:="admin", Password:="")
   Set objData = objSpace.OpenDatabase(Name:="C:\Temp\Test.accdb")
   Debug.Print objData.Version
   objData.Close
```

```
   Set objSpace = Nothing
   Set objData = Nothing
End Sub
```

Auf diesen Objekten lässt sich mit der Methode *Execute* und SQL-Anweisungen arbeiten. Die aktuelle Datenbank kann mit der Methode *CurrentDB* referenziert werden. In Prozedur (Code 6.20) schreibt eine Aktionsabfrage die *Umsätze* in eine neu angelegte Spalte *UmsatzZugfedern*, wenn die Produktkategorie mit *Zugfedern* angegeben ist.

Code 6.20 Die Prozedur füllt eine neue Spalte mit Daten

```
Sub DAOExecuteDemo()
   Dim objData    As Database
   Dim sSql       As String

   On Error GoTo ErrorHandler
   Set objData = CurrentDb
   sSql = "UPDATE [Verkaufsliste] " & _
      "SET [UmsatzZugfedern] = [Umsatz] " & _
      "WHERE [Produktkategorie] = 'Zugfedern'"
   objData.Execute sSql
   Application.RefreshDatabaseWindow
   Set objData = Nothing
   Exit Sub

ErrorHandler:
   MsgBox Err.Description
   Set objData = Nothing
End Sub
```

Die Methode *Execute* funktioniert wie die Methode *DoCmd.SQL*. Während *DoCmd.SQL* einfacher zu handhaben ist, funktioniert *Execute* auch auf gespeicherten Abfragen.

6.3.3 Transaktionen

Mit dem Arbeiten auf Datenbanken ist immer auch das Risiko verbunden, dass Änderungen nur teilweise ausgeführt werden, weil es in der laufenden Aktion zu einem Abbruch kam, z. B. durch Stromausfall. Dann muss mit dem Neustart der Datenbank dieses Problem vom System erkannt und der Urzustand wieder hergestellt werden.

Access stellt dazu die Methode *Transaktionen* zur Verfügung. Vor allen wichtigen Aktionen wird die Anweisung *BeginTrans* gesetzt und danach mit *CommitTrans* bestätigt. Erst durch die Bestätigung werden die Änderungen in der Datenbank akzeptiert. In der

Prozedur (Code 6.21) steht nach der Änderungsabfrage ein *Stop*. Wird mit dem Halt die Datenbank geschlossen und wieder geöffnet, so werden die Änderungen nicht durchgeführt, obwohl die Anweisung *Execute* bereits ausgeführt wurde.

Code 6.21 Demo zur Anwendung von Transaktionen

```
Sub DAOTransAction()
   Dim objData    As Database
   Dim sSql       As String

   BeginTrans
   Set objData = CurrentDb
   sSql = "UPDATE [Verkaufsliste] " & _
      "SET [UmsatzZugfedern] = [Umsatz] " & _
      "WHERE [Produktkategorie] = 'Zugfedern'"
   objData.Execute sSql
   Application.RefreshDatabaseWindow
   Stop
   CommitTrans
   Set objData = Nothing
End Sub
```

Ein Database Objekt enthält als Unterobjekte folgende Objektlisten:

- TableDefs,
- QueryDefs,
- Relations,
- RecordSets.

6.3.4 Tabellendefinitionen

Die Objektliste *TableDefs* bezieht sich auf die Tabellen einer *Database*-Sitzung und ein einzelnes Objekt *TableDef* enthält die Definitionswerte einer Tabelle. Die Prozedur (Code 6.22) nennt alle Feld-Objekte *Fields* der Tabelle *Verkaufsliste*. Die Feldnamen werden im Direktfenster ausgegeben.

Code 6.22 Die Prozedur liest alle Feldnamen der Tabelle

```
Sub ReadTableFields()
   Dim objData    As Database
   Dim objTable   As TableDef
   Dim objField   As Field
```

```
   Set objData = CurrentDb
   Set objTable = objData.TableDefs("Verkaufsliste")
   For Each objField In objTable.Fields
      Debug.Print objField.Name
   Next
   Set objTable = Nothing
   Set objData = Nothing
End Sub
```

Die zuvor manuell angelegte Spalte *UmsatzZugfedern* in der Tabelle *Verkaufsliste* lässt sich automatisch mit der Prozedur (Code 6.23) erzeugen.

Code 6.23 Die Prozedur hängt der Tabelle ein neues Feldelement an

```
Sub CreateTableField()
   Dim objData    As Database
   Dim objTable   As TableDef
   Dim objField   As Field

   Set objData = CurrentDb
   Set objTable = objData.TableDefs("Verkaufsliste")
   Set objField = objTable.CreateField("UmsatzZugfedern", dbDouble)
   objTable.Fields.Append objField
   Application.RefreshDatabaseWindow
   Set objField = Nothing
   Set objTable = Nothing
   Set objData = Nothing
End Sub
```

Im Objektkatalog sind unter *DataTypeEnum* alle Datentypen (wie hier *dbDecimal*) zu finden.

Mit dem Objekt *TableDef* lassen sich auch neue Tabellen generieren. Die Prozedur (Code 6.24) erzeugt eine Tabelle mit den Umsätzen einer ausgewählten Region.

Code 6.24 Die Prozedur erstellt eine neue Tabelle mit CreateTableDef

```
Sub CreateNewTable()
   Dim objData    As Database
   Dim objTable   As TableDef
   Dim sSql       As String

'Neue Tabelle mit CreateTableDef
   Set objData = CurrentDb
   Set objTable = objData.CreateTableDef("UmsatzWest")
```

```
   With objTable
      .Fields.Append .CreateField("Kunde", dbText, 150)
      .Fields.Append .CreateField("Umsatz", dbDouble)
   End With
   objData.TableDefs.Append objTable

'Füllen der Tabelle mit Execute SQL
   sSql = "INSERT INTO UmsatzWest (Kunde, Umsatz) " & _
      "SELECT Kunde, Umsatz FROM Verkaufsliste " & _
      "WHERE (((Verkaufsliste.Region) Like ""West""));"
   objData.Execute sSql
   objData.Close
   Application.RefreshDatabaseWindow
   Set objTable = Nothing
   Set objData = Nothing
End Sub
```

6.3.5 Abfragedefinitionen

Die Objektliste *QueryDefs* bezieht sich auf die Abfragen einer *Database*-Sitzung und ein einzelnes Objekt *QueryDef* enthält die Definitionswerte einer Abfrage. Die Prozedur (Code 6.25) nennt alle Feld-Objekte *Fields* der Tabelle *Verkaufsliste*. Die Feldnamen werden im Direktfenster ausgegeben.

Code 6.25 Die Prozedur liest alle Feldnamen der Abfrage

```
Sub ReadQueryFields()
   Dim objData    As Database
   Dim objTable   As QueryDef
   Dim objField   As Field

   Set objData = CurrentDb
   Set objTable = objData.QueryDefs("UmsatzOst")
   For Each objField In objTable.Fields
      Debug.Print objField.Name
   Next
   Set objTable = Nothing
   Set objData = Nothing
End Sub
```

Ebenso können Abfragen mit der Methode *CreateQueryDef* erzeugt werden (Code 6.26).

Code 6.26 Die Prozedur schreibt die Feldnamen einer Abfrage ins Direktfenster

```
Sub CreateNewQuery()
   Dim objData     As Database
   Dim objQuery    As QueryDef
   Dim sSql        As String

   sSql = "SELECT Verkaufsliste.Produktkategorie, " & _
      "Verkaufsliste.Kunde, Verkaufsliste.Umsatz " & _
      "FROM Verkaufsliste " & _
      "WHERE (((Verkaufsliste.Produktkategorie) Like ""*federn""));"
   Set objData = CurrentDb
   Set objQuery = objData.CreateQueryDef("UmsatzFedern", sSql)
   Application.RefreshDatabaseWindow
   Set objQuery = Nothing
   Set objData = Nothing
End Sub
```

6.3.6 Relationen

Zu einer Relationalen Datenbank gehören natürlich auch *Relationen*. Sie verwalten die Beziehungen zwischen Feldelementen von Tabellen. Im Beispiel existieren eine Verkaufsliste und eine Kundenliste. Um in einer Abfrage die Umsätze der Kunden zu sammeln, muss zuvor eine Relation zwischen beiden Tabellen erstellt werden (Abb. 6.4).

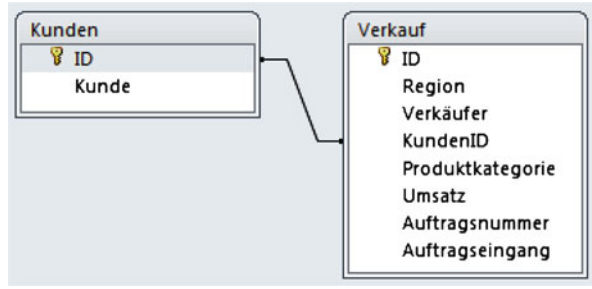


Abb. 6.4 Relation zwischen den Tabellen Kunden und Verkauf

Die Prozedur (Code 6.27) erstellt eine 1:n-Beziehung zwischen den Tabellen Kunden und Verkauf. Zuvor werden eventuell vorhandene Beziehungen gelöscht. Der Name der neuen Relation wird nach der Struktur *Tabelle1_Schlüssel1_Tabelle2_Schlüssel2* konstruiert, so dass die Felder *Schlüssel1* und *Schlüssel2* im Diagramm eine Beziehung aufweisen. In diesem Fall lautet der Beziehungsname *Kunden_ID_Verkauf_KundenID*.

Code 6.27 Die Prozedur erstellt eine 1:n Beziehung zwischen zwei Tabellen

```
Sub CreateRelation()
   Dim objData       As Database
   Dim objRelation   As Relation
   Dim objField      As Field
   Dim sRelation     As String
   Dim sFields       As String
   Dim lRelCount     As Long
   Dim lCount        As Long

'löschen vorhandener Beziehungen
   Set objData = CurrentDb
   lRelCount = objData.Relations.Count
   For lCount = lRelCount - 1 To 0 Step -1
      objData.Relations.Delete (objData.Relations(lCount).Name)
   Next lCount
'Relation erstellen
   sRelation = "Kunden_ID_Verkauf_KundenID"
   Set objRelation = objData.CreateRelation()
   With objRelation
      .Name = sRelation          'Relationsname
      .Table = "Kunden"          'Primärschlüssel
      .ForeignTable = "Verkauf"  'Fremdschlüssel
      .Attributes = dbRelationDontEnforce '1:n Beziehung
   End With
'Feld-Beziehungen
   sFields = "ID_KundenID"
   Set objField = objRelation.CreateField(sFields, dbLong)
   With objField
      .Name = "ID"
      .ForeignName = "KundenID"
   End With
'Feldbeziehung der Relation hinzufügen
   objRelation.Fields.Append objField
'Relation der Datenbank hinzufügen
   objData.Relations.Append objRelation

   Set objField = Nothing
   Set objRelation = Nothing
   Set objData = Nothing
End Sub
```

Die Konstanten zur Anweisung *CreateRelation* stehen unter dem Begriff *RelationAttributeEnum* im Objektkatalog.

Mit der vorhandenen Beziehung kann die Abfrage für die Kundenumsätze erstellt werden (Code 6.28).

Code 6.28 Die Prozedur wertet Daten aus zwei Tabellen aus

```
Sub CreateKundenUmsatz()
   Dim objData    As Database
   Dim objQuery   As QueryDef
   Dim sSql       As String

   sSql = "SELECT Kunden.ID, Kunden.Kunde, " & _
      "Sum(Verkauf.Umsatz) AS SummevonUmsatz " & _
      "FROM Kunden " & _
      "INNER JOIN Verkauf " & _
      "ON Kunden.ID = Verkauf.KundenID " & _
      "GROUP BY Kunden.ID, Kunden.Kunde;"
   Set objData = CurrentDb
   Set objQuery = objData.CreateQueryDef("KundenUmsatz", sSql)
   Application.RefreshDatabaseWindow
   Set objQuery = Nothing
   Set objData = Nothing
End Sub
```

Eine Relation kann durch die Methode *Delete* auch wieder gelöscht werden (Code 6.29).

Code 6.29 Die Prozedur löscht eine bestimmte Beziehung

```
Sub DeleteRelation()
   On Error Resume Next 'falls nicht vorhanden
   CurrentDb.Relations.Delete "Kunden_ID_Verkauf_KundenID"
End Sub
```

6.3.7 Datensätze

Das Datensatz-Objekt *RecordSet* bezieht sich in einer Tabelle oder Abfrage auf eine Zeile, die aus mehreren Feldelementen besteht. Das übergeordnete Objekt ist auch hier *Database*. Die Prozedur (Code 6.30) liest alle Datensätze der Tabelle Kunden und gibt die Werte der Felder im Direktfenster aus.

Code 6.30 Die Prozedur liest alle Datensätze einer Tabelle

```
Sub ReadTable()
   Dim objData    As Database
   Dim objRecord  As Recordset
```

```
    Set objData = CurrentDb
    Set objRecord = objData.OpenRecordset _
       ("Kunden", dbOpenTable, dbReadOnly)
    With objRecord
       Do Until .EOF
          Debug.Print .Fields("ID"), .Fields("Kunde")
          .MoveNext
       Loop
    End With
    Set objRecord = Nothing
    Set objData = Nothing
End Sub
```

Den jeweils aktuellen Datensatz markiert ein Zeiger, der mit der Instanziierung des *RecordSets* auf den ersten Datensatz zeigt. Die Grenzen des Zeigers werden durch Konstante (Anhang 6, Tab. A6.8) gesteuert.

Die Bewegung des Datensatzzeigers wird durch *Move*-Methoden gesteuert (Anhang 6, Tab. A6.9).

Ein Datensatz wird mit der Methode *OpenRecordset* instanziiert.

```
'Syntax:
   OpenRecordset (Name, [Typ], [Optionen], [Modus])
```

Als *Name* wird der Name der Tabelle oder Abfrage eingesetzt. Als *Typ* wird eine Konstante (Anhang 6, Tab. A6.10) je nach Anwendung eingetragen. Mit *Optionen* (Anhang 6, Tab. A6.11) können mehrere Angaben gesetzt werden. Es ist dann die Summe der Werte. Der letzte Parameter *Modus* wird in (Anhang 6, Tab. A6.12) beschrieben.

So einfach wie Datensätze über ein Objekt *Recordset* gelesen werden (Code 6.30), so einfach können sie auch editiert werden. Mit der Methode *Edit* werden die Datenfelder des Datensatzes in einen Puffer kopiert. Wertzuweisungen an die Datenfelder stehen danach auf dem Puffer. Mit der Eigenschaft *OriginalValue* können immer noch die Originalwerte abgefragt werden. Mit der Methode *Update* werden die Änderungen vom Puffer an den Datensatz endgültig übertragen.

Die Prozedur (Code 6.31) ändert im Datenfeld Verkäufer alle Datensätze mit dem Eintrag *Lehrer* in *Müller*.

Code 6.31 Die Prozedur ändert alle Datensätze mit einem bestimmten Datenfeldinhalt

```
Sub UpdateTable()
   Dim objData    As Database
   Dim objRecord  As Recordset
```

```
   Set objData = CurrentDb
   Set objRecord = objData.OpenRecordset _
      ("Verkauf", dbOpenTable, dbOpenTable)
   With objRecord
      Do Until .EOF
         If .Fields("Verkäufer") = "Lehrer" Then
            .Edit
            !Verkäufer = "Müller"
            .Update
         End If
         .MoveNext
      Loop
   End With
   Set objRecord = Nothing
   Set objData = Nothing
End Sub
```

Mit der Methode *AddNew* wird ein neuer Puffer erzeugt, der nach Wertzuweisungen an die Datenfelder mit der Methode *Update* an die vorhandenen Datensätze angehängt wird. Die Methode *Delete* löscht den aktuellen Datensatz.

Ähnlich wie die Methode *Move* gibt es die Methode *Find* auf Datensätze, die zum Suchen dient. Anhang 6, Tab. A6.13 enthält die *Find*-Methoden.

Die Prozedur (Code 6.32) findet alle Datensätze in der Abfrage *KundenUmsatz*, deren Umsatzsumme größer als 100.000 ist.

Code 6.32 Die Prozedur findet alle Datensätze in einer Abfrage, deren Umsatz größer als 100.000 ist

```
Sub FindInQuery()
   Dim objData    As Database
   Dim objRecord  As Recordset
   Dim sFind      As String

   sFind = "[SummevonUmsatz] > 100000"
   Set objData = CurrentDb
   Set objRecord = objData.OpenRecordset _
      ("KundenUmsatz", dbOpenDynaset)
   With objRecord
      .FindFirst sFind
      Do Until .NoMatch = True
         Debug.Print !Id, !Kunde, !SummevonUmsatz
         .FindNext sFind
      Loop
   End With
```

```
   Set objRecord = Nothing
   Set objData = Nothing
End Sub
```

Alternativ zur Methode *Find* können Datensätze auch über die Eigenschaft *Filter* eines Datensatzes gefunden werden (Code 6.33). Dabei hilft auch, dass ein Datensatz aus einem Datensatz erzeugt werden kann.

Code 6.33 Die Prozedur erzeugt ein Recordset aus einem Recordset.Filter

```
Sub FilterQuery()
   Dim objData    As Database
   Dim objRec1    As Recordset
   Dim objRec2    As Recordset
   Dim sFilter    As String

   sFilter = "[SummevonUmsatz] > 100000"
   Set objData = CurrentDb
   Set objRec1 = objData.OpenRecordset _
      ("KundenUmsatz", dbOpenDynaset)
   objRec1.Filter = "SummevonUmsatz > 100000"
   Set objRec2 = objRec1.OpenRecordset
   With objRec2
      Do Until .EOF
         Debug.Print !Id, !Kunde, !SummevonUmsatz
         .MoveNext
      Loop
   End With
   Set objRec2 = Nothing
   Set objRec1 = Nothing
   Set objData = Nothing
End Sub
```

Auf ähnliche Weise lässt sich auch mit der Eigenschaft *Sort* verfahren (Code 6.34).

Code 6.34 Die Prozedur erzeugt ein Recordset aus einem Recordset.Sort

```
Sub SortQuery()
   Dim objData    As Database
   Dim objRec1    As Recordset
   Dim objRec2    As Recordset
   Dim sSort      As String
```

```
    sSort = "SummevonUmsatz DESC"
    Set objData = CurrentDb
    Set objRec1 = objData.OpenRecordset _
       ("KundenUmsatz", dbOpenDynaset)
    objRec1.Sort = sSort
    Set objRec2 = objRec1.OpenRecordset
    With objRec2
       Do Until .EOF
          Debug.Print !Id, !Kunde, !SummevonUmsatz
          .MoveNext
       Loop
    End With
    Set objRec2 = Nothing
    Set objRec1 = Nothing
    Set objData = Nothing
End Sub
```

6.4 ActiveX Data Objekt-Bibliothek

Bei der Betrachtung des Objektmodells der *ActiveX Data Objects Library* (ADO) ist keine Strukturhierarchie wie in den anderen Bibliotheken zu finden. Vielmehr gibt es mehrere unabhängige Objektgruppen.

6.4.1 Datenbank-Verbindungen

In der ADO-Bibliothek ist ein *Connection* Objekt, wie der Name schon sagt, die Verbindung zu einer Datenbank. Die Prozedur (Code 6.35) erstellt eine solche Verbindung und gibt über die Eigenschaft *ConnectionString* die Daten der Verbindung aus (Abb. 6.5).

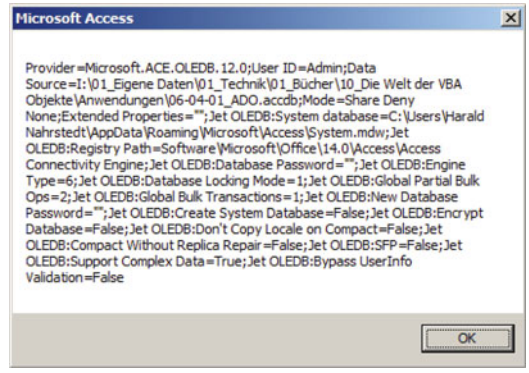


Abb. 6.5 Beispiel von Verbindungsdaten zur aktuellen Datenbank

Code 6.35 Die Prozedur zeigt die Daten einer Verbindung zur aktuellen Datenbank

```
Sub ConnectionDatabase()
   Dim objConnect As Connection

   Set objConnect = CurrentProject.Connection
   MsgBox objConnect.ConnectionString
   Set objConnect = Nothing
End Sub
```

Mit der Methode *Open* einer Verbindung lassen sich auch externe Datenbanken öffnen (Code 6.36).

Code 6.36 Die Prozedur erstellt eine Verbindung zu einer externen Datenbank

```
Sub ConnectionExternDatabase()
   Dim objConn As ADODB.Connection

   Set objConn = New Connection
   objConn.Open "Provider=Microsoft.ACE.OLEDB.12.0;" & _
      "Data Source=C:\Temp\Test.accdb"
   MsgBox objConn.ConnectionString
   objConn.Close
   Set objConn = Nothing
End Sub
```

Für den Fall, dass eine Verbindung zu einer älteren Datenbank vom Typ (*.mdb) erstellt werden soll, muss ein anderer *Provider* verwendet werden. Eine Verbindung verfügt über die Eigenschaft *Provider*, so dass dieser bereits vor der Methode *Open* gesetzt werden kann (Code 6.37).

Code 6.37 Die Prozedur erstellt eine Verbindung zu einer älteren Datenbank

```
Sub ConnectionExternDatabaseOld()
   Dim objConn As ADODB.Connection

   Set objConn = New Connection
   With objConn
      .Provider = "Microsoft.Jet.OLEDB.4.0"
      .Open "Data Source=C:\Temp\Test.mdb"
      MsgBox .ConnectionString
      .Close
   End With
   Set objConn = Nothing
End Sub
```

Die Methode *Execute* der Verbindung erlaubt die Ausführung von SQL-Anweisungen (Code 6.38).

Code 6.38 Die Prozedur wertet Datensätze aus und beschreibt eine neue Spalte

```
'Die Spalte UmsatzDruckfedern manuell erstellen
Sub ConnectionExecute()
   Dim objConn As ADODB.Connection
   Dim sSql    As String
   Dim lCount  As Long

   sSql = "UPDATE Verkaufsliste " & _
      "SET UmsatzDruckfedern = Umsatz " & _
      "WHERE Produktkategorie = 'Druckfedern'"
   Set objConn = CurrentProject.Connection
   objConn.Execute sSql, lCount
   MsgBox lCount & " Datensätze gefunden!"
   Application.RefreshDatabaseWindow
   Set objConn = Nothing
End Sub
```

6.4.2 Transaktionen

Mit Objekten der ADO-Bibliothek können ebenso wie mit den Objekten der DAO-Bibliothek *Transaktionen* durchgeführt werden. Sind es unter DAO die Objekte *Workspace*, unter denen die *Transaktionen* ablaufen, so sind es unter ADO die Objekte *Connection*. Die Namen der Methoden sind identisch und lauten auch hier *BeginTrans*, *CommitTrans* und *RollbackTrans*. Die Prozedur (Code 6.39) zeigt eine Anwendung dieser Methoden.

Code 6.39 Demo zur Anwendung von Transaktionen

```
Sub ADOTransAction()
   Dim objConn As ADODB.Connection
   Dim sSql       As String

   sSql = "UPDATE Verkaufsliste " & _
      "SET UmsatzDruckfedern = Umsatz " & _
      "WHERE Produktkategorie = 'Druckfedern'"
   Set objConn = CurrentProject.Connection
   With objConn
      .BeginTrans
      .Execute sSql
      Application.RefreshDatabaseWindow
```

```
      Stop
      .CommitTrans
      Application.RefreshDatabaseWindow
   End With
   Set objConn = Nothing
End Sub
```

6.4.3 Befehle

Ein Objekt *Command* besitzt wie ein Objekt *Connection* ebenfalls die Methode *Execute*. Mit ihr lassen sich Aufgaben erledigen, die auch die Methode *Execute* des Objekts *Connection* erledigen kann. Doch die Methode *Execute* ist unter dem Objekt *Command* wesentlich leistungsfähiger.

```
'Syntax:
Command.Execute (RecordsAffected, Parameters, Options)
```

Anhang 6, Tab. A6.14 beschreibt die Parameter der Methode Execute.

Betrachten wir dazu eine Parameterabfrage mit dem Namen *VerkäuferUmsatz* die zuvor manuell erstellt wurde. Die Entwurfsansicht dazu zeigt (Abb. 6.6). In der Spalte Verkäufer wird als Parameter der Verkäufername erwartet. Diese Abfrage nutzt die Prozedur (Code 6.40) für eine Auswertung mit dem Verkäufernamen *Lehmann* und gibt das Ergebnis als Datensatz im Direktfenster aus.

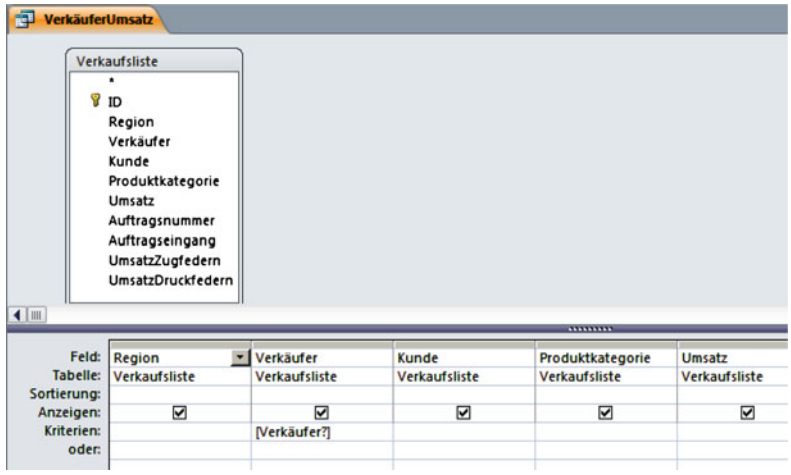


Abb. 6.6 Entwurfsansicht einer Parameterabfrage

Code 6.40 Die Prozedur nutzt eine Parameterabfrage zur Auswertung im Direktfenster

```
'die Parameterabfrage VerkäuferUmsatz muss existieren
Sub CommandExecute()
   Dim objConn As ADODB.Connection
   Dim objCom  As ADODB.Command

   Set objConn = CurrentProject.Connection
   Set objCom = New ADODB.Command
   With objCom
      .ActiveConnection = objConn
      .CommandText = "VerkäuferUmsatz"
      .CommandType = adCmdTable
      Debug.Print .Execute(, "Lehmann").GetString
   End With
   Set objCom = Nothing
   Set objConn = Nothing
End Sub
```

Die Methode *GetString* gehört zum Objekt *Recordset*, das nachfolgend besprochen wird.

6.4.4 Datensätze

Die Datensatz-Objekte *Recordsets* der ADO-Bibliothek sind nahezu identisch mit den Objekten der DAO-Bibliothek. Lediglich ein paar Unterschiede müssen beachtet werden.

Ein Datensatz unter ADO wird zunächst instanziiert und bekommt dann mit der Methode *Open* die entsprechenden Daten zugewiesen.

```
'Syntax:
Open Source, ActiveConnection, CursorType, LockType, Options
```

Anhang 6, Tab. A6.15 zeigt die Parameter der Methode *Open* des Recordsets-Objekts.

Das übergeordnete Objekt ist auch das Objekt *Connection*. Die Prozedur (Code 6.41) liest alle Datensätze der Tabelle *Verkaufsliste* und gibt die Werte der Felder im Direktfenster aus.

Code 6.41 Die Prozedur liest alle Datensätze einer Tabelle

```
Sub ReadTable()
   Dim objConn As ADODB.Connection
   Dim objRec  As ADODB.Recordset
```

```
   Set objConn = CurrentProject.Connection
   Set objRec = New ADODB.Recordset
   objRec.Open "Verkaufsliste", objConn, _
      adOpenDynamic, adLockOptimistic
   With objRec
      Do Until .EOF
         Debug.Print .Fields("ID"), .Fields("Kunde"), _
            .Fields("Umsatz")
         .MoveNext
      Loop
   End With
   Set objRec = Nothing
   Set objConn = Nothing
End Sub
```

Das Navigieren in den Datensätzen mittels Datensatzzeiger funktioniert mit den gleichen Methoden wie bei den DAO-Objekten, ebenso die Grenzwerte BOF und EOF.

Einfacher ist das Ändern von Feldwerten eines Datensatzes. Es muss eine Änderung weder mit *Edit* angezeigt, noch mit *Update* übernommen werden. Werte können den Datenfeldern direkt zugewiesen werden. Mit dem Verlassen eines Datensatzes werden die Änderungen automatisch übernommen.

Die Prozedur (Code 6.42) ändert im Datenfeld *Verkäufer* alle Datensätze mit dem Eintrag *Weinreich* in *Steinreich*.

Code 6.42 Die Prozedur ändert alle Datensätze mit einem bestimmten Datenfeldinhalt

```
Sub UpdateTable()
   Dim objConn As ADODB.Connection
   Dim objRec  As ADODB.Recordset

   Set objConn = CurrentProject.Connection
   Set objRec = New ADODB.Recordset
   objRec.Open "Verkaufsliste", objConn, _
      adOpenDynamic, adLockOptimistic

   With objRec
      Do Until .EOF
         If .Fields("Verkäufer") = "Weinreich" Then
            !Verkäufer = "Steinreich"
         End If
         .MoveNext
      Loop
   End With
```

```
   Application.RefreshDatabaseWindow
   Set objRec = Nothing
   Set objConn = Nothing
End Sub
```

Mit der Methode *AddNew* wird ein neuer Puffer erzeugt, der nach Wertzuweisungen an die Datenfelder mit der Methode *Update* an die vorhandenen Datensätze angehängt wird. Die Methode *Delete* löscht den aktuellen Datensatz.

Auch bei diesem Recordset gibt es die Methode *Find*. Doch anders als beim DAO-Datensatz mit den Methoden *FindFirst* und *FindNext* gibt es hier nur die Methode *Find*, die nur ein Suchkriterium (Datenfeld) verarbeitet (Code 6.43). Es gibt auch nicht die Möglichkeit, Datenfelder mit boolschen Operatoren zu verknüpfen.

```
'Syntax:
Find (Criteria, SkipRows, SearchDirection, Start)
```

Anhang 6, Tab. A6.16 zeigt die Parameter der Methode *Find* des Recordset-Objekts.

Code 6.43 Die Prozedur sucht ausgewählte Datensätze

```
Sub SearchRecords()
   Dim objConn As ADODB.Connection
   Dim objRec  As ADODB.Recordset
   Dim sSQL    As String

   sSQL = "Produktkategorie LIKE 'Metallschrauben'"
   Set objConn = CurrentProject.Connection
   Set objRec = New ADODB.Recordset
   With objRec
      .Open "Verkaufsliste", objConn, adOpenStatic
      .Find sSQL
      Do While Not .EOF
         Debug.Print _
            "ID: " & !Id & _
            " Kunde: " & !Kunde & _
            " Umsatz: " & !Umsatz
         .Find sSQL, 1 'nächster Datensatz
      Loop
   End With
   Set objRec = Nothing
   Set objConn = Nothing
End Sub
```

Die Anwendung der Methode *Filter* ist bei ADO deutlich einfacher. Über die Zuweisung einer SQL-Anweisung an die Eigenschaft *Filter* des ADO-Datensatzes wird diese direkt ausgeführt (Code 6.44).

Code 6.44 Die Prozedur erzeugt ein Recordset aus einem Recordset.Filter

```
Sub FilterRecords()
   Dim objConn As ADODB.Connection
   Dim objRec  As ADODB.Recordset
   Dim sSQL    As String

   sSQL = "Region = 'Nord'"
   Set objConn = CurrentProject.Connection
   Set objRec = New ADODB.Recordset
   sSQL = "Region = 'Nord'"
   With objRec
      .Open "Verkaufsliste", objConn, adOpenStatic
      .Filter = sSQL
      Do Until .EOF
         Debug.Print !Id, !Kunde, !Umsatz
         .MoveNext
      Loop
   End With
   Set objRec = Nothing
   Set objConn = Nothing
End Sub
```

Ebenso wird die Methode *Sort* angewendet (Code 6.45).

Code 6.45 Die Prozedur erzeugt ein Recordset aus einem Recordet.Sort

```
Sub SortRecords()
   Dim objConn As ADODB.Connection
   Dim objRec  As ADODB.Recordset
   Dim sOpen   As String
   Dim sSort   As String
   Dim sx As String

   Set objConn = CurrentProject.Connection
   Set objRec = New ADODB.Recordset
   objRec.CursorLocation = adUseClient
   sOpen = "SELECT * FROM Verkaufsliste"
   objRec.Open sOpen, objConn, adOpenStatic, adLockReadOnly
   sSort = "Verkäufer ASC, Kunde ASC"
   With objRec
      .Sort = sSort
```

```
      Do Until .EOF
         Debug.Print !Id, !Verkäufer, !Kunde, !Umsatz
         .MoveNext
      Loop
   End With
   Set objRec = Nothing
   Set objConn = Nothing
End Sub
```

In der Prozedur wird die Eigenschaft *CursorLocation* gesetzt und ist für die Anwendung auch erforderlich. Die Eigenschaft ermöglicht zwischen verschiedenen Cursorbibliotheken auszuwählen, auf die der Anbieter zugreifen kann, in der Regel zwischen einer clientseitigen Cursorbibliothek und einer Cursorbibliothek auf dem Server. Die Zuweisung wirkt sich nur auf Verbindungen aus, die nach dem Festlegen der Eigenschaft eingerichtet wurden. Das Ändern der *CursorLocation*-Eigenschaft hat keine Auswirkung auf vorhandene Verbindungen.

6.5 Access-Formulare

In Abschn. 6.2 wurde bereits das Lesen von Formularen und Berichten beschrieben. Mit den danach behandelten Objekten und ihren Methoden und Eigenschaften können nun auch Formulare und Berichte neu erstellt werden.

6.5.1 Ungebundene Formulare

Betrachten wir im ersten Schritt die Erstellung eines Formulars (Code 6.46).

Code 6.46 Die Prozedur erzeugt ein leeres Formular

```
Sub CreateFormular()
   Dim objForm As Form
   Dim sName   As String

   Set objForm = Application.CreateForm
   sName = objForm.Name
   With DoCmd
      .SetWarnings False
      .Restore
      .Save acForm, sName
      .Close acForm, sName
      .Rename "Demo", acForm, sName
      .SetWarnings True
```

```
    End With
    Set objForm = Nothing
End Sub
```

Die Methode *CreateForm* des Anwendungs-Objekts *Application* erzeugt ein leeres Formular. Das System vergibt für dieses Formular einen Namen, der in der Regel *Formular*(Nr) lautet, wobei Nr eine fortlaufende Nummerierung darstellt. Mit der Methode *Save* des Objekts *DoCmd* wird das Formular unter dem erzeugten Namen gespeichert. Zuvor sorgt die Methode *SetWarnings* dafür, dass **keine** Fehlermeldung erfolgt, wenn ein Formular mit diesem Namen bereits existiert. Ebenso bei der Methode *Rename*, die das vorhandene Formular in *Demo* umbenennt. Danach wird die Fehlerausgabe wieder freigeschaltet. Wird das Formular minimiert dargestellt, dann hilft die Methode *Restore* zu einer Maximaldarstellung.

Aus dem Kapitel über Formulare wissen wir, dass zu einer Oberfläche auch ein Formularmodul gehört. In der Entwicklungsumgebung wird unter Einfügen aber kein Formular vorgegeben, so wie in anderen Office-Anwendungen. Hier hat Access seine Eigenart. Solange ein Formular keinen Code enthält, wird im Projekt-Explorer auch das zugehörige Formularmodul nicht erscheinen.

Die Prozedur (Code 6.47) öffnet das leere Formular erneut und setzt die Eigenschaft *HasModule* des Formulars auf *True*. Dadurch wird dann auch das Formularmodul sichtbar (Abb. 6.7).

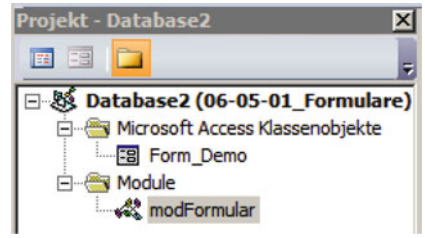


Abb. 6.7 Neues Formularmodul der Klassenobjekte

Code 6.47 Die Prozedur setzt die Eigenschaft HasModul eines Formulars auf True

```
Sub CreateForm()
    Dim objForm As Form
    Dim sName   As String

    sName = "Demo"
    DoCmd.OpenForm sName, acDesign
    Set objForm = Forms(sName)
    objForm.HasModule = True
    Set objForm = Nothing
End Sub
```

Der Name des Formularmoduls setzt sich aus dem Namen des Formulars und der Präfix *Form_* zusammen. Die Objektliste *Forms* enthält alle geöffneten Formulare der aktuellen Datenbank. Im Codefenster des Moduls finden sich dann auch alle Ereignisprozeduren (Abb. 6.8).

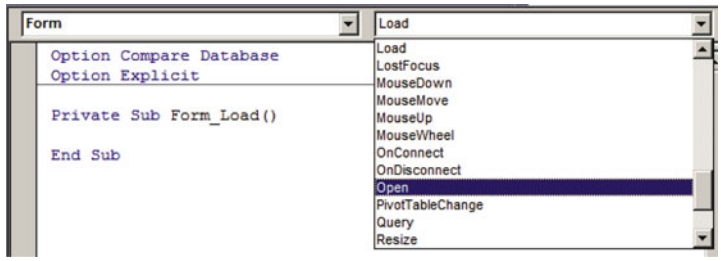


Abb. 6.8 Codefenster eines Formularmoduls

Ein Unterobjekt des Formulars ist die Objektliste der Bereiche, *Section* 0 bis 2. Sie werden über einen Index (Anhang 6, Tab. A6.17) angesprochen und ihre Lage zeigt (Abb. 6.9).

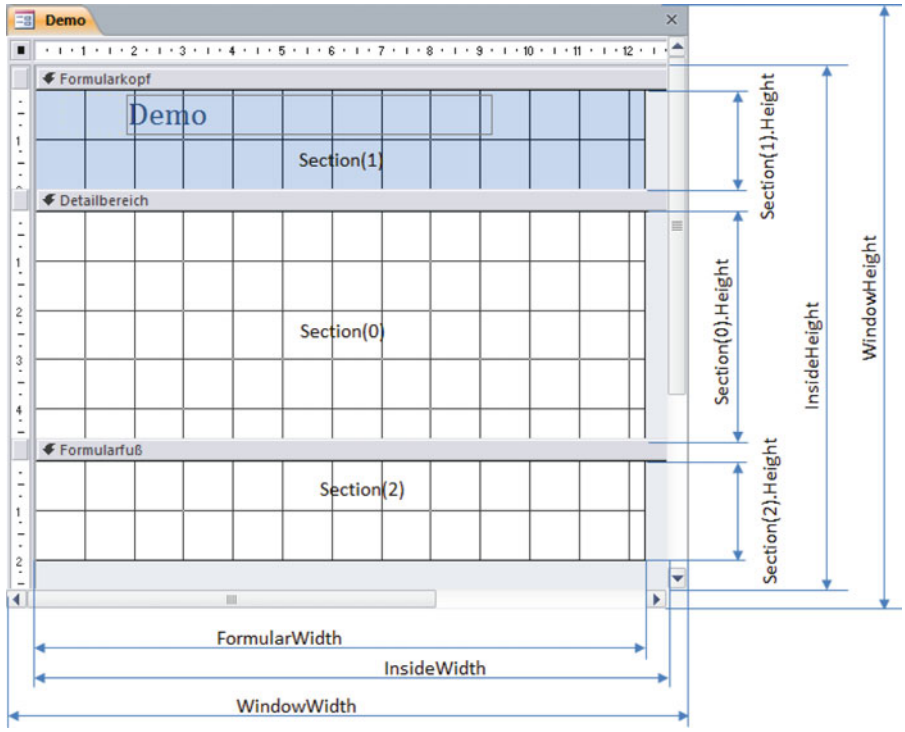


Abb. 6.9 Formular Design-Eigenschaften

Im Formularmodul können alle weiteren Eigenschaften des Formulars gesetzt werden. Die Prozedur (Code 6.48) passt die Abmessungen des Formulars beim Event *Load* an die vorhandene Fenstergröße an.

Code 6.48 Die Prozedur passt ein Formular dem nutzbaren Fenster an

```
Private Sub Form_Load()
   Dim iFormH  As Integer
   Dim iFormB  As Integer
   Dim iWindowH   As Integer
   Dim iWindowB   As Integer

   With Me
      iFormH = .Section(0).Height _
         + .Section(1).Height _
         + .Section(2).Height
      iFormB = .Width
      iWindowH = .InsideHeight
      iWindowB = .InsideWidth
      If iWindowB <> iFormB Then
         .InsideWidth = iFormB
      End If
      If iWindowH <> iFormH Then
         .InsideHeight = iFormH
      End If
   End With
End Sub
```

Neben den Abmessungen gibt es noch eine Vielzahl von Designmöglichkeiten mit Farben und Formen. Die Eigenschaft *Visible* eines Bereichs legt fest, ob dieser sichtbar ist. Die Eigenschaft *DisplayWhen* legt für einen Bereich fest, ob dieser nur dargestellt, nur gedruckt oder für beides sichtbar ist.

Die Vielzahl der Ereignisse tritt oft in einer Ablauffolge nacheinander auf. So werden die Events *Open*, *Load*, *Resize*, *Activate*, *GotFocus* und *Current* nacheinander aufgerufen, wenn ein Formular geöffnet wird. Bei der Programmierung muss dies in jedem Fall beachtet werden. Ebenso bei Maus- und Tastatur-Ereignissen.

Ein Formular verfügt ebenfalls über Filter- und Datenoperations-Ereignisse. Die wichtigsten zeigt Anhang 6, Tab. A6.18.

Neben den vielen Ereignissen besitzt ein Formular nur wenige Methoden. So setzen die Methoden *GoToPage* und *SetFocus* den Focus, während *Recalc*, *Refresh*, *Repaint* und *Requery* eine Aktualisierung der entsprechenden Objekte vornehmen. Eine Zusammenstellung steht im Objektkatalog unter *Elemente des Form-Objekts*.

6.5.2 Steuerelemente

Ein Formular ohne Steuerelemente ist nahezu wertlos. Bereits in Abschn. 1.3 wurden Steuerelemente und ihre Anwendung auf Formularen beschrieben. Mit der Methode *CreateControl* werden Steuerelemente auf dem Formular erzeugt. Den möglichen Typnamen eines Controls gibt die Aufzählung unter *AcControlType* in der Access-Hilfe. So ist *acTextBox* ein Textfeld und *acLabel* ein Beschriftungsfeld.

Die Prozedur (Code 6.49) öffnet das erzeugte Formular *Demo* im Entwurfsmodus, denn nur in diesem lassen sich Steuerelemente erzeugen, und instanziiert ein Textfeld mit einem Bezeichnungsfeld. Die Positionierung wird durch die Angaben von Bereich und Abmessungen gestaltet (Abb. 6.10).

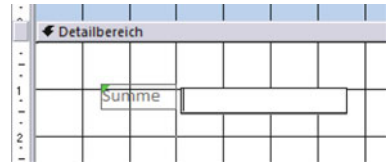


Abb. 6.10 Die erzeugten Controls

Code 6.49 Die Prozedur erzeugt zwei Controls auf einem Formular

```
Sub CreateControls()
   Dim objForm    As Form
   Dim ctlText    As Control
   Dim ctlLabel   As Control
   Dim sName      As String

   sName = "Demo"
   DoCmd.OpenForm sName, acDesign
   Set objForm = Forms(sName)

   Set ctlText = Application.CreateControl _
      (FormName:=objForm.Name, _
      ControlType:=acTextBox, _
      Section:=acDetail, _
      Left:=1000, Top:=300, Width:=2000, Height:=300)
   ctlText.Properties("Name") = "Ergebnis"

   Set ctlLabel = Application.CreateControl _
      (FormName:=objForm.Name, _
      ControlType:=acLabel, _
      Section:=acDetail, _
      Left:=100, Top:=300, Width:=900, Height:=300)
   ctlLabel.Properties("Caption") = "Summe"
```

```
   Set ctlText = Nothing
   Set ctlLabel = Nothing
   Set objForm = Nothing
End Sub
```

Die Prozedur (Code 6.50) erstellt ein Formular zur Berechnung von Kreisparametern, das nur über die Eingabewerte agiert. In den Textfeldern können wahlweise Werte wie *Radius*, *Inhalt* oder *Umfang* eingegeben werden. Die Beziehung der Werte untereinander definiert ein Klassenmodul, denn auch diese gibt es in der Access-Anwendung.

Code 6.50 Die Prozedur erstellt ein Formular zur Berechnung von Kreisparametern

```
Sub CreateCalculate()
   Dim objForm As Form
   Dim ctlText    As Control
   Dim ctlLabel   As Control
   Dim sName   As String

'Create Formular
   Set objForm = Application.CreateForm
   sName = objForm.Name
   With DoCmd
      .SetWarnings False
      .Restore
      .Save acForm, sName
      .Close acForm, sName
      .Rename "Kreis", acForm, sName
      .SetWarnings True
   End With

'Create Controls
   sName = "Kreis"
   DoCmd.OpenForm sName, acDesign
   Set objForm = Forms(sName)
   objForm.HasModule = True

   Set ctlLabel = Application.CreateControl _
      (FormName:=objForm.Name, _
      ControlType:=acLabel, _
      Section:=acDetail, _
      Left:=500, Top:=400, Width:=3100, Height:=400)
   ctlLabel.Properties("Caption") = "Kreisberechnung"
   ctlLabel.Properties("FontSize") = 18
   Set ctlLabel = Nothing
```

```
    Set ctlLabel = Application.CreateControl _
        (FormName:=objForm.Name, _
        ControlType:=acLabel, _
        Section:=acDetail, _
        Left:=500, Top:=1000, Width:=1000, Height:=300)
    ctlLabel.Properties("Caption") = "Radius"
    Set ctlLabel = Nothing
    Set ctlText = Application.CreateControl _
        (FormName:=objForm.Name, _
        ControlType:=acTextBox, _
        Section:=acDetail, _
        Left:=1600, Top:=1000, Width:=2000, Height:=300)
    ctlText.Properties("Name") = "Radius"
    Set ctlText = Nothing

    Set ctlLabel = Application.CreateControl _
        (FormName:=objForm.Name, _
        ControlType:=acLabel, _
        Section:=acDetail, _
        Left:=500, Top:=1500, Width:=1000, Height:=300)
    ctlLabel.Properties("Caption") = "Inhalt"
    Set ctlLabel = Nothing
    Set ctlText = Application.CreateControl _
        (FormName:=objForm.Name, _
        ControlType:=acTextBox, _
        Section:=acDetail, _
        Left:=1600, Top:=1500, Width:=2000, Height:=300)
    ctlText.Properties("Name") = "Inhalt"
    Set ctlText = Nothing

    Set ctlLabel = Application.CreateControl _
        (FormName:=objForm.Name, _
        ControlType:=acLabel, _
        Section:=acDetail, _
        Left:=500, Top:=2000, Width:=1000, Height:=300)
    ctlLabel.Properties("Caption") = "Umfang"
    Set ctlLabel = Nothing
    Set ctlText = Application.CreateControl _
        (FormName:=objForm.Name, _
        ControlType:=acTextBox, _
        Section:=acDetail, _
        Left:=1600, Top:=2000, Width:=2000, Height:=300)
    ctlText.Properties("Name") = "Umfang"
    Set ctlText = Nothing
    Set objForm = Nothing
End Sub
```

Das mit der Prozedur (Code 6.50) erstellte Formular zeigt (Abb. 6.11). Ein Klassenmodul mit dem Namen *clsKreis* liefert die Parameter-Beziehungen (Code 6.51).

Abb. 6.11 Formular zur Kreisberechnung

Code 6.51 Die Property-Prozeduren der Klasse clsKreis

```
Option Compare Database
Option Explicit

Const PI = 3.14162
Private dRadius As Double

Property Let Radius(Radius As Double)
   dRadius = Radius
End Property
Property Let Inhalt(Inhalt As Double)
   dRadius = Sqr(Inhalt / PI)
End Property
Property Let Umfang(Umfang As Double)
   dRadius = Umfang / 2 / PI
End Property

Property Get Radius() As Double
   Radius = dRadius
End Property
Property Get Inhalt() As Double
   Inhalt = PI * dRadius * dRadius
End Property
Property Get Umfang() As Double
   Umfang = 2 * PI * dRadius
End Property
```

Über die Ereignis-Prozeduren des Formulars *Form_Kreis* wird die Ein- und Ausgabe zu den Textfeldern gesteuert (Code 6.52).

Code 6.52 Die Ereignisprozeduren im Formularmodul Form_Kreis

```
Dim Krs As New clsKreis

Private Sub Radius_KeyUp(KeyCode As Integer, Shift As Integer)
   Krs.Radius = Val(Radius.Text)
```

```
    Inhalt.Value = Format(Krs.Inhalt, "#,##0.00")
    Umfang.Value = Format(Krs.Umfang, "#,##0.00")
End Sub

Private Sub Inhalt_KeyUp(KeyCode As Integer, Shift As Integer)
    Krs.Inhalt = Val(Inhalt.Text)
    Radius.Value = Format(Krs.Radius, "#,##0.00")
    Umfang.Value = Format(Krs.Umfang, "#,##0.00")
End Sub

Private Sub Umfang_KeyUp(KeyCode As Integer, Shift As Integer)
    Krs.Umfang = Val(Umfang.Text)
    Radius.Value = Format(Krs.Radius, "#,##0.00")
    Inhalt.Value = Format(Krs.Inhalt, "#,##0.00")
End Sub
```

6.5.3 Gebundene Formulare

Die Aufgabe von Formularen ist zum größten Teil die Präsentation von vorhandenen Daten und deren Auswertungen. So sind Formulare und Steuerelemente mehr oder weniger direkt mit Tabellen und Abfragen verbunden. Man spricht hier von gebundenen Formularen.

Die Prozedur (Code 6.53) erzeugt mit jedem Aufruf ein neues Formular, das zunächst gespeichert und dann zur Bearbeitung im Entwurfsmodus wieder geöffnet wird. Danach wird ein Datensatz der Tabelle Verkaufsliste erstellt, welches als Konstruktionsvorlage für das Formular dient. Entsprechend der ersten acht Felder (lLoop = 0 bis 7) wird ein Bezeichnungsfeld mit dem Inhalt des Feldnamens und dahinter ein Datenfeld mit dem Namen des Feldnamens erstellt (Abb. 6.12). Die Variable *lTop* sorgt dafür, dass diese nicht übereinander, sondern untereinander erzeugt werden.

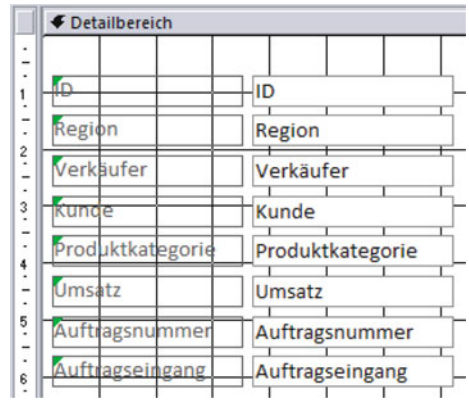


Abb. 6.12 Erzeugte Datenfelder und ihre Beschriftung

Der Typ des Datenfeldes kann außer einer *Textbox* auch eine *ComboBox* oder *Checkbox* sein. Wie dies gestaltet werden kann, wird im Code nur für eine *ComboBox* angedeutet, auch wenn im Beispiel nur *TextBoxen* zum Einsatz kommen. Für eine Darstellung in *Listenform* bekommt die Spaltebezeichnung ebenfalls den Feldnamen. Nachdem die Datenfelder mit ihren Bezeichnungen erzeugt sind, erfolgt die Anbindung an die Tabelle über eine SQL-Anweisung, hier Auswahl aller Datensätze. Mit der Eigenschaft *DefaultView* (Anhang 6, Tab. A6.19) wird noch die Darstellungform gewählt.

Code 6.53 Die Prozedur erzeugt ein gebundenes Formular

```
Sub CreateFormularVerkauf()
   Dim objData    As Database
   Dim objForm    As Form
   Dim objRec     As Recordset
   Dim objField   As Field2
   Dim ctrLabel   As Control
   Dim ctrField   As Control
   Dim cbxField   As Control
   Dim objLbl     As Control
   Dim sName      As String
   Dim sSql       As String
   Dim lCtrType   As Long
   Dim lLoop      As Long
   Dim lTop       As Long

'Formular neu erstellen
   Set objForm = Application.CreateForm
   sName = objForm.Name
   With DoCmd
      .SetWarnings False
      .Restore
      .Save acForm, sName
      .Close acForm, sName
      .Rename "Verkauf", acForm, sName
      .SetWarnings True
   End With
   Set objForm = Nothing

'Formular zur Bearbeitung öffnen
   lTop = 0
   sName = "Verkauf"
   DoCmd.OpenForm sName, acDesign
   Set objForm = Forms(sName)
```

```
Set objData = CurrentDb
Set objRec = objData.openrecordset("Verkaufsliste", _
   dbOpenTable, dbReadOnly)
For lLoop = 0 To 7
   Set objField = objRec.Fields(lLoop)
   lCtrType = acTextBox
   On Error Resume Next
   lCtrType = objField.Properties("DisplayControl")
   On Error GoTo 0

   lTop = lTop + 400
'Feldbezeichnung
   Set ctrLabel = Application.CreateControl _
      (sName, acLabel, acDetail, _
      Left:=100, Top:=lTop, Width:=1900, Height:=300)
   ctrLabel.Properties("Caption") = objField.Name
   Set ctrLabel = Nothing
'Datenfeld
   Set ctrField = Application.CreateControl _
      (sName, lCtrType, acDetail, _
      Left:=2100, Top:=lTop, Width:=2000, Height:=300)
   ctrField.ControlSource = objField.SourceField
'Feldtyp
   Select Case lCtrType
   Case acComboBox
      Set cbxField = ctrField
      With cbxField
         .RowSource = objField.Properties("RowSource")
         .ColumnCount = objField.Properties("ColumnCount")
         .ColumnWidths = objField.Properties("ColumnWidths")
      End With
   End Select
'Spaltenbeschriftung
   Set ctrLabel = Application.CreateControl _
      (sName, acLabel, acDetail, ctrField.Name)
   With ctrLabel
      .Caption = objField.Name
      On Error Resume Next
      .Caption = objField.Properties("Caption")
      On Error GoTo 0
   End With

   Set ctrLabel = Nothing
   Set ctrField = Nothing
Next lLoop
```

```
'Zugriff auf Tabelle mit der Darstellungsform
   sSql = "SELECT * FROM Verkaufsliste"
   objForm.RecordSource = sSql
   objForm.DefaultView = acDefViewDatasheet
   'objForm.DefaultView = acDefViewPivotChart
   'objForm.DefaultView = acDefViewPivotTable
   'objForm.DefaultView = acDefViewSingle
   'objForm.DefaultView = acDefViewSplitForm
   DoCmd.Close acForm, objForm.Name, acSaveYes
   Set objForm = Nothing
End Sub
```

Die Darstellungsmöglichkeiten (Abb. 6.13–6.17) sind im Code auskommentiert.

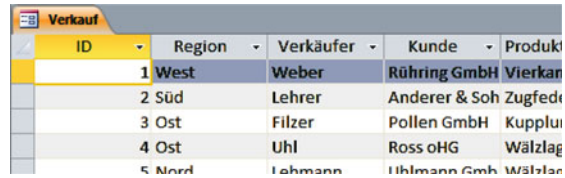


Abb. 6.13 Formular in Tabellenform

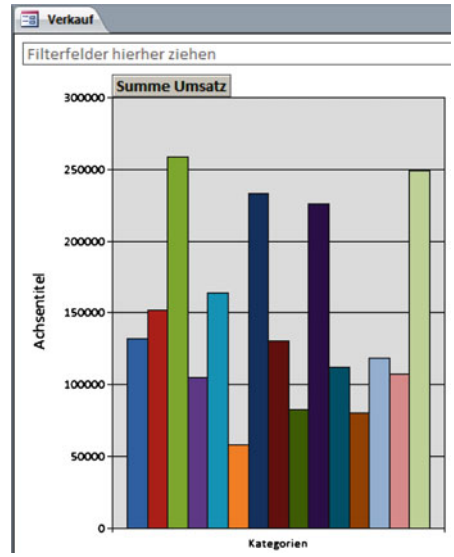


Abb. 6.14 Formular als PivotChart

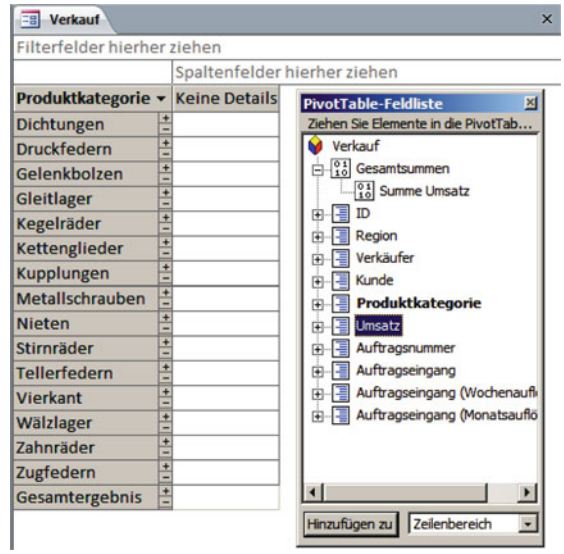


Abb. 6.15 Formular als Pivot-Table

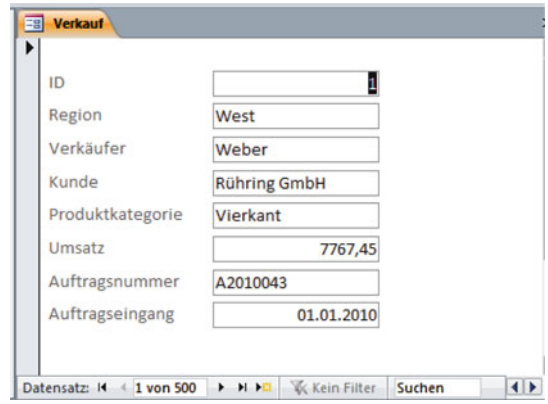


Abb. 6.16 Formular als Datensatz

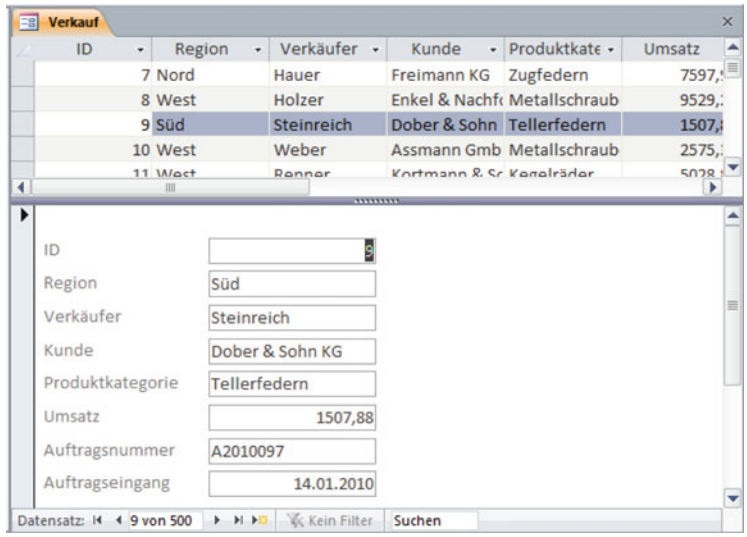


Abb. 6.17 Geteiltes Formular

6.5.4 Unterformulare

Unterformulare werden wie Formulare erstellt und dann über eine Methode in ein Hauptformular platziert. Auch das Ziehen des Unterformulars auf das Hauptformular ist eine Methode. Hier wollen wir die Erstellung per VBA durchführen.

Unterformulare eignen sich sehr gut zur Darstellung einer 1:n Beziehung, wobei das Hauptformular den einen Datensatz repräsentiert und das Unterformular die n zugehörigen Datensätze. Als Beispiel dient die Verkäuferliste die in einer 1:n Beziehung zur Verkaufsliste steht (Abb. 6.18).

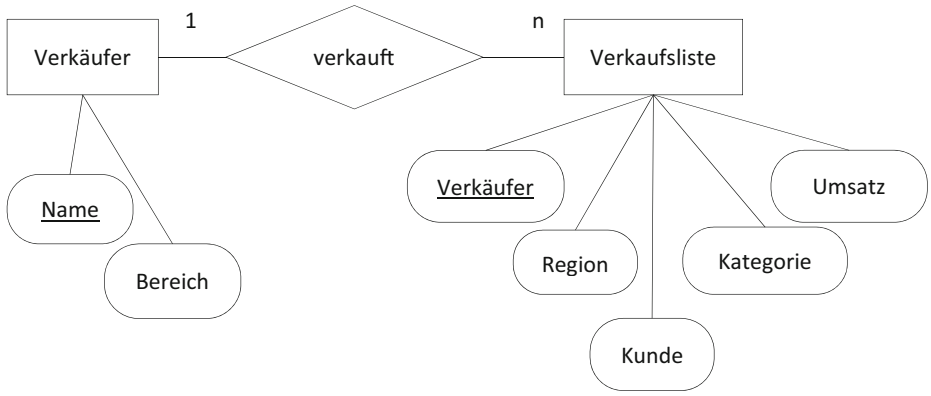


Abb. 6.18 Entity-Relationship-Model einer 1:n Beziehung

Aus der Verkaufsliste lässt sich sehr schnell per Abfrage eine Verkäuferliste erstellen (Code 6.54), wie in Abschn. 6.3 gezeigt.

Code 6.54 Die Prozedur erzeugt eine neue Tabelle ohne doppelte Einträge

```
Sub CreateVerkäuferTable()
   Dim objData     As Database
   Dim objTable    As TableDef
   Dim sSql        As String

'Neue Tabelle mit CreateTableDef
   Set objData = CurrentDb
   Set objTable = objData.CreateTableDef("Verkäufer")
   With objTable
      .Fields.Append .CreateField("Name", dbText, 100)
      .Fields.Append .CreateField("Region", dbText, 100)
   End With
   objData.TableDefs.Append objTable

'Füllen der Tabelle mit Execute SQL
   sSql = "INSERT INTO Verkäufer (Name, Region) " & _
      "SELECT DISTINCT Verkäufer, Region FROM Verkaufsliste ;"
   objData.Execute sSql
   objData.Close
   Application.RefreshDatabaseWindow
   Set objTable = Nothing
   Set objData = Nothing
End Sub
```

Ebenfalls in Abschn. 6.3 wurde der Aufbau einer Beziehung (Relation) beschrieben. Die Prozedur (Code 6.55) erzeugt eine 1:n Relation zwischen Verkäufer und Verkaufsliste (Abb. 6.19).

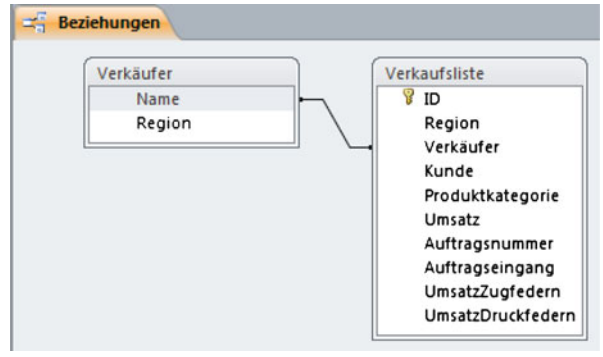


Abb. 6.19 Darstellung der Relation in Access

Code 6.55 Die Prozedur stellt eine 1:n Beziehung zwischen zwei Tabellen her

```
Sub CreateRelation()
   Dim objData       As Database
   Dim objRelation   As Relation
   Dim objField      As Field
   Dim sRelation     As String
   Dim sFields       As String
   Dim lRelCount     As Long
   Dim lCount        As Long

'löschen vorhandener Beziehungen
   Set objData = CurrentDb
   lRelCount = objData.Relations.Count
   For lCount = lRelCount - 1 To 0 Step -1
      objData.Relations.Delete (objData.Relations(lCount).Name)
   Next lCount

'Relation erstellen
   sRelation = "Verkäufer_Name_Verkaufsliste_Verkäufer"
   Set objRelation = objData.CreateRelation()
   With objRelation
      .Name = sRelation                'Relationsname
      .Table = "Verkäufer"             'Primärschlüssel
      .ForeignTable = "Verkaufsliste"  'Fremdschlüssel
      .Attributes = dbRelationDontEnforce '1:n Beziehung
   End With

'Feld-Beziehungen
   sFields = "Name_Verkäufer"
   Set objField = objRelation.CreateField(sFields, dbLong)
   With objField
      .Name = "Name"
      .ForeignName = "Verkäufer"
   End With

'Feldbeziehung der Relation hinzufügen
   objRelation.Fields.Append objField

'Relation der Datenbank hinzufügen
   objData.Relations.Append objRelation

   Set objField = Nothing
   Set objRelation = Nothing
   Set objData = Nothing
End Sub
```

Als Unterformular soll das bereits zuvor erstellte Formular *Verkauf* in Tabellenform dienen (Code 5.56).

Code 6.56 Die Prozedur erstellt das Formular Verkauf

```
Sub CreateSubFormular()
   Dim objData    As Database
   Dim objForm    As Form
   Dim objRec     As Recordset
   Dim objField   As Field2
   Dim ctrLabel   As Control
   Dim ctrField   As Control
   Dim cbxField   As Control
   Dim objLbl     As Control
   Dim sName      As String
   Dim sSql       As String
   Dim lCtrType   As Long
   Dim lLoop      As Long
   Dim lTop       As Long

'Formular neu erstellen
   Set objForm = Application.CreateForm
   sName = objForm.Name
   objForm.HasModule = True
   With DoCmd
      .SetWarnings False
      .Restore
      .Save acForm, sName
      .Close acForm, sName
      .Rename "Verkauf", acForm, sName
      .SetWarnings True
   End With
   Set objForm = Nothing

'Formular zur Bearbeitung öffnen
   lTop = 0
   sName = "Verkauf"
   DoCmd.OpenForm sName, acDesign
   Set objForm = Forms(sName)

   Set objData = CurrentDb
   Set objRec = objData.openrecordset _
      ("Verkaufsliste", dbOpenTable, dbReadOnly)
   For lLoop = 0 To 7
      Set objField = objRec.Fields(lLoop)
      lCtrType = acTextBox
```

```
      On Error Resume Next
      lCtrType = objField.Properties("DisplayControl")
      On Error GoTo 0

      lTop = lTop + 400
   'Feldbezeichnung
      Set ctrLabel = Application.CreateControl _
         (sName, acLabel, acDetail, _
         Left:=100, Top:=lTop, Width:=1900, Height:=300)
      ctrLabel.Properties("Caption") = objField.Name
      Set ctrLabel = Nothing

   'Datenfeld
      Set ctrField = Application.CreateControl _
         (sName, lCtrType, acDetail, _
         Left:=2100, Top:=lTop, Width:=2000, Height:=300)
      ctrField.ControlSource = objField.SourceField

   'Feldtyp
      Select Case lCtrType
      Case acComboBox
         Set cbxField = ctrField
         With cbxField
            .RowSource = objField.Properties("RowSource")
            .ColumnCount = objField.Properties("ColumnCount")
            .ColumnWidths = objField.Properties("ColumnWidths")
         End With
      End Select

   'Spaltenbeschriftung
      Set ctrLabel = Application.CreateControl _
         (sName, acLabel, acDetail, ctrField.Name)
      With ctrLabel
         .Caption = objField.Name
         On Error Resume Next
         .Caption = objField.Properties("Caption")
         On Error GoTo 0
      End With

      Set ctrLabel = Nothing
      Set ctrField = Nothing
   Next lLoop
```

```
'Zugriff auf Tabelle mit der Darstellungsform
   sSql = "SELECT * FROM Verkaufsliste"
   objForm.RecordSource = sSql
   objForm.DefaultView = acDefViewDatasheet
   DoCmd.Close acForm, objForm.Name, acSaveYes
   Set objData = Nothing
   Set objRec = Nothing
   Set objForm = Nothing
End Sub
```

Zur Ergänzung erhält der Formularfuß ein ungebundenes Datenfeld der Summenbestimmung. Die Prozedur (Code 6.57) öffnet das vorhandene Formular und stellt das Datenfeld mit Bezeichnung im Formularfuß ein (Abb. 6.20).

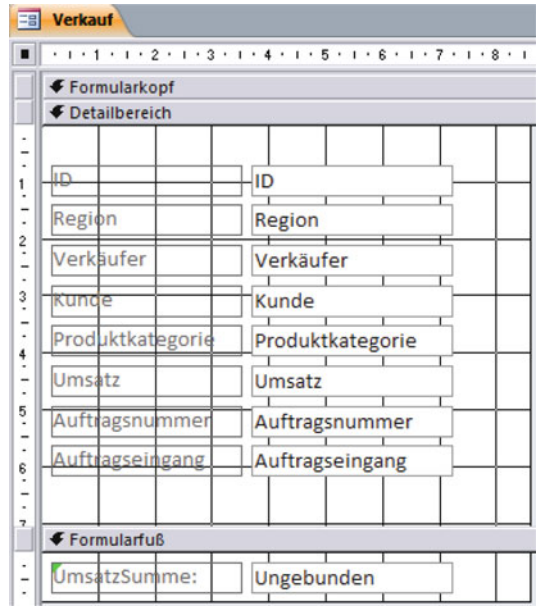


Abb. 6.20 Zusätzliches Summenfeld im Formularfuß

Code 6.57 Die Prozedur erzeugt ein Textfeld im Formularfuß

```
Sub CreateSubSumField()
   Dim objForm    As Form
   Dim ctlText    As Control
   Dim ctlLabel   As Control
   Dim sName      As String
```

```
    sName = "Verkauf"
    DoCmd.OpenForm sName, acDesign
    Set objForm = Forms(sName)
    On Error GoTo ErrorHandler

    Set ctlLabel = Application.CreateControl _
       (FormName:=objForm.Name, _
       ControlType:=acLabel, _
       Section:=acFooter, _
       Left:=2000, Top:=100, Width:=1600, Height:=300)
    ctlLabel.Properties("Caption") = "UmsatzSumme:"

    Set ctlText = Application.CreateControl _
       (FormName:=objForm.Name, _
       ControlType:=acTextBox, _
       Section:=acFooter, _
       Left:=3800, Top:=100, Width:=2000, Height:=300)
    ctlText.Properties("Name") = "UmsatzSumme"
    ctlText.ControlSource = "=Sum([Umsatz])"
    ctlText.Properties("Format") = "Currency"
    With objForm
       .Section(0).Height = 4000
       .Section(1).Height = 0
       .Section(2).Height = 500
    End With

    Set ctlText = Nothing
    Set ctlLabel = Nothing
    Set objForm = Nothing
    Exit Sub

ErrorHandler:
    DoCmd.RunCommand acCmdFormHdrFtr
    Resume
End Sub
```

In der Formularansicht enthält das Textfeld *UmsatzSumme* die Summe der gesamten Umsätze (Abb. 6.21).

Abb. 6.21 Summenfeld in der Formularansicht

Mit der gleichen Prozedur wie zuvor das Formular *Verkauf* erstellt wurde, wird auch das Hauptformular *Umsatz* erzeugt, lediglich ein paar Änderungen sind erforderlich (Code 6.58). Das Formular weist zwei Datenfelder auf (Abb. 6.22).

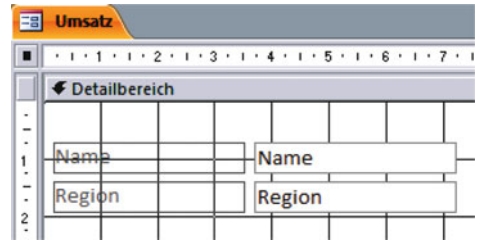


Abb. 6.22 Entwurfsansicht von Formular Umsatz

Code 6.58 Die Prozedur erzeugt ein Formular Umsatz

```
Sub CreateMainFormular()
   Dim objData     As Database
   Dim objForm     As Form
   Dim objRec      As Recordset
   Dim objField    As Field2
   Dim ctrLabel    As Control
   Dim ctrField    As Control
```

```
   Dim cbxField   As Control
   Dim objLbl     As Control
   Dim sName      As String
   Dim sSql       As String
   Dim lCtrType   As Long
   Dim lLoop      As Long
   Dim lTop       As Long

'Formular neu erstellen
   Set objForm = Application.CreateForm
   sName = objForm.Name
   With DoCmd
      .SetWarnings False
      .Restore
      .Save acForm, sName
      .Close acForm, sName
      .Rename "Umsatz", acForm, sName
      .SetWarnings True
   End With
   Set objForm = Nothing

'Formular zur Bearbeitung öffnen
   lTop = 0
   sName = "Umsatz"
   DoCmd.OpenForm sName, acDesign
   Set objForm = Forms(sName)

   Set objData = CurrentDb
   Set objRec = objData.openrecordset("Verkäufer", dbOpenTable, _
     dbReadOnly)
   For lLoop = 0 To 1
      Set objField = objRec.Fields(lLoop)
      lCtrType = acTextBox
      On Error Resume Next
      lCtrType = objField.Properties("DisplayControl")
      On Error GoTo 0

      lTop = lTop + 400
   'Feldbezeichnung
      Set ctrLabel = Application.CreateControl _
         (sName, acLabel, acDetail, _
         Left:=100, Top:=lTop, Width:=1900, Height:=300)
      ctrLabel.Properties("Caption") = objField.Name
      Set ctrLabel = Nothing
```

```
   'Datenfeld
      Set ctrField = Application.CreateControl _
         (sName, lCtrType, acDetail, _
         Left:=2100, Top:=lTop, Width:=2000, Height:=300)
      ctrField.ControlSource = objField.SourceField
   'Feldtyp
      Select Case lCtrType
      Case acComboBox
         Set cbxField = ctrField
         With cbxField
            .RowSource = objField.Properties("RowSource")
            .ColumnCount = objField.Properties("ColumnCount")
            .ColumnWidths = objField.Properties("ColumnWidths")
         End With
      End Select
   'Spaltenbeschriftung
      Set ctrLabel = Application.CreateControl _
         (sName, acLabel, acDetail, ctrField.Name)
      With ctrLabel
         .Caption = objField.Name
         On Error Resume Next
         .Caption = objField.Properties("Caption")
         On Error GoTo 0
      End With

      Set ctrLabel = Nothing
      Set ctrField = Nothing
   Next lLoop

'Zugriff auf Tabelle mit der Darstellungsform
   sSql = "SELECT * FROM Verkäufer"
   objForm.RecordSource = sSql
   objForm.DefaultView = acDefViewSingle
   DoCmd.Close acForm, objForm.Name, acSaveYes
   Set objData = Nothing
   Set objRec = Nothing
   Set objForm = Nothing
End Sub
```

Im nächsten Schritt wird das Formular *Verkauf* aus dem Navigationsbereich auf das im Entwurfmodus geöffnete Formular *Umsatz* gezogen (Abb. 6.23).

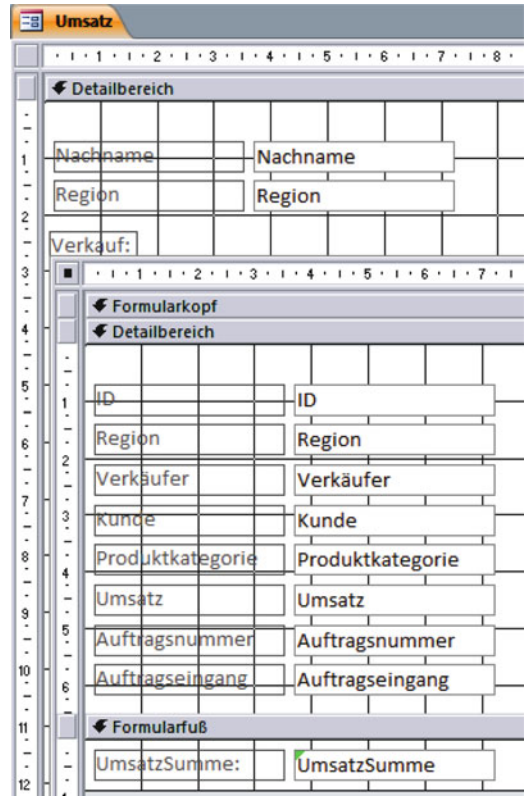


Abb. 6.23 Formular Verkauf als SubForm auf Formular Umsatz

Durch Ziehen der Formularränder wird die Darstellung angepasst. Nun müssen die beiden Formulare mit ihren Schlüsselfeldern noch verlinkt werden. Dies geschieht im Formular *Form_Umsatz* des Hauptformulars (Code 6.59).

Code 6.59 Die Ereignisprozedur verknüpft Main- und Sub-Formular über ihre Schlüsselfelder

```
Private Sub Form_Open(Cancel As Integer)
   Verkauf.LinkChildFields = "Verkäufer"
   Verkauf.LinkMasterFields = "Nachname"
End Sub
```

Das Formular *Umsatz* mit dem verlinkten Unterformular *Verkauf* zeigt in der Formularansicht (Abb. 6.24) zu jedem ausgewählten Verkäufer dessen zugehörige Datensätze. Da das Unterformular in Tabellenform eingebunden ist, bleibt die Umsatzsumme verborgen. Doch es gibt die Möglichkeit, dieses Auswertungsfeld mit einem Textfeld auf dem Hauptformular zu verknüpfen.

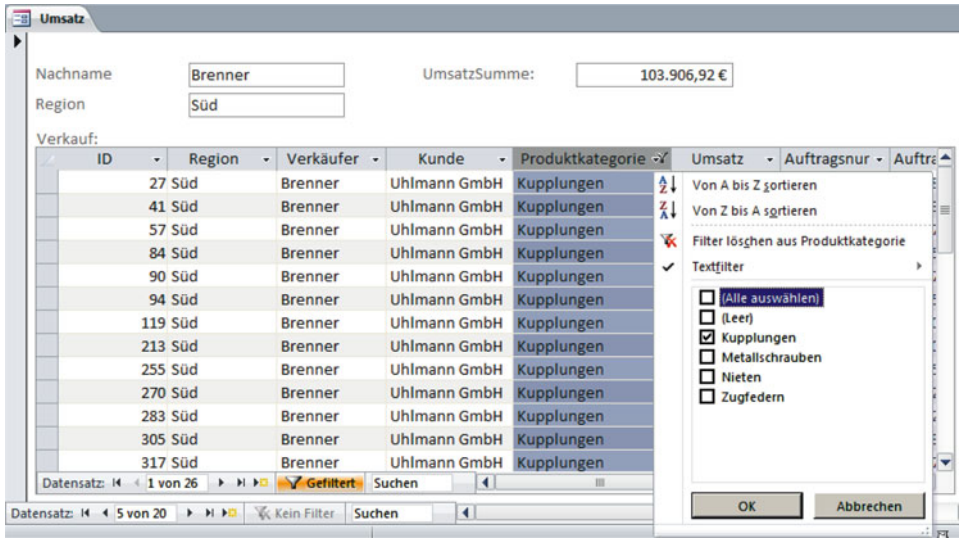


Abb. 6.24 Die verlinkten Formulare

Die Prozedur (Code 6.60) erzeugt ein weiteres Textfeld im Hauptformular zur Übernahme der Umsatzsumme.

Code 6.60 Die Prozedur verlinkt Textfelder zwischen Haupt- und Unterformular

```
Sub CreateMainSumField()
   Dim objForm     As Form
   Dim ctlText     As Control
   Dim ctlLabel    As Control
   Dim sName       As String

   sName = "Umsatz"
   DoCmd.OpenForm sName, acDesign
   Set objForm = Forms(sName)
   Set ctlLabel = Application.CreateControl _
      (sName, acLabel, acDetail, _
      Left:=5100, Top:=400, Width:=1900, Height:=300)
   ctlLabel.Properties("Caption") = "UmsatzSumme:"
   Set ctlText = Application.CreateControl _
      (sName, acTextBox, acDetail, _
      Left:=7100, Top:=400, Width:=2000, Height:=300)
   ctlText.Properties("Name") = "GesamtSumme"
   ctlText.ControlSource = "=[Verkauf].[Formular]![UmsatzSumme]"
   ctlText.Properties("Format") = "Currency"
```

```
    Set ctlText = Nothing
    Set ctlLabel = Nothing
    Set objForm = Nothing
End Sub
```

Zusammengefasst ergeben sich vier Schritte zum Unterformular:

1. Erstellung eines Formulars als Unterformular in Listenform.
2. Erstellung eines Hauptformulars in Einzeldarstellung.
3. Mit Drag & Drop das Unterformular auf das Hauptformular ziehen.
4. Im Hauptformular die Verlinkung der Primärfelder herstellen.

Die Gestaltung der Formulare mit Überschrift und Farben überlasse ich dem Leser.

6.6 Access-Berichte

Die Berichts-Objekte *Reports* haben die Aufgabe, vorhandene Daten in aufbereiteter Form als Druckvorlage bereitzustellen. Ähnlich wie bei den Formularen gibt es in Access unterschiedliche Berichtsformen. Aber es gibt auch Ähnlichkeiten bei den Eigenschaften, Methoden und Ereignissen. Zusätzlich verfügt das Berichts-Objekt über einige optische Gestaltungsmöglichkeiten.

6.6.1 Berichtsaufbau

Im Gegensatz zum Formular verfügt ein Bericht über zwei Kopf- und Fuß-Bereiche (Abb. 6.25).

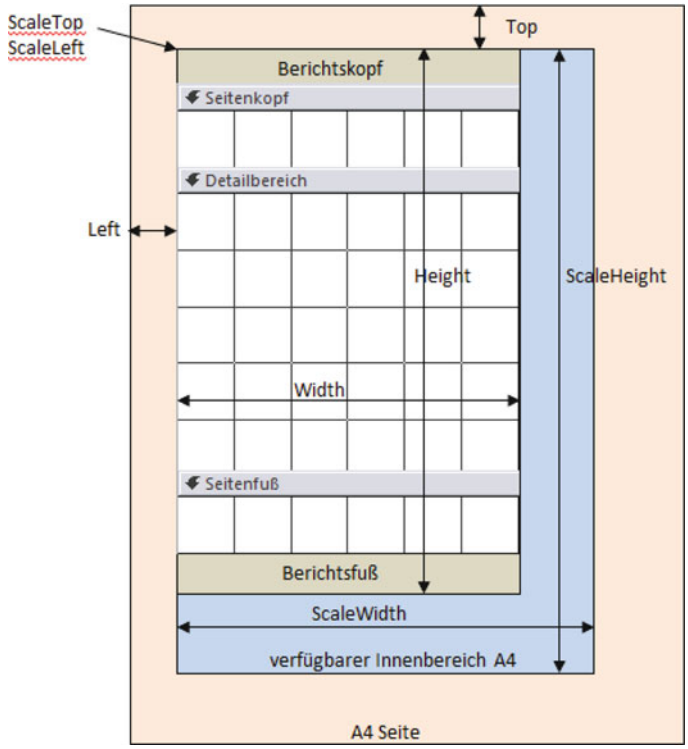


Abb. 6.25 Bereiche eines Reports

Die Prozedur (Code 6.61) erzeugt einen leeren Bericht und setzt die Eigenschaft *HasModule* des Berichts auf *True*. Dadurch wird dann auch das Berichtmodul sichtbar.

Code 6.61 Die Prozedur erzeugt einen leeren Bericht

```
Sub CreateReport()
   Dim objReport  As Report
   Dim sName      As String

   Set objReport = Application.CreateReport
   sName = objReport.Name
   objReport.HasModule = True
   With DoCmd
      .SetWarnings False
      .Restore
      .Save acReport, sName
      .Close acReport, sName
      .Rename "Bericht", acReport, sName
      .SetWarnings True
```

```
   End With
   Set objReport = Nothing
End Sub
```

Die Prozedur (Code 6.62) liest alle vorhandenen Berichte.

Code 6.62 Die Prozedur liest alle Berichte in der aktuellen Anwendung

```
Sub ReadAllReports()
   Dim obj As AccessObject
   Dim dbs As Object

   Set dbs = Application.CurrentProject
   For Each obj In dbs.AllReports
      Debug.Print obj.Name
   Next obj
End Sub
```

Mit dem Berichtsmodul, dessen Name sich aus den Begriffen Report und Berichtsname zusammensetzt, in diesem Fall also *Report_Bericht*, stehen auch eine Menge Ereignisprozeduren zum Bericht zur Verfügung (Abb. 6.26).

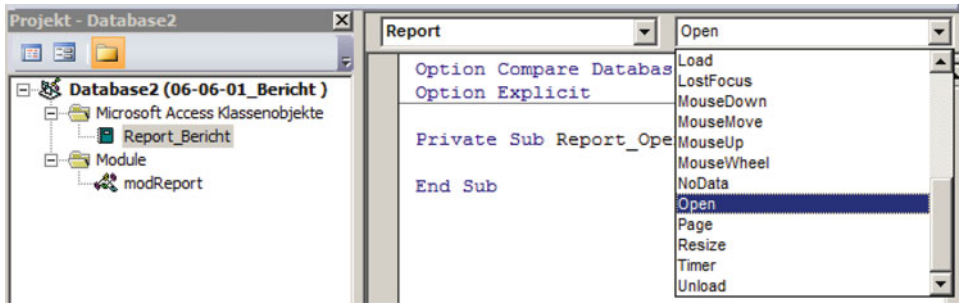


Abb. 6.26 Das Klassenobjekt Report_Bericht und seine Ereignisprozeduren

Mit den Ereignisprozeduren können auch Eigenschaften und Methoden des Berichts genutzt werden. Die Ereignisprozedur *Report_Load* (Code 6.63) stellt die Eigenschaft *BackColor* für die Bereiche Detail, Seitenkopf und Seitenfuß ein und nutzt dabei drei mögliche Methoden der Farbdefinition.

Code 6.63 Die Ereignis-Prozedur setzt Hintergrundfarben beim Laden der Berichtsansicht

```
Private Sub Report_Load()
   With Me
      .Caption = "Demo Bericht"
```

```
      .Section(acDetail).BackColor = QBColor(14)
      .Section(acPageHeader).BackColor = RGB(255, 0, 0)
      .Section(acPageFooter).BackColor = vbBlue
   End With
   'ReadProperties
End Sub
```

Der Aufruf der Prozedur *ReadProperties* (Code 6.64) ist auskommentiert, da er nur einmal zur Information aufgerufen wird. Die Prozedur liefert alle Eigenschaften des Berichts (Abb. 6.27) mit den Eigenschaften *Properties*, die teilweise auch schreibgeschützt sind.

```
RecordSource
Caption       Demo Bericht
PopUp         Falsch
Modal         Falsch
DisplayOnSharePointSite      1
DefaultView    1
AllowReportView             Wahr
AllowLayoutView             Wahr
PictureType    0
Picture       (keines)
PictureTiling Falsch
PictureAlignment             2
PictureSizeMode              0
Width          6994
AutoCenter    Falsch
AutoResize    Wahr
FitToPage     Wahr
BorderStyle    2
...
```

Abb. 6.27 Auszug der Eigenschaftsnamen und Werte

Code 6.64 Die Prozedur liefert eine Auflistung aller Eigenschaften eines Steuerelements

```
Sub ReadProperties()
   Dim objPro As Property
   On Error Resume Next
   For Each objPro In Me.Properties
      Debug.Print objPro.Name, objPro.Value
   Next
End Sub
```

Damit wird deutlich, dass ein Bericht selbst ebenfalls ein Steuerelement ist.

6.6.2 Steuerelemente

Auf einem Bericht lassen sich wie beim Formular ebenfalls ungebundene und gebundene Steuerelemente positionieren. Dazu wird die Methode *CreateReportControl* verwendet. Im Unterschied zu einem Formular können auf einem Bericht auch Zeichenobjekte dargestellt werden.

Die Prozedur (Code 6.65) erzeugt zwei Steuerelemente auf einem geöffneten Bericht, wie sie schon im Formular genutzt wurden. Mit Hilfe der Funktion *DSum* wird der Umsatz in der Tabelle *Verkaufsliste* summiert und in ein Bezeichnungsfeld gesetzt (Abb. 6.28).

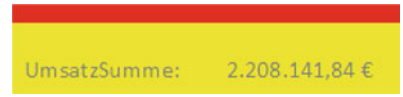


Abb. 6.28 Die Bezeichnungsfelder mit Inhalt

Code 6.65 Die Prozedur erzeugt zwei Bezeichnungsfelder auf einem geöffneten Bericht

```
Sub CreateReportControls()
   Dim objReport  As Report
   Dim ctlLabel   As Control
   Dim sName      As String
   Dim dWert      As Currency

   sName = "Bericht"
   Set objReport = Application.Reports(sName)
   Set ctlLabel = Application.CreateReportControl _
      (objReport.Name, acLabel, acDetail, "", "", _
      2100, 300, 2000, 300)
   dWert = DSum("[Umsatz]", "Verkaufsliste")
   With ctlLabel
      .Properties("Caption") = Format(dWert, "#,##0.00 €")
      .Properties("FontName") = "Calibri"
      .Properties("FontSize") = 12
      .Properties("FontBold") = False
      .Properties("Borderstyle") = 0
   End With
   Set cltlabel = Nothing
   Set ctlLabel = Application.CreateReportControl _
      (objReport.Name, acLabel, acDetail, "", "", _
      100, 300, 1800, 300)
   With ctlLabel
      .Properties("Caption") = "UmsatzSumme: "
      .Properties("FontName") = "Calibri"
      .Properties("FontSize") = 12
```

```
      .Properties("FontBold") = False
      .Properties("Borderstyle") = 0
   End With
   Set ctlLabel = Nothing
   Set objReport = Nothing
End Sub
```

6.6.3 Befehle

Für eine einfache Berichtserstellung eignet sich gut das Objekt *DoCmd*. Mit dessen Methode *OpenReport* lassen sich vorhandene Berichte über den Parameter *View* öffnen.

```
'Syntax:
DoCmd.OpenReport (ReportName, [View], [FilterName], _
   [WhereCondition], [WindowMode], [OpenArgs])
```

Anhang 6, Tab. A6.20 zeigt die Konstanten für den *View*-Parameter.

Die Prozedur (Code 6.66) erzeugt zuerst einen Bericht aus der Verkaufsliste nach Verkäufern sortiert und speichert diesen unter dem Namen *VerkaufUmsatz*. Danach wird der Bericht im Entwurfsmodus geöffnet und eine Gruppierung nach Verkäufern erstellt. Im Gruppenfuß werden die Anzahl der Datensätze je Verkäufer und der erzielte Umsatz eingestellt. Das aktuelle Datum im Seitenkopf und die aktuelle Seitenzahl im Seitenfuß ergänzen den Bericht (Abb. 6.29).

Verkäufer:	ID:	Region:	Kunde:	Produktkategorie:	Umsatz:
Altmann	452	Nord	Ross oHG	Metallschrauben	1.929
Altmann	148	Nord	Rühring GmbH	Druckfedern	8.409
Altmann	175	Nord	Raabe & Partner	Stirnräder	2.702
Altmann	181	Nord	Raabe & Partner	Stirnräder	2.702
Altmann	251	Nord	Ross oHG	Metallschrauben	1.929
Altmann	499	Nord	Raabe & Partner	Stirnräder	2.702
Altmann	19	Nord	Raabe & Partner	Stirnräder	2.702
34				**Umsatzsumme:**	**141.844**
Amtmann	304	Süd	Philipps GmbH	Gelenkbolzen	7.530
Amtmann	475	Süd	Münch GmbH	Tellerfedern	6.005
Amtmann	364	Süd	Münch GmbH	Tellerfedern	6.005
Amtmann	42	Süd	Münch GmbH	Tellerfedern	6.005
Amtmann	387	Süd	Philipps GmbH	Gelenkbolzen	7.530
Amtmann	96	Süd	Rüther & Hübner	Dichtungen	2.655
Amtmann	114	Süd	Philipps GmbH	Gelenkbolzen	7.530
Amtmann	55	Süd	Rüther & Hübner	Dichtungen	2.655

Seite 1 von 13

Abb. 6.29 Ausschnitt aus dem Umsatzreport in der Seitenansicht

Code 6.66 Die Prozedur erzeugt einen Bericht in Gruppenform mit Auswertung

```
Sub CreateReportVerkaufUmsatz()
   Dim objData    As Database
   Dim objReport  As Report
   Dim objRec     As Recordset
   Dim objField   As Field2
   Dim ctrLabel   As Control
   Dim ctrField   As Control
   Dim cbxField   As Control
   Dim objLbl     As Control
   Dim sTable     As String
   Dim sReport    As String
   Dim sFind      As String
   Dim sName      As String
   Dim sSql       As String
   Dim sFormat    As String
   Dim lCtrType   As Long
   Dim lLoop      As Long
   Dim lLeft      As Long
   Dim lWidth     As Long
   Dim lNum       As Long
   Dim vGrpLevel  As Variant

'Formular neu erstellen
   sTable = "Verkaufsliste"
   sReport = "VerkaufUmsatz"
   Set objReport = Application.CreateReport
   With objReport
'       .HasModule = True
       .RecordSource = "SELECT * " & _
          " FROM " & sTable & " ORDER BY Verkäufer"
       sName = .Name
       .Section(0).Height = 300   'Detailbereich
       .Section(3).Height = 600   'Seitenkopf
       .Section(4).Height = 300   'Seitenfuß
   End With
   With DoCmd
       .SetWarnings False
       .Restore
       .Save acReport, sName
       .Close acReport, sName
       .Rename sReport, acReport, sName
       .SetWarnings True
   End With
   Set objReport = Nothing

'Formular zur Bearbeitung im Entwurfsmodus öffnen
```

```
    lLeft = 550
    sName = sReport
    DoCmd.OpenReport sName, acViewDesign
    Set objReport = Reports![VerkaufUmsatz]

'Gruppenfuß gestalten
    vGrpLevel = Application.CreateGroupLevel(sName, "Verkäufer", _
      False, True)
    objReport.Section(acGroupLevel1Footer).Height = 400

    'Anzahl
    Set ctrField = Application.CreateReportControl _
       (sName, acTextBox, acGroupLevel1Footer, "", "", _
       550, 50, 500, 400)
    sFormat = "=Count([Verkäufer])"
    With ctrField
       .Properties("ControlSource") = sFormat
       .Properties("Format") = "##0"
       .Properties("FontName") = "Calibri"
       .Properties("FontSize") = 11
       .Properties("FontBold") = True
       .Properties("Borderstyle") = 0
       .SizeToFit
    End With
    Set ctrField = Nothing

    'Umsatzsumme
    Set ctrLabel = Application.CreateReportControl _
       (sName, acLabel, acGroupLevel1Footer, "", "", _
       5000, 50, 2500, 400)
    With ctrLabel
       .Properties("Caption") = "Umsatzsumme:"
       .Properties("FontName") = "Calibri"
       .Properties("FontSize") = 11
       .Properties("FontBold") = True
       .SizeToFit
    End With
    Set ctrLabel = Nothing
    Set ctrField = Application.CreateReportControl _
       (sName, acTextBox, acGroupLevel1Footer, "", "", _
       7650, 50, 1000, 400)
    sFormat = "=Sum([Umsatz])"
    With ctrField
       .Properties("ControlSource") = sFormat
       .Properties("Format") = "#,##0"
```

```
      .Properties("FontName") = "Calibri"
      .Properties("FontSize") = 11
      .Properties("FontBold") = True
      .Properties("Borderstyle") = 0
      .SizeToFit
   End With
   Set ctrField = Nothing

'Seitenkopf Überschrift
   Set ctrLabel = Application.CreateReportControl _
      (sName, acLabel, acPageHeader, "", "", _
      lLeft, 10, 3000, 400)
   With ctrLabel
      .Properties("Caption") = "Umsatzliste"
      .Properties("FontName") = "Calibri"
      .Properties("FontSize") = 16
      .Properties("FontBold") = True
      .SizeToFit
   End With
   Set ctrLabel = Nothing

'Seitenkopf Datum
   Set ctrField = Application.CreateReportControl _
      (sName, acTextBox, acPageHeader, "", "", _
      6000, 10, 3000, 300)
   sFormat = "=Format(Date())"
   With ctrField
      .Properties("ControlSource") = sFormat
      .Properties("FontName") = "Calibri"
      .Properties("FontSize") = 9
      .Properties("FontBold") = False
      .Properties("Borderstyle") = 0
      .Properties("TextAlign") = 3
      .SizeToFit
   End With
   Set ctrField = Nothing

'Seitenfuß Seite
   Set ctrField = Application.CreateReportControl _
      (sName, acTextBox, acPageFooter, "", "", _
      6000, 10, 3000, 300)
   sFormat = "=""Seite "" & [Page] & "" von  "" & [Pages] "
   With ctrField
      .Properties("ControlSource") = sFormat
      .Properties("FontName") = "Calibri"
```

```
      .Properties("FontSize") = 9
      .Properties("FontBold") = False
      .Properties("Borderstyle") = 0
      .Properties("TextAlign") = 3
      .SizeToFit
   End With
   Set ctrField = Nothing

'Listenstruktur
   Set objData = CurrentDb
   Set objRec = objData.OpenRecordset(sTable, _
      dbOpenTable, dbReadOnly)

   For lLoop = 0 To 5
      Select Case lLoop
      Case 0
         lNum = 2
         lWidth = 1200
      Case 1
         lNum = 0
         lWidth = 600
      Case 2
         lNum = 1
         lWidth = 800
      Case 3
         lNum = 3
         lWidth = 2000
      Case 4
         lNum = 4
         lWidth = 2000
      Case 5
         lNum = 5
         lWidth = 1000
      End Select
      Set objField = objRec.Fields(lNum)

   'Datenfeld im Detailbereich
      Set ctrField = Application.CreateReportControl _
         (sName, acTextBox, acDetail, "", objField.Name, _
         lLeft, 10, lWidth, 300)
      With ctrField
         .Properties("FontName") = "Calibri"
         .Properties("FontSize") = 11
         .Properties("Borderstyle") = 0
         .SizeToFit
      End With

   'Feldbezeichnung im Seitenkopf
```

```
      Set ctrLabel = Application.CreateReportControl _
         (sName, acLabel, acPageHeader, "", objField.Name & ":", _
         lLeft, 1000, lWidth, 300)
      With ctrLabel
         .Properties("FontName") = "Calibri"
         .Properties("FontSize") = 11
         .Properties("Borderstyle") = 0
         .SizeToFit
      End With

      lLeft = lLeft + lWidth + 100
      Set ctrLabel = Nothing
      Set ctrField = Nothing
   Next lLoop

   DoCmd.Close acReport, sName, acSaveYes

   Set objReport = Nothing
End Sub
```

6.7 Access-Makros

Access-Makros sind nicht zu Verwechseln mit VBA-Makros. Daher bezeichne ich auch gerne die Makros unter VBA als Prozeduren, ein Begriff der aus der Informatik stammt und eine logische Einheit von Anweisungen darstellt, ähnlich den Funktionen.

Access-Makros haben die Aufgabe, Aktionen für VBA-Unkundige mit umgangssprachlichen Elementen in einer Liste so zusammen zu fassen, dass sie nacheinander automatisch abgearbeitet werden. Hier wollen wir uns lediglich auf die Verknüpfung zwischen Datenmakros und VBA-Prozeduren beschränken.

6.7.1 Makros allgemein

Ein einfaches Makro (Abb. 6.30) öffnet ein Formular und zeigt Datensätze, deren Umsatz größer als 9000 ist (Abb. 6.31).

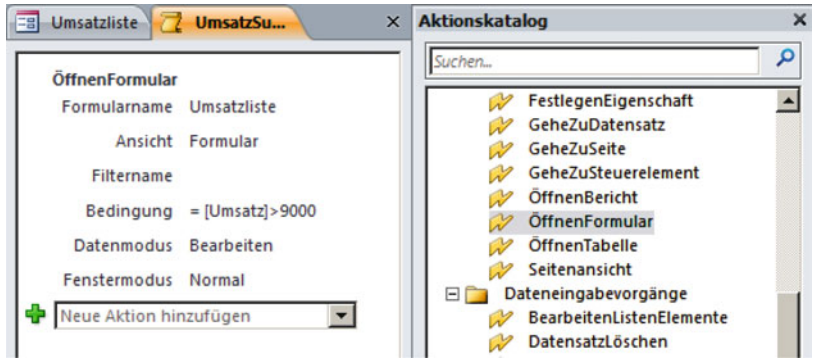


Abb. 6.30 Anlegen eines Makros

ID	Region	Verkäufer	Kunde	Produktkategorie	Umsatz	Auftragsnummer	Auftragseingang
3	West	Holzer	Enkel & Nachfol	Metallschraube	9.529	A2010008	10.01.2010
14	West	Bauer	Zimmermann oH	Dichtungen	9.323	A2010051	21.01.2010
22	West	Holzer	Enkel & Nachfol	Metallschraube	9.529	A2010008	04.02.2010
40	Nord	Lehmann	Enkel & Nachfol	Stirnräder	9.959	A2010040	12.03.2010
46	West	Näher	Arnsberg GmbH	Metallschraube	9.402	A2010053	19.03.2010
50	Nord	Lehmann	Freimann KG	Gelenkbolzen	9.090	A2010087	29.03.2010
76	Ost	Uhl	Kramer GmbH	Kegelräder	9.004	A2010089	16.05.2010
86	West	Holzer	Enkel & Nachfol	Metallschraube	9.529	A2010008	02.06.2010
87	Nord	Lehmann	Freimann KG	Gelenkbolzen	9.090	A2010087	03.06.2010
104	West	Näher	Schnieder KG	Nieten	9.630	A2010057	30.06.2010

Abb. 6.31 Verkaufsliste mit großen Umsätzen

Im Entwurfsmodus des Makros wird im Menüband unter *Tools* eine *Konvertierung zu VBA* ausgewiesen. Das Ergebnis (Code 6.67) führt wieder zu einer *DoCmd*-Anweisung, eingebettet in einer Funktion.

Code 6.67 Der VBA-Code des konvertierten Makros

```
Function UmsatzSuche()
   On Error GoTo UmsatzSuche_Err

   DoCmd.OpenForm "Umsatzliste", acNormal, "", _
      "[Umsatz]>9000", acEdit, acNormal

UmsatzSuche_Exit:
   Exit Function

UmsatzSuche_Err:
   MsgBox Error$
   Resume UmsatzSuche_Exit

End Function
```

Die Funktion enthält ein Fehlerhandling, falls das Auswahlkriterium nicht erfüllbar ist. Aus der bisherigen Entwicklung würde eine VBA-Prozedur das Anwendungs-Objekt nutzen und hätte eine etwas andere Form (Code 6.68).

Code 6.68 Die Prozedur erfüllt die gleiche Aufgabe wie das Makro

```
Sub AppDoCommand()
   Dim appAccess As Application

   Set appAccess = Application
   With appAccess.DoCmd
      .OpenForm "Umsatzliste", acNormal, "", _
         "[Umsatz]>9000", acFormEdit, acWindowNormal
   End With
   Set appAccess = Nothing
End Sub
```

Aus der Handhabung des Makro-Editors wird schnell ersichtlich, dass eine VBA-Prozedur oder -Funktion viel leistungsfähiger ist als ein Access-Makro.

6.7.2 AutoExec

Ein Access-Makro mit dem Namen *AutoExec* genießt eine Sonderstellung unter den Access-Makros. Sein Inhalt wird mit dem Öffnen der Datenbank ausgeführt. Damit lassen sich wieder, wie bei den anderen Anwendungen auch, Steuerelement im Menüband platzieren.

Im Makro genügt der Aufruf einer Funktion (Abb. 6.32).

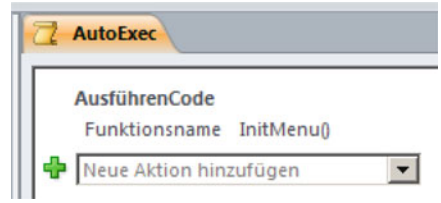


Abb. 6.32 Beispiel für ein Makro AutoExec

In einem Code-Modul steht dann die eigentliche Menüprozedur (Code 6.69).

Code 6.69 Die Prozedur installiert Steuerelemente im Menüband

```
Const sMenuName As String = "MenuTitle"

Function InitMenu()
   Dim objMenuBar       As CommandBar
   Dim objMenuGroup     As CommandBarControl
   Dim objMenuButton    As CommandBarControl

   Set objMenuBar = Application.CommandBars.Add(sMenuName, msoBarTop)
   objMenuBar.Visible = True

'--- Auswahlliste / Menükopf ---
   Set objMenuGroup = objMenuBar.Controls.Add _
      (Type:=msoControlPopup, Temporary:=False)
   With objMenuGroup
      .Caption = sMenuName
      .Tag = sMenuName
      .TooltipText = "Aktion wählen ..."
   End With

'--- Button ---
   Set objMenuButton = objMenuGroup.Controls.Add _
      (Type:=msoControlButton, Temporary:=True)
   With objMenuButton
      .BeginGroup = False
      .Caption = "Formular öffnen"
      .FaceId = 59
```

```
        .OnAction = "AppDoCommand"
        .Style = msoButtonIconAndCaption
        .TooltipText = "Formular öffnen"
        .Tag = "Formular öffnen"
    End With

    Set objMenuBar = Nothing
    Set objMenuGroup = Nothing
    Set objMenuBar = Nothing
End Function
```

Im Beispiel wird ein Steuerelement *Popup* als Container für mehrere *Button*-Steuerelemente installiert (Abb. 6.33). In diesem Fall ist es nur der Aufruf der Prozedur *AppDoCommand* (Code 6.70), die ihrerseits das Formular *Umsatzliste* mit der Auswertung öffnet (Abb. 6.34).

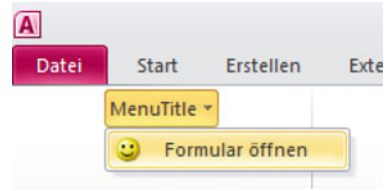


Abb. 6.33 Einträge im Menüband unter Add-Ins

Umsatzliste

Verkaufsliste

ID	Region	Verkäufer	Kunde	Produktkategorie	Umsatz	Auftragsnummer	Auftragseingang
8	West	Holzer	Enkel & Nachfol	Metallschraube	9.529	A2010008	10.01.2010
14	West	Bauer	Zimmermann oH	Dichtungen	9.323	A2010051	21.01.2010
22	West	Holzer	Enkel & Nachfol	Metallschraube	9.529	A2010008	04.02.2010
40	Nord	Lehmann	Enkel & Nachfol	Stirnräder	9.959	A2010040	12.03.2010
46	West	Näher	Arnsberg GmbH	Metallschraube	9.402	A2010053	19.03.2010
50	Nord	Lehmann	Freimann KG	Gelenkbolzen	9.090	A2010087	29.03.2010
76	Ost	Uhl	Kramer GmbH	Kegelräder	9.004	A2010089	16.05.2010
86	West	Holzer	Enkel & Nachfol	Metallschraube	9.529	A2010008	02.06.2010

Abb. 6.34 Umsatzliste aller Umsätze deren Wert größer als 9000 ist

Code 6.70 Die Prozedur erzeugt eine Umsatzliste mit großen Umsätzen

```
Sub AppDoCommand()
   Dim appAccess As Application

   Set appAccess = Application
   With appAccess.DoCmd
      .OpenForm "Umsatzliste", acNormal, "", _
         "[Umsatz]>9000", acFormEdit, acWindowNormal
   End With
   Set appAccess = Nothing
End Sub
```

6.8 Access-Interaktionen

Die Stärke von Access sind Relationen und deren Auswertung, dargestellt in Formularen und Berichten.

6.8.1 Access-Tabellen in Excel auswerten

Die Methode *TransferSpreadsheet* des *DoCmd*-Objekts ist ein mächtiges Werkzeug und erlaubt den Export, Import oder die Verknüpfung zwischen Access-Formularen und Excel-Worksheets.

```
'Syntax
  DoCmd.TransferSpreadsheet( _
     TransferType, _
     SpreadsheetType, _
     TableName, _
     FileName, _
     HasFieldNames, _
     Range)
```

Die Syntax wird in der VBA-Hilfe umfassend erläutert. Anhang 6, Tab. A6.21 zeigt die Konstanten für den *TransferType* und Anhang 6, Tab. A6.22 wichtige Konstanten für den *SpreadsheetType*.

Die Prozedur (Code 6.71) erzeugt ein neues Excel-Workbook in dem vorgegebenen Pfad und exportiert die Daten aus dem Formblatt *Verkaufsliste* in das erste Worksheet. Dies bekommt automatisch den Namen des Formblatts.

Code 6.71 Die Prozedur exportiert die Daten aus einem Formular in ein Excel-Workbook

```
Sub TableToExcel()
   Dim sTable As String
   Dim sExcel As String

   DoCmd.OpenForm FormName:="Verkaufsliste"
   sTable = Forms("Verkaufsliste").RecordSource
   sExcel = "C:\Temp\Liste.xlsx"
   DoCmd.TransferSpreadsheet _
      TransferType:=acExport, _
      SpreadsheetType:=acSpreadsheetTypeExcel12Xml, _
      TableName:=sTable, _
      FileName:=sExcel, _
      HasFieldNames:=True
End Sub
```

Leider gibt es in der VBA-Hilfe ein paar Ungereimtheiten. So wird als Konstante für den SpreadsheetType nur *acSpreadsheetTypeExcel9* angegeben. Damit ist nur ein Transfer in den xls-Dateityp möglich. Mit der Angabe *xlsx* kann die Datei nicht geöffnet werden. Gibt man in der Programmierung hinter der Konstanten ein + Zeichen ein, dann zeigt die Intellisence-Funktion (Abb. 6.35) noch zwei weitere *SpreadsheetType*-Konstante.

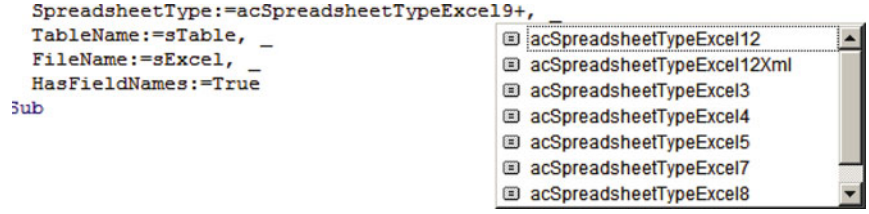


Abb. 6.35 SpreadsheetType-Konstante in der Auswahl

Nur die *SpreadsheetType*-Konstante *acSpreadsheetTypeExcel12Xml* liefert das gewünschte xlsx-Format (Abb. 6.36).

	A	B	C	D	E	F	G	H	I	J	K
1	ID	Region	Verkäufer	Kunde	Produktka	Umsatz	Auftragsn	Auftragsei	UmsatzZu	UmsatzDruckfedern	
2	1	West	Weber	Rühring G	Vierkant	7767,45	A2010043	########			
3	2	Süd	Lehrer	Anderer &	Zugfederr	120,56	A2010094	########	120,56		
4	3	Ost	Filzer	Pollen Gm	Kupplung	1490,03	A2010099	########			
5	4	Ost	Uhl	Ross oHG	Wälzlager	1610,7	A2010070	########			
6	5	Nord	Lehmann	Uhlmann (	Wälzlager	5213,46	A2010073	########			
7	6	Süd	Sandmanr	Hobel Gm	Druckfede	7921	A2010002	########		7.921,00	
8	7	Nord	Hauer	Freimann	Zugfederr	7597,92	A2010060	########	7597,92		
9	8	West	Holzer	Enkel & N	Metallsch	9529,23	A2010008	########			
10	9	Süd	Steinreich	Dober & S	Tellerfede	1507,88	A2010097	########			
11	10	West	Weber	Assmann (	Metallsch	2575,39	A2010080	########			
12	11	West	Renner	Kortmann	Kegelräde	5028,85	A2010100	########			
13	12	Nord	Lehmann	Bahr oHG	Tellerfede	2118,03	A2010049	########			
14	13	West	Weber	Pollen Gm	Gleitlager	7891,77	A2010020	########			
15	14	West	Bauer	Zimmerm	Dichtunge	9322,63	A2010051	########			
16	15	Ost	Wanda	Ross oHG	Gelenkbo	7264,24	A2010061	########			
17	16	Ost	Wanda	Conen oH	Stirnräder	1712,06	A2010084	########			
18	17	Ost	Wanda	Freimann	Gleitlager	6500,32	A2010059	########			
19	18	West	Näher	Brommel	Gleitlager	2191,89	A2010034	########			

Abb. 6.36 Die exportierten Daten

Da Excel der Spezialist für Diagramme ist, kann aus der exportierten Tabelle zum Beispiel noch ein Kreisdiagramm auf einer PowerPoint-Folie erstellt werden, natürlich aus Access heraus.

Die Prozedur (Code 6.72) öffnet das zuvor erstellte Excel-Workbook und erstellt eine Pivot-Tabelle zur Verkaufsliste. Mit der *Region* in der Zeilenbelegung und der *Umsatz-Summe* im Datenfeld wird daraus ein Kreisdiagramm. Anhang 6, Tab. A6.23 enthält alle Konstante für die Eigenschaft *Orientation* eines Feldes in der Pivot-Tabelle. Das Kreisdiagramm erhält ein eigenes *Sheet* und wird in die Zwischenablage kopiert.

Eine neu eröffnete Präsentation übernimmt das Diagramm. Während die Excel-Anwendung im Hintergrund bleibt, ist die Präsentation nach der Erstellung weiterhin sichtbar und kann bearbeitet werden (Abb. 6.37).

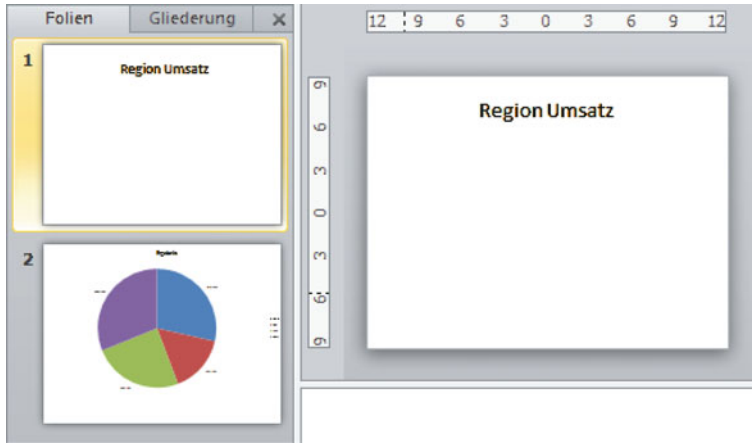


Abb. 6.37 Das Ergebnis der Excel-Pivotisierung

Code 6.72 Die Prozedur erstellt aus einer Excel-Tabelle eine Folie mit Diagramm

```
Sub CreateDiagrammFolie()
   Dim exlApp      As Excel.Application
   Dim exlBook     As Excel.Workbook
   Dim exlSheet    As Excel.Worksheet
   Dim exlPivot    As Excel.Worksheet
   Dim objTable    As Excel.PivotTable
   Dim objField    As Excel.PivotField
   Dim objShape    As Excel.Shape
   Dim pptDemo     As PowerPoint.Application
   Dim pptShow     As PowerPoint.Presentation
   Dim pptFolie    As PowerPoint.Slide
   Dim sExcel      As String

'Öffnen der Excel-Verkaufsliste
   sExcel = "C:\Temp\Liste.xlsx"
   Set exlApp = Excel.Application
   Set exlBook = exlApp.Workbooks.Open(sExcel)
   Set exlSheet = exlBook.Worksheets("Verkaufsliste")
'Öffnen der Pivot-Tabelle
   exlSheet.Select
   exlSheet.Range("A1").Select
   Set objTable = exlSheet.PivotTableWizard
'Zeilenbelegung
   Set objField = objTable.PivotFields("Region")
   objField.Orientation = xlRowField
   Set objField = Nothing
```

```
'Datenfeldbelegung mit Summenbildung
   Set objField = objTable.PivotFields("Umsatz")
   objField.Orientation = xlDataField
   objField.Function = xlSum
   objField.NumberFormat = "#,##0"
   Set objField = Nothing
'Chart erstellen
   Set exlPivot = exlBook.Worksheets("Tabelle1")
   exlPivot.Range("A3:B6").Select
   Set objShape = exlPivot.Shapes.AddChart
   With objShape.Chart
      .ChartType = xlPie
      .SeriesCollection(1).Select
      .SetElement (msoElementDataLabelOutSideEnd)
      .ChartArea.Select
      .Location Where:=xlLocationAsNewSheet, Name:="Region Umsatz"
   End With
   exlBook.Sheets("Region Umsatz").Select
   Selection.Copy
   exlBook.Saved = True
   exlBook.Close

'Neue Präsentation öffnen
   Set pptDemo = CreateObject("PowerPoint.Application")
   With pptDemo
      .Visible = True
      .Presentations.Add
   End With
   Set pptShow = pptDemo.ActivePresentation
'Erste Folie mit Titel
   Set pptFolie = pptShow.Slides.Add(1, ppLayoutTitleOnly)
   With pptFolie
      .Shapes(1).TextFrame.TextRange.Text = "Region Umsatz"
   End With
   Set pptFolie = Nothing
'Zweite Folie mit Chart
   Set pptFolie = pptShow.Slides.Add(2, ppLayoutBlank)
   With pptFolie
      .Shapes.PasteSpecial (ppPasteDefault)
   End With
   Set pptFolie = Nothing
   Set pptShow = Nothing
   Set pptDemo = Nothing
   Set objShape = Nothing
   Set objTable = Nothing
```

```
    Set exlPivot = Nothing
    Set exlSheet = Nothing
    Set exlBook = Nothing
    Set exlApp = Nothing
End Sub
```

6.8.2 Excel-Tabellen in Access verbinden

In diesem Beispiel sind zwei Excel-Tabellen gegeben, die in Verbindung zueinander stehen (Abb. 6.38 und 6.39). In Excel kann zwar mit MSQuery eine Relation erstellt und daraus auch Abfragen abgeleitet werden, doch ist dies in Access um ein Vielfaches leichter, zumal das Ergebnis als Formular oder Bericht ansprechend gestaltet und ausgegeben werden kann.

Abb. 6.38 Excel-Worksheet Produktionsdaten

	A	B	C	D	E	F
1	Produkt	Bauteil	Anzahl	Std	Gehalt	MNr
2	A	1	7	3,5	29,80 €	6
3	A	2	1	4,3	69,52 €	1
4	A	1	4	2,5	27,45 €	5
5	A	3	2	4,4	76,31 €	14
6	A	4	0	6,9	57,09 €	16
7	A	5	17	9,5	85,88 €	16
8	A	6	1	7,5	79,51 €	1
9	A	7	7	4,8	86,93 €	15
10	A	8	19	1,4	27,68 €	15

Produktionsdaten

Abb. 6.39 Excel-Worksheet Maschinendaten

	A	B	C
1	MNr	Stundensatz	
2	1	31,25 €	
3	2	45,77 €	
4	3	69,87 €	
5	4	55,26 €	
6	5	39,96 €	
7	6	41,28 €	
8	7	21,42 €	
9	8	67,34 €	
10	9	58,17 €	

Maschinendaten

Doch auch hier will ich nur einen Minimalweg zeigen. Die beiden Tabellen befinden sich in zwei unterschiedlichen Excel-Workbooks, da die Methode *TransferSpreadsheet* immer nur auf das erste Arbeitsblatt eines Workbooks zugreift. Nach dem Import wird eine Relation zwischen den Tabellen aufgebaut, mit deren Hilfe dann die Personal- und Maschinenkosten in einer Abfrage bestimmt werden. Die Ausgabe soll in diesem Fall eine PDF-Datei sein.

In der Tabelle *Produktionsdaten* sind Produkte aufgeführt, die aus einer unterschiedlichen Anzahl von Bauteilen bestehen. Zur Fertigung ist die angegebene Zeit in Stunden nötig. Dafür ist ein Mitarbeiter mit dema ngegebenen Gehalt/Stunde erforderlich. Ebenso wird in dem Zeitraum eine Maschine (MNr) eingesetzt, deren Stundenkosten in der Tabelle *Maschinendaten* stehen.

Die Prozedur (Code 6.73) erzeugt zuerst in einer leeren Datenbank die erforderlichen Tabellen-Templates. Sind die Tabellen bereits vorhanden, dann werden sie gelöscht. An mehreren Stellen befindet sich in der Prozedur die Anwendungs-Methode *RefreshDatabaseWindow*. So können Anweisungsergebnisse gut im Einzelschritt (F8) getestet werden.

Code 6.73 Die Prozedur erstellt Tabellen-Templates

```
Sub CreateTables()
   Dim objData    As Database
   Dim objTable   As TableDef
   Dim sTable     As String

   Set objData = CurrentDb
   On Error Resume Next
'Neue Tabelle Produktionsdaten
   sTable = "Produktionsdaten"
   objData.TableDefs.Delete (sTable)
   Application.RefreshDatabaseWindow
   Set objTable = objData.CreateTableDef(sTable)
   With objTable
      .Fields.Append .CreateField("Produkt", dbText)
      .Fields.Append .CreateField("Bauteil", dbLong)
      .Fields.Append .CreateField("Anzahl", dbLong)
      .Fields.Append .CreateField("Std", dbDouble)
      .Fields.Append .CreateField("Gehalt", dbDouble)
      .Fields.Append .CreateField("MNr", dbLong)
   End With
   objData.TableDefs.Append objTable
   Application.RefreshDatabaseWindow
   Set objTable = Nothing
'Neue Tabelle Maschinendaten
   sTable = "Maschinendaten"
   objData.TableDefs.Delete (sTable)
   Application.RefreshDatabaseWindow
   Set objTable = objData.CreateTableDef(sTable)
   With objTable
      .Fields.Append .CreateField("MNr", dbLong)
      .Fields.Append .CreateField("Stundensatz", dbDouble)
   End With
```

```
   objData.TableDefs.Append objTable
   Application.RefreshDatabaseWindow
   On Error GoTo 0
   Set objTable = Nothing
   Set objData = Nothing
End Sub
```

Liegen die Tabellen in Access vor, dann überträgt Prozedur (Code 6.74) die Daten mit der *DoCmd*-Methode *TransferSpreadsheet*.

Code 6.74 Die Prozedur überträgt die Excel-Daten

```
Sub TransferData()
   Dim sExcel     As String
   sExcel = "C:\Temp\06-08-02_Produktionsdaten.xlsx"
   DoCmd.TransferSpreadsheet _
      TransferType:=acImport, _
      SpreadsheetType:=acSpreadsheetTypeExcel12Xml, _
      TableName:="Produktionsdaten", _
      FileName:=sExcel, _
      HasFieldNames:=True
   sExcel = "C:\Temp\06-08-02_Maschinendaten.xlsx"
   DoCmd.TransferSpreadsheet _
      TransferType:=acImport, _
      SpreadsheetType:=acSpreadsheetTypeExcel12Xml, _
      TableName:="Maschinendaten", _
      FileName:=sExcel, _
      HasFieldNames:=True
End Sub
```

Die Prozedur (Code 6.75) erstellt eine Relation zwischen den Tabellen über den MNr-Schlüssel.

Code 6.75 Die Prozedur erstellt eine Relation

```
Sub CreateRelation()
   Dim objData       As Database
   Dim objRelation   As Relation
   Dim objField      As Field
   Dim sRelation     As String
   Dim sFields       As String
   Dim lRelCount     As Long
   Dim lCount        As Long
```

```
'löschen vorhandener Beziehungen
   Set objData = CurrentDb
   lRelCount = objData.Relations.Count
   For lCount = lRelCount - 1 To 0 Step -1
      objData.Relations.Delete (objData.Relations(lCount).Name)
   Next lCount
'Relation erstellen
   sRelation = "Maschinendaten_MNr_Produktionsdaten_MNr"
   Set objRelation = objData.CreateRelation()
   With objRelation
      .Name = sRelation                    'Relationsname
      .Table = "Maschinendaten"            'Primärschlüssel
      .ForeignTable = "Produktionsdaten"   'Fremdschlüssel
      .Attributes = dbRelationDontEnforce '1:n Beziehung
   End With
'Feld-Beziehungen
   sFields = "MNr_MNr"
   Set objField = objRelation.CreateField(sFields, dbLong)
   With objField
      .Name = "MNr"
      .ForeignName = "MNr"
   End With
'Feldbeziehung der Relation hinzufügen
   objRelation.Fields.Append objField
'Relation der Datenbank hinzufügen
   objData.Relations.Append objRelation

   Set objField = Nothing
   Set objRelation = Nothing
   Set objData = Nothing
End Sub
```

Die Prozedur (Code 6.76) erstellt in einer Abfrage die Personal- und Maschinenkosten, die dann als PDF-Datei (Abb. 6.40) ausgegeben wird.

Abb. 6.40 Ausgabe der Produktkosten als PDF-Datei

ProduktKosten

Produkt	Bauteil	Personal	Maschine
A	1	68,625	99,9
A	1	104,3	144,48
A	2	298,936	134,375
A	3	335,764	234,96
A	4	393,921	212,382
A	5	815,86	292,41
A	6	596,325	234,375
A	7	417,264	243,504
A	8	38,752	71,022
A	9	25,816	149,52
A	10	604,412	616,602
A	11	582,271	184,375
A	12	696,575	633,369

Code 6.76 Die Prozedur erzeugt eine Abfrage und die als PDF-Datei aus

```
Sub CreateKostenReport()
   Dim objData    As Database
   Dim objQuery   As QueryDef
   Dim sSql       As String

   sSql = "SELECT Produktionsdaten.Produkt, " & _
      "Produktionsdaten.Bauteil, " & _
      "[Std]*[Gehalt] AS Personal, " & _
      "[Std]*[Stundensatz] AS Maschine " & _
      "FROM Maschinendaten " & _
      "INNER JOIN Produktionsdaten " & _
      "ON Maschinendaten.MNr = Produktionsdaten.MNr " & _
      "GROUP BY Produktionsdaten.Produkt, " & _
      "Produktionsdaten.Bauteil, " & _
      "[Std]*[Gehalt], [Std]*[Stundensatz];"
   Set objData = CurrentDb
   Set objQuery = objData.CreateQueryDef("ProduktKosten", sSql)
   Application.RefreshDatabaseWindow
   DoCmd.OutputTo ObjectType:=acOutputQuery, _
      ObjectName:="ProduktKosten", _
      OutputFile:="C:\Temp\Report.pdf", _
      OutputFormat:=acFormatPDF
   Set objQuery = Nothing
   Set objData = Nothing
End Sub
```

Anhang 6, Tab. A6.24 zeigt die wichtigsten Objekttypen für die Methode *OutputTo*.

Visio-Objekte

7

Visio ist mit seinen vielen Vorlagen ein gelungenes Business-Grafikprogramm und da es zur Familie der Office-Anwendungen gehört, auch mit einer VBA-Entwicklungsumgebung ausgestattet. Auch hier muss für andere Anwendungen zur frühen Bindung die Visio-Objektbibliothek unter Verweise eingebunden werden. Visio ist kein Ersatz für ein CAD-Programm.

7.1 Visio-Anwendungen

Das Anwendungs-Objekt *Application* ist das oberste Objekt der Visio-Objekt-Hierarchie (Abb. 7.1). Im Gegensatz zu Access gibt es hier eine klare Objektstruktur.

Im Objektkatalog befindet sich eine ausführliche Darstellung der Visio-Objektmodellreferenz. Mit ihr kann durch die Dokumentation navigiert werden. Ebenso gibt es eine Aufstellung aller Elemente des Application-Objekts und die der nachfolgenden Objekte.

7.1.1 Eigenschaften und Methoden

Das Anwendungs-Objekt von Visio verhält sich anders als das Anwendungsobjekt von Excel. Mit dem Start einer Visio-Anwendung, und es können mehrere gleichzeitig geöffnet sein, wird die Auswahl einer Vorlage angeboten. Wird die Auswahl abgebrochen, so enthält die Visio-Anwendung ein dunkles Fenster und die Einträge im Menüband sind alle ausgegraut. Ohne eine Auswahl ist Visio nicht arbeitsfähig.

Von allen Eigenschaften, die eine Visio-Anwendung besitzt, werden im Beispiel (Code 7.1) einige ausgegeben. Im Objektkatalog finden sich noch weitere, für die sich eine ausführliche Betrachtung lohnt.

H. Nahrstedt, *Die Welt der VBA-Objekte*, DOI 10.1007/978-3-658-13891-2_7

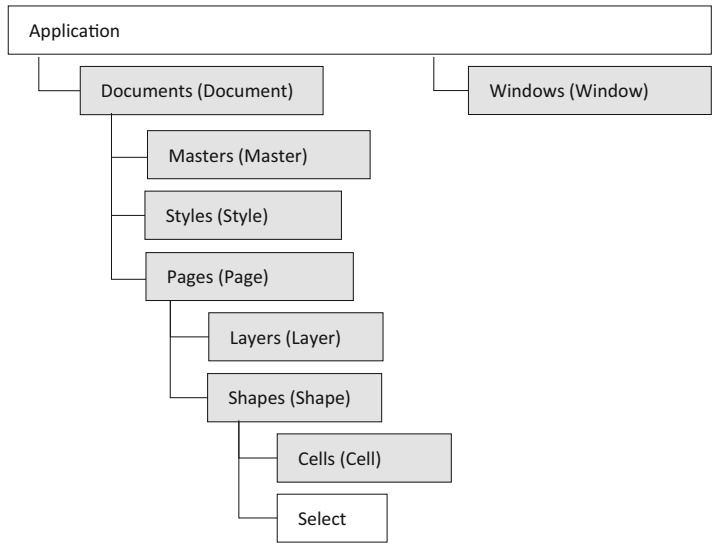


Abb. 7.1 Die wichtigsten Unterobjekte und Objektlisten (*grau*) des Visio Application-Objekts

Code 7.1 Die Prozedur öffnet eine Anwendung und liefert Informationen

```
Sub ApplicationStart()
   Dim vsoApp     As Visio.Application
   Dim sInfo      As String
   Dim sReturn    As Variant

   Set vsoApp = New Application
   With vsoApp
      .Visible = True
      Debug.Print "Version: " & .Version
      Debug.Print "SystemPfad: " & .Path
      Debug.Print "Aktiver Drucker: " & .ActivePrinter
      Debug.Print "Benutzer Name: " & .UserName
      Debug.Print "Arbeitspfad: " & .DrawingPaths
      .Quit
   End With
   Set vsoApp = Nothing
End Sub
```

Im vorangegangenen Beispiel wurde die Methode *Quit* zum Schließen der instanziierten Anwendung genutzt. Methoden unter Visio, von denen es einige gibt, werden erst interessant, wenn das eigentliche Arbeitsobjekt *Document* in der Anwendung existiert. Das gilt auch für die Ereignisse.

Die Prozedur (Code 7.2) liest alle Zeichnungsordner der aktuellen Anwendung.

Code 7.2 Die Prozedur liest alle aktuellen Zeichnungsordner

```
Sub ReadDrawingFolder()
   Dim vsoApp     As Visio.Application
   Dim sFolder()  As String
   Dim iMin       As Integer
   Dim iMax       As Integer
   Dim iLoop      As Integer

   Set vsoApp = Application
   With vsoApp
      .DrawingPaths = "C:\Temp"
      .EnumDirectories .DrawingPaths, sFolder
      iMin = LBound(sFolder)
      iMax = UBound(sFolder)
      For iLoop = iMin To iMax
         Debug.Print sFolder(iLoop)
      Next iLoop
   End With

   Set vsoApp = Nothing
End Sub
```

7.1.2 Windows

Ein Unterobjekt der Anwendung ist, wie in fast allen anderen Anwendungen, das Objekt *Window*. Es liegt als Objektsammlung *Windows* vor und ein einzelnes Fenster kann über einen Index ausgewählt werden.

Die Prozedur (Code 7.3) liefert den Kopfzeilentext aller geöffneten Fenster in der aktuellen Anwendung.

Code 7.3 Die Prozedur liefert alle Fenstertexte der aktuellen Anwendung

```
Sub ReadWindows()
   Dim iLoop As Integer

   For iLoop = 1 To Visio.Application.Windows.Count
      Debug.Print Visio.Application.Windows(iLoop).Caption
   Next iLoop
End Sub
```

Ein besonderes Windows-Objekt ist das *ActiveWindow*, das derzeit aktuelle Fenster. Die wichtigsten Methoden von Window sind *Active*, *Close* und *SetWindowRect*.

7.1.3 ThisDocument

Mit dem Öffnen einer leeren Zeichnung unter *Möglichkeiten für einen Anfang* existiert im Projekt-Explorer das Visio-Objekt *ThisDocument*. Es bringt neben Eigenschaften und Methoden auch viele Ereignisprozeduren mit. Insbesondere die Ereignisse *DocumentOpened* und *BeforeDocumentClose*, die nach dem Öffnen eines Anwendungsdokuments und vor dem Schließen aufgerufen werden, so wie bei anderen Anwendungen (Code 7.4).

Code 7.4 Ereignisprozeduren im Dokument

```
Private Sub Document_DocumentOpened(ByVal Doc As IVDocument)
   InitMenu
End Sub

Private Sub Document_BeforeDocumentClose(ByVal Doc As IVDocument)
   RemoveMenu
End Sub
```

Die Prozedur (Code 7.5) gibt einige Eigenschaften des Dokuments im Direktfenster aus.

Code 7.5 Die Prozedur zeigt einige Eigenschaften von ThisDocument im Direktfenster

```
Sub DocumentProperties()
   Dim vsoApp  As Visio.Application
   Dim vsoDoc  As Visio.Document

   Set vsoApp = Visio.Application
   Set vsoDoc = vsoApp.ActiveDocument
   With vsoDoc
      Debug.Print ".................Name: " & .Name
      Debug.Print "................Titel: " & .Title
      Debug.Print ".................Pfad: " & .Path
      Debug.Print "...............Format: " & .PaperSize
      Debug.Print ".............Drucker: " & .Printer
      Debug.Print ".............Vorlage: " & .Template
      Debug.Print "...........Erzeugt am: " & .TimeCreated
      Debug.Print "Letzte Bearbeitung am: " & .TimeEdited
   End With

   Set vsoDoc = Nothing
   Set vsoApp = Nothing
End Sub
```

Die Prozedur (Code 7.6) öffnet eine neue Anwendung und erzeugt ein Dokument nach einer Vorlage. Das Dokument wird danach unter einem Namen gespeichert und die Anwendung geschlossen.

Code 7.6 Die Prozedur öffnet eine neue Anwendung

```
Sub CreateNewDocument()
   Dim vsoApp     As Visio.Application
   Dim vsoDoc     As Visio.Document

   Set vsoApp = New Visio.Application
   Set vsoDoc = vsoApp.Documents.Add("C:\Temp\Demo.vsd")
   vsoDoc.SaveAs ("C:\Temp\Demo2.vsd")
   vsoApp.Quit
   Set vsoDoc = Nothing
   Set vsoApp = Nothing
End Sub
```

7.2 Visio-Dokumente

Das Dokument-Objekt *Document* ist vergleichbar mit einer Excel-Arbeitsmappe. Eine Excel-Anwendung kann mehrere Arbeitsmappen enthalten, die wiederum mehrere Arbeitsblätter besitzen können. Eine Visio-Anwendung kann mehrere Dokumente enthalten, die wiederum mehrere Seiten besitzen können (Abb. 7.2).

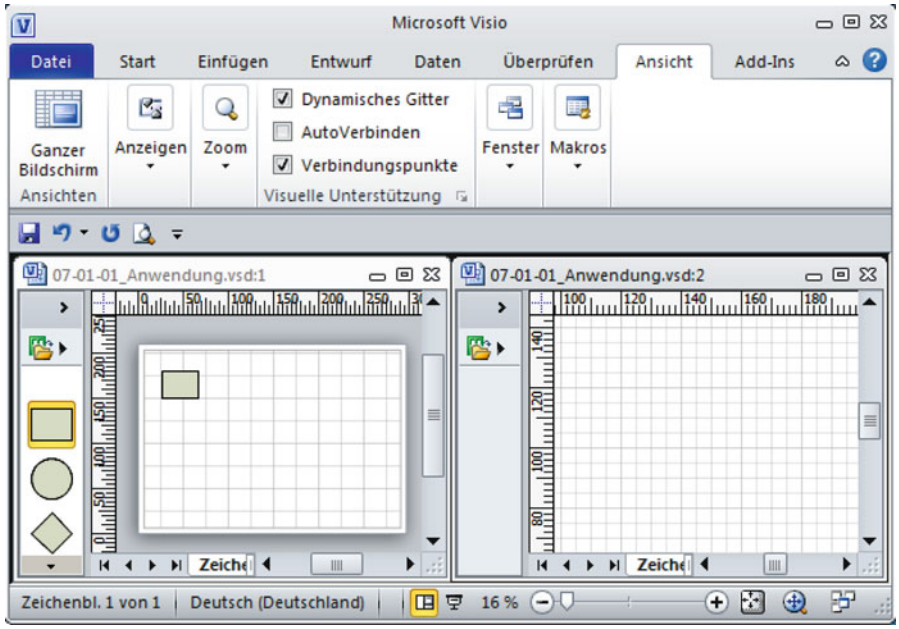


Abb. 7.2 Zwei Dokumente in einer Visio-Anwendung

Die Dokumente in einer Visio-Anwendung haben hinter dem Titel der Anwendung eine fortlaufende Nummer, getrennt durch einen Doppelpunkt.

7.2.1 Eigenschaften

Die Prozedur (Code 7.7) liest alle vorhandenen Dokumente der aktuellen Anwendung und gibt einige ihrer Eigenschaften im Direktfenster aus.

Code 7.7 Die Prozedur gibt Eigenschaften aktiver Dokumente aus

```
Sub ReadAllDocuments()
   Dim vsoApp  As Visio.Application
   Dim vsoDoc  As Visio.Document
```

```
   Set vsoApp = Visio.Application
   For Each vsoDoc In vsoApp.Documents
      With vsoDoc
         Debug.Print ""
         Debug.Print ".....Name: " & .Name
         Debug.Print ".....Type: " & .Type
         Debug.Print "alternate Names: " & .AlternateNames
         Debug.Print "...Beschreibung: " & .Description
      End With
   Next
   Set vsoApp = Nothing
End Sub
```

7.2.2 Methoden

Wie ein Dokument in einer neuen Anwendung erstellt wird wurde bereits gezeigt. Die Prozedur (Code 7.8) erzeugt ein neues Dokument in einer bereits vorhandenen Anwendung mit Dokument(en).

Code 7.8 Die Prozedur erzeugt ein weiteres Dokument in der Anwendung

```
Sub CreateInsideNewDocument()
   Dim vsoApp     As Visio.Application
   Dim vsoDoc     As Visio.Document
   Const VORLAGE  As String = "C:\Temp\Demo.vst"

   Set vsoApp = Visio.Application
   Set vsoDoc = vsoApp.Documents.Add(VORLAGE)
   With vsoDoc
      .SaveAs ("C:\Temp\Demo2.vst")
      .Close
   End With
   Set vsoApp = Nothing
End Sub
```

Mit der Methode *Print* lassen sich Dokumente ausgeben. Visio bietet zusätzliche Print-Parameter, mit denen die Ausgabe gestaltet werden kann (Anhang 7, Tab. A7.1).

Die Ränder einer Druckausgabe lassen sich durch Eigenschaften (Anhang 7, Tab. A7.2) bestimmen.

Ein neuer Wert sollte zusammen mit einer Maßeinheit (Anhang 7, Tab. A7.3) angegeben werden.

Die Prozedur (Code 7.9) setzt die Ränder der Druckausgabe und die Druckform, bevor das Dokument ausgedruckt wird.

Code 7.9 Die Prozedur druckt das aktuelle Dokument

```
Sub PrintDocument()
   Dim vsoApp  As Visio.Application
   Dim vsoDoc  As Visio.Document

   Set vsoApp = Visio.Application
   Set vsoDoc = vsoApp.ActiveDocument
   With vsoDoc
      .LeftMargin(visMillimeters) = 20
      .RightMargin(visMillimeters) = 20
      .TopMargin(visMillimeters) = 20
      .BottomMargin(visMillimeters) = 20
      .PrintLandscape = True
      .Print
   End With
   Set vsoApp = Nothing
End Sub
```

7.2.3 Ereignisse

Das Visio-Dokument bringt einige Ereignisse mit, die im Objektkatalog ausreichend beschrieben sind. Aber auch das Unterobjekt *Window* bringt ähnliche Ereignisse mit. Unter anderem gibt es hier einige Mausereignisse. Um sie zu nutzen, wird in der Klasse *clsMouseAction* eine Objektvariable *vsoWindow* mit den Events von *Window* definiert (Code 7.10).

Code 7.10 Die Klasse der Mausaktionen

```
Dim WithEvents vsoWindow As Visio.Window

Private Sub Class_Initialize()
   Set vsoWindow = ActiveWindow
End Sub
Private Sub Class_Terminate()
   Set vsoWindow = Nothing
End Sub

Private Sub vsoWindow_MouseDown(ByVal Button As Long, _
   ByVal KeyButtonState As Long, ByVal x As Double, _
   ByVal y As Double, CancelDefault As Boolean)
   If Button = 1 Then
      Debug.Print "Left mouse botton down"
```

```
   ElseIf Button = 2 Then
      Debug.Print "Right mouse button down"
   ElseIf Button = 16 Then
      Debug.Print "Center mouse button down"
   End If
End Sub

Private Sub vsoWindow_MouseMove(ByVal Button As Long, _
   ByVal KeyButtonState As Long, ByVal x As Double, _
   ByVal y As Double, CancelDefault As Boolean)
   Debug.Print "x-Pos: "; x
   Debug.Print "y-Pos: "; y
End Sub

Private Sub vsoWindow_MouseUp(ByVal Button As Long, _
   ByVal KeyButtonState As Long, ByVal x As Double, _
   ByVal y As Double, CancelDefault As Boolean)
   If Button = 1 Then
      Debug.Print "Left mouse botton up"
   ElseIf Button = 2 Then
      Debug.Print "Right mouse button up"
   ElseIf Button = 16 Then
      Debug.Print "Center mouse button up"
   End If
End Sub
```

In der Klasse erzeugt der Konstruktor *Class_Initialize* die Objektvariable *objWindow*, die dann mit dem Destruktor *Class_Terminate* wieder zerstört wird. Das Ereignis *MouseDown* meldet im Direktfenster welche Maustaste gedrückt wurde. Das Ereignis *MouseUp* meldet im Direktfenster welche Maustaste freigegeben wird. Und letztlich schreibt das Ereignis *MouseMove* die Position der Maus auf dem Dokument fortlaufend ins Direktfenster.

Das Ereignis *DocumentSaved* im Objekt *ThisDocument* (Code 7.11) instanziiert die erforderliche Objektvariable. Mit dem Schließen des Dokuments *BeforeDocumentClose* wird die Instanziierung wieder aufgehoben. Der Start erfolgt nach dem Öffnen der Anwendung mit der Menüanweisung *Speichern*.

Code 7.11 Instanziierung und Terminierung der Mausereignisse

```
Dim vsoMouse As clsMouseAction

Private Sub Document_DocumentSaved(ByVal doc As IVDocument)
   Set vsoMouse = New clsMouseAction
End Sub
```

```
Private Sub Document_BeforeDocumentClose(ByVal doc As IVDocument)
   Set vsoMouse = Nothing
End Sub
```

7.2.4 Formatvorlagen

Dokumenten können Formatvorlagen zugewiesen werden und beinhalten entsprechende Eigenschaftswerte. Mit der Erstellung eines Dokuments wird eine Standard-Formatvorlage gesetzt.

Die Prozedur (Code 7.12) liest alle vorhandenen Formatvorlagen und gibt ihre Namen im Direktfenster aus.

Code 7.12 Die Prozedur liest alle vorhandenen Styles

```
Sub ReadStyles()
   Dim vsoApp     As Visio.Application
   Dim vsoDoc     As Visio.Document
   Dim vsoStyle   As Style

   Set vsoApp = Visio.Application
   Set vsoDoc = vsoApp.ActiveDocument
   For Each vsoStyle In vsoDoc.Styles
      Debug.Print vsoStyle.Name
   Next

   Set vsoApp = Nothing
   Set vsoDoc = Nothing
End Sub
```

Die Prozedur (Code 7.13) weist dem aktiven Dokument einen neuen Formatnamen zu und löscht ihn anschließend wieder. Die neue Formatvorlage kann auf einer vorhandenen basieren und wird dann unter *BasedOn* angegeben.

Code 7.13 Die Prozedur erzeugt einen neuen Style

```
Sub SetDocumentStyle()
   Dim vsoApp As Visio.Application
   Dim vsoDoc As Visio.Document

   Set vsoApp = Visio.Application
   Set vsoDoc = vsoApp.ActiveDocument
   On Error Resume Next
```

```
   vsoDoc.Styles.Add _
      stylename:="Neuer Stil", _
      BasedOn:="", _
      fincludesText:=True, _
      fincludesLine:=True, _
      fincludesfill:=True
   vsoDoc.Styles("Neuer Stil").Delete

   Set vsoApp = Nothing
   Set vsoDoc = Nothing
End Sub
```

In Formatvorlagen lassen sich Seitenabmessungen, Formenvorlagen, Farben und weitere Eigenschaften einstellen, so dass der Start einer neuen Zeichnung vereinfacht wird.

7.3 Visio-Seiten

Jedes Visio-Dokument kann mehrere Seiten besitzen. Daher finden wir hier auch die Objektliste *Pages* vor.

7.3.1 Eigenschaften

Wie bei allen Objektlisten gibt es die Eigenschaft *Count*, mit der die Anzahl vorhandener Objekte abgefragt werden kann.

Die Prozedur (Code 7.14) liest die vorhandenen Seiten des aktiven Dokuments und einige Eigenschaften der Seite im Direktfenster aus.

Code 7.14 Die Prozedur liest alle vorhandenen Dokumentseiten

```
Sub ReadPages()
   Dim vsoApp  As Visio.Application
   Dim vsoDoc  As Visio.Document
   Dim vsoPage As Visio.Page

   Set vsoApp = Visio.Application
   Set vsoDoc = vsoApp.ActiveDocument
   For Each vsoPage In vsoDoc.Pages
      With vsoPage
         Debug.Print .Name
         Debug.Print .ObjectType
```

```
         Debug.Print .PageSheet
         Debug.Print .Stat
         Debug.Print .Type
      End With
   Next
   Set vsoApp = Visio.Application
   Set vsoDoc = vsoApp.ActiveDocument
End Sub
```

7.3.2 Methoden

Ebenso wie bei allen Objektlisten gibt es auch hier die Methoden *Add*, *Quit*, *Delete* und *Print*.

Die Prozedur (Code 7.15) erstellt im aktiven Dokument eine neue Seite und gibt ihr einen Namen.

Code 7.15 Die Prozedur erzeugt ein Hintergrundblatt

```
Sub CreatePage()
   Dim vsoDoc  As Visio.Document
   Dim vsoPage As Visio.Page

   Set vsoDoc = Application.ActiveDocument
   Set vsoPage = vsoDoc.Pages.Add
   With vsoPage
      .Name = "Hintergrund"
      .Background = True
   End With
   vsoDoc.Pages("Zeichenblatt-1").BackPage = "Hintergrund"
   Set vsoDoc = Nothing
   Set vsoPage = Nothing
End Sub
```

Die Eigenschaft *Background* der neuen Seite wird auf *True* gesetzt. Dadurch wird die Seite zu einer Hintergrundseite. Eine Hintergrundseite kann hinter einer Vordergrundseite angezeigt und so von mehreren Vordergrundseiten genutzt werden. Eine Hintergrundseite eignet sich zur Darstellung eines Firmenlogos und zur gleichmäßigen Einteilung der Seiten.

Einer Vordergrundseite kann eine Hintergrundseite über das Kontextmenü der Registerkarte und der Auswahl *Seite einrichten* zugewiesen bekommen. Im Code erledigt das die Eigenschaft *Backpage*.

7.3.3 Ereignisse

Die meisten Ereignisse auf einer Seite beschäftigen sich eher mit den Darstellungselementen als mit der Seite selbst. Neben den Ereignissen *BeforePageDelete* und *PageDeleteCanceled* ist *PageChanged* sicher das interessanteste Ereignis.

Die Prozedur (Code 7.16) meldet Veränderungen auf der aktuellen Seite, wie z. B. die einer Namensänderung auf der Registerkarte.

Code 7.16 Die Ereignisprozedur meldet Seitenänderungen

```
Private Sub Document_PageChanged(ByVal Page As IVPage)
   MsgBox Page.Name & " wurde verändert!"
End Sub
```

Die Ereignisprozedur *PageChanged* gehört zum Dokument-Objekt und kann nur ins Klassenmodul *ThisDocument* geschrieben werden.

7.4 Visio-Formen

Das eigentliche Arbeitsobjekt unter Visio sind die Form-Objekte *Shapes*, die äußerst vielseitige Darstellungen aufweisen können, von einfachen Linien und Rechtecken bis hin zu sehr detailreichen Formen. Sie können Gebäude und Straßen in einem Lageplan sein, Computersymbole in einem Netzwerkdiagramm, Stationen und Schienen in einem Metroplan oder Möbel und Wände in einer Einrichtungsplanung. Alle Möglichkeiten aufzuführen würde hier zu weit führen.

Bei ihrer Darstellung unterscheidet man eindimensionale (1D) und zweidimensionale (2D) Formen. 1D-Formen zeigen, wenn sie markiert sind, einen Anfangs- und Endpunkt (Abb. 7.3). Mit der Maus können sie verschoben werden. 2D-Formen zeigen markiert acht Ziehpunkte und können damit horizontal, vertikal oder diagonal in ihrer Größe verändert werden. Zusätzliche Ziehpunkte wie gelbe Rauten steuern das Verhalten der Formen.

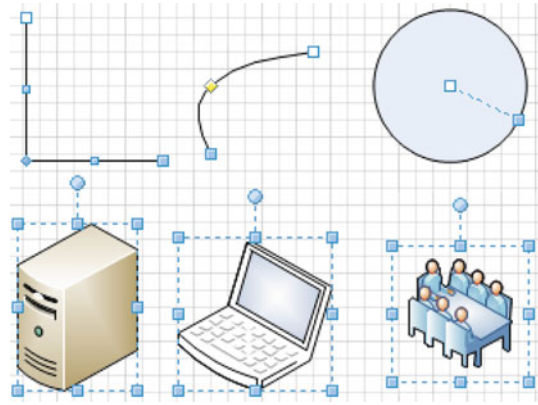


Abb. 7.3 1D- und 2D-Formen

Die Prozedur (Code 7.17) liest alle vorhandenen Formen auf einer vorgegebenen Seite.

Code 7.17 Die Prozedur liest alle Shapes einer vorgegebenen Page

```
Sub ReadShapes()
   Dim vsoApp     As Visio.Application
   Dim vsoDoc     As Visio.Document
   Dim vsoPage    As Visio.Page
   Dim vsoShape   As Visio.Shape

   Set vsoApp = Visio.Application
   Set vsoDoc = vsoApp.ActiveDocument
   Set vsoPage = vsoDoc.Pages("Zeichenblatt-3")
   For Each vsoShape In vsoPage.Shapes
      With vsoShape
         Debug.Print .Name
         Debug.Print .Text
         Debug.Print .FillStyle
      End With
   Next
   Set vsoPage = Nothing
   Set vsoApp = Nothing
   Set vsoDoc = Nothing
End Sub
```

Mithilfe einiger Zeichnungsmethoden können einfache Formen in VBA erstellt werden.

7.4.1 Gerade Formen

Mit der Methode *DrawLine* werden Linien auf der Zeichenfläche *Page* erstellt.

```
'Syntax
Page.DrawLine (xBegin, yBegin, xEnd, yEnd)
```

Als Parameter werden die Koordinaten des Anfangs- und Endpunktes (Anhang 7, Tab. A7.4) der Linie in der Maßeinheit Inch angegeben. Die Reihenfolge der Parameter muss stets eingehalten werden. Mit den Methoden *SwapEnds* (1D) und *ReverseEnds* (2D) kann die Reihenfolge vertauscht werden.

Mit der Methode *DrawRectangle* können Rechtecke erstellt werden. Als Parameter werden die Koordinaten des Rechtecks (Anhang 7, Tab. A7.5) ebenfalls in Inch angegeben.

```
'Syntax
Page.DrawRectangle (xBegin, yBegin, xEnd, yEnd)
```

Die Prozedur (Code 7.18) erstellt eine Linie und ein Rechteck auf der aktuellen Seite. Zwei Funktionen sorgen für die Umrechnung der Maßeinheiten.

Code 7.18 Die Prozedur zeichnet eine Linie und ein Rechteck

```
Sub DrawLines()
   Dim vsoApp     As Visio.Application
   Dim vsoDoc     As Visio.Document
   Dim vsoPage    As Visio.Page
   Dim vsoShape   As Visio.Shape

   Set vsoApp = Visio.Application
   Set vsoDoc = vsoApp.ActiveDocument
   Set vsoPage = vsoDoc.Pages("Zeichenblatt-2")
   Set vsoShape = vsoPage.DrawLine( _
      m2i(50), m2i(50), m2i(100), m2i(100))
   With vsoShape
      .Name = "Linie"
      .Text = "Linie"
   End with
   Set vsoApp = Nothing
   Set vsoShape = vsoPage.DrawRectangle( _
      m2i(150), m2i(100), m2i(200), m2i(150))
   With vsoShape
      .Name = "Rechteck"
      .Text = "Rechteck"
   End with
   Set vsoApp = Nothing
   Set vsoDoc = Nothing
   Set vsoPage = Nothing
   Set vsoShape = Nothing
End Sub

Function m2i(ByVal mm As Double)
   m2i = mm / 25.4
End Function

Function i2m(ByVal inch As Double)
   i2m = inch * 25.4
End Function
```

7.4.2 Gebogene Formen

Mit der Methode *DrawArcByThreePoints* wird, wie der Name schon sagt, ein Bogen durch drei als Parameter vorgegebene Punkte gezeichnet. Der Bogen verfügt somit über einen Anfangs- und Endpunkt sowie einen mittleren Kontrollpunkt.

```
'Syntax
Page.DrawArcByThreePoints (xBegin, yBegin, xEnd, yEnd, xControl, _
  yControl)
```

Die Methode *DrawOval* zeichnet eine Ellipse mit vier Koordinaten. Im Sonderfall wird daraus ein Kreis.

```
'Syntax
Page.DrawOval (xBegin,yBegin,xEnd,yEnd)
```

Die Prozedur (Code 7.19) zeichnet einen Bogen und einen Kreis auf das aktuelle Zeichenblatt.

Code 7.19 Die Prozedur zeichnet einen Bogen und eine Ellipse

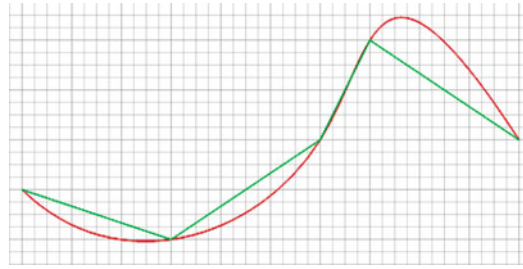
```
Sub DrawArcs()
   Dim vsoApp     As Visio.Application
   Dim vsoDoc     As Visio.Document
   Dim vsoPage    As Visio.Page
   Dim vsoShape   As Visio.Shape

   Set vsoApp = Visio.Application
   Set vsoDoc = vsoApp.ActiveDocument
   Set vsoPage = vsoDoc.Pages("Zeichenblatt-2")
   Set vsoShape = vsoPage.DrawArcByThreePoints( _
      m2i(90), m2i(80), m2i(160), m2i(120), _
      m2i(95), m2i(85))
   With vsoShape
      .Name = "Bogen"
      .Text = "Bogen"
   End with
   Set vsoApp = Nothing
   Set vsoShape = vsoPage.DrawOval( _
      m2i(220), m2i(40), m2i(260), m2i(130))
   With vsoShape
      .Name = "Oval"
      .Text = "Oval"
   End with
```

```
    Set vsoApp = Nothing
    Set vsoDoc = Nothing
    Set vsoPage = Nothing
    Set vsoShape = Nothing
End Sub
```

Etwas komplexer sind die Kurven (Anhang 7, Tab. A7.6), die durch Draw-Methoden erstellt werden können. Es müssen dazu weitere Parameter definiert werden.

Die Prozedur (Code 7.20) nutzt die Methode *DrawSpline* um eine Approximationskurve durch die vorgegebenen Punkte nach der Methode der kleinsten Fehlerquadrate zu erstellen. Gleichfalls wird mit der Methode *DrawPolyline* durch die gleichen Punkte eine Verbindungslinie erzeugt (Abb. 7.4).

Abb. 7.4 Erzeugte Kurven



Code 7.20 Die Prozedur erzeugt Kurven zwischen mehreren Punkten

```
Sub DrawCurves()
    Dim vsoApp          As Visio.Application
    Dim vsoDoc          As Visio.Document
    Dim vsoPage         As Visio.Page
    Dim vsoShape        As Visio.Shape
    Dim dP(1 To 10)     As Double

    dP(1) = m2i(40)
    dP(2) = m2i(60)
    dP(3) = m2i(100)
    dP(4) = m2i(40)
    dP(5) = m2i(160)
    dP(6) = m2i(80)
    dP(7) = m2i(180)
    dP(8) = m2i(120)
    dP(9) = m2i(240)
    dP(10) = m2i(80)
```

```
    Set vsoApp = Visio.Application
    Set vsoDoc = vsoApp.ActiveDocument
    Set vsoPage = vsoDoc.Pages("Zeichenblatt-3")
    Set vsoShape = vsoPage.DrawSpline(dP, m2i(20), 1)
    With vsoShape
       .Name = "Spline"
       .Text = "Spline"
       .CellsSRC(visSectionObject, visRowLine, _
          visLineColor).FormulaU = "THEMEGUARD(RGB(255,0,0))"
    End With
    Set vsoShape = Nothing
    Set vsoShape = vsoPage.DrawPolyline(dP, 1)
    With vsoShape
       .Name = "Polyline"
       .Text = "Polyline"
       .CellsSRC(visSectionObject, visRowLine, _
          visLineColor).FormulaU = "THEMEGUARD(RGB(0,255,0))"
    End With
    Set vsoShape = Nothing
    Set vsoApp = Nothing
    Set vsoDoc = Nothing
    Set vsoPage = Nothing
End Sub
```

7.4.3 Formenblätter

Jede Form *Shape* besitzt ein Formblatt *ShapeSheet*, in dem alle Eigenschaften einer Form festgehalten werden. Form und Formblatt sind unterschiedliche Darstellungen der gleichen Datenmenge, die eine grafisch und die andere in einer Eigenschaftsliste. Für die Darstellung einer Form wird das Formblatt zuvor ausgewertet.

Das Formblatt besitzt eine tabellarische Struktur und ist wegen der vielen Eigenschaften in Bereiche unterteilt. Aufgerufen wird das Formblatt (Abb. 7.5) einer markierten Form über das Menüband Register *Entwicklertools* / *Shape-Design* / *ShapeSheet anzeigen*.

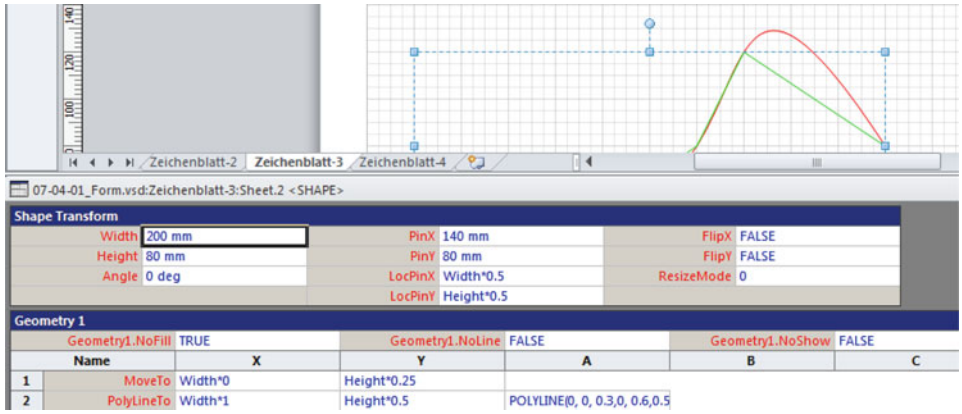


Abb. 7.5 Shape und ShapeSheet-Ausschnitt

Die Prozedur (Code 7.21) liest die Formeln aus dem zugehörigen Formblatt der durch die Methode *DrawPolyline* erzeugten Form mit dem Namen *Curve2* und gibt die Daten im Direktfenster aus (Abb. 7.6).

Abb. 7.6 Die Ausgabe im Direktfenster

```
Direktbereich
Zeile:  1
Geometry1.NoFill            TRUE
Geometry1.NoLine            FALSE
Geometry1.NoShow            FALSE
Geometry1.NoSnap            FALSE
Geometry1.Path
Geometry1.NoQuickDrag       FALSE
Zeile:  2
Geometry1.X1  Width*0
Geometry1.Y1  Height*0,25
Zeile:  3
Geometry1.X2  Width*1
Geometry1.Y2  Height*0,5
Geometry1.A2  POLYLINIE(0; 0; 0,3;0; 0,6;0,5; 0,7;1)
```

Code 7.21 Die Prozedur liest die Daten der Polyline Kurve aus dem ShapeSheet

```
Sub ReadShapeSheet()
   Dim vsoApp     As Visio.Application
   Dim vsoDoc     As Visio.Document
   Dim vsoPage    As Visio.Page
   Dim vsoShape   As Visio.Shape
   Dim vsoSect    As Visio.Section
   Dim iRow       As Integer
   Dim iCol       As Integer
```

```
    Set vsoApp = Visio.Application
    Set vsoDoc = vsoApp.ActiveDocument
    Set vsoPage = vsoDoc.Pages("Zeichenblatt-3")
    Set vsoShape = vsoPage.Shapes("Curve2")
    Set vsoSect = vsoShape.Section(visSectionProp)
    On Error Resume Next
    With vsoSectvisSectionProp
       For iRow = 1 To .Count
          Debug.Print "Zeile: "; iRow
          For iCol = 0 To 10
             Debug.Print .Row(iRow - 1).Cell(iCol).Name, _
                .Row(iRow - 1).Cell(iCol).Formula
          Next iCol
       Next iRow
    End With
    On Error GoTo 0
    Set vsoSect = Nothing
    Set vsoShape = Nothing
    Set vsoApp = Nothing
    Set vsoDoc = Nothing
    Set vsoPage = Nothing
End Sub
```

Mit *visSectionFirstComponent* wird der erste Geometriebereich eines Formblatts gekennzeichnet. Der nächste Bereich wird durch die Konstante *visSectionFirstComponent + 1* bestimmt, und so fort, bis mit *visSectionLastComponent* der letzte Geometriebereich angewählt werden kann. Alle Bereiche einer Form stehen unter *VisSectionIndices-Konstanten* in der Visio-Hilfe.

Da die Anzahl der Zellen in den Bereichen unterschiedlich ist, leistet die Anweisung *On Error Resume Next* gute Dienste. Mit der Methode *SectionExists* kann geprüft werden ob der Bereich bereits angelegt wurde. Mit der Methode *CellExists* kann dies für jede Zelle erfolgen.

Durch Schreiben von Werten in das Formblatt lässt sich die Form ebenso leicht verändern, besonders wenn exakte Werte erforderlich sind. Die Prozedur (Code 7.22) erzeugt ein Rechteck auf der aktiven Seite und verändert nachträglich dessen Maße und Darstellung.

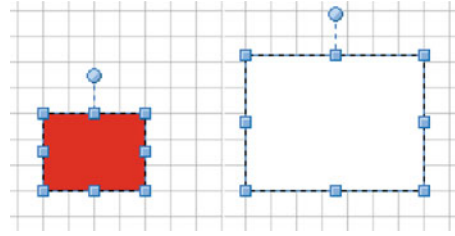
Code 7.22 Die Prozedur erzeugt ein Rechteck und überschreibt die ShapSheet-Zellen

```
Sub WriteToSheet()
   Dim appVisio   As Visio.Application
   Dim docVisio   As Visio.Document
   Dim pagVisio   As Visio.Page
   Dim shpVisio   As Visio.Shape
   Dim secVisio   As Visio.Section
```

```
   Set appVisio = Application
   Set docVisio = appVisio.ActiveDocument
   Set pagVisio = ActivePage
   Set shpVisio = pagVisio.DrawRectangle( _
      m2i(100), m2i(70), m2i(220), m2i(160))
   With shpVisio
      .Cells("Width").Result("mm") = 110
      .Cells("Height").Result("mm") = 80
      .Cells("PinX").Result(visMillimeters) = 185
      .Cells("Width").Result(visMillimeters) = 145
      .Cells("Angle").Result(visDegrees) = 30
      .CellsSRC(visSectionObject, visRowFill, _
         visFillForegnd).FormulaU = "THEMEGUARD(RGB(255,255,0))"
   End With
   Set shpVisio = Nothing
   Set pagVisio = Nothing
   Set appVisio = Nothing
   Set docVisio = Nothing
End Sub
```

Aber nicht nur Eigenschaftswerte sondern auch Verhaltensfunktionen können eingesetzt werden. Die Prozedur (Code 7.23) erzeugt ein Rechteck auf der aktuellen Seite und schreibt im Formblatt-Bereich *Fill Format* in die Zelle *FillForegnd* eine *IF*-Anweisung. Die sorgt dafür, dass die Füllfarbe von Weiß auf Rot wechselt, sobald mindestens eines der beiden Maße *Width* und *Height* unter 50 mm gezogen wird (Abb. 7.7).

Abb. 7.7 Bedingte Füllfarbe



Code 7.23 Die Prozedur erzeugt ein Rechteck mit funktionellem Verhalten

```
Sub SetFunction()
   Dim vsoApp     As Visio.Application
   Dim vsoPage    As Visio.Page
   Dim vsoShape   As Visio.Shape
   Dim iBorder    As Integer
   Dim iColor1    As Integer
   Dim iColor2    As Integer
   Dim sFormel    As String
```

```
   Set vsoApp = Application
   Set vsoPage = vsoApp.ActivePage
   Set vsoShape = vsoPage.DrawRectangle( _
      m2i(100), m2i(70), m2i(220), m2i(160))
   With vsoShape
      iBorder = m2i(50)
      sFormel = "IF(OR(Width<" & Str(iBorder) & _
         ", Height<" & Str(iBorder) & ")," & _
         visRed & "," & visWhite & ")"
      .Cells("FillForegnd").FormulaU = sFormel
   End With
   Set vsoShape = Nothing
   Set vsoPage = Nothing
   Set vsoApp = Nothing
End Sub
```

7.4.4 Formen gruppieren

Formen besitzen die Möglichkeit mit anderen Formen eine Gruppe bilden zu können. Dadurch können sehr komplexe Formen entstehen. Eine Gruppe bildet wiederum eine Form und besitzt daher alle Eigenschaften und Methoden von Formen, einschließlich eines Formblatt-Objekts *ShapeSheet*.

Die Prozedur (Code 7.24) erzeugt drei Formen, markiert sie und fasst sie mit der Methode *Group* zu einer neuen Form zusammen.

Code 7.24 Die Prozedur erzeugt eine Formengruppe

```
Sub CreateGroup()
   Dim vsoApp     As Visio.Application
   Dim vsoPage    As Visio.Page
   Dim vsoShape   As Visio.Shape

   Set vsoApp = Visio.Application
   Set vsoPage = vsoApp.ActivePage
'erzeugt Rechtecke
   Set vsoShape = vsoPage.DrawRectangle( _
      m2i(100), m2i(50), m2i(260), m2i(120))
   vsoShape.Name = "Rechteck1"
   Set vsoShape = Nothing
   Set vsoShape = vsoPage.DrawRectangle( _
      m2i(200), m2i(140), m2i(250), m2i(180))
   vsoShape.Name = "Rechteck2"
   Set vsoShape = Nothing
```

```
'erzeugt Ellipse
   Set vsoShape = vsoPage.DrawOval( _
      m2i(150), m2i(80), m2i(190), m2i(150))
   vsoShape.Name = "Ellipse"
   Set vsoShape = Nothing
'Gruppierung
   With ActiveWindow
      .Select vsoPage.Shapes("Rechteck1"), visSelect
      .Select vsoPage.Shapes("Rechteck2"), visSelect
      .Select vsoPage.Shapes("Ellipse"), visSelect
      .Group
   End With
   Set vsoApp = Nothing
   Set vsoPage = Nothing
End Sub
```

Die Objekte *ActiveWindow*, *ActiveDocument* und *ActivePage* erleichtern die Programmierung durch direkten Zugriff auf aktive Objekte.

7.4.5 Formen verbinden

Formen stehen oft in einer Relation zueinander und dies wird durch das Verbindungslinien-Objekt *ConnectShape* dargestellt. Diese Verbindungen sollen nicht verloren gehen, wenn eine Form auf der Seite verschoben wird.

Zur Erstellung einer Verbindung besitzen Formen Verbindungspunkte. Sie werden mit einer Verbindungsform, im einfachsten Fall eine Linie, verbunden. Die Prozedur (Code 7.25) erzeugt ein Rechteck, einen Kreis und verbindet beide Formen mit einer Verbindungslinie.

Code 7.25 Die Prozedur erzeugt zwei Formen und verbindet sie

```
Sub CreateLink()
   Dim vsoApp     As Visio.Application
   Dim vsoPage    As Visio.Page
   Dim vsoShape1  As Visio.Shape
   Dim vsoShape2  As Visio.Shape

   Set vsoApp = Visio.Application
   Set vsoPage = vsoApp.ActivePage
'erzeugt Rechteck
   Set vsoShape1 = vsoPage.DrawRectangle( _
      m2i(50), m2i(40), m2i(140), m2i(120))
```

```
'erzeugt Ellipse
   Set vsoShape2 = vsoPage.DrawOval( _
      m2i(160), m2i(80), m2i(240), m2i(150))
'erzeugt Verbinder
   With ActiveWindow
      .DeselectAll
      .Select vsoShape1, visSelect
      .Select vsoShape2, visSelect
      Application.ActiveWindow.Selection.ConnectShapes
   End With
   Set vsoShape1 = Nothing
   Set vsoShape2 = Nothing
   Set vsoApp = Nothing
   Set vsoPage = Nothing
End Sub
```

Shapes lassen sich aber auch durch die Methode *GlueTo* dauerhaft „verkleben“. Bei dieser Methode werden die Klebepunkte angegeben. Die Prozedur (Code 7.26) verbindet zwei Rechtecke nach dieser Methode.

Code 7.26 Die Prozedur verbindet zwei Shapes mit der Methode GlueTo

```
Sub GlueShapes()
   Dim vsoApp        As Visio.Application
   Dim vsoPage       As Visio.Page
   Dim vsoShape1     As Visio.Shape
   Dim vsoShape2     As Visio.Shape
   Dim vsoArrow      As Visio.Shape
   Dim celGlueBegin  As Visio.Cell
   Dim celGlueEnd    As Visio.Cell
   Dim celGlueShp1   As Visio.Cell
   Dim celGlueShp2   As Visio.Cell

   Set vsoApp = Visio.Application
   Set vsoPage = vsoApp.ActivePage

'erzeugt Rechteck
   Set vsoShape1 = vsoPage.DrawRectangle( _
      m2i(50), m2i(40), m2i(140), m2i(120))
'erzeugt Ellipse
   Set vsoShape2 = vsoPage.DrawRectangle( _
      m2i(160), m2i(80), m2i(240), m2i(150))
'erzeugt Verbinder
   Set vsoArrow = vsoPage.DrawLine( _
      m2i(10), m2i(10), m2i(20), m2i(20))
```

```
'verkleben
   Set celGlueBegin = vsoArrow.Cells("BeginX")
   Set celGlueEnd = vsoArrow.Cells("EndX")
   Set celGlueShp1 = vsoShape1.Cells("Geometry1.X3")
   Set celGlueShp2 = vsoShape2.Cells("Geometry1.X1")
   celGlueBegin.GlueTo celGlueShp1
   celGlueEnd.GlueTo celGlueShp2

   Set celGlueBegin = Nothing
   Set celGlueEnd = Nothing
   Set celGlueShp1 = Nothing
   Set celGlueShp2 = Nothing
   Set vsoShape1 = Nothing
   Set vsoShape2 = Nothing
   Set vsoApp = Nothing
   Set vsoPage = Nothing
End Sub
```

7.4.6 Vorlagen und Schablonen

Damit die Erstellung von Zeichnungen schneller erfolgen kann, stellt Visio Vorlagenobjekte *Templates* (vom Typ *.VST) und Schablonen-Objekte *Stencils* (vom Typ *.VSS) in eigenen Dateien zur Verfügung. Die Dateien beinhalten eine Sammlung von Formen meist zu einem bestimmten Thema. Eine Form in einer Schablone wird als *MasterShape* bezeichnet und kann per *Drag & Drop* aus der Schablone auf die Seite gezogen werden.

In VBA können die Schablonen mit Methoden geöffnet und wieder geschlossen werden. Fast jede Zeichnung unter Visio enthält mehrere Schablonen. Die Prozedur (Code 7.27) öffnet ein neues Dokument über die Methode *Add* zusammen mit dem Template für *Brainstorming*-Formen und der dazugehörigen Schablone für *Legenden*-Formen. Danach wird die Schablone für *StandardFlussdiagramm*-Shapes geöffnet und wieder geschlossen.

Code 7.27 Die Prozedur öffnet und schließt Schablonen

```
Sub StencilInOut()
   Dim vsoApp  As Visio.Application
   Dim vsoDocs As Visio.Documents
   Dim sPath   As String

   Set vsoApp = Visio.Application
   Set vsoDocs = vsoApp.Documents
   sPath = "C:\Program Files (x86)\Microsoft Office\Office14\1031\"
```

```
   With vsoDocs
      .Add FileName:=sPath & "BSTORM_M.VST"
      .OpenEx sPath & "BASFLO_M.VSS", visOpenDocked
   End With
   Documents("BASFLO_M.vss").Close
   Set vsoDocs = Nothing
   Set vsoApp = Nothing
End Sub
```

Da es für Schablonen einen Standardordner gibt, ist eine Pfadangabe nicht erforderlich. Die Prozedur (Code 7.28) öffnet die Schablone *Netzwerk* und überträgt das darin enthaltene *MasterShape*-Objekt *PC* auf die Seite an die vorgegebene Stelle.

Code 7.28 Die Prozedur öffnet eine Standardschablone

```
Sub OpenStencil()
   Dim vsoApp        As Visio.Application
   Dim vsoDocs       As Visio.Documents
   Dim vsoPage       As Visio.Page
   Dim vsoStencil    As Visio.Document
   Dim vsoMaster     As Visio.Master
   Dim vsoShape      As Visio.Shape

   Set vsoApp = Visio.Application
   Set vsoDocs = vsoApp.Documents
   Set vsoPage = Application.ActivePage
   Set vsoStencil = vsoDocs.OpenEx("COMPS_M.VSS", _
      visOpenRO + visOpenDocked)

   Set vsoMaster = vsoStencil.Masters("PC")
   Set vsoShape = vsoPage.Drop(vsoMaster, m2i(200), m2i(100))

   Set vsoShape = Nothing
   Set vsoMaster = Nothing
   Set vsoStencil = Nothing
   Set vsoPage = Nothing
   Set vsoDocs = Nothing
   Set vsoApp = Nothing
End Sub
```

Die Prozedur (Code 7.29) öffnet die Basis-Schablone und überträgt drei verschiedene Formen. Die Positionen werden diesmal in Inch angegeben.

Code 7.29 Die Prozedur erzeugt drei verschiedene Basisformen auf der aktiven Seite

```
Sub UsedStencil()
   Dim vsoApp        As Visio.Application
   Dim vsoDocs       As Visio.Documents
   Dim vsoStencil    As Visio.Document
   Dim vsoPage       As Visio.Page
   Dim vsoMaster     As Visio.Master
   Dim vsoShape      As Visio.Shape
   Dim vsoStarMaster As Visio.Master
   Dim vsoStarShape  As Visio.Shape
   Dim vsoHexaMaster As Visio.Master
   Dim vsoHexaShape  As Visio.Shape

   Set vsoApp = Visio.Application
   Set vsoDocs = vsoApp.Documents
   Set vsoPage = Application.ActivePage
   Set vsoStencil = vsoDocs.OpenEx("BASIC SHAPES.VSS", visOpenDocked)

   Set vsoMaster = vsoStencil.Masters("Rechteck")
   Set vsoShape = vsoPage.Drop(vsoMaster, 4.25, 5.5)
   vsoShape.Text = "Rechteck"
   Set vsoShape = Nothing
   Set vsoMaster = Nothing

   Set vsoMaster = vsoStencil.Masters("Stern 7")
   Set vsoShape = vsoPage.Drop(vsoMaster, 2#, 5.5)
   vsoShape.Text = "Stern 7"
   Set vsoShape = Nothing
   Set vsoMaster = Nothing

   Set vsoMaster = vsoStencil.Masters("Sechseck")
   Set vsoShape = vsoPage.Drop(vsoMaster, 7#, 5.5)
   vsoShape.Text = "Sechseck"
   Set vsoShape = Nothing
   Set vsoMaster = Nothing
   Set vsoStencil = Nothing
   Set vsoPage = Nothing
   Set vsoDocs = Nothing
   Set vsoApp = Nothing
End Sub
```

7.4.7 Ereignisse

In Visio gibt es drei Möglichkeiten auftretende Ereignisse zu verarbeiten. Die einfachste besteht darin, die Ereignisse des *ThisDocument* Objekts zu nutzen. Dies wurde ja bereits mehrfach bei anderen Anwendungen gezeigt. Die zweite Möglichkeit, ebenfalls bereits genutzt, besteht in der Instanziierung einer Objektvariablen mit der *WithEvents* Angabe.

Eine dritte Möglichkeit bietet die Anwendung der Funktion *AddAdvice*. In einem Klassenmodul wird mit der Anweisung *Implements* das Interface *Visio.IvisEventProc* erstellt. Diese Schnittstelle liefert die Prozedur *VisEventProc*.

Zunächst wird in einem Klassenmodul *clsEvents* die Schnittstelle *IVisEventProc* installiert. Dann wird entsprechend dem Ereignisparameter *nEventCode* das Ereignis identifiziert und im Direktfenster ausgegeben. Die Ereigniskonstanten stehen unter *VisEventCode*-Konstanten in der Visio-Hilfe. Sie können auch als Summe ein Ereignis darstellen, so stehen *visEvtPage* + *visEvtAdd* in der Summe für Ereignis von Page + Ereignis Added = Ereignis Page.Added.

Die Codezeilen (Code 7.30) bilden den Inhalt des Klassenmoduls.

Code 7.30 Klassenmodul clsEvent

```
Implements Visio.IVisEventProc

'definiert visEvtAdd als 2-Byte Wert
'das verhindert einen run-time overflow error
Private Const visEvtAdd As Integer = &H8000

Private Function IVisEventProc_VisEventProc( _
   ByVal nEventCode As Integer, _
   ByVal pSourceObj As Object, _
   ByVal nEventID As Long, _
   ByVal nEventSeqNum As Long, _
   ByVal pSubjectObj As Object, _
   ByVal vMoreInfo As Variant) As Variant

   Dim sText As String
'Ereignis-Auswahl
   Select Case nEventCode
   Case visEvtCodeDocSave
      sText = "DocumentSaved (" & nEventCode & ")"
   Case (visEvtPage + visEvtAdd)
      sText = "PageAdded (" & nEventCode & ")"
   Case visEvtCodeShapeDelete
      sText = "ShapesDeleted(" & nEventCode & ")"
   Case Else
      sText = "Other (" & nEventCode & ")"
   End Select
```

```
'Ausgabe Ereignisname & Code
   Debug.Print sText
End Function
```

In einem Codemodul stehen die Prozeduren *CreateEventObjects* und *DeleteEventObjects* (Code 7.31). In der ersten werden alle notwendigen Objekte instanziiert, wie die Objektvariable der Klasse und drei Ereignisprozeduren über die Methode *AddAdvice*. Die zweite Prozedur terminiert alle instanziierten Objekte.

Code 7.31 Initialisierung und Terminierung der Ereignis-Objekte in modEvent

```
Private objEvent     As clsEvents

Dim vsoEvents        As Visio.EventList
Dim vsoEventDocSav   As Visio.Event
Dim vsoEventPagAdd   As Visio.Event
Dim vsoEventShpDel   As Visio.Event

'deklariert visEvtAdd als 2-byte Datentyp
'um einen run-time overflow Fehler zu vermeiden
Private Const visEvtAdd As Integer = &H8000

Public Sub CreateEventObjects()
'instanziiert ein Objekt der Ereignisklasse
   Set objEvent = New clsEvents
'instanziiert Ereignisliste für das aktive Dokument
   Set vsoEvents = ActiveDocument.EventList
'instanziiert ein DocumentSaved event
   Set vsoEventDocSav = _
      vsoEvents.AddAdvise( _
      visEvtCodeDocSave, objEvent, "", "Document saved...")
'instanziiert ein PageAdded event
   Set vsoEventPagAdd = _
      vsoEvents.AddAdvise( _
      visEvtAdd + visEvtPage, objEvent, "", "Page added...")
'instanziiert ein ShapesDelete event
   Set vsoEventShpDel = _
      vsoEvents.AddAdvise( _
      visEvtCodeShapeDelete, objEvent, "", "Shapes deleted...")
End Sub

Public Sub DeleteEventObjects()
'löscht DocumentSaved event
    vsoEventDocSav.Delete
    Set vsoEventDocSav = Nothing
```

```
'löscht PageAdded event
    vsoEventPagAdd.Delete
    Set vsoEventPagAdd = Nothing
'löscht ShapesDeleted event
    vsoEventShpDel.Delete
    Set vsoEventShpDel = Nothing
End Sub
```

Nach dem Start und der Auslösung der drei Ereignisse, stehen die Protokollzeilen im Direktfenster (Abb. 7.8).

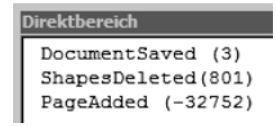


Abb. 7.8 Abb. 7.8 Ausgabe der Ereignistexte

7.5 Visio-Schichten

Visio bietet dem Anwender die Möglichkeit, Formen einem Schicht-Objekt *Layer* zuzuordnen. Schichten sind mit aufeinanderliegenden durchsichtigen Folien vergleichbar. Es können einzelne, mehrere oder alle übereinander betrachtet, bzw. gedruckt werden. Auch der Schutz vor Veränderung kann schichtweise erfolgen. So lassen sich Formen und Formengruppen thematisch zuordnen.

Bei der Instanziierung einer Form gibt es noch keine Zuordnung zu einer Schicht. Eine Schicht muss existieren, damit eine Zuweisung erfolgen kann. Schichten können instanziiert, verändert und auch wieder gelöscht werden. Einige *MasterShapes* verfügen bereits über Schichten.

7.5.1 Schichten verwalten

Schichten verhalten sich wie Seiten und sind in der Objektstruktur ihnen untergeordnet. Die Prozedur (Code 7.32) gibt die vorhandenen Schichten eines Dokuments im Direktfenster aus.

Code 7.32 Die Prozedur nennt alle vorhandenen Layer der ActivePage

```
Sub ReadAllLayers()
   Dim vsoApp     As Visio.Application
   Dim vsoPage    As Visio.Page
   Dim vsoLayer   As Visio.Layer
```

```
   Set vsoApp = Visio.Application
   Set vsoPage = vsoApp.ActivePage
   For Each vsoLayer In vsoPage.Layers
      Debug.Print vsoLayer.Name; ": "; _
         vsoLayer.Document
   Next
   Set vsoLayer = Nothing
   Set vsoPage = Nothing
   Set vsoApp = Nothing
End Sub
```

Die Prozedur (Code 7.33) erzeugt drei Formen auf der aktiven Seite *ActivePage* zusammen mit zwei Schichten und weist die Formen den Schichten zu.

Code 7.33 Die Prozedur erzeugt drei Shapes in zwei Layern

```
Sub CreateLayer()
   Dim vsoApp     As Visio.Application
   Dim vsoPage    As Visio.Page
   Dim vsoShape   As Visio.Shape
   Dim vsoLayers  As Visio.Layers
   Dim vsoLayer   As Visio.Layer

   Set vsoApp = Visio.Application
   Set vsoPage = vsoApp.ActivePage
   Set vsoLayers = vsoPage.Layers

'erzeugt zwei Rechtecke mit Layer
   Set vsoShape = vsoPage.DrawRectangle( _
      m2i(100), m2i(50), m2i(260), m2i(120))
   Set vsoLayer = vsoLayers.Add("Gerade Formen")
   vsoLayer.Add vsoShape, 1
   Set vsoShape = Nothing
   Set vsoShape = vsoPage.DrawRectangle( _
      m2i(200), m2i(140), m2i(250), m2i(180))
   Set vsoLayer = vsoLayers.Add("Gerade Formen")
   vsoLayer.Add vsoShape, 1
   Set vsoShape = Nothing
   Set vsoLayer = Nothing

'erzeugt Ellipse mit Layer
   Set vsoShape = vsoPage.DrawOval( _
      m2i(150), m2i(80), m2i(190), m2i(150))
   Set vsoLayer = vsoLayers.Add("Gebogene Formen")
```

```
   vsoLayer.Add vsoShape, 1
   Set vsoLayer = Nothing
   Set vsoShape = Nothing
   Set vsoPage = Nothing
   Set vsoApp = Nothing
End Sub
```

Die Zuordnung einer Form zu einer Schicht erfolgt mit der Methode *Add* und der Angabe des Form-Objekts. Der zweite Parameter gibt an, ob bei *ShapeGroups* die *KindObjects* ausgeschlossen werden (0 = false) oder nicht (1 = true). Analog kann mit der Methode *Remove* eine Form auch wieder von der Schicht entfernt werden. Mit der Methode *Delete* wird eine Schicht von einer Seite gelöscht.

7.5.2 Schichten Eigenschaften

Das Aus- und Einblenden von Schichten kann im Menüband unter *Start / Bearbeiten / Layer / Layereigenschaften* über die *Aktivität* erfolgen. In VBA zeigt die Prozedur (Code 7.34) diese Möglichkeit durch Wertveränderung in der zugehörigen Zelle des Formblatts der Form.

Code 7.34 Die Prozedur blendet die vorhandenen Schichten aus und ein

```
Sub ShowLayers()
   Dim vsoApp     As Visio.Application
   Dim vsoPage    As Visio.Page
   Dim vsoLayer   As Visio.Layer

   Set vsoApp = Visio.Application
   Set vsoPage = vsoApp.ActivePage
   For Each vsoLayer In vsoPage.Layers
      vsoLayer.CellsC(visLayerVisible).FormulaU = "0"
      Stop
      vsoLayer.CellsC(visLayerVisible).FormulaU = "1"
   Next
   Set vsoLayer = Nothing
   Set vsoPage = Nothing
   Set vsoApp = Nothing
End Sub
```

7.5.3 Schichten und Formen

Über die Methode *CreateSelection* einer aktiven Seite *ActivePage* ist es möglich, die Formen einer Schicht direkt anzusprechen. Die Prozedur (Code 7.35) markiert alle Formen einer vorgegebenen Schicht und verändert deren Eigenschaft *Style*.

Code 7.35 Die Prozedur markiert alle Shapes eines Layers und ändert deren Styles

```
Sub LayerShapesSelect()
   Dim vsoApp     As Visio.Application
   Dim vsoPage    As Visio.Page
   Dim vsoLayer   As Visio.Layer
   Dim vsoSelect  As Visio.Selection

   Set vsoApp = Visio.Application
   Set vsoPage = vsoApp.ActivePage
   With vsoPage
      Set vsoLayer = .Layers.ItemU("Gerade Formen")
      Set vsoSelect = .CreateSelection(visSelTypeByLayer, _
         visSelModeSkipSuper, vsoLayer)
   End With
   With Application.ActiveWindow
      .Selection = vsoSelect
      .Selection.FillStyle = "Guide"
      .Selection.LineStyle = "Guide"
      '.Selection.Style = "Normal" 'alles zurücksetzen
   End With

   Set vsoLayer = Nothing
   Set vsoPage = Nothing
   Set vsoApp = Nothing
End Sub
```

Doch es gibt nur wenige Eigenschaften, auf die über das Auswahl-Object *Selection* zugegriffen werden kann. Anders ist es bei einem direkten Zugriff auf die Formen, dann ist auch ein Zugriff auf deren Formblatt möglich.

Die Prozedur (Code 7.36) markiert zunächst alle Formen einer Schicht um dann über eine *For-Each-Next* Schleife auf sie zuzugreifen.

Code 7.36 Die Prozedur verändert nacheinander alle Formen einer Schicht

```
Sub LayerShapesChange()
   Dim vsoApp     As Visio.Application
   Dim vsoPage    As Visio.Page
   Dim vsoLayer   As Visio.Layer
   Dim vsoSelect  As Visio.Selection
   Dim vsoShape   As Visio.Shape

   Set vsoApp = Visio.Application
   Set vsoPage = vsoApp.ActivePage
   With vsoPage
      Set vsoLayer = .Layers.ItemU("Gerade Formen")
```

```
      Set vsoSelect = .CreateSelection(visSelTypeByLayer, _
         visSelModeSkipSuper, vsoLayer)
   End With

   Application.ActiveWindow.Selection = vsoSelect

   For Each vsoShape In Application.ActiveWindow.Selection
      Debug.Print vsoShape.Name
      With vsoShape
         .CellsSRC(visSectionObject, visRowFill, _
            visFillForegnd).FormulaU = "THEMEGUARD(RGB(255,255,0))"
         .CellsSRC(visSectionObject, visRowFill, _
            visFillBkgnd).FormulaU = "THEMEGUARD(SHADE(FillForegnd, _
            LUMDIFF(THEME(""FillColor""),THEME(""FillColor2""))))"
         .CellsSRC(visSectionObject, visRowFill, _
            visFillPattern).FormulaU = "1"
      End With
   Next

   Set vsoLayer = Nothing
   Set vsoSelect = Nothing
   Set vsoPage = Nothing
   Set vsoApp = Nothing
End Sub
```

In der Methode *CreateSelection* ist als erster Parameter die Konstante *visSelTypeByLayer* gesetzt. Andere Konstante (Anhang 7, Tab. A7.7) erlauben den Zugriff auf andere Form-Gruppierungen.

7.6 Visio-Container

Unter dem Sammler-Objekt *Container* werden in Visio Anordnungen von Formen verstanden, die von einem sichtbaren Rahmen umgeben sind. Durch das Ziehen von Formen mit der Maus auf den Container entsteht eine Sammlung. Mit dem Verschieben von Containern werden auch die zugehörigen Formen verschoben. Das gleiche Verhalten zeigt sich beim Kopieren und Löschen von Containern.

7.6.1 Container erstellen

Die Sammlungs-Objekte *Container* werden aus einer eigens dafür vorgesehenen Schablone erstellt. Die Prozedur (Code 7.37) bestimmt im ersten Schritt den Standardpfad unter

dem sich diese Schablone befindet. Anschließend wird die Schablone für Container instanziiert. Sie öffnet sich dabei in einem ausgeblendeten Fenster. Als *Master* wird der erste Container der Schablone gewählt.

Im nächsten Schritt wird der erste Container als Kopie des Masters an der vorgegebenen Position auf der ersten Seite mit der Methode *Drop* erstellt. Der zweite Container wird mit der Methode *DropContainer* erstellt und hat den ersten Container zum Inhalt. Folglich bedarf es keiner Positionsangabe. Der dritte Container wird mit einer leeren Auswahl erstellt. Wenn keine Positionsangabe erfolgt, wird der Container mittig eingestellt. Im letzten Schritt wird ein Rechteck-Shape erstellt und dem vierten Container bei der Instanziierung zugewiesen.

Code 7.37 Die Prozedur erstellt vier Container mit unterschiedlichem Inhalt

```
Sub PageDropContainer()
   Dim vsoPage       As Visio.Page
   Dim vsoStencil    As Visio.Document
   Dim vsoMaster     As Visio.master
   Dim vsoCont       As Visio.Shape
   Dim vsoShape      As Visio.Shape
   Dim vsoSel        As Visio.Selection
   Dim sContPath     As String

'Schablone und Master
   sContPath = Application.GetBuiltInStencilFile( _
      visBuiltInStencilContainers, visMSDefault)
   Set vsoStencil = Application.Documents.OpenEx( _
      sContPath, Visio.visOpenHidden)
   Set vsoMaster = vsoStencil.Masters.Item(1)

'Container 1
   Set vsoCont = ThisDocument.Pages(1). _
      Drop(vsoMaster, 5, 5)
   vsoCont.Name = "Container_1"
   vsoCont.Text = "Container No. 1"
   Set vsoCont = Nothing

'Container 2
   Set vsoCont = ThisDocument.Pages(1). _
      DropContainer(vsoMaster, _
      ThisDocument.Pages(1).Shapes(1))
   vsoCont.Name = "Container_2"
   vsoCont.Text = "Container No. 2"
   Set vsoCont = Nothing
```

```
'Container 3
   Set vsoSel = Nothing
   Set vsoCont = ThisDocument.Pages(1). _
      DropContainer(vsoMaster, vsoSel)
   vsoCont.Name = "Container_3"
   vsoCont.Text = "Container No. 3"
   Set vsoCont = Nothing

'Shape und Container 4
   Set vsoShape = ActivePage.DrawRectangle( _
      m2i(100), m2i(50), m2i(160), m2i(80))
   Set vsoCont = ThisDocument.Pages(1). _
      DropContainer(vsoMaster, vsoShape)
   vsoCont.Name = "Container_4"
   vsoCont.Text = "Container No. 4"
   Set vsoCont = Nothing
   vsoStencil.Close

   Set vsoMaster = Nothing
   Set vsoStencil = Nothing
End Sub
```

Mit der Methode *Delete* können Container samt Inhalt wieder entfernt werden.

7.6.2 Container verwalten

Container verfügen nicht über eine Objektliste, sondern müssen über ihre ID angesprochen werden (Code 7.38).

Code 7.38 Die Prozedur nennt alle vorhandenen Container der ActivePage

```
Sub ReadContainer()
   Dim vsoApp     As Visio.Application
   Dim vsoPage    As Visio.Page
   Dim vsoShape   As Visio.Shape
   Dim vContID    As Variant

   Set vsoApp = Visio.Application
   Set vsoPage = vsoApp.ActivePage
   For Each vContID In _
     vsoPage.GetContainers(visContainerIncludeNested)
      Set vsoShape = vsoPage.Shapes.ItemFromID(vContID)
      Debug.Print vsoShape.Name; " : "; vsoShape.Text
   Next
```

```
   Set vsoShape = Nothing
   Set vsoPage = Nothing
   Set vsoApp = Nothing
End Sub
```

Die Prozedur (Code 7.39) bestimmt alle Shapes eines vorgegebenen Containers.

Code 7.39 Die Prozedur bestimmt alle Shapes eines Containers

```
Sub ReadContainerShapes()
   Dim vsoApp     As Visio.Application
   Dim vsoPage    As Visio.Page
   Dim vsoShape   As Visio.Shape
   Dim vsoCont    As Visio.Shape
   Dim vContID    As Variant

   Set vsoApp = Visio.Application
   Set vsoPage = vsoApp.ActivePage
   Set vsoCont = vsoPage.Shapes("Container_4")
   For Each vContID In vsoCont.ContainerProperties. _
         GetMemberShapes(visContainerFlagsDefault)
      Set vsoShape = vsoPage.Shapes.ItemFromID(vContID)
      Debug.Print vsoShape.Name; " : "; vsoShape.Text
   Next
   Set vsoShape = Nothing
   Set vsoCont = Nothing
   Set vsoPage = Nothing
   Set vsoApp = Nothing
End Sub
```

7.6.3 Eigenschaften und Methoden

Eine Eigenschaft der Formen ist *ContainerProperties*. Ist ein Container angewählt, dann lässt sich über diese Eigenschaft das Design eines Containers ändern (Code 7.40).

Code 7.40 Die Prozedur verändert das Design eines Containers

```
Sub ChangeContProperties()
   Dim vsoApp     As Visio.Application
   Dim vsoPage    As Visio.Page
   Dim vsoShape   As Visio.Shape
   Dim vsoCont    As Visio.Shape
   Dim vContID    As Variant
```

```
   Set vsoApp = Visio.Application
   Set vsoPage = vsoApp.ActivePage
   Set vsoCont = vsoPage.Shapes("Container_2")
   With vsoCont.ContainerProperties
      .ContainerStyle = 2
      .HeadingStyle = 2
      .SetMargin visMillimeters, 5
   End With
   Set vsoCont = Nothing
   Set vsoPage = Nothing
   Set vsoApp = Nothing
End Sub
```

Während die Methode *Delete* einen Container samt Inhalt löscht, entfernt die Methode *Disband* nur den Container (Code 7.41).

Code 7.41 Die Prozedur entfernt nur den Container

```
Sub RemoveCont()
   Dim vsoApp     As Visio.Application
   Dim vsoPage    As Visio.Page
   Dim vsoShape   As Visio.Shape
   Dim vsoCont    As Visio.Shape
   Dim vContID    As Variant

   Set vsoApp = Visio.Application
   Set vsoPage = vsoApp.ActivePage
   Set vsoCont = vsoPage.Shapes("Container_4")
   vsoCont.ContainerProperties.Disband
   Set vsoCont = Nothing
   Set vsoPage = Nothing
   Set vsoApp = Nothing
End Sub
```

Mit der Methode *AddMember* der Eigenschaft *ContainerProperties* lassen sich Formen auch nachträglich einem Container zuweisen. Die Prozedur (Code 7.42) erstellt zuerst einen Container und danach eine Form. Erst danach erfolgt die Zuweisung zum Container.

Code 7.42 Die Prozedur weist eine Form einem vorhandenen Container zu

```
Sub AddShapeToCont()
   Dim vsoStencil    As Visio.Document
   Dim vsoMaster     As Visio.master
   Dim vsoCont       As Visio.Shape
   Dim vsoShape      As Visio.Shape
   Dim sContPath     As String
```

```
'Schablone und Master
   sContPath = Application.GetBuiltInStencilFile( _
      visBuiltInStencilContainers, visMSDefault)
   Set vsoStencil = Application.Documents.OpenEx( _
      sContPath, Visio.visOpenHidden)
   Set vsoMaster = vsoStencil.Masters.Item(1)

'Container erstellen
   Set vsoCont = ThisDocument.Pages(2). _
      DropContainer(vsoMaster, vsoShape)
   vsoCont.Name = "Container_1"
   vsoCont.Text = "Container No. 1"
   vsoStencil.Close

'Shape hinzufügen
   Set vsoShape = ActivePage.DrawRectangle( _
      m2i(50), m2i(50), m2i(100), m2i(80))
   vsoCont.ContainerProperties.AddMember vsoShape, _
      visMemberAddExpandContainer
   Set vsoShape = Nothing
   Set vsoCont = Nothing
   Set vsoMaster = Nothing
   Set vsoStencil = Nothing
End Sub
```

7.7 Visio-Interaktionen

Visio wird aufgrund der vielen Vorlagen meist direkt genutzt. Doch es ergeben sich auch Interaktionen mit anderen Anwendungen, wenn diese bereits Aufstellungen zu Objekten verwalten.

7.7.1 Klassendiagramme mit Excel darstellen

In der Objektorientierten Programmierung (OOP) werden Klassen mit Eigenschaften, Methoden und Ereignissen zwar meist in eigenen Codecontainern geschrieben, zur Darstellung von Assoziationen aber benötigt man eine grafische Darstellung, das Klassendiagramm. Es liegt also nahe, die textliche Form einer Klasse automatisch in eine grafische umzuwandeln.

In diesem Beispiel wird für jede Klasse ein Excel-Arbeitsblatt erstellt (Abb. 7.9). Alle Klassen werden dann in Visio nach diesen Daten grafisch gezeichnet. Vergleichbar ist dies mit den Objekten Form und Formblatt.

	A	B	C	D	E
1	Typ	Scope	Name	Datentyp	Description
2	A	+	sKey	String	Personalschlüssel
3	A	+	sName	String	Name
4	A	+	sVorname	Integer	Vorname
5	A	+	sAbteilung	Integer	Abteilungsname
6	M	+	Property Let Key()	Key As String	
7	M	+	Property Let Name()	Name As String	
8	M	+	Property Get Key()	String	
9	M	+	Property Get Name()	String	
10	M	+	Sub Erfasse()	Key, Name, Vorname, Abteilung	
11	M	+	Sub Zeige()		

Cover clsAufgabe clsPerson clsPlanung

Abb. 7.9 Beispiel einer Klassenbeschreibung

Die Prozedur (Code 7.43) in einem Excel-Codemodul erstellt zuerst ein Visio-Dokument, diesmal mit später Bindung. Danach werden in einer *For-Each-Next*-Schleife alle Excel-Arbeitsblätter ausgewertet.

Code 7.43 Die Prozedur bestimmt die Ausgabedaten

```
Sub CreateClassDiagram()
   Dim objBook        As Object
   Dim objTab         As Object
   Dim lRow           As Long
   Dim lRowMax        As Long
   Dim sType          As String
   Dim sScope         As String
   Dim sName          As String
   Dim sData          As String
   Dim sAtt           As String
   Dim sMet           As String
   Dim sClass         As String
   Dim iAtt           As Integer
   Dim iMet           As Integer
   Dim iPos           As Integer

   Dim objVisio       As Object
   Dim PageName       As String
   Dim PageWidth      As String
   Dim PageHeight     As String
   Dim DrawX0         As Double   'Nullpunkt x
   Dim DrawY0         As Double   'Nullpunkt y
   Dim DrawXS         As Double   'Scalefaktor x
   Dim DrawYS         As Double   'Scalefaktor y
```

```
PageName = "Klassendiagramm"
PageWidth = "594 mm"          'Seitenbreite in mm
PageHeight = "420 mm"         'Seitenhöhe in mm
DrawX0 = 0.8
DrawY0 = 1.5
DrawXS = 1
DrawYS = 0.94

Set objVisio = CreateObject(Class:="Visio.Application")

With objVisio
  .documents.Add ("")
  .ActivePage.Name = "Overview"
  .ActivePage.Shapes("thePage").Cells("PageWidth").FormulaU = _
     PageWidth
  .ActivePage.Shapes("thePage").Cells("PageHeight").FormulaU = _
     PageHeight
  .ActivePage.Shapes("thePage").Cells("DrawingSizeType").FormulaU _
     = "5"
End With

Set objBook = ThisWorkbook
iPos = 0
For Each objTab In objBook.Worksheets
   If Not objTab.Name = "Cover" Then
      sAtt = ""
      sMet = ""
      iAtt = 0
      iMet = 0
      sClass = objTab.Name
      lRowMax = objTab.UsedRange.Rows.Count
      For lRow = 2 To lRowMax
         sType = objTab.Cells(lRow, 1)
         sScope = objTab.Cells(lRow, 2)
         sName = objTab.Cells(lRow, 3)
         sData = objTab.Cells(lRow, 4)
         Select Case sType
         Case "A"
            iAtt = iAtt + 1
            If sAtt = "" Then
               sAtt = sScope & " " & sName
            Else
               sAtt = sAtt & vbLf & sScope & " " & sName
            End If
            sAtt = sAtt & " (" & sData & ")"
```

```
            Case "M"
               iMet = iMet + 1
               If sMet = "" Then
                  sMet = sScope & " " & sName
               Else
                  sMet = sMet & vbLf & sScope & " " & sName
               End If
               sMet = sMet & " (" & sData & ")"
            End Select
         Next lRow
         iPos = iPos + 1
         Call DrawRec(iPos, objVisio, sClass, iAtt, sAtt, iMet, sMet)
      End If
   Next
   Set objTab = Nothing
   Set objBook = Nothing
   Set objVisio = Nothing
End Sub
```

Je Blatt bzw. Klasse werden die Attribute und Methoden als Text gesammelt, um danach deren Umfang bestimmen zu können. Die Prozedur (Code 7.44) erstellt aus diesen Texten die Klasse in grafischer Form (Abb. 7.10).

Abb. 7.10 Die generierten Klassen

Code 7.44 Die Prozedur zeichnet eine Klasse

```
Private Sub DrawRec(iPos, objVisio, sClass, iAtt, sAtt, iMet, sMet)
   Dim objShape1 As Object
   Dim objShape2 As Object
   Dim objShape3 As Object

   Set objShape1 = objVisio.ActiveWindow.Page.DrawRectangle(0, 10, _
      3, 9.5)
   With objShape1
      .Text = sClass
      .Cells("Para.HorzAlign").ResultIUForce = "1"
      .Cells("FillForegnd").FormulaForceU = 2
      .Cells("LineColor").FormulaU = 0
      .Cells("Char.Color[1]").Formula = 0
      .Cells("Char.Style").Formula = 17#
      .Cells("Char.Size").ResultIU = 0.2
   End With
   Set objShape2 = objVisio.ActiveWindow.Page.DrawRectangle(0, 9.5, _
      3, 9.5 - (iAtt + 2) * 0.25)
   With objShape2
      .Text = sAtt
      .Cells("Para.HorzAlign").ResultIUForce = "0"
      .Cells("FillForegnd").FormulaForceU = 1
      .Cells("LineColor").FormulaU = 0
      .Cells("Char.Color[1]").Formula = 0
      .Cells("Char.Style").Formula = 17#
      .Cells("Char.Size").ResultIU = 0.2
   End With
   Set objShape3 = objVisio.ActiveWindow.Page.DrawRectangle(0, 9.5 - _
     (iAtt + 2) * 0.25, 3, 9.5 - (iAtt + 2) * 0.25 - (iMet + 2) * 0.3)
   With objShape3
      .Text = sMet
      .Cells("Para.HorzAlign").ResultIUForce = "0"
      .Cells("FillForegnd").FormulaForceU = 1
      .Cells("LineColor").FormulaU = 0
      .Cells("Char.Color[1]").Formula = 0
      .Cells("Char.Style").Formula = 17#
      .Cells("Char.Size").ResultIU = 0.2
   End With
```

```
    objVisio.ActiveWindow.Select objShape1, 2
    objVisio.ActiveWindow.Select objShape2, 2
    objVisio.ActiveWindow.Select objShape3, 2
    objVisio.ActiveWindow.Group
    objVisio.ActiveWindow.Selection.Move 0 + (iPos - 1) * 3, 5

    Set objShape1 = Nothing
    Set objShape2 = Nothing
    Set objShape3 = Nothing
End Sub
```

Durch die Klarheit der Viso-Objektstruktur und den beschriebenen Elementen dürfte es bei der Programmierung hier die wenigsten Probleme geben. Daher verzichte ich auf weitere Anwendungsbeispiele, um für das letzte Kapitel *Project* genügend viele Seiten zu haben.

Project-Objekte

8

Project ist ein wichtiges Hilfsmittel des Projektmanagements auf der Grundlage der Netzplantechnik. Es hilft bei der Planung, Verwaltung und dem Controlling von Projekten. Die wichtigsten Parameter eines Projekts sind Kosten, Ressourcen und Zeit. Betrachtet werden Vorgänge, die Einzelaufgaben, Arbeitspakete oder gar Unterprojekte sein können.

8.1 Project-Anwendungen

Das Anwendungs-Objekt *Application* ist das oberste Objekt der Project-Objekt-Hierarchie (Abb. 8.1). Die Struktur ist so umfangreich, so dass ich hier nur die bedeutendsten Unterobjekte wiedergebe. In der VBA-Hilfe werden unter *Elemente des Application-Objekts* alle Methoden, Eigenschaften und Ereignisse aufgeführt.

Auch bei Project kann eine Anwendung gleichzeitig mehrere Projekte geöffnet verwalten. Die Prozedur (Code 8.1) bestimmt die Anzahl geöffneter Projekte und gibt deren Namen im Direktfenster aus.

Code 8.1 Die Prozedur bestimmt alle geöffneten Projekte der Anwendung

```
Sub ReadAllProjects()
   Dim objPro As Project

   Debug.Print Application.Projects.Count
   For Each objPro In Application.Projects
      Debug.Print objPro.Name
   Next
End Sub
```

Im *ProjectGlobal* (Global.MPT) abgelegte Prozeduren gelten für alle Anwendungen, während die Prozeduren im einem VBAProject nur für dieses Gültigkeit haben.

H. Nahrstedt, *Die Welt der VBA-Objekte*, DOI 10.1007/978-3-658-13891-2_8

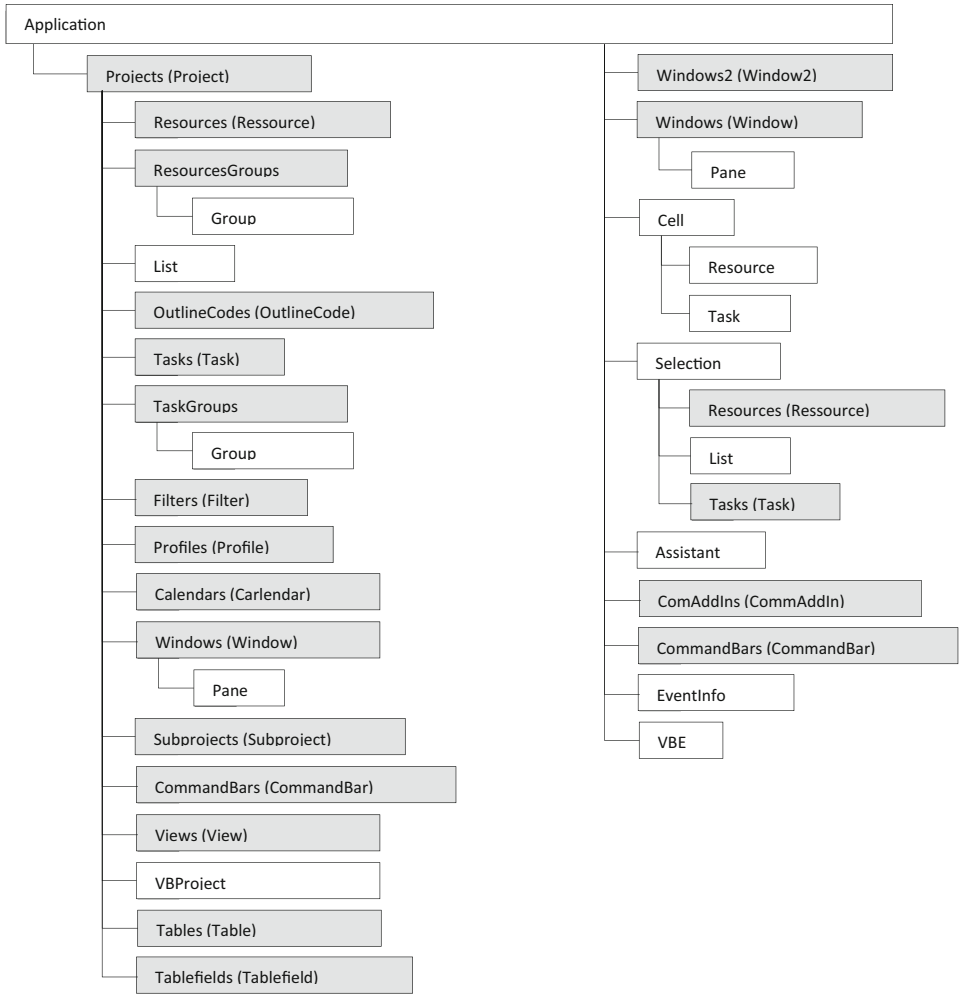


Abb. 8.1 Die wichtigsten Unterobjekte und Objektlisten (*grau*) des Application-Objekts

8.1.1 Eigenschaften

Eigenschaften sind erst mit dem Objekt *Project* wirklich interessant. Doch es gibt auch zum Anwendungs-Objekt ein paar Informationen, die Prozedur (Code 8.2) ausliest.

Code 8.2 Die Prozedur gibt einige Eigenschaften der aktuellen Anwendung aus

```
Sub ReadApplicationInfos()
   With Application
      Debug.Print "Obere Pos...: "; .Top
      Debug.Print "Linke Pos...: "; .Left
      Debug.Print "Breite.....: "; .Width
      Debug.Print "Höhe.......: "; .Height
      Debug.Print "Version....: "; .Version
      Debug.Print "Pfad.......: "; .Path
      Debug.Print "Startansicht: "; .DefaultView
      Debug.Print "Kopfzeile...: "; .Caption
   End With
End Sub
```

Die Eigenschaft *DefaultView* bestimmt die Ansicht, die beim Start von Project angezeigt wird. Der Defaultwert ist *Gantt mit Zeitvorgaben*. In der VBA-Hilfe werden unter *Application.DefaultView* weitere Möglichkeiten gezeigt.

8.1.2 Methoden

Die Methode *About* gibt Informationen zur Anwendung (Code 8.3) in einem Dialogfenster aus. Neben der Version werden Angaben zur Lizensierung und zum Urheberrecht angezeigt. Über Schaltflächen können weitere Systeminformationen und Daten zum Software-Service aufgerufen werden.

Code 8.3 Die Prozedur verwendet die Methode About zur Informationsausgabe

```
Sub ReadProjectInfo()
   Application.About
End Sub
```

Mit der Methode *AppExecute* können andere Anwendungen gestartet werden. Die Prozedur (Code 8.4) startet nacheinander den Texteditor und eine Excel-Anwendung. Dann wird das Dialogfenster zur Autokorrektur mit der Methode *Autocorrect* geöffnet. Die Methode *EditCopyPicture* kopiert das aktuelle Fenster als GIF-Datei in das vorgegebene Verzeichnis. Die Methode *CalculateAll* führt alle Berechnungen erneut durch.

Code 8.4 Die Prozedur startet verschiedene andere Anwendungen

```
Sub StartApp()
   With Application
      .AppExecute Command:="Notepad.exe"
      .AppExecute Command:="Excel.exe"
      .AutoCorrect
      .EditCopyPicture forprinter:=pjGIF, FileName:="C:\Temp\gif"
      .CalculateAll
   End With
End Sub
```

Die Methode *FileCloseEx* schließt das aktuelle Projekt, während die Methode *FileCloseAllEx* alle geöffneten Projekte schließt. Die Methode *FileExit* beendet die Project-Anwendung.

Das Studium weiterer Methoden ist empfehlenswert und liefert neue Erkenntnisse. Doch auch viele Methoden sind erst mit der Kenntnis des Projekt-Objekts und dessen Unterobjekten wirklich interessant. So gibt es Methoden für die Darstellung des Ganttdiagramms, für die Behandlung der eingetragenen Vorgänge und viele andere Designänderungen.

8.1.3 Ereignisse

Das Anwendungs-Objekt bringt auch eine Vielzahl vor Ereignisprozeduren mit. Aber auch hier sind die meisten erst in einem Projekt sinnvoll anwendbar. Mit der Definition einer Eventvariablen im Klassenmodul *ThisProject* (Code 8.5) werden auch die Ereignisprozeduren sichtbar (Abb. 8.2). Das Ereignis *ApplicationBeforeClose* ist ja von anderen Anwendungen bereits bekannt.

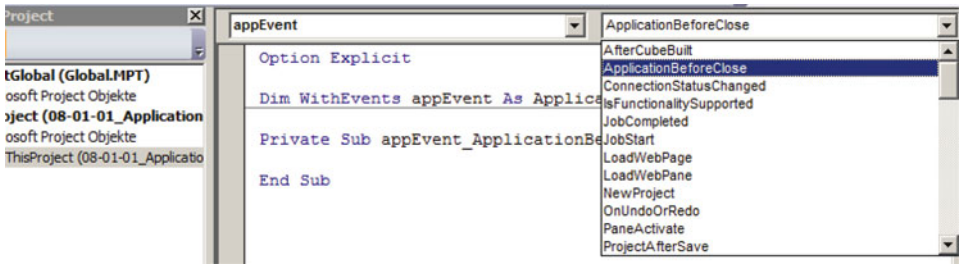


Abb. 8.2 Auswahl der Anwendungs-Ereignisprozeduren

Code 8.5 Definition eines Ereignis-Objekts

```
Option Explicit
Dim WithEvents appEvent As Application
Private Sub appEvent_ApplicationBeforeClose(ByVal Info As EventInfo)
End Sub
```

Viele Ereignisse betreffen die Unterobjekte *Project* und *Window* und werden daher in den nächsten Kapiteln behandelt.

8.1.4 ActiveWindow

Das Fenster-Objekt *ActiveWindow* ist ein Unterobjekt des Objekts *VBE*, das Stammobjekt für alle Objekte und Objektlisten in VBA. Das Objekt *ActiveWindows* wird an dieser Stelle erwähnt, weil es das Fenster unter allen geöffneten ist, auf dem der Focus liegt.

Die Prozedur (Code 8.6) erlaubt die Veränderung der Eigenschaft *Caption* (Text in der Kopfzeile) über einen Dialog mit dem Anwender.

Code 8.6 Die Prozedur ändert den Kopftext des aktiven Fensters

```
Sub ChangeWindowCaption()
   Dim sText As String

   sText = InputBox("Eingabe eines neuen Fenstertextes" & _
      vbCrLf & "für das aktuelle Projekt-Fenster." & vbCrLf & _
      "Nur OK oder Abbrechen " & vbCrLf & _
      "lässt den alten Text bestehen")
   If sText = Empty Then
      Exit Sub
   Else
      ActiveWindow.Caption = sText
   End If
End Sub
```

8.2 Project-Projekte

Das Projekt-Objekt *Project* ist das Arbeitsobjekt von Project, wie sich auch unschwer aus dem Objektmodell ablesen lässt. Bereits im Abschn. 8.1 wurden die geöffneten Projekte der aktuellen Projekt-Anwendung bestimmt. Mit *Projects* handelt es sich um eine Objektliste, so dass die einzelnen Objekte mit der *For-Each-Next*-Schleife oder durch ihren *Index* ansprechbar sind. Eine Besonderheit stellt das Objekt *ActiveProject* dar, es ist das aktuelle Projekt in der aktuellen Anwendung.

Die Prozedur (Code 8.7) öffnet neben dem aktiven Projekt ein zweites.

Code 8.7 Die Prozedur öffnet ein vorhandenes Projekt

```
Sub UsedActiveProjects()
   Dim proDemo1 As Project
   Dim proDemo2 As Project

   Set proDemo1 = ActiveProject
   Application.FileOpenEx "C:\Temp\Test.mpp"
   Set proDemo2 = ActiveProject
   Debug.Print "Demo1: "; proDemo1.Name
   Debug.Print "Demo2: "; proDemo2.Name
   Application.FileCloseEx (pjDoNotSave)
   proDemo1.Activate
   Set proDemo2 = Nothing
   Set proDemo1 = Nothing
End Sub
```

In der VBA-Hilfe unter *Elemente des Project-Objekts* werden alle Eigenschaften, Methoden und Ereignisse aufgeführt.

8.2.1 Eigenschaften

Aus der Vielzahl von Eigenschaften nutzt die Prozedur (Code 8.8) einige Projekt-Informationen des aktiven Projekts.

Code 8.8 Die Prozedur zeigt einige Projekt-Informationen im Direktfenster

```
Sub ActiveProjectInfo()
   With ActiveProject
      Debug.Print "Name.........: "; .Name
      Debug.Print "Projekttyp...: "; .Type
      Debug.Print "Pfad.........: "; .Path
      Debug.Print "CodeName.....: "; .CodeName
      Debug.Print "Erstelldatum.: "; .CreationDate
      Debug.Print "Startdatum...: "; .ProjectStart
      Debug.Print "Startzeit....: "; .DefaultStartTime
      Debug.Print "Endedatum....: "; .ProjectFinish
      Debug.Print "Endezeit.....: "; .DefaultFinishTime
      Debug.Print "h/Tag........: "; .HoursPerDay
      Debug.Print "h/Woche......: "; .HoursPerWeek
      Debug.Print "Tage/Monat...: "; .DaysPerMonth
      Debug.Print "Akt.Datum....: "; .CurrentDate
      Debug.Print "Akt.Tabelle..: "; .CurrentTable
      Debug.Print "Akt.Ansicht..: "; .CurrentView
      Debug.Print "Anz.Aufgaben.: "; .NumberOfTasks
      Debug.Print "Vorlage......: "; .Template
      Debug.Print "Notizen......: "; .ProjectNotes
   End With
End Sub
```

8.2.2 Methoden

Mit der Methode *Activate* wird ein geöffnetes Projekt zum Objekt *ActiveProject*. Die Methode *Add* der Auflistung *Projects* des Anwendungs-Objekts erzeugt ein weiteres leeres Projekt, wenn nicht mit dem Parameter *Template* eine Vorlage vorgegeben wird (Code 8.9).

Code 8.9 Die Prozedur nutzt die Methode Activate

```
Sub CreateNewProject()
   Dim proDemo1 As Project
   Dim proDemo2 As Project

   Set proDemo1 = ActiveProject
   Set proDemo2 = Application.Projects.Add
   ActiveProject.SaveAs Name:="C:\Temp\Demo2.mpp"
   proDemo1.Activate
   Debug.Print "Demo1: "; proDemo1.Name
   Debug.Print "Demo2: "; proDemo2.Name
```

```
   Set proDemo2 = Nothing
   Set proDemo1 = Nothing
End Sub
```

8.2.3 Ereignisse

Im Objekt *ThisProject* (Abb. 8.3) gibt es zum *Project*-Objekt einige Ereignisse. Darunter die aus anderen Anwendungen bekannten *Open* und *BeforeClose*. Mit ihnen können wieder eigene Einträge im Menüband gestaltet werden.

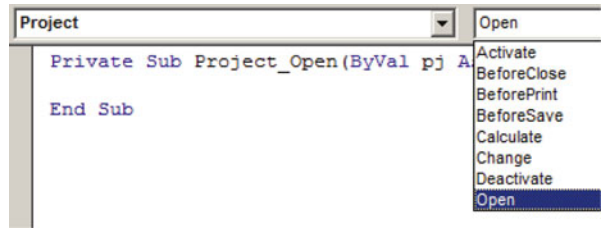


Abb. 8.3 Auswahlmöglichkeit der Project-Ereignisse

Ein sehr wichtiges Ereignis des Projekt-Objekts ist *Change*. Die Ereignisprozedur (Code 8.10) erlaubt die Reaktion auf Veränderungen im Projekt.

Code 8.10 Die Ereignisprozedur registriert lokale Veränderungen

```
Private Sub Project_Change(ByVal pj As Project)
   MsgBox "Change_local"
End Sub
```

8.2.4 Projektmanagement- und Projekt-Handbuch

Zum Start eines Projekts sind die Erstellung eines Projektmanagement- und Projekt-Handbuches unerlässlich. Sie liegen zur einheitlichen Handhabung in einem Unternehmen oft als Excel-Liste in numerischer Form vor. Daraus können dann Dokumentationen und Aufgaben generiert werden.

Die Themen eines Projektmanagement-Handbuches sind:

- Einleitende Erklärungen,
- Grundlagen wie Begriffe, Normen und Organisation,
- Verantwortlichkeiten der Stakeholder,
- Projektablauf mit Planung, Realisierung und Abschluss,
- Methoden mit Lastenheft, Risikoanalyse, Strukturplan, Kick-off, Entscheidungen,
- Hilfsmittel wie Projektantrag, Vorgangsliste, Terminplan, Kapazitätsplan, Kostenplan,
- Berichts- und Dokumentationswesen.

Die Themen eines Projekt-Handbuches sind:

- Projektziele,
- Terminziele,
- Kostenziele,
- Verantwortlichkeiten des Projektleiters und der Teammitglieder,
- Externe Teammitglieder,
- Normen, Richtlinien und Standards,
- Projektablauf,
- Projektreporting.

Im nachfolgenden Kapitel werden vereinfachte Excellisten zur Generierung von Projektdaten verwendet.

8.3 Project-Einzelobjekte

8.3.1 Vorgänge

Das Vorgangs-Objekt *Task* beschreibt einen Vorgang im Projektplan. Eigentlich heißt Task wörtlich übersetzt Aufgabe, doch in der deutschen Version von Project wird der Begriff Vorgang verwendet. Alle Vorgänge eines Projekts werden in der Objektliste *Tasks* verwaltet. Die Prozedur (Code 8.11) erstellt ein neues Projekt mit dem Beispiel *Netzplan* aus Abschn. 1.5.6 und überträgt die Daten aus einem Excel-Arbeitsblatt (Abb. 8.4) in ein leeres Projekt.

Abb. 8.4 Projektvorlage in Excel

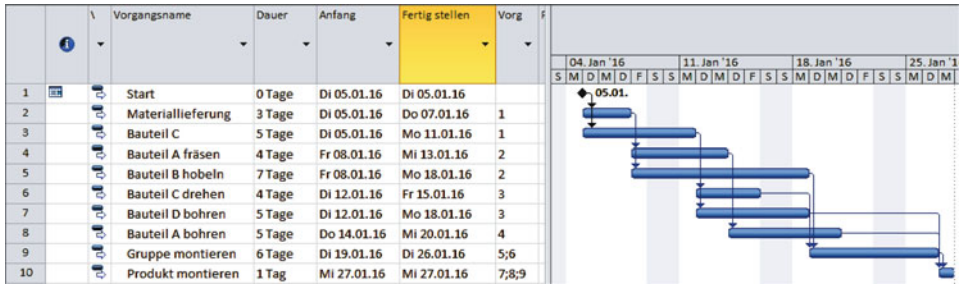
	A	B	C
1	Vorgangsname	Dauer	Vorgänger
2	Start	0t	
3	Materiallieferung	3t	1
4	Bauteil C	5t	1
5	Bauteil A fräsen	4t	2
6	Bauteil B hobeln	7t	2
7	Bauteil C drehen	4t	3
8	Bauteil D bohren	5t	3
9	Bauteil A bohren	5t	4
10	Gruppe montieren	6t	5;6
11	Produkt montieren	1t	7;8;9
12			

Tasks

Code 8.11 Die Prozedur erstellt ein neues Projekt

```
Sub ImportProjectExample()
   Dim exlApp  As Excel.Application
   Dim wbkPlan As Excel.Workbook
   Dim wshPlan As Excel.Worksheet
   Dim lRow    As Long
   Dim lRowMax As Long
   Dim proNetz As Project
   Dim proTask As Task

   Set exlApp = Excel.Application
   Set wbkPlan = exlApp.Workbooks.Open _
      (FileName:="C:\Temp\Tasks.xlsx")
   Set wshPlan = wbkPlan.Worksheets("Tasks")
   lRowMax = wshPlan.UsedRange.Rows.Count
   Set proNetz = Application.Projects.Add

   For lRow = 2 To lRowMax
      Set proTask = proNetz.Tasks.Add
      With proTask
         .Name = wshPlan.Cells(lRow, 1)
         .Duration = Trim(wshPlan.Cells(lRow, 2)) & "t"
         .Predecessors = wshPlan.Cells(lRow, 3)
         .Manual = False
         If lRow = 2 Then
            .Start = Date + 7
         End If
      End With
      Set proTask = Nothing
   Next lRow
   wbkPlan.Close
   ActiveProject.SaveAs Name:="C:\Temp\NetzPlan.mpp"
   Set proNetz = Nothing
   Set exlApp = Nothing
   Set wbkPlan = Nothing
   Set wshPlan = Nothing
End Sub
```

Das Resultat der Übertragung (Abb. 8.5) nach Project ist eine Standardansicht mit der Vorgangstabelle im linken Teil und dem Gantt-Diagramm im rechten Teil des Bildschirms. Durch die Zuordnung *Manual* = *false* werden alle Vorgänge mit den Vorgängern automatisch verbunden, beginnend ab dem aktuellen Datum + 7 Tage.



	Vorgangsname	Dauer	Anfang	Fertig stellen	Vorg
1	Start	0 Tage	Di 05.01.16	Di 05.01.16	
2	Materiallieferung	3 Tage	Di 05.01.16	Do 07.01.16	1
3	Bauteil C	5 Tage	Di 05.01.16	Mo 11.01.16	1
4	Bauteil A fräsen	4 Tage	Fr 08.01.16	Mi 13.01.16	2
5	Bauteil B hobeln	7 Tage	Fr 08.01.16	Mo 18.01.16	2
6	Bauteil C drehen	4 Tage	Di 12.01.16	Fr 15.01.16	3
7	Bauteil D bohren	5 Tage	Di 12.01.16	Mo 18.01.16	3
8	Bauteil A bohren	5 Tage	Do 14.01.16	Mi 20.01.16	4
9	Gruppe montieren	6 Tage	Di 19.01.16	Di 26.01.16	5;6
10	Produkt montieren	1 Tag	Mi 27.01.16	Mi 27.01.16	7;8;9

Abb. 8.5 Vorgänge mit automatischer Verknüpfung im Gantt-Diagramm

Über die Methode *ViewApplyEx* des Application-Objekts können die Ansichten geändert werden. Die Prozedur (Code 8.12) schaltet auf ein Gantt-Diagramm mit Angabe des *Kritischen Pfades* um (Abb. 8.6). Als Kritischer Pfad wird die Folge von Vorgängen bezeichnet, bei der jede zeitliche Änderung Auswirkungen auf den Fertigstellungstermin hat.

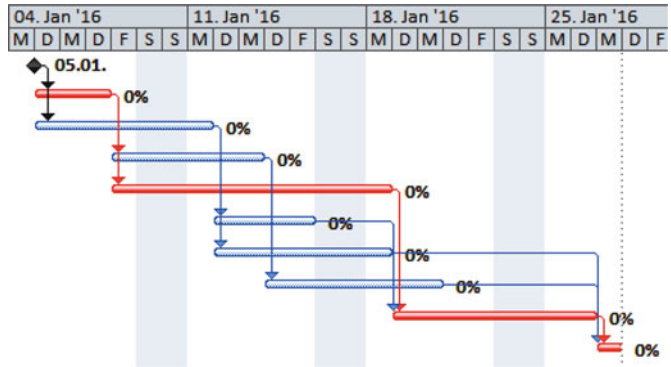


Abb. 8.6 Darstellung des Kritischen Pfades im Gantt-Diagramm

Code 8.12 Die Prozedur zeigt das Gantt-Diagramm mit Kritischem Pfad

```
Sub ViewCritivalPath()
   Application.ViewApplyEx _
      Name:="Gantt-&Diagramm: Überwachung", _
      ApplyTo:=0
End Sub
```

Die Prozedur (Code 8.13) zeigt wieder die Standardansicht des Gantt-Diagramms.

Code 8.13 Die Prozedur zeigt die Standardansicht des Gantt-Diagramms

```
Sub ViewGanttDia()
   Application.ViewApplyEx _
      Name:="&Balkendiagramm (Gantt)", _
      ApplyTo:=0
End Sub
```

Eine weitere Darstellungsmöglichkeit ist der bereits bekannte Netzplan (Abb. 8.7), bei dem die Vorgänge als Knoten mit ihren Informationen dargestellt werden.

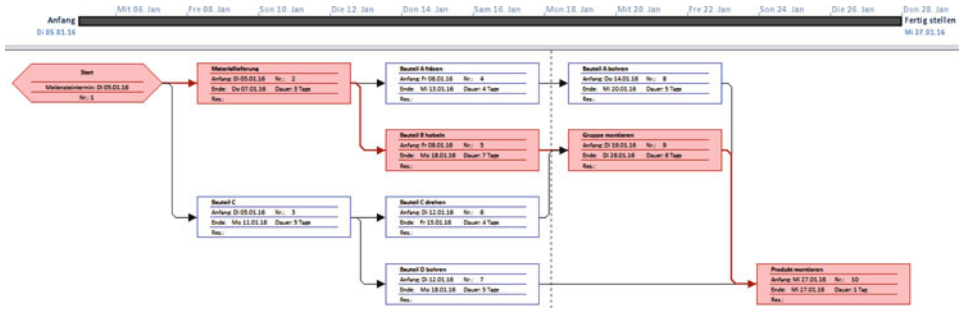


Abb. 8.7 Darstellung des Netzplans zum Beispiel

Die Prozedur (Code 8.14) erzeugt mit der Methode *ViewApply* des Application-Objekts den Netzplan.

Code 8.14 Die Prozedur zeigt den Netzplan eines Projekts

```
Sub ViewNetzPlan()
   Application.ViewApply _
      Name:="Netzplandiagramm"
End Sub
```

Vorgänge werden mit der Methode *InsertTask* eingefügt und mit der Methode *EditDelete* gelöscht. Zuvor muss eventuell die entsprechende Zeile mit der Methode *SelectRow* markiert werden. Die Prozedur (Code 8.15) erzeugt nach der vierten Zeile einen weiteren Vorgang zum Bauteil A mit dem Namen *Bauteil A schleifen*, der im Anschluss an den Vorgang *Bauteil A fräsen* erfolgt. Der Vorgang *Bauteil A bohren* kann erst nach *Bauteil A schleifen* erfolgen.

Code 8.15 Die Prozedur führt Änderungen im Projektplan durch

```
Sub InsertNewTasks()
   With Application
      .SelectRow Row:=5, RowRelative:=False
      .InsertTask
      .SetTaskField Field:="Name", Value:="Bauteil A schleifen"
      .SelectTaskField Row:=0, Column:="Vorgangsmodus"
      .SetTaskField Field:="Vorgangsmodus", Value:="Nein"
      .SelectTaskField Row:=0, Column:="Dauer"
      .SetTaskField Field:="Dauer", Value:="2"
      .SelectTaskField Row:=0, Column:="Vorgänger"
      .SetTaskField Field:="Vorgänger", Value:="4"
      .SelectRow Row:=9, RowRelative:=False
      .SetTaskField Field:="Vorgänger", Value:="5"
      .SelectTaskField Row:=1, Column:="Vorgänger"
      .SetTaskField Field:="Vorgänger", Value:="6;7"
      .SelectTaskField Row:=1, Column:="Vorgänger"
      .SetTaskField Field:="Vorgänger", Value:="8;9;10"
   End With
End Sub
```

Der neu eingefügte Vorgang (Abb. 8.8) wird im Gantt-Diagramm farblich markiert.

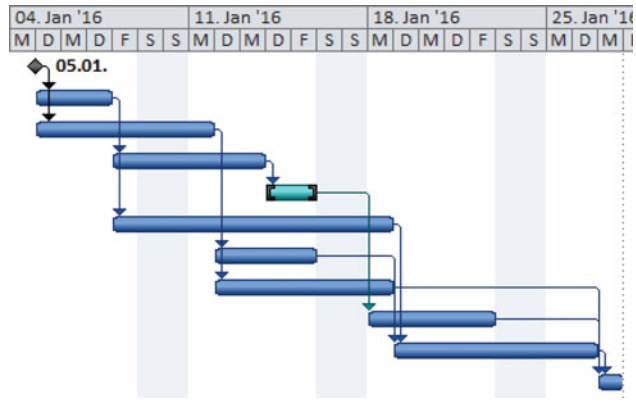
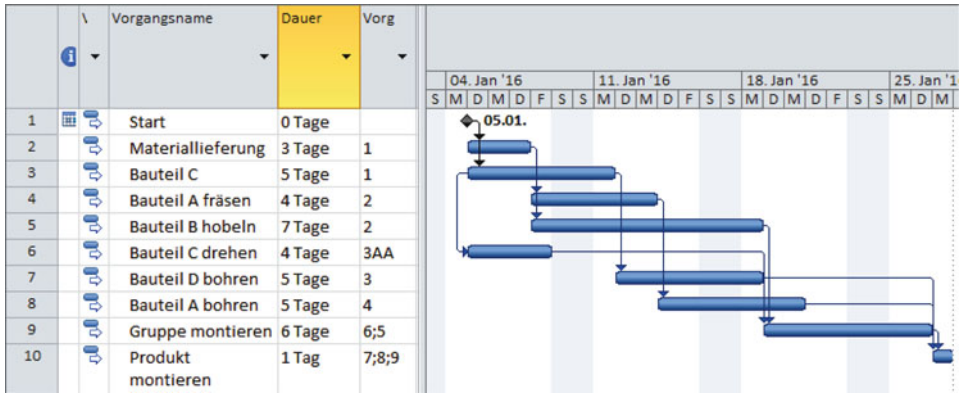


Abb. 8.8 Markierte Änderung im Gantt-Diagramm

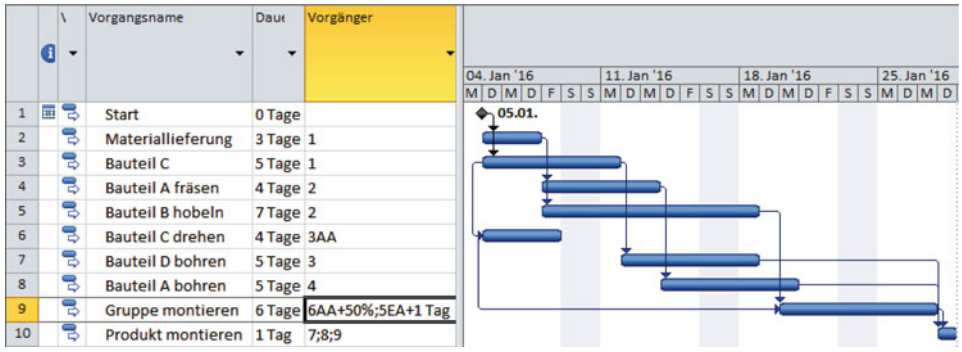
Neben der Angabe des Vorgängers oder der Vorgänger kann zusätzlich eine von vier Verknüpfungsarten (Anhang 8, Tab. A8.1) angegeben werden. Mit der Angabe *3AA* für den Vorgang 7 mit dem Vorgänger 3, starten beide Vorgänge zur gleichen Zeit (Abb. 8.9).



	Vorgangsname	Dauer	Vorg
1	Start	0 Tage	
2	Materiallieferung	3 Tage	1
3	Bauteil C	5 Tage	1
4	Bauteil A fräsen	4 Tage	2
5	Bauteil B hobeln	7 Tage	2
6	Bauteil C drehen	4 Tage	3AA
7	Bauteil D bohren	5 Tage	3
8	Bauteil A bohren	5 Tage	4
9	Gruppe montieren	6 Tage	6;5
10	Produkt montieren	1 Tag	7;8;9

Abb. 8.9 Veränderungen im Projektplan

Neben der Verknüpfungsart können die Vorgänger auch mit Statuswerten und Zeitangaben versehen werden. In Zeile 9 (Abb. 8.10) hat der Vorgang zwei Vorgänger. Der erste Vorgänger ist mit 6AA+50 % angegeben, so dass der Vorgang 9 erst beginnen kann wenn 50 % vom Vorgang 6 bereits abgeschlossen ist. Der zweite Vorgänger ist mit 5EA+1t angegeben, so dass der Vorgang 9 ebenso erst beginnen kann, wenn der Vorgang 5 bereits 1 Tag abgeschlossen ist. Sind beide Bedingungen erfüllt, kann der Vorgang 9 starten.



	Vorgangsname	Dauc	Vorgänger
1	Start	0 Tage	
2	Materiallieferung	3 Tage	1
3	Bauteil C	5 Tage	1
4	Bauteil A fräsen	4 Tage	2
5	Bauteil B hobeln	7 Tage	2
6	Bauteil C drehen	4 Tage	3AA
7	Bauteil D bohren	5 Tage	3
8	Bauteil A bohren	5 Tage	4
9	Gruppe montieren	6 Tage	6AA+50%;5EA+1 Tag
10	Produkt montieren	1 Tag	7;8;9

Abb. 8.10 Beispiel einer Verknüpfungsart

Alle diese Einträge lassen sich mit auch mit der Methode *SetTaskField* vornehmen.

Milestones (Meilensteinen, Vorgänge mit Dauer = 0) werden an wichtigen Positionen eines Projekts gesetzt. Im Beispiel ist der Vorgang *Start* (Raute) ein solcher Meilenstein. Eine andere Kontrollmöglichkeit sind Stichtage zur Fortschrittsüberwachung.

Die Prozedur (Code 8.16) generiert einen Stichtag für den Vorgang 8 und setzt den Termin 3 Tage vor Beginn des Vorgangs.

Code 8.16 Die Prozedur erzeugt einen Stichtag

```
Sub CreateStichtag()
   Dim dStart As Date
   dStart = ActiveProject.Tasks(8).Start - 3
   With Application
      .SelectRow Row:=8, RowRelative:=False
      .SetTaskField Field:="Stichtag", _
         Value:=dStart, AllSelectedTasks:=True
   End With
End Sub
```

8.3.2 Ressourcen

Project unterscheidet drei Arten des Ressourcen-Objekts *Resources*:

- Arbeit,
- Material,
- Kosten.

Zur Ressource Arbeit zählen Personen und Geräte, die zur Durchführung eines Vorgangs benötigt werden. Aufgewendet wird Zeit.

Zur Ressource Material zählen Verbrauchsmaterialien und Hilfsmittel, die für einen Vorgang erforderlich sind. Auch die entsprechenden Lagerbestände. Aufgewendet wird Material.

Zur Ressource Kosten zählen die für einen Vorgang anfallenden Kosten, unabhängig von der anfallenden Zeit. Aufgewendet wird ein Betrag.

Erfasst werden die Ressourcen in einer Tabelle. Die Prozedur (Code 8.17) übernimmt die Einträge aus einer Excel-Tabelle (Abb. 8.11).

Abb. 8.11 Excel-Vorlage für Ressourcen

	A	B	C	D	E
1	Name	Art	Max.Einh.	StandardSatz	ÜberstundenSatz
2	Einkäufer	Arbeit	80%	60,00 €/Std.	80,00 €/Std.
3	Mechaniker	Arbeit	60%	40,00 €/Std.	60,00 €/Std.
4	Monteur	Arbeit	100%	70,00 €/Std.	90,00 €/Std.
5	Projektleiter	Arbeit	100%	80,00 €/Std.	100,00 €/Std.
6	Fräsmaschine	Arbeit	100%	65,00 €/Std.	
7	Hobelbank	Arbeit	100%	64,00 €/Std.	
8	Drehbank	Arbeit	100%	66,00 €/Std.	
9	Bohrmaschine	Arbeit	100%	62,00 €/Std.	
10	Kühlmittel	Material		12,00 €/Std.	
11	Schmiermittel	Material		14,00 €/Std.	
12	Büromaterial	Kosten			

Code 8.17 Die Prozedur importiert aus einem Excel-Sheet Ressourcen-Daten

```
Sub ImportResourcenExample()
   Dim exlApp  As Excel.Application
   Dim wbkPlan As Excel.Workbook
   Dim wshPlan As Excel.Worksheet
   Dim lRow    As Long
   Dim lRowMax As Long
   Dim proNetz As Project
   Dim proRes  As Resource
   Dim dValue  As Double

   Set exlApp = Excel.Application
   Set wbkPlan = exlApp.Workbooks.Open _
      (FileName:="C:\Temp\Resourcen.xlsx")
   Set wshPlan = wbkPlan.Worksheets("Resourcen")
   lRowMax = wshPlan.UsedRange.Rows.Count
   Set proNetz = Application.ActiveProject

   For lRow = 2 To lRowMax
      Set proRes = proNetz.Resources.Add
      With proRes
         .Name = wshPlan.Cells(lRow, 1)
         Select Case wshPlan.Cells(lRow, 2)
         Case "Arbeit"
            .Type = pjResourceTypeWork
         Case "Material"
            .Type = pjResourceTypeMaterial
         Case "Kosten"
            .Type = pjResourceTypeCost
         Case Else
            .Type = ""
         End Select
         dValue = CDbl(wshPlan.Cells(lRow, 3))
         If dValue > 0 Then .MaxUnits = dValue
         dValue = CDbl(wshPlan.Cells(lRow, 4))
         If dValue > 0 Then .StandardRate = dValue
         dValue = CDbl(wshPlan.Cells(lRow, 5))
         If dValue > 0 Then .OvertimeRate = dValue
         dValue = CDbl(wshPlan.Cells(lRow, 6))
         If dValue > 0 Then .CostPerUse = dValue
      End With
      Set proRes = Nothing
   Next lRow
   wbkPlan.Close
   ActiveProject.SaveAs Name:="C:\Temp\NetzPlan.mpp"
```

```
    Set proNetz = Nothing
    Set exlApp = Nothing
    Set wbkPlan = Nothing
    Set wshPlan = Nothing
End Sub
```

Die Zuordnung der Ressourcen zu den Vorgängen erfolgt in der Vorgangs-Tabelle und übernimmt Prozedur (Code 8.18). Die Einträge erfolgen in der Spalte Ressorcennamen (Abb. 8.12).

	Ressourcennamen
1	
2	Projektleiter[2%];Einkäufer[60%];Büromaterial
3	Einkäufer[60%];Projektleiter[2%];Büromaterial
4	Kühlmittel[1];Projektleiter[5%];Mechaniker[80%];Fräsmaschine;Büromaterial
5	Kühlmittel[1];Projektleiter[5%];Mechaniker[80%];Hobelbank;Büromaterial
6	Kühlmittel[1];Projektleiter[5%];Mechaniker[80%];Drehbank;Büromaterial
7	Kühlmittel[1];Projektleiter[5%];Mechaniker[80%];Bohrmaschine;Büromaterial
8	Kühlmittel[1];Projektleiter[5%];Mechaniker[80%];Bohrmaschine;Büromaterial
9	Projektleiter[1%];Monteur;Schmiermittel[1];Büromaterial
10	Projektleiter[1%];Monteur;Schmiermittel[1];Büromaterial

Abb. 8.12 Die erfolgte Zuordnung im Beispiel

Code 8.18 Die Prozedur ordnet die Ressourcen den Vorgängen zu

```
Sub TasksRelationResources()
   'Ansicht Vorgangs-Tabelle
   With Application
      .SelectRow Row:=2, RowRelative:=False
      .ResourceAssignment Resources:="Einkäufer"
      .ResourceAssignment Resources:="Projektleiter[2]"
      .ResourceAssignment Resources:="Büromaterial"
      .SelectRow Row:=1
      .ResourceAssignment Resources:="Einkäufer"
      .ResourceAssignment Resources:="Projektleiter[2]"
      .ResourceAssignment Resources:="Büromaterial"
      .SelectRow Row:=1
      .ResourceAssignment Resources:="Mechaniker"
      .ResourceAssignment Resources:="Fräsmaschine"
      .ResourceAssignment Resources:="Kühlmittel"
      .ResourceAssignment Resources:="Projektleiter[5]"
      .ResourceAssignment Resources:="Büromaterial"
```

```
      .SelectRow Row:=1
      .ResourceAssignment Resources:="Mechaniker"
      .ResourceAssignment Resources:="Hobelbank"
      .ResourceAssignment Resources:="Kühlmittel"
      .ResourceAssignment Resources:="Projektleiter[5]"
      .ResourceAssignment Resources:="Büromaterial"
      .SelectRow Row:=1
      .ResourceAssignment Resources:="Mechaniker"
      .ResourceAssignment Resources:="Drehbank"
      .ResourceAssignment Resources:="Kühlmittel"
      .ResourceAssignment Resources:="Projektleiter[5]"
      .ResourceAssignment Resources:="Büromaterial"
      .SelectRow Row:=1
      .ResourceAssignment Resources:="Mechaniker"
      .ResourceAssignment Resources:="Bohrmaschine"
      .ResourceAssignment Resources:="Kühlmittel"
      .ResourceAssignment Resources:="Projektleiter[5]"
      .ResourceAssignment Resources:="Büromaterial"
      .SelectRow Row:=1
      .ResourceAssignment Resources:="Mechaniker"
      .ResourceAssignment Resources:="Bohrmaschine"
      .ResourceAssignment Resources:="Kühlmittel"
      .ResourceAssignment Resources:="Projektleiter[5]"
      .ResourceAssignment Resources:="Büromaterial"
      .SelectRow Row:=1
      .ResourceAssignment Resources:="Monteur"
      .ResourceAssignment Resources:="Schmiermittel"
      .ResourceAssignment Resources:="Projektleiter[1]"
      .ResourceAssignment Resources:="Büromaterial"
      .SelectRow Row:=1
      .ResourceAssignment Resources:="Monteur"
      .ResourceAssignment Resources:="Schmiermittel"
      .ResourceAssignment Resources:="Projektleiter[1]"
      .ResourceAssignment Resources:="Büromaterial"
   End With
End Sub
```

Die Werte und Zuordnungen sind vereinfacht und ohne reale Grundlage.

8.3.3 Kalender

Project unterscheidet mehrere Arten von Kalender-Objekten *Calendars*.

- Basiskalender als Vorlage für alle Projekte mit den Formen Standard, 24-Stunden-Kalender und Nachtschicht
- Projektkalender der aus dem Basiskalender hervorgeht
- Kalender für einzelne Vorgänge
- Kalender für Ressourcen

Als Grundeinstellung sind drei Kalender vorhanden. Die Kalender *Standard* (Projektkalender), *Nachtschicht* und *24 Stunden*.

Die Prozedur (Code 8.19) ändert die Arbeitszeiten für die Arbeitstage im *Standard*-Kalender.

Code 8.19 Die Prozedur ändert Arbeitszeiten im Kalender Standard

```
Sub SetCalendarTimes()
   With ActiveProject.BaseCalendars("Standard")
      .WeekDays(1).Default 'Sonntag
      With .WeekDays(2) 'Montag
         .Shift1.Start = "08:00"
         .Shift1.Finish = "12:00"
         .Shift2.Start = "12:30"
         .Shift2.Finish = "16:30"
         .Shift3.Clear
         .Shift4.Clear
         .Shift5.Clear
      End With
      With .WeekDays(3) 'Dienstag
         .Shift1.Start = "07:00"
         .Shift1.Finish = "11:00"
         .Shift2.Start = "12:00"
         .Shift2.Finish = "16:00"
         .Shift3.Clear
         .Shift4.Clear
         .Shift5.Clear
      End With
```

```
      With .WeekDays(4) 'Mittwoch
         .Shift1.Start = "07:00"
         .Shift1.Finish = "11:30"
         .Shift2.Start = "12:00"
         .Shift2.Finish = "16:30"
         .Shift3.Clear
         .Shift4.Clear
         .Shift5.Clear
      End With
      With .WeekDays(5) 'Donnerstag
         .Shift1.Start = "07:00"
         .Shift1.Finish = "11:00"
         .Shift2.Start = "12:00"
         .Shift2.Finish = "16:00"
         .Shift3.Clear
         .Shift4.Clear
         .Shift5.Clear
      End With
      With .WeekDays(6) 'Freitag
         .Shift1.Start = "07:30"
         .Shift1.Finish = "11:30"
         .Shift2.Start = "12:00"
         .Shift2.Finish = "16:00"
         .Shift3.Clear
         .Shift4.Clear
         .Shift5.Clear
      End With
      .WeekDays(7).Default 'Samstag
   End With
End Sub
```

Die Prozedur (Code 8.20) öffnet das Dialogfenster für die Arbeitszeit-Verwaltung (Abb. 8.13).

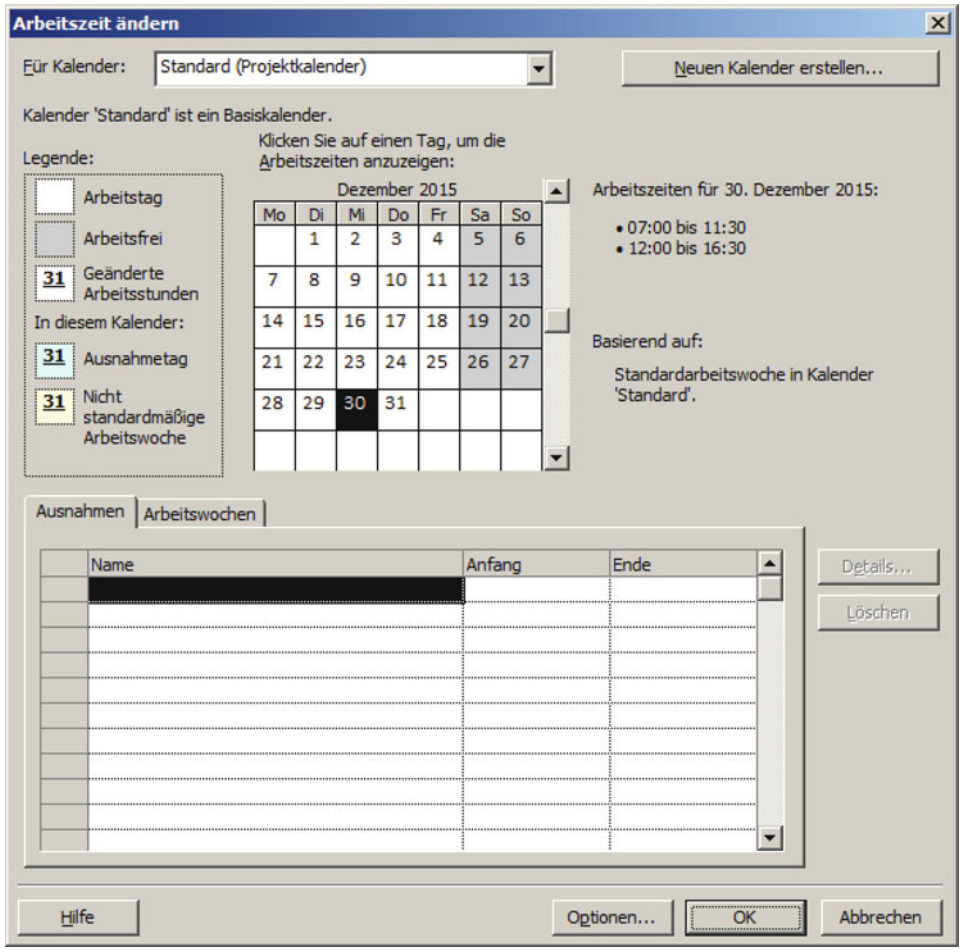


Abb. 8.13 Das Dialogfenster zur Arbeitszeiten-Verwaltung

Code 8.20 Die Prozedur öffnet das Dialogfenster zur Arbeitszeitverwaltung

```
Sub ViewCalendarDialog()
   Application.ChangeWorkingTimeEx
End Sub
```

Die Prozedur (Code 8.21) erzeugt einen neuen Kalender mit dem Namen *NeuerStandard* als Kopie des Kalenders *Standard*, den die Prozedur (Code 8.22) wieder löscht.

Code 8.21 Die Prozedur erzeugt einen neuen Kalender

```
Sub CreateNewCalendar()
   Application.BaseCalendarCreate Name:="NeuerStandard", _
      FromName:="Standard"
End Sub
```

Code 8.22 Die Prozedur löscht einen vorgegebenen Kalender

```
Sub DeteleCalendar()
   Application.BaseCalendarDelete Name:="NeuerStandard"
End Sub
```

Im letzten Schritt müssen den Vorgängen die vorhandenen Kalender zugewiesen werden. Die Prozedur (Code 8.23) weist allen Vorgängen den Standardkalender zu (Abb. 8.14). In der Indikatorenspalte wird dies durch ein Symbol *Vorgangsliste mit Kalender* gekennzeichnet.

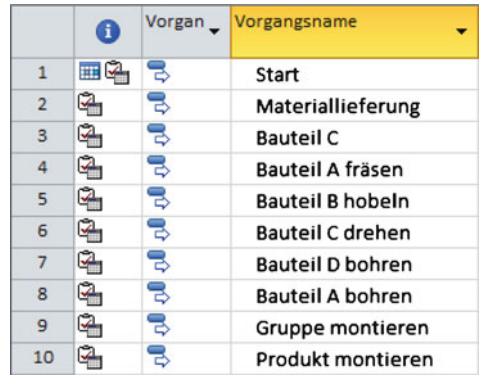


Abb. 8.14 Symbole in der Indikatorenspalte kennzeichnen die Kalenderzuordnung

Code 8.23 Die Prozedur weist allen Vorgängen den Standardkalender zu

```
Sub SetTasksCalendar()
   'Ansicht Gantt-Diagramm
   Dim lRow As Long
   With Application
      lRow = .ActiveProject.Tasks.Count
      .SelectRow Row:=1, RowRelative:=False, Height:=lRow - 1
      .SetTaskField Field:="Vorgangskalender", _
         Value:="Standard", AllSelectedTasks:=True
   End With
End Sub
```

Vor dem Start des Projekts wird der so erstellte Plan als *Basisplan* im Rahmen der gesamten Projektdatei gespeichert. Dadurch können spätere Änderungen im Projekt dokumentiert werden. Bei größeren Projekten werden in Zeitabständen oft mehrere Basispläne abgelegt. Man spricht dann von *Baselines*. Die Prozedur (Code 8.24) erstellt den ersten Basisplan.

Code 8.24 Die Prozedur erstellt die erste Baseline

```
Sub CreateBaseline()
    Application.BaselineSave All:=True, Copy:=0, Into:=0
End Sub
```

Die Prozedur (Code 8.25) zeigt die erste Baseline.

Code 8.25 Die Prozedur zeigt die erste Baseline als Gantt-Diagramm

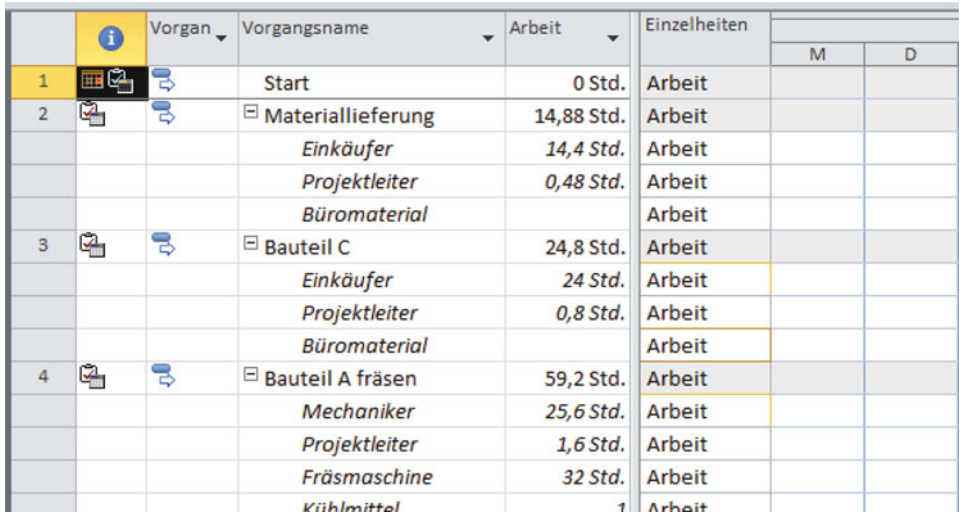
```
Sub ShowBaseline()
    Application.GanttBarStyleBaseline Baseline:=0, Show:=True
End Sub
```

8.4 Project-Ansichten

Neben den bisher gezeigten Ansichten *Balkendiagramm (Gantt), Gantt-Diagramm: Überwachung*, *Netzplan* und *Ressourcenliste* gibt es noch weitere wichtige Ansichten. Einige dienen der Analyse und folgen später.

8.4.1 Weitere wichtige Ansichten

Die Ansicht *Vorgang Einsatz* listet zu allen Vorgängen die Ressourcen mit auf und ermöglich in einer Tabelle die wöchentliche Verteilung der Arbeiten (Abb. 8.15). Die Prozedur (Code 8.26) erstellt die Ansicht *Vorgang Einsatz*.



		Vorgan	Vorgangsname	Arbeit	Einzelheiten	M	D
1			Start	0 Std.	Arbeit		
2			Materiallieferung	14,88 Std.	Arbeit		
			Einkäufer	*14,4 Std.*	Arbeit		
			Projektleiter	*0,48 Std.*	Arbeit		
			Büromaterial		Arbeit		
3			Bauteil C	24,8 Std.	Arbeit		
			Einkäufer	*24 Std.*	Arbeit		
			Projektleiter	*0,8 Std.*	Arbeit		
			Büromaterial		Arbeit		
4			Bauteil A fräsen	59,2 Std.	Arbeit		
			Mechaniker	*25,6 Std.*	Arbeit		
			Projektleiter	*1,6 Std.*	Arbeit		
			Fräsmaschine	*32 Std.*	Arbeit		
			Kühlmittel	*1*	Arbeit		

Abb. 8.15 Die Ansicht Vorgang Einsatz am Beispiel

Code 8.26 Die Prozedur stellt die Ansicht Vorgang Einsatz ein

```
Sub ViewTasksUse()
   Application.ViewApplyEx _
      Name:="Vorgang Einsatz", ApplyTo:=0
End Sub
```

Ebenso wichtig für die Projektplanung sind *Pufferzeiten*. Sie zeigen den Spielraum, der bei notwendigen Verschiebungen vorhanden ist. Zur Darstellung gibt es zwei Möglichkeiten. Mit dem Objekt *GanttBarStyleSlack* werden die Pfufferzeiten im Gantt-Diagramm als Linien eingetragen. Die Prozeduren (Code 8.27) schalten die Darstellung ein und aus.

Code 8.27 Prozeduren zum Ein- und Ausschalten der grafischen Darstellung von Pufferzeiten

```
Sub ShowStyleSlacksOn()
    Application.GanttBarStyleSlack Show:=True
End Sub
```

```
Sub ShowStyleSlacksOff()
    Application.GanttBarStyleSlack Show:=False
End Sub
```

Übersichtlicher ist die Darstellung in einer Spalte der Vorgangstabelle (Abb. 8.16). Die Prozedur (Code 8.28) erstellt diese zusätzliche Spalte.

	Vorgangsname	Freie Pufferzeit	Gesamte Pufferzeit
1	Start	0 Tage	0 Tage
2	Materiallieferung	0 Tage	4 Tage
3	Bauteil C	0 Tage	0 Tage
4	Bauteil A fräsen	0 Tage	12 Tage
5	Bauteil B hobeln	4 Tage	4 Tage
6	Bauteil C drehen	0 Tage	0 Tage
7	Bauteil D bohren	26 Tage	26 Tage
8	Bauteil A bohren	12 Tage	12 Tage
9	Gruppe montieren	0 Tage	0 Tage
10	Produkt montieren	0 Tage	0 Tage

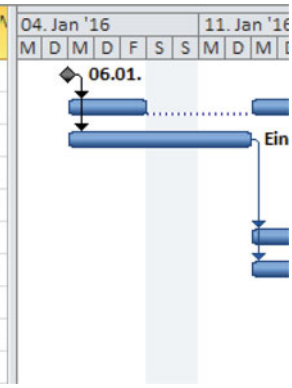


Abb. 8.16 Ausgabe von Pufferzeiten als Spaltenwerte

Code 8.28 Die Prozedur erstellt in der Vorgangstabelle Spalten mit Pufferzeiten

```
Sub CreateTaskSlackColumn()
    TableApply Name:="Berechnete Termine"
End Sub
```

Eine der neueren Ansichten in Project ist der *Teamplaner* (Abb. 8.17). In ihm werden zu den Ressourcen die zugeordneten Vorgänge auf einer Zeitschiene dargestellt. Vorgänge ohne Ressourcenzuordnung werden unterhalb einer Trennlinie angezeigt und können per Interaktion durch Maus und Verschieben einer Ressource (oder mehreren) zugeordnet werden. Die Grafik zeigt sehr anschaulich welche Bereiche zur Verfügung stehen. Die Prozedur (Code 8.29) erstellt die Ansicht *Teamplaner*.

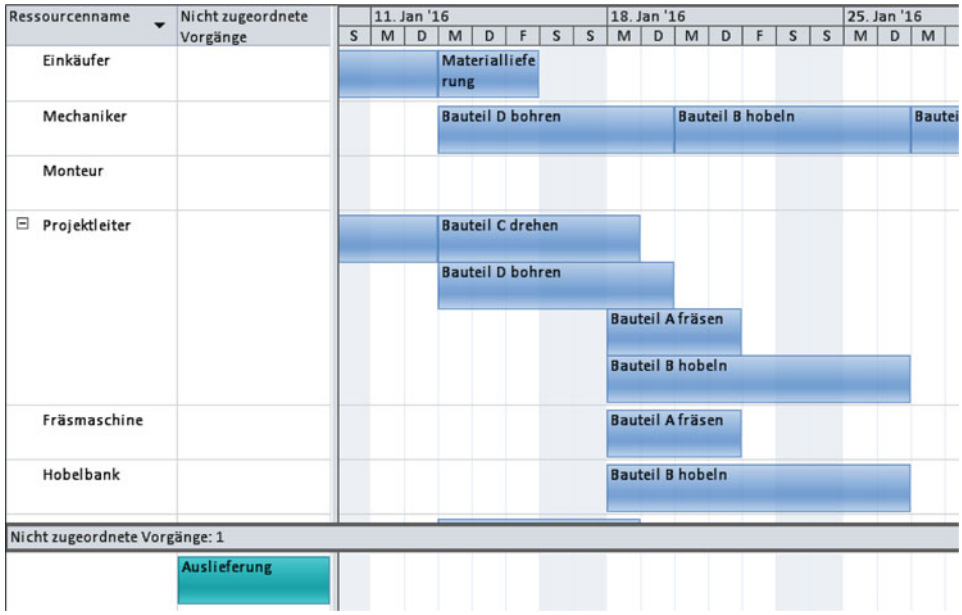


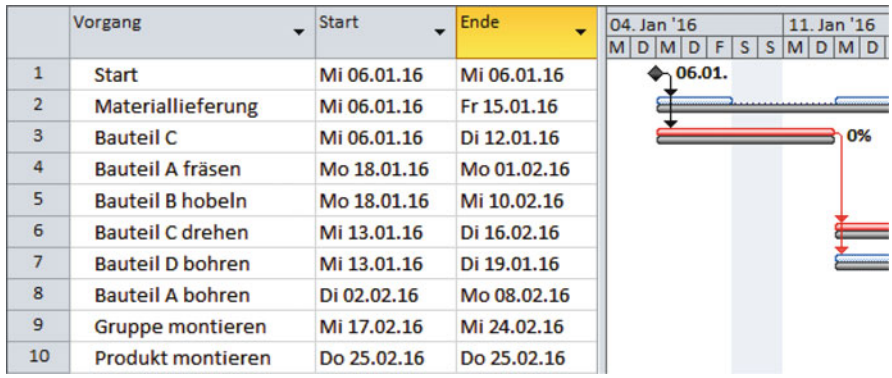
Abb. 8.17 Die Ansicht Teamplaner am Beispiel

Code 8.29 Die Prozedur stellt die Ansicht Teamplaner ein

```
Sub ViewTeamplaner()
   Application.ViewApplyEx _
      Name:="Teamplanung", ApplyTo:=0
End Sub
```

8.4.2 Eigene Ansichten erstellen

Neben einer Vielzahl vorgegebener Ansichten können auch eigene Ansichten kreiert werden. Die Prozedur (Code 8.30) erstellt eine neue Tabelle *Termine* als Kopie der Vorgangstabelle und übernimmt daraus nur die Spalten *Nr.*, *Vorgangsname*, *Anfang* und *Fertig stellen*. Dabei werden die Spaltentitel abgeändert in *Vorgang*, *Start* und *Ende* (Abb. 8.18).



	Vorgang	Start	Ende
1	Start	Mi 06.01.16	Mi 06.01.16
2	Materiallieferung	Mi 06.01.16	Fr 15.01.16
3	Bauteil C	Mi 06.01.16	Di 12.01.16
4	Bauteil A fräsen	Mo 18.01.16	Mo 01.02.16
5	Bauteil B hobeln	Mo 18.01.16	Mi 10.02.16
6	Bauteil C drehen	Mi 13.01.16	Di 16.02.16
7	Bauteil D bohren	Mi 13.01.16	Di 19.01.16
8	Bauteil A bohren	Di 02.02.16	Mo 08.02.16
9	Gruppe montieren	Mi 17.02.16	Mi 24.02.16
10	Produkt montieren	Do 25.02.16	Do 25.02.16

Abb. 8.18 Ansicht der neuen Tabelle

Code 8.30 Die Prozedur erstellt eine neue Tabelle

```
Sub CreateNewTable()
   On Error Resume Next
   With Application
      .TableEditEx Name:="Termine", TaskTable:=True, _
         Create:=True, OverwriteExisting:=True, _
         FieldName:="Nr.", Title:="", _
         Width:=6, Align:=1, ShowInMenu:=False, _
         LockFirstColumn:=True, DateFormat:=255, _
         RowHeight:=1, AlignTitle:=0, WrapText:=False
      .TableEditEx Name:="Termine", TaskTable:=True, _
         NewFieldName:="Name", Title:="Vorgang", _
         Width:=23, Align:=0, LockFirstColumn:=True, _
         DateFormat:=255, RowHeight:=1, _
         AlignTitle:=0, WrapText:=True
      .TableEditEx Name:="Termine", TaskTable:=True, _
         NewFieldName:="Anfang", Title:="Start", _
         Width:=13, Align:=0, LockFirstColumn:=True, _
         DateFormat:=255, RowHeight:=1, _
         AlignTitle:=0, WrapText:=False
      .TableEditEx Name:="Termine", TaskTable:=True, _
         NewFieldName:="Fertig stellen", Title:="Ende", _
         Width:=13, Align:=0, LockFirstColumn:=True, _
         DateFormat:=255, RowHeight:=1, _
         AlignTitle:=0, WrapText:=False, ShowAddNewColumn:=True
      .TableApply Name:="Termine"
   End With
End Sub
```

8.4.3 Ansichten Teilung

Das Feld-Objekt *Pane* stellt einen Bereich in der aktuellen Ansicht dar. Wird es aktiviert, dann erfolgt eine horizontale Ansichtsteilung und im unteren Bereich wird eine Maske mit Details zu dem markierten Eintrag der oberen Tabelle eingeblendet. Die Prozeduren (Code 8.31) öffnen und schließen die Darstellung. Die Anwendung in der Vorgangsliste (Abb. 8.19) wird anschließend dargestellt.

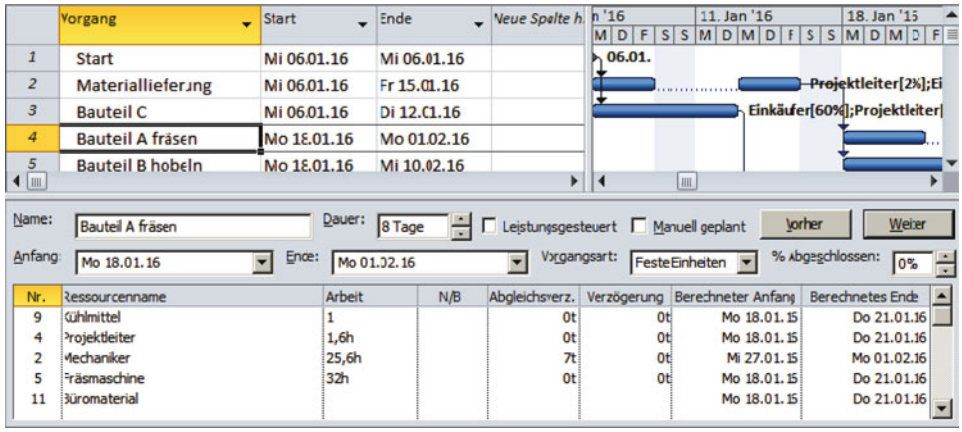


Abb. 8.19 Einblendung der Vorgang-Details am Beispiel

Code 8.31 Die Prozeduren schalten die Darstellung von Details ein und aus

```
Sub CreatePaneView()
   Application.PaneCreate
End Sub

Sub ClosePaneView()
   Application.PaneClose
End Sub
```

Durch Ziehen der Trennlinien zwischen den Bildschirmausschnitten kann die Größe der Darstellung angepasst werden. Durch Ziehen der Trennlinie an den Rand entfällt die Trennung.

Neben der Vorgangs-Maske lassen sich noch viele weitere Details einblenden. Die Prozedur (Code 8.32) blendet die Maske Ressource Einsatz ein (Abb. 8.20).

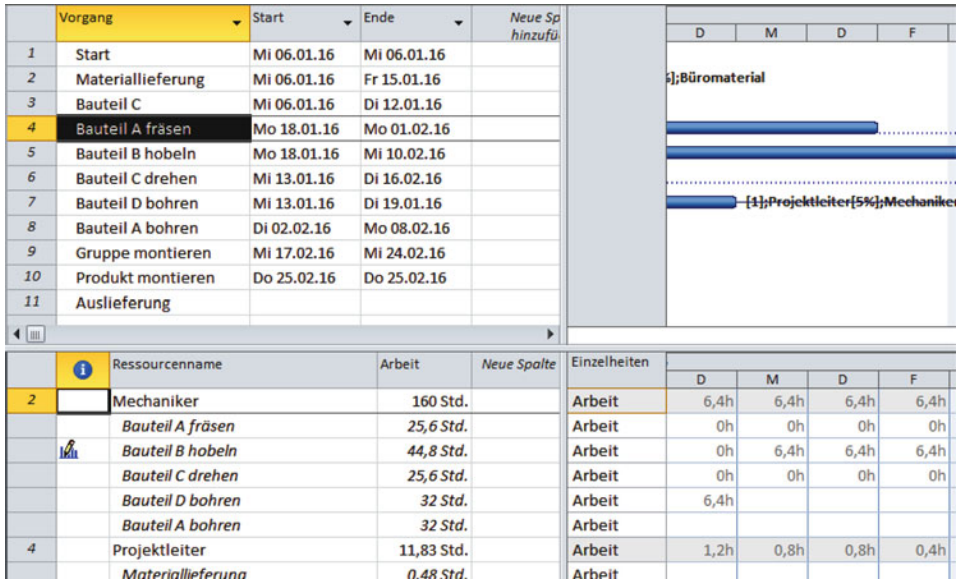


Abb. 8.20 Einblendung der Ressourcen Einsatz-Details

Hier wird auch die Aufgabe des Parameters *ApplyTo* in der Methode *ViewApplyEx* deutlich. Er gibt den Teilbereich der Ansicht vor.

Code 8.32 Die Prozedur stellt den Ressourcen-Einsatz im Detailbereich ein

```
Sub ViewResourcesUse()
   Application.ViewApplyEx _
      Name:="Ressource Einsatz", ApplyTo:=1
End Sub
```

8.4.4 Darstellungen in der Zeitachse

Einen genauen Überblick über den gesamten Terminplan bietet das *Timeline*-Objekt (Zeitachse). Die Prozedur (Code 8.33) schaltet zwischen Darstellung (Abb. 8.21) und Nichtdarstellung mit jedem Aufruf um.

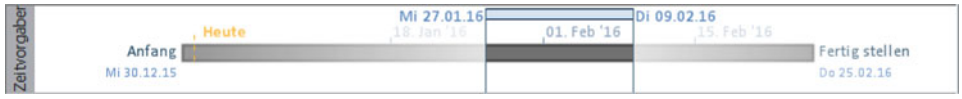


Abb. 8.21 Darstellung der Zeitachse am Beispiel

Code 8.33 Die Prozedur blendet die Zeitachse ein und aus

```
Sub ToggleTimeLine()
   Application.TimelineViewToggle
End Sub
```

Außerdem besteht die Möglichkeit ausgewählte Vorgänge in der Zeitachse darzustellen. Die Prozedur (Code 8.34) blendet die Vorgänge zum Bauteil A ein (Abb. 8.22).

Abb. 8.22 Darstellung von Vorgängen in der Zeitachse

Code 8.34 Die Prozedur blendet Vorgänge in der Zeitachse ein

```
Sub InsertTasksInTimeLine()
   With Application
      .WindowActivate TopPane:=False
      .TaskOnTimeline TaskID:=4
      .TaskOnTimeline TaskID:=8
      .TaskOnTimeline ShowDialog:=False
   End With
End Sub
```

Die Prozedur (Code 8.35) entfernt die Vorgänge wieder aus der Zeitachse.

Code 8.35 Die Prozedur entfernt die Vorgänge aus der Zeitachse

```
Sub RemoveTasksInTimeLine()
   With Application
      .TaskOnTimeline TaskID:=4, Remove:=True
      .TaskOnTimeline TaskID:=8, Remove:=True
      .TaskOnTimeline ShowDialog:=False
   End With
End Sub
```

Mit der Anweisung *ShowDialog:=True* wird ein Dialogfenster zur manuellen Auswahl der Vorgänge eingeblendet. Mit der Auswahl vieler Vorgänge wächst die Darstellung der Ebenen in der Zeitachse (Abb. 8.23).

Abb. 8.23 Darstellung sich überlagernder Vorgänge in der Zeitachse

Es besteht auch die Möglichkeit die Zeitachse zu drucken, sie per E-Mail (Abb. 8.24) zu senden oder für eine PowerPoint-Präsentation in die Zwischenablage zu kopieren. Die Prozedur (Code 8.36) kopiert die Zeitachse mit unterschiedlichen Größen für die Anwendung in die Zwischenablage.

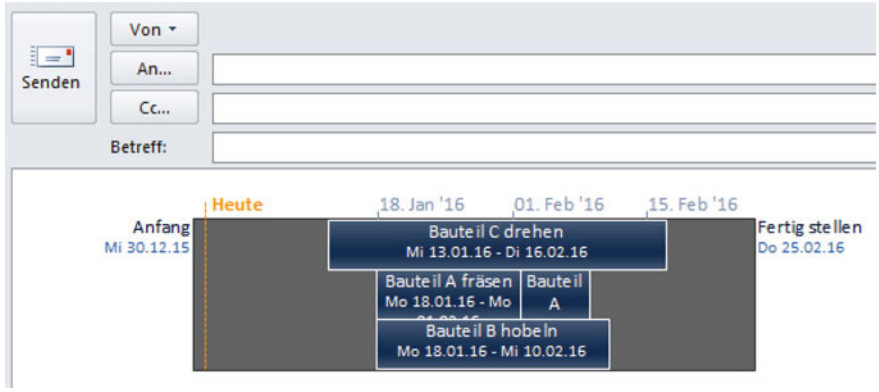


Abb. 8.24 Import der Zeitachse in ein E-Mail-Formular

Code 8.36 Export der Zeitachse mit unterschiedlichen Größen in die Zwischenablage

```
Sub ExportTimeLine()
   With Application
      .TimelineExport ExportWidth:=600 'Mail
      .TimelineExport ExportWidth:=940 'Präsentation
      .TimelineExport                  'volle Größe
   End With
End Sub
```

8.4.5 Kostenkontrolle

Eine der Hauptaufgaben eines Projektmanagers ist die ständige Kostenkontrolle. Dazu stehen ihm zwei Ansichten zur Verfügung. Die erste erzeugt Prozedur (Code 8.37) und hat den Titel *ViewProjectStatistik* (Abb. 8.25).

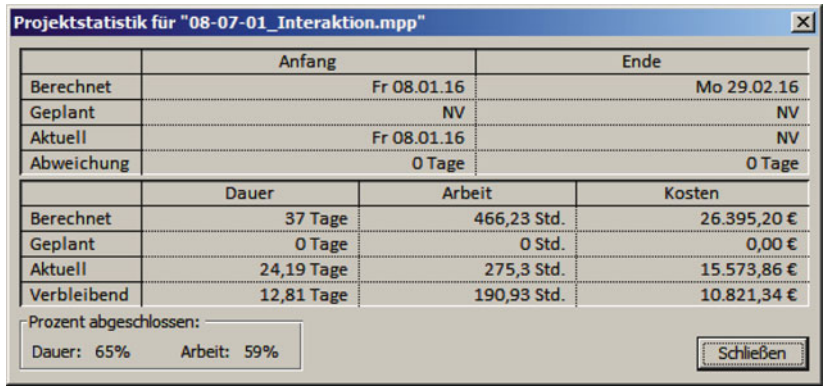


Abb. 8.25 Projektstatistik

Code 8.37 Die Prozedur ruft die aktuelle Projektstatistik auf

```
Sub ViewProjectStatistic()
    ProjectStatistics Project:="08-07-01_Interaktion.mpp"
End Sub
```

Die zweite Möglichkeit ist die Kostenanalyse. Die Prozedur (Code 8.38) erstellt eine Kostenanalyse vom aktuellen Projekt (Abb. 8.26).

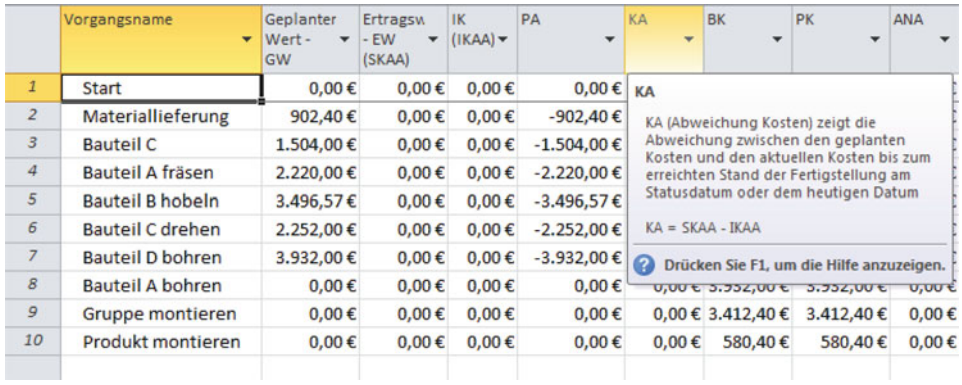


Abb. 8.26 Ausschnitt aus einer Kostenanalyse

Code 8.38 Die Prozedur ruft die Kostenanalyse auf

```
Sub ViewCostsAnalysis()
    TableApply Name:="Kostenanalyse"
End Sub
```

Wird die Maus auf eine Spaltenüberschrift gesetzt, so erfolgt die Einblendung einer Erklärung zum verwendeten Kürzel und der zugrunde liegenden Formel.

8.5 Project-Terminplanung

Im täglichen Projektgeschäft läuft leider nicht immer alles so rund wie vorher geplant. Oft müssen Einschränkungen berücksichtigt werden. Änderungen und Verschiebungen sind die Folge.

8.5.1 Vorgangseinschränkungen

Die Prozedur (Code 8.39) erzeugt beim Vorgang Nr. 3 die Einschränkungsart *Anfang nicht früher als* mit dem Datum *08.01.2016*.

Code 8.39 Die Prozedur definiert eine Einschränkung

```
Sub TaskTimeRestriction()
   With Application
      .SelectTaskField Row:=3, Column:="Name", RowRelative:=False
      .SetTaskField Field:="Einschränkungsart", _
         Value:="Anfang nicht früher als", AllSelectedTasks:=True
      .SetTaskField Field:="Einschränkungstermin", _
         Value:="Fr 08.01.16", AllSelectedTasks:=True
   End With
End Sub
```

Die Einschränkung zieht damit Probleme nach sich (Abb. 8.27). Natürlich können die Warnhinweise ignoriert werden, doch besser ist es sie zu beheben.

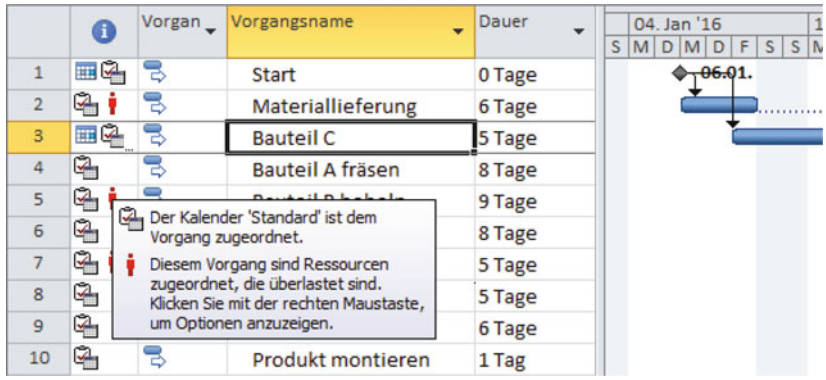


Abb. 8.27 Überlastungen durch die definierte Einschränkung

Die Prozedur (Code 8.40) verschiebt automatisch den Vorgang Nr. 2 auf den nächstmöglichen Termin. Erreicht wird dies durch eine Unterbrechung des Vorgangs.

Code 8.40 Die Prozedur bewirkt eine Terminverschiebung

```
Sub NextAvailableTime()
   With Application
      .SelectTaskField Row:=2, Column:="Indikatoren", _
         RowRelative:=False
      .TaskDrivers
      .LevelSelected ResolveMethod:=pjResolveNextAvailableTime
   End With
End Sub
```

8.5.2 Zeitraumgrenzen

Mitunter kommt es vor, dass Vorgänge nur in bestimmten Zeiträumen durchführbar sind. Die Prozedur (Code 8.41) erzeugt vor dem betreffenden Vorgang (hier Nr. 7) einen Meilenstein (Abb. 8.28). Der erhält die Einschränkung *Anfang nicht früher als*. Der Vorgang selbst erhält die Einschränkung *Ende nicht später als*. Ebenso müssen die Vorgänger umgesetzt werden.

	Vorgan	Vorgangsname	Dauer	Anfang	Fertig stellen	Vorgänger	R
1		Start	0 Tage	Mi 06.01.16	Mi 06.01.16		
2		Materiallieferung	4 Tage	Mi 06.01.16	Fr 15.01.16	1	F
3		Bauteil C	5 Tage	Fr 08.01.16	Do 14.01.16	1	E
4		Bauteil A fräsen	8 Tage	Mo 18.01.16	Mo 01.02.16	2	[
5		Bauteil B hobeln	11 Tage	Mo 18.01.16	Fr 12.02.16	2	[
6		Bauteil C drehen	8 Tage	Fr 15.01.16	Do 18.02.16	3	[
7		Bauteil D	0 Tage	Do 14.01.16	Do 14.01.16	3	
8		Bauteil D bohren	5 Tage	Mi 20.01.16	Di 26.01.16	7	[
9		Bauteil A bohren	5 Tage	Di 02.02.16	Mo 08.02.16	4	[
10		Gruppe montieren	6 Tage	Fr 19.02.16	Fr 26.02.16	5;6	F
11		Produkt montieren	1 Tag	Mo 29.02.16	Mo 29.02.16	8;9;10	F

Abb. 8.28 Der eingefügte Meilenstein Bauteil D Nr. 7 ermöglicht eine Zeitraumangabe

Der neue Meilenstein wird durch Einfügen einer neuen Zeile Nr. 7 möglich. Seine Dauer ist Null.

Code 8.41 Die Prozedur erzeugt einen Aktionszeitraum für einen Vorgang

```
Sub CreateTaskPeriod()
   With Application
      .SelectTaskField Row:=7, Column:="Name", RowRelative:=False
      .InsertTask
      .SetTaskField Field:="Name", Value:="Bauteil D  "
      .SelectTaskField Row:=0, Column:="Dauer"
      .SetTaskField Field:="Dauer", Value:="0"
      .SelectTaskField Row:=0, Column:="Vorgangsmodus"
      .SetTaskField Field:="Vorgangsmodus", Value:="Nein"
      .SelectTaskField Row:=0, Column:="Vorgänger"
      .SetTaskField Field:="Vorgänger", Value:="3"
      .SelectTaskField Row:=1, Column:="Vorgänger"
      .SetTaskField Field:="Vorgänger", Value:="7"
      .SelectTaskField Row:=7, Column:="Name", RowRelative:=False
      .SetTaskField Field:="Einschränkungstermin", _
         Value:="Mo 11.01.16", AllSelectedTasks:=True
      .SetTaskField Field:="Einschränkungsart", _
         Value:="Anfang nicht früher als", AllSelectedTasks:=True
      .SelectTaskField Row:=1, Column:="Name"
      .SetTaskField Field:="Einschränkungstermin", _
         Value:="Do 23.01.16", AllSelectedTasks:=True
      .SetTaskField Field:="Einschränkungsart", _
         Value:="Ende nicht später als", AllSelectedTasks:=True
   End With
End Sub
```

8.5.3 Vorgangsunterbrechung

Vorgänge können mit der Methode *Split* bewusst unterbrochen werden. Die Prozedur (Code 8.42) unterbricht den Vorgang Nr. 5 am 22.01.16, um in dann einen Tag später am 23.01.16 fortzusetzen. Danach erfolgt noch einmal eine Korrektur, die den Start des zweiten Teils auf den 26.01.16 verschiebt (Abb. 8.29).

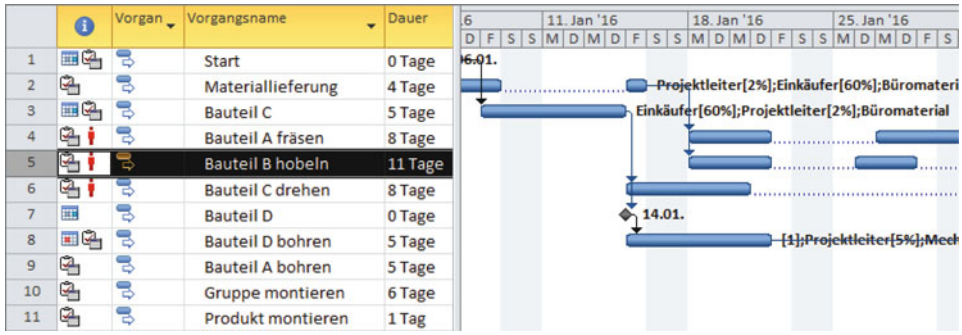


Abb. 8.29 Unterbrechung eines Vorgangs am Beispiel Vorgang Nr. 5

Code 8.42 Die Prozedur unterbricht einen Vorgang

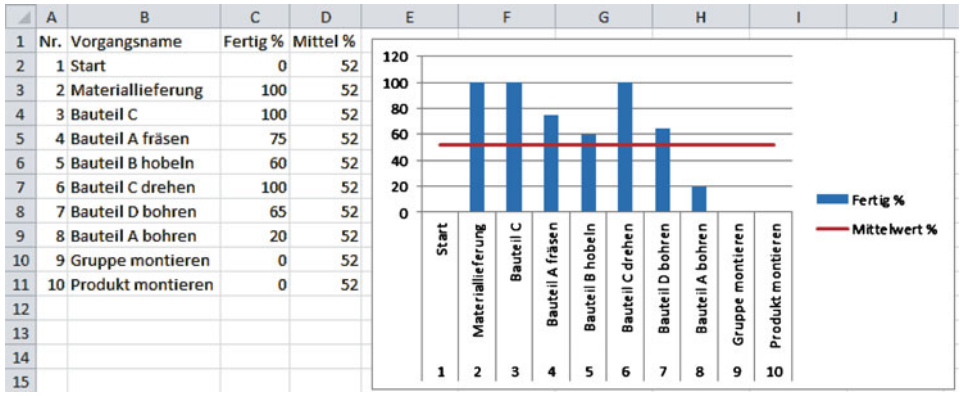
```
Sub SplitTask()
  With Application.ActiveProject.Tasks(5)
    .Split StartSplitOn:="22.01.16 08:00", EndSplitOn:="23.01.16 08:00"
    .SplitParts(2).Start = "26.01.16 08:00"
  End With
End Sub
```

8.6 Project-Interaktionen

Project bietet sehr viele Ansichten, so dass ein Controlling sehr einfach zu handhaben ist. In wenigen Dingen können da andere Anwendungen nützlich sein. Der Klassiker ist auch hier eine Visualisierung mit Diagrammen.

8.6.1 Datenexport nach Excel

Am Anfang dieses Kapitels wurden bereits die Grunddaten aus Excel nach Project importiert. Hier soll nun die andere Richtung zur Anwendung kommen. Die Prozedur (Code 8.43) exportiert den Status jedes Vorgangs in Prozent in ein neues Excel-Arbeitsblatt. Die Daten werden dann in einem Balkendiagramm visualisiert (Abb. 8.30).



Nr.	Vorgangsname	Fertig %	Mittel %
1	Start	0	52
2	Materiallieferung	100	52
3	Bauteil C	100	52
4	Bauteil A fräsen	75	52
5	Bauteil B hobeln	60	52
6	Bauteil C drehen	100	52
7	Bauteil D bohren	65	52
8	Bauteil A bohren	20	52
9	Gruppe montieren	0	52
10	Produkt montieren	0	52

Abb. 8.30 Daten und Diagramm im Excel-Worksheet

Code 8.43 Die Prozedur exportiert den Vorgangsstatus nach Excel mit Visualisierung

```
Sub ExportProjectUtilization()
   Dim proActive  As Project
   Dim proTask    As Task
   Dim exlApp     As Excel.Application
   Dim exlPro     As Excel.Workbook
   Dim exlPlan    As Excel.Worksheet
   Dim exlShape   As Excel.Shape
   Dim exlChart   As Excel.Chart
   Dim sAddress   As String
   Dim lRow       As Long
   Dim sRow       As String
   Dim sRange     As String

   Set exlApp = CreateObject("Excel.Application")
   Set exlPro = exlApp.Workbooks.Add
   exlApp.Visible = True
   Set exlPlan = exlPro.Worksheets(1)
   exlPlan.Name = "Status"
   exlPlan.Range("A1:D1") = _
      Array("Nr.", "Vorgangsname", "Fertig %", "Mittel %")

'Datenübertragung
   Set proActive = ActiveProject
   With proActive
      lRow = 1
      For Each proTask In .Tasks
         lRow = lRow + 1
         exlPlan.Cells(lRow, 1) = proTask.WBS
```

```
            exlPlan.Cells(lRow, 2) = proTask.Name
            exlPlan.Cells(lRow, 3) = proTask.PercentComplete
        Next
        exlPlan.Columns("A:D").EntireColumn.AutoFit
    End With

'Mittelwertbildung
    lRow = exlPlan.UsedRange.Rows.Count
    sRow = Trim(Str(lRow))
    sRange = "Status!$C$2:$C$" & sRow & ""
    exlPlan.Range(sRange).Select
    exlPro.Names.Add Name:="Daten", RefersTo:=exlPlan.Range(sRange)
    sRange = "Status!$D$2:$D$" & sRow
    exlPlan.Range(sRange).Select
    exlPlan.Range(sRange).FormulaArray = "=AVERAGE(Daten)"
'Säulendiagramm
    sAddress = exlPlan.UsedRange.Address
    exlPlan.Range("$B$2:$C$" & sRow).Select
    exlPlan.UsedRange.Select
    Set exlShape = exlPlan.Shapes.AddChart
    Set exlChart = exlShape.Chart
    exlChart.ChartType = xlColumnClustered

'Mittelwert-Linie
    exlChart.SeriesCollection(2).Name = "=""Mittelwert %"""
    exlChart.SeriesCollection(2).Values = sRange
    exlChart.SeriesCollection(2).Type = xlLine
    exlChart.SeriesCollection(2).Select
    Selection.MarkerStyle = xlNone
    Set exlChart = Nothing
    Set proActive = Nothing
    Set proTask = Nothing
    Set exlApp = Nothing
    Set exlPro = Nothing
    Set exlPlan = Nothing
End Sub
```

Das so gewonnene Diagramm kann natürlich weiter exportiert werden, z. B. in eine PowerPoint-Folie. Wie das funktioniert, wurde an anderer Stelle bereits gezeigt.

8.6.2 Teamkommunikation

In größeren Projekten werden Excel-Tools gerne zur Kommunikations-Organisation zwischen den Teammitgliedern verwendet. Dort lassen sich dann detaillierte Teamaufgaben

und Meetings verwalten. Mit einer Anbindung an Outlook können Team-E-Mails für Meeting-Einladungen und Teamaufgaben direkt gesendet werden.

Dazu müssen zunächst einmal wieder Teamdaten nach Excel gesendet werden. Um Teammitglieder von anderen Arbeits-Ressourcen zu trennen, kann z. B. für das Kürzel in der Ressourcen-Tabelle eine Personalnummer verwendet werden (Abb. 8.31). Die Prozedur (Code 8.44) erstellt eine Teamliste in Excel zur weiteren Organisation (Abb. 8.32).

Abb. 8.31 Kennung der Teammitglieder in der Ressourcenliste

	Ressourcenname	Art		Kürzel
1	Einkäufer	Arbeit		P01
2	Mechaniker	Arbeit		P02
3	Monteur	Arbeit		P03
4	Projektleiter	Arbeit		P04
5	Fräsmaschine	Arbeit		F
6	Hobelbank	Arbeit		H
7	Drehbank	Arbeit		D
8	Bohrmaschine	Arbeit		B
9	Kühlmittel	Material		K
10	Schmiermittel	Material		S
11	Büromaterial	Kosten		B

	A	B	C	D	E	F
1	PNr	Name	Max.	Std.Satz	ÜbStd.Satz	Kosten/E
2	P01	Einkäufer	60%	60,00 €/Std.	80,00 €/Std.	0
3	P02	Mechaniker	80%	40,00 €/Std.	60,00 €/Std.	0
4	P03	Monteur	100%	70,00 €/Std.	90,00 €/Std.	0
5	P04	Projektleiter	100%	80,00 €/Std.	100,00 €/Std.	0
6						

Abb. 8.32 Die erstellte Teamliste

Code 8.44 Die Prozedur erstellt eine Teamliste als Excel-Worksheet

```
Sub ExportTeam()
   Dim proActive  As Project
   Dim proRes     As Resource

   Dim exlApp     As Excel.Application
   Dim exlPro     As Excel.Workbook
   Dim exlTeam    As Excel.Worksheet
   Dim exlShape   As Excel.Shape
   Dim exlChart   As Excel.Chart
```

```
    Dim sAddress    As String
    Dim lRow        As Long
    Dim sRow        As String
    Dim sRange      As String
    Dim dValue      As Double

    Set exlApp = CreateObject("Excel.Application")
    Set exlPro = exlApp.Workbooks.Add
    exlApp.Visible = True
    Set exlTeam = exlPro.Worksheets(1)
    exlTeam.Name = "Team"
    exlTeam.Range("A1:F1") = _
       Array("PNr", "Name", "Max.", "Std.Satz", _
       "ÜbStd.Satz", "Kosten/E")

'Datenübertragung
    Set proActive = ActiveProject
    With proActive
       lRow = 1
       For Each proRes In .Resources
          If Left(proRes.Initials, 1) = "P" Then
             lRow = lRow + 1
             exlTeam.Cells(lRow, 1) = proRes.Initials
             exlTeam.Cells(lRow, 2) = proRes.Name
             dValue = CDbl(proRes.MaxUnits)
             exlTeam.Cells(lRow, 3) = Format(dValue, "##0,0%")
             exlTeam.Cells(lRow, 4) = proRes.StandardRate
             exlTeam.Cells(lRow, 5) = proRes.OvertimeRate
             exlTeam.Cells(lRow, 6) = proRes.CostPerUse
          End If
       Next
       exlTeam.Columns("A:D").EntireColumn.AutoFit
    End With
    Set proActive = Nothing
    Set proRes = Nothing
    Set exlApp = Nothing
    Set exlPro = Nothing
    Set exlTeam = Nothing
End Sub
```

Und mit der Teamliste geht es weiter. Wird nun mit laufender Spalte ein Meeting-Termin in der ersten Zeile eingetragen (Abb. 8.33) und darunter vermerkt wer teilnehmen soll, so lässt sich mit dieser Struktur und der Prozedur (Code 8.45) eine Sammel-E-Mail als Einladung versenden (Abb. 8.34).

	A	B	C	D	E	F	G	H
1	PNr	Name	Max.	Std.Satz	ÜbStd.Satz	Kosten/E	E-Mail-Adresse	10.01.2016
2	P01	Einkäufer	60%	60,00 €/Std.	80,00 €/Std.	0	einkauf@muster.de	x
3	P02	Mechaniker	80%	40,00 €/Std.	60,00 €/Std.	0	mechanik@muster.de	x
4	P03	Monteur	100%	70,00 €/Std.	90,00 €/Std.	0	monteur@muster.de	
5	P04	Projektleiter	100%	80,00 €/Std.	100,00 €/Std.	0	leitung@muster.de	x

Abb. 8.33 Erweiterung der Teamliste mit Mailadresse und fortlaufenden Meeting-Terminen

Abb. 8.34 E-Mail-Beispiel

Code 8.45 Die Prozedur erzeugt Sammel-E-Mails

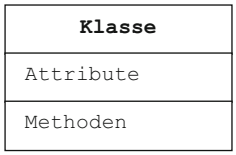
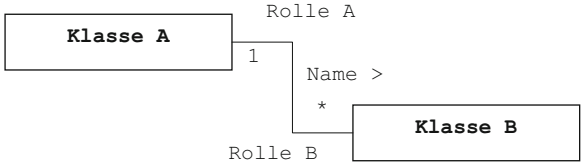
```
Sub SendTeamMails()
   Dim wshTeam     As Worksheet
   Dim outApp      As Outlook.Application
   Dim outMail     As Outlook.MailItem
   Dim lRow        As Long
   Dim lRowMax     As Long
   Dim lColMax     As Long
   Dim dTime       As Date
   Dim sMail       As String

   Set wshTeam = ThisWorkbook.Worksheets("Team")
   With wshTeam.UsedRange
      lRowMax = .Rows.Count
      lColMax = .Columns.Count
   End With
   dTime = wshTeam.Cells(1, lColMax)
   Set outApp = CreateObject("Outlook.Application")
   For lRow = 2 To lRowMax
      sMail = wshTeam.Cells(lRow, 7)
      If wshTeam.Cells(lRow, lColMax) = "x" And _
         Not sMail = "" Then
         Set outMail = outApp.CreateItem(0)
         With outMail
```

```
            .Subject = "Teammeeting am " & dTime
            .Body = "Einladung zum Teammeeting am " & dTime & _
               vbCrLf & "Gruß Projectleitung"
            .To = sMail
            .Attachments.Add "C:\Temp\Agenda.docx"
            .Display
            '.Send
         End With
         Set outMail = Nothing
      End If
   Next lRow
   Set outApp = Nothing
   Set wshTeam = Nothing
End Sub
```

Eine andere Variante ist die Serienmail-Funktion in Word. Diese überlasse ich zur Übung wieder dem Leser.

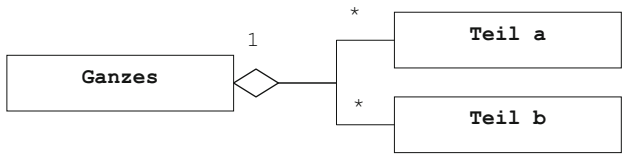
Anhang 1

Tab. A1.1 Die wichtigsten Klassendiagramm Elemente

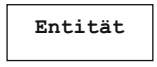
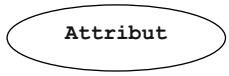
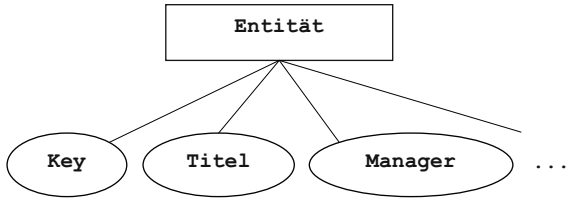
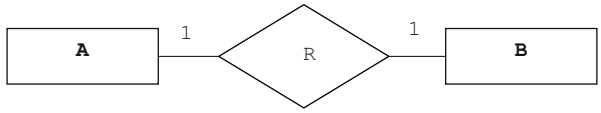
	Eine **Klasse** wird vereinfacht nur als ein Rechteck dargestellt, in dem der Name der Klasse steht. Ausführlicher werden in Rechtecken darunter Attribute und Methoden notiert. Attribute werden als Liste untereinander aufgeführt. Es sind alle für die Modellbildung wichtigen Eigenschaften. Methoden, auch als Member bezeichnet, sind Funktionen mit Parametern.
	Eine **abstrakte** Klasse wird kursiv geschrieben und besitzt keine Objekte, sondern dient nur der Strukturierung.
	Eine **Assoziation** ist eine Beziehung zwischen Klassen. Ihre Objekte tauschen über die Assoziation ihre Nachrichten aus. Sie hat einen **Namen** und kann einen **Pfeil** für die Richtung haben. An den Enden können die **Rollen** der Klassen und ihre **Multiplizität** angegeben werden.
	Eine **gerichtete** Assoziation hat einen offenen Pfeil und Nachrichten werden nur in dieser Richtung getauscht.

H. Nahrstedt, *Die Welt der VBA-Objekte*, DOI 10.1007/978-3-658-13891-2

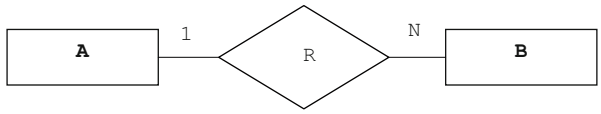
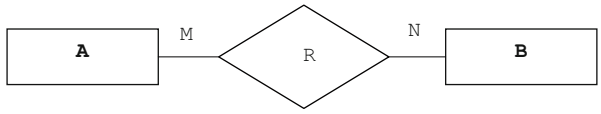
Tab. A1.1 (Fortsetzung)

	Eine **Vererbung** wird mit einem leeren Pfeil dargestellt. Die Oberklasse vererbt ihre Eigenschaften an die Unterklasse.
	Eine **Aggregation** ist eine Teile-Ganzes-Beziehung und wird durch eine Raute ausgedrückt. Sind die Teile existenzabhängig vom Ganzen, spricht man von einer **Komposition** und die Raute ist gefüllt.

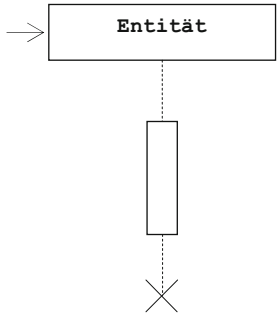
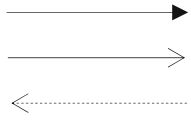
Tab. A1.2 Die wichtigsten Entity-Relationship-Model-Elemente

	Eine Entität ist ein Objekt, über welches die Daten (Modell) gespeichert werden. Sie werden als Rechtecke dargestellt.
	Attribute werden als Ovale dargestellt.
	Zugehörigkeiten zwischen Attributen und Entität werden durch gerade Linien dargestellt
 Es werden drei Typen von Relationen unterschieden:	Beziehungen zwischen Entitäten werden als Rhomben dargestellt und beschrieben. Entitäten und Beziehungen werden durch Konnektoren verbunden. Zur genaueren Definition können an die Konnektoren Kardinalitäten gesetzt werden (N, 1).
	Eine Instanz der Entität A ist genau mit einer Instanz der Entität B verbunden.

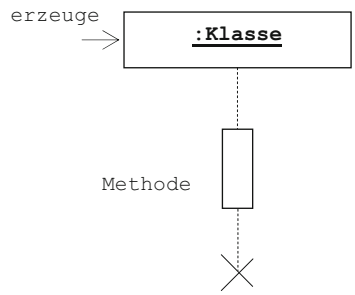
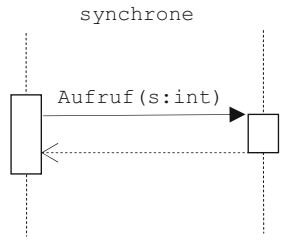
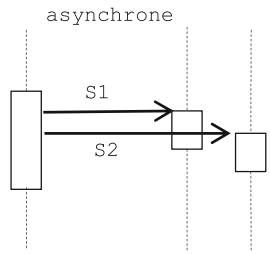
Tab. A1.2 (Fortsetzung)

	Eine Instanz der Entität A kann mit keiner, einer oder mehreren Instanzen der Entität B verbunden sein. Eine Instanz der Entität B kann genau eine Instanz der Entität A konnektieren.
	Eine Instanz der Entität A kann mit keiner, einer oder mehreren Instanzen der Entität B verbunden sein. Gleiches gilt auch für die Umkehrung.

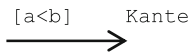
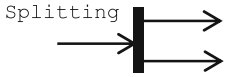
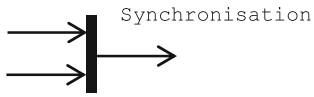
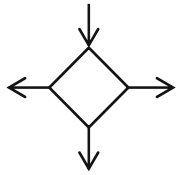
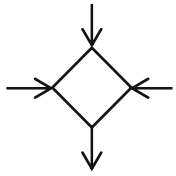
Tab. A1.3 Die wichtigsten Sequenzdiagramm-Elemente

	Eine **Strichfigur** stellt den Nutzer der Szene dar. Eine Szene kann aber auch durch das System angestoßen werden.
	In einem **Rechteck** wird Objektname und Klasse genannt, so wie in der Instanzspezifikation. Ein Pfeil an das Objekt kennzeichnet die Instanziierung. Eine **senkrechte gestrichelte Linie** stellt die Lebenslinie (lifeline) des Objekts dar. Die Lebenslinie endet mit gekreuzten Linien.
	Nachrichten werden als **Pfeile** eingezeichnet. Die Bezeichnung der Nachricht steht am Pfeil. Ein geschlossener Pfeil kennzeichnet eine synchrone Nachricht, ein geöffneter eine asynchrone Nachricht. Eine gestrichelte Linie kennzeichnet die Rückmeldung (Return).

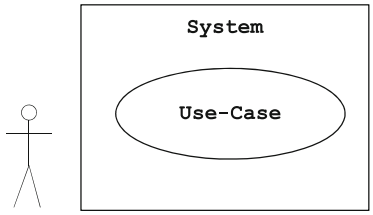
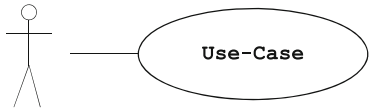
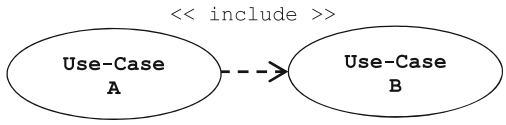
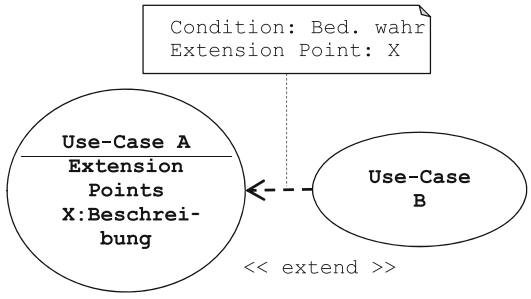
Tab. A1.4 Wichtige Teilszenen im Sequenzdiagramm

	Die **Instanziierung** eines Objekts wird durch eine Nachricht an den Rahmen des Objekts dargestellt. Damit beginnt die Lebenslinie unterhalb des Objekts als gestrichelte Linie. Ein X am Ende der Lebenslinie kennzeichnet ihr **Ende**.
	Ein geschlossener Pfeil kennzeichnet **synchrone** Nachrichten. Das Sender-Objekt wartet, bis das Empfänger-Objekt ein Return sendet und setzt dann die Verarbeitung fort. Der Aufruf hat einen Namen und in Klammern können Parameter (Name und Typ) angegeben werden.
	Ein geöffneter Pfeil kennzeichnet **asynchrone** Nachrichten. Das Sender-Objekt wartet nicht auf eine Rückmeldung, sondern setzt nach dem Senden die Verarbeitung fort.

Tab. A1.5 Die wichtigsten Aktivitätsdiagramm-Elemente

	Der **Startknoten** zeigt einen Eintrittspunkt in ein System. Ein System kann mehrere Eintrittspunkte besitzen.
	Der **Endknoten** ist das Gegenstück zum Startknoten. Er definiert den Austrittspunkt aus dem System. Ein System kann mehrere Austrittspunkte besitzen.
	Ein **Ablaufende** terminiert einen Pfad einer Aktivität, die Aktivität selbst läuft weiter.
	Eine **Aktion** ist eine Teilaktivität, die sich im Sinne des Aktivitätsdiagramms nicht weiter unterteilen lässt. Eine Aktion verbraucht Zeit und ändert das System.
	Eine **Kante** beschreibt den Fluss zwischen verbundenen Elementen. In eckigen Klammern kann eine Bedingung angegeben werden, die erfüllt sein muss, damit die Kante überquert wird.
	Eine **Splittung** teilt den aktuellen Pfad in parallel ablaufende Pfade.
	Eine **Synchronisation** führt parallel laufende Pfade wieder zusammen.
	An einer **Verzweigung** wird aufgrund von Regeln entschieden, welche weiteren Pfade getrennt ausgeführt werden.
	Eine **Zusammenführung** vereint zuvor getrennte Pfade wieder zu einem gemeinsamen Pfad.
	**Objektknoten** repräsentieren an einer Aktion beteiligte Objekte, wie z. B. Daten.

Tab. A1.6 Die wichtigsten Anwendungsfalldiagramm-Elemente

	Im Use-Case-Diagramm gibt es den **Akteur** und das **System**. Der Akteur ist der Anwender und das System die Software. Das System wird als Rechteck dargestellt und beinhaltet nur wesentliche Anforderungen, dargestellt in Ellipsen. Jede Ellipse enthält genau eine Funktion und wird knapp beschrieben.
	Beziehungen zwischen Use-Cases und Akteur werden durch Verbindungslinien, den Assoziationen dargestellt. Die Art der Assoziation wird durch den Linientyp dargestellt. Eine durchgezogene Linie ist eine Assoziation zwischen Akteur und Use-Case.
	Gestrichelte Linien stellen Assoziationen zwischen zwei Use-Cases dar. Neben die Linie wird ein Schlüsselwort in doppelt spitze Klammern gestellt. Sie werden als Stereotypen bezeichnet. Bei einer include-Assoziation wird B immer dann ausgeführt wenn auch A ausgeführt wird.
	Bei einer extend-Assoziation wird A abhängig von Bedingungen in B ausgeführt. Use-Case A ist um einen Erweiterungspunkt X ergänzt worden. Ein Use-Case kann beliebig viele Erweiterungspunkte besitzen. Die Bedingung zur Ausführung von Use-Case A wird durch einen Notizzettel an die Assoziation gesetzt.

Tab. A1.7 Eigenschaften und Methoden der Klasse Collection

Add (Item, Key, Before, After)	Hinzufügen eines Elements. Item: Element beliebigen Datentyps Key: **Textschlüssel** für den Zugriff Before, After: Relative Positionierung der Zufügung
Remove (Index)	Entfernt ein Element. Index: Key oder Zahl in der Liste
Item	1 <= n <= Collection-Object.Count oder Key der Liste
Count	Liefert die Anzahl der Elemente in der Liste.

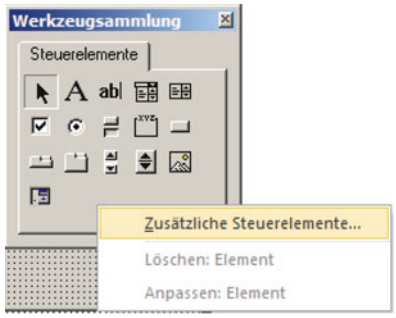
Tab. A1.8 Eigenschaften und Methoden der Klasse Dictionary

Add (Key, Item)	Hinzufügen eines Elements. Key: Beliebiger Schlüssel für den Zugriff Item: Element beliebigen Datentyps Achtung! Andere Folge wie bei Collection
Exists(Key)	Erlaubt die Abfrage eines Schlüssels. Wichtig! Liefert True oder False.
Items()	Liefert alle Elemente im Dictionary.
Keys()	Liefert einen Array-Container mit allen Keys im Dictionary.
Remove (Key)	Entfernt ein Element mit dem Schlüssel.
RemoveAll()	Löscht alle Schlüssel und Items im Dictionary.
Item(Key)[=newItem]	Liefert das Element mit dem Schlüssel. Ein nicht vorhandener Schlüssel wird neu angelegt.
Count	Liefert die Anzahl der Elemente in der Liste.

Tab. A1.9 Die wichtigsten ActiveX-Steuerelemente

Symbol Name	Beschreibung
 Label	Enthält Text zur Beschriftung von anderen Steuerelementen. Kann auch kurze Anweisungen enthalten.
 TextBox	Dient zur Ein- und Ausgabe von Daten in Textform.
 ComboBox	Stellt eine Kombination zwischen TextBox und Listenfeld dar. Ein angeklicktes Datenelement im Listenfeld erscheint in der TextBox.
 ListBox	Kann mehrere Textelemente enthalten, die zur Auswahl dienen. Es können auch mehrere Elemente ausgewählt werden.
 CheckBox	Das Kontrollfeld dient zur Wahl oder Abwahl einer Option. Es kann auch eine Mehrfachwahl getroffen werden
 OptionButton	Erlaubt die Auswahl einer einzigen Option aus mehreren Möglichkeiten, die auch in Frames zusammengefasst werden können.
 ToggleButton	Zeigt einen Zustand wie Ja/Nein oder An/Aus oder 0/1 und dient zur Umschaltung.
 Frame	Ein Rahmen, der mehrere Steuerelemente logisch zusammenfasst. Wird häufig auf Optionsfelder und Kontrollkästchen angewendet.

Tab. A1.9 (Fortsetzung)

Symbol Name	Beschreibung
CommandButton	Kann mit Klick eine VBA-Prozedur ausführen.
TabStrip	Erlaubt die Zusammenfassung von Steuerelementen auf einer Fläche.
MultiPage	Erlaubt die Zusammenfassung von Steuerelementen auf einer Seite.
ScrollBar	Dient zur Steuerung des sichtbaren Teils eines Steuerelements.
SpinButton	Dient zur Einstellung eines Wertes zwischen Minimum und Maximum.
Image	Dient zur Aufnahme eines Bildes.
	Mit dem Kontextmenü auf eine freie Fläche der Werkzeugsammlung können weitere Steuerelemente und Steuerelement-Bibliotheken aufgerufen werden.

Tab. A1.10 Verbindungsformen für Shapes

Verbindungsform	Konstante	Wert
Gerade Verbindung	msoConnectorStraight	1
Gewinkelte Verbindung	msoConnectorElbow	2
Gekrümmte Verbindung	msoConnectorCurve	3

Anhang 2

Tab. A2.1 Dialogfeld-Konstante und -Parameter

Dialogfeldkonstante	Argumentliste(n)
xlDialogActivate	window_text, pane_num
xlDialogActiveCellFont	font, font_style, size, strikethrough, superscript, subscript, outline, shadow, underline, color, normal, background, start_char, char_count
xlDialogAddChartAutoformat	name_text, desc_text
xlDialogAddinManager	operation_num, addinname_text, copy_logical
xlDialogAlignment	horiz_align, wrap, vert_align, orientation, add_indent
xlDialogApplyNames	name_array, ignore, use_rowcol, omit_col, omit_row, order_num, append_last
xlDialogApplyStyle	style_text
xlDialogAppMove	x_num, y_num
xlDialogAppSize	x_num, y_num
xlDialogArrangeAll	arrange_num, active_doc, sync_horiz, sync_vert
xlDialogAssignToObject	macro_ref
xlDialogAssignToTool	bar_id, position, macro_ref
xlDialogAttachText	attach_to_num, series_num, point_num
xlDialogAttachToolbars	
xlDialogAutoCorrect	correct_initial_caps, capitalize_days
xlDialogAxes	x_primary, y_primary, x_secondary, y_secondary
xlDialogAxes	x_primary, y_primary, z_primary
xlDialogBorder	outline, left, right, top, bottom, shade, outline_color, left_color, right_color, top_color, bottom_color
xlDialogCalculation	type_num, iter, max_num, max_change, update, precision, date_1904, calc_save, save_values, alt_exp, alt_form
xlDialogCellProtection	locked, hidden

Tab. A2.1 (Fortsetzung)

Dialogfeldkonstante	Argumentliste(n)
xlDialogChangeLink	old_text, new_text, type_of_link
xlDialogChartAddData	ref, rowcol, titles, categories, replace, series
xlDialogChartLocation	
xlDialogChartOptionsDataLabels	
xlDialogChartOptionsDataTable	
xlDialogChartSourceData	
xlDialogChartTrend	type, ord_per, forecast, backcast, intercept, equation, r_squared, name
xlDialogChartType	
xlDialogChartWizard	long, ref, gallery_num, type_num, plot_by, categories, ser_titles, legend, title, x_title, y_title, z_title, number_cats, number_titles
xlDialogCheckboxProperties	value, link, accel_text, accel2_text, 3d_shading
xlDialogClear	type_num
xlDialogColorPalette	file_text
xlDialogColumnWidth	width_num, reference, standard, type_num, standard_num
xlDialogCombination	type_num
xlDialogConditionalFormatting	
xlDialogConsolidate	source_refs, function_num, top_row, left_col, create_links
xlDialogCopyChart	size_num
xlDialogCopyPicture	appearance_num, size_num, type_num
xlDialogCreateNames	top, left, bottom, right
xlDialogCreatePublisher	file_text, appearance, size, formats
xlDialogCustomizeToolbar	category
xlDialogCustomViews	
xlDialogDataDelete	
xlDialogDataLabel	show_option, auto_text, show_key
xlDialogDataSeries	rowcol, type_num, date_num, step_value, stop_value, trend
xlDialogDataValidation	
xlDialogDefineName	name_text, refers_to, macro_type, shortcut_text, hidden, category, local
xlDialogDefineStyle	style_text, number, font, alignment, border, pattern, protection
xlDialogDefineStyle	style_text, attribute_num, additional_def_args, …
xlDialogDeleteFormat	format_text
xlDialogDeleteName	name_text
xlDialogDemote	row_col
xlDialogDisplay	formulas, gridlines, headings, zeros, color_num, reserved, outline, page_breaks, object_num

Tab. A2.1 (Fortsetzung)

Dialogfeldkonstante	Argumentliste(n)
xlDialogDisplay	cell, formula, value, format, protection, names, precedents, dependents, note
xlDialogEditboxProperties	validation_num, multiline_logical, vscroll_logical, password_logical
xlDialogEditColor	color_num, red_value, green_value, blue_value
xlDialogEditDelete	shift_num
xlDialogEditionOptions	edition_type, edition_name, reference, option, appearance, size, formats
xlDialogEditSeries	series_num, name_ref, x_ref, y_ref, z_ref, plot_order
xlDialogErrorbarX	include, type, amount, minus
xlDialogErrorbarY	include, type, amount, minus
xlDialogExternalDataProperties	
xlDialogExtract	unique
xlDialogFileDelete	file_text
xlDialogFileSharing	
xlDialogFillGroup	type_num
xlDialogFillWorkgroup	type_num
xlDialogFilter	
xlDialogFilterAdvanced	operation, list_ref, criteria_ref, copy_ref, unique
xlDialogFindFile	
xlDialogFont	name_text, size_num
xlDialogFontProperties	font, font_style, size, strikethrough, superscript, subscript, outline, shadow, underline, color, normal, background, start_char, char_count
xlDialogFormatAuto	format_num, number, font, alignment, border, pattern, width
xlDialogFormatChart	layer_num, view, overlap, angle, gap_width, gap_depth, chart_depth, doughnut_size, axis_num, drop, hilo, up_down, series_line, labels, vary
xlDialogFormatCharttype	apply_to, group_num, dimension, type_num
xlDialogFormatFont	color, backgd, apply, name_text, size_num, bold, italic, underline, strike, outline, shadow, object_id, start_num, char_num
xlDialogFormatFont	name_text, size_num, bold, italic, underline, strike, color, outline, shadow
xlDialogFormatFont	name_text, size_num, bold, italic, underline, strike, color, outline, shadow, object_id_text, start_num, char_num
xlDialogFormatLegend	position_num
xlDialogFormatMain	type_num, view, overlap, gap_width, vary, drop, hilo, angle, gap_depth, chart_depth, up_down, series_line, labels, doughnut_size
xlDialogFormatMove	x_offset, y_offset, reference

Tab. A2.1 (Fortsetzung)

Dialogfeldkonstante	Argumentliste(n)
xlDialogFormatMove	x_pos, y_pos
xlDialogFormatMove	explosion_num
xlDialogFormatNumber	format_text
xlDialogFormatOverlay	type_num, view, overlap, gap_width, vary, drop, hilo, angle, series_dist, series_num, up_down, series_line, labels, doughnut_size
xlDialogFormatSize	width, height
xlDialogFormatSize	x_off, y_off, reference
xlDialogFormatText	x_align, y_align, orient_num, auto_text, auto_size, show_key, show_value, add_indent
xlDialogFormulaFind	text, in_num, at_num, by_num, dir_num, match_case, match_byte
xlDialogFormulaGoto	reference, corner
xlDialogFormulaReplace	find_text, replace_text, look_at, look_by, active_cell, match_case, match_byte
xlDialogFunctionWizard	
xlDialogGallery3dArea	type_num
xlDialogGallery3dBar	type_num
xlDialogGallery3dColumn	type_num
xlDialogGallery3dLine	type_num
xlDialogGallery3dPie	type_num
xlDialogGallery3dSurface	type_num
xlDialogGalleryArea	type_num, delete_overlay
xlDialogGalleryBar	type_num, delete_overlay
xlDialogGalleryColumn	type_num, delete_overlay
xlDialogGalleryCustom	name_text
xlDialogGalleryDoughnut	type_num, delete_overlay
xlDialogGalleryLine	type_num, delete_overlay
xlDialogGalleryPie	type_num, delete_overlay
xlDialogGalleryRadar	type_num, delete_overlay
xlDialogGalleryScatter	type_num, delete_overlay
xlDialogGoalSeek	target_cell, target_value, variable_cell
xlDialogGridlines	x_major, x_minor, y_major, y_minor, z_major, z_minor, 2D_effect
xlDialogImportTextFile	
xlDialogInsert	shift_num
xlDialogInsertHyperlink	
xlDialogInsertNameLabel	
xlDialogInsertObject	object_class, file_name, link_logical, display_icon_logical, icon_file, icon_number, icon_label
xlDialogInsertPicture	file_name, filter_number

Tab. A2.1 (Fortsetzung)

Dialogfeldkonstante	Argumentliste(n)
xlDialogInsertTitle	chart, y_primary, x_primary, y_secondary, x_secondary
xlDialogLabelProperties	accel_text, accel2_text, 3d_shading
xlDialogListboxProperties	range, link, drop_size, multi_select, 3d_shading
xlDialogMacroOptions	macro_name, description, menu_on, menu_text, shortcut_on, shortcut_key, function_category, status_bar_text, help_id, help_file
xlDialogMailEditMailer	to_recipients, cc_recipients, bcc_recipients, subject, enclosures, which_address
xlDialogMailLogon	name_text, password_text, download_logical
xlDialogMailNextLetter	
xlDialogMainChart	type_num, stack, 100, vary, overlap, drop, hilo, overlap%, cluster, angle
xlDialogMainChartType	type_num
xlDialogMenuEditor	
xlDialogMove	x_pos, y_pos, window_text
xlDialogNew	type_num, xy_series, add_logical
xlDialogNewWebQuery	
xlDialogNote	add_text, cell_ref, start_char, num_chars
xlDialogObjectProperties	placement_type, print_object
xlDialogObjectProtection	locked, lock_text
xlDialogOpen	file_text, update_links, read_only, format, prot_pwd, write_res_pwd, ignore_rorec, file_origin, custom_delimit, add_logical, editable, file_access, notify_logical, converter
xlDialogOpenLinks	document_text1, document_text2, . . . , read_only, type_of_link
xlDialogOpenMail	subject, comments
xlDialogOpenText	file_name, file_origin, start_row, file_type, text_qualifier, consecutive_delim, tab, semicolon, comma, space, other, other_char, field_info
xlDialogOptionsCalculation	type_num, iter, max_num, max_change, update, precision, date_1904, calc_save, save_values
xlDialogOptionsChart	display_blanks, plot_visible, size_with_window
xlDialogOptionsEdit	incell_edit, drag_drop, alert, entermove, fixed, decimals, copy_objects, update_links, move_direction, autocomplete, animations
xlDialogOptionsGeneral	R1C1_mode, dde_on, sum_info, tips, recent_files, old_menus, user_info, font_name, font_size, default_location, alternate_location, sheet_num, enable_under
xlDialogOptionsListsAdd	string_array
xlDialogOptionsListsAdd	import_ref, by_row
xlDialogOptionsME	def_rtl_sheet, crsr_mvmt, show_ctrl_char, gui_lang

Tab. A2.1 (Fortsetzung)

Dialogfeldkonstante	Argumentliste(n)
xlDialogOptionsTransition	menu_key, menu_key_action, nav_keys, trans_eval, trans_entry
xlDialogOptionsView	formula, status, notes, show_info, object_num, page_breaks, formulas, gridlines, color_num, headers, outline, zeros, hor_scroll, vert_scroll, sheet_tabs
xlDialogOutline	auto_styles, row_dir, col_dir, create_apply
xlDialogOverlay	type_num, stack, 100, vary, overlap, drop, hilo, overlap%, cluster, angle, series_num, auto
xlDialogOverlayChartType	type_num
xlDialogPageSetup	head, foot, left, right, top, bot, hdng, grid, h_cntr, v_cntr, orient, paper_size, scale, pg_num, pg_order, bw_cells, quality, head_margin, foot_margin, notes, draft
xlDialogPageSetup	head, foot, left, right, top, bot, size, h_cntr, v_cntr, orient, paper_size, scale, pg_num, bw_chart, quality, head_margin, foot_margin, draft
xlDialogPageSetup	head, foot, left, right, top, bot, orient, paper_size, scale, quality, head_margin, foot_margin, pg_num
xlDialogParse	parse_text, destination_ref
xlDialogPasteNames	
xlDialogPasteSpecial	paste_num, operation_num, skip_blanks, transpose
xlDialogPasteSpecial	rowcol, titles, categories, replace, series
xlDialogPasteSpecial	paste_num
xlDialogPasteSpecial	format_text, pastelink_logical, display_icon_logical, icon_file, icon_number, icon_label
xlDialogPatterns	apattern, afore, aback, newui
xlDialogPatterns	lauto, lstyle, lcolor, lwt, hwidth, hlength, htype
xlDialogPatterns	bauto, bstyle, bcolor, bwt, shadow, aauto, apattern, afore, aback, rounded, newui
xlDialogPatterns	bauto, bstyle, bcolor, bwt, shadow, aauto, apattern, afore, aback, invert, apply, newfill
xlDialogPatterns	lauto, lstyle, lcolor, lwt, tmajor, tminor, tlabel
xlDialogPatterns	lauto, lstyle, lcolor, lwt, apply, smooth
xlDialogPatterns	lauto, lstyle, lcolor, lwt, mauto, mstyle, mfore, mback, apply, smooth
xlDialogPatterns	type, picture_units, apply
xlDialogPhonetic	
xlDialogPivotCalculatedField	
xlDialogPivotCalculatedItem	
xlDialogPivotClientServerSet	
xlDialogPivotFieldGroup	start, end, by, periods
xlDialogPivotFieldProperties	name, pivot_field_name, new_name, orientation, function, formats

Tab. A2.1 (Fortsetzung)

Dialogfeldkonstante	Argumentliste(n)
xlDialogPivotFieldUngroup	
xlDialogPivotShowPages	name, page_field
xlDialogPivotSolveOrder	
xlDialogPivotTableOptions	
xlDialogPivotTableWizard	type, source, destination, name, row_grand, col_grand, save_data, apply_auto_format, auto_page, reserved
xlDialogPlacement	placement_type
xlDialogPrint	range_num, from, to, copies, draft, preview, print_what, color, feed, quality, y_resolution, selection, printer_text, print_to_file, collate
xlDialogPrinterSetup	printer_text
xlDialogPrintPreview	
xlDialogPromote	rowcol
xlDialogProperties	title, subject, author, keywords, comments
xlDialogProtectDocument	contents, windows, password, objects, scenarios
xlDialogProtectSharing	
xlDialogPublishAsWebPage	
xlDialogPushbuttonProperties	default_logical, cancel_logical, dismiss_logical, help_logical, accel_text, accel_text2
xlDialogReplaceFont	font_num, name_text, size_num, bold, italic, underline, strike, color, outline, shadow
xlDialogRoutingSlip	recipients, subject, message, route_num, return_logical, status_logical
xlDialogRowHeight	height_num, reference, standard_height, type_num
xlDialogRun	reference, step
xlDialogSaveAs	document_text, type_num, prot_pwd, backup, write_res_pwd, read_only_rec
xlDialogSaveCopyAs	document_text
xlDialogSaveNewObject	
xlDialogSaveWorkbook	document_text, type_num, prot_pwd, backup, write_res_pwd, read_only_rec
xlDialogSaveWorkspace	name_text
xlDialogScale	cross, cat_labels, cat_marks, between, max, reverse
xlDialogScale	min_num, max_num, major, minor, cross, logarithmic, reverse, max
xlDialogScale	cat_labels, cat_marks, reverse, between
xlDialogScale	series_labels, series_marks, reverse
xlDialogScale	min_num, max_num, major, minor, cross, logarithmic, reverse, min
xlDialogScenarioAdd	scen_name, value_array, changing_ref, scen_comment, locked, hidden
xlDialogScenarioCells	changing_ref

Tab. A2.1 (Fortsetzung)

Dialogfeldkonstante	Argumentliste(n)
xlDialogScenarioEdit	scen_name, new_scenname, value_array, changing_ref, scen_comment, locked, hidden
xlDialogScenarioMerge	source_file
xlDialogScenarioSummary	result_ref, report_type
xlDialogScrollbarProperties	value, min, max, inc, page, link, 3d_shading
xlDialogSelectSpecial	type_num, value_type, levels
xlDialogSendMail	recipients, subject, return_receipt
xlDialogSeriesAxes	axis_num
xlDialogSeriesOptions	
xlDialogSeriesOrder	chart_num, old_series_num, new_series_num
xlDialogSeriesShape	
xlDialogSeriesX	x_ref
xlDialogSeriesY	name_ref, y_ref
xlDialogSetBackgroundPicture	
xlDialogSetPrintTitles	titles_for_cols_ref, titles_for_rows_ref
xlDialogSetUpdateStatus	link_text, status, type_of_link
xlDialogShowDetail	rowcol, rowcol_num, expand, show_field
xlDialogShowToolbar	bar_id, visible, dock, x_pos, y_pos, width, protect, tool_tips, large_buttons, color_buttons
xlDialogSize	width, height, window_text
xlDialogSort	orientation, key1, order1, key2, order2, key3, order3, header, custom, case
xlDialogSort	orientation, key1, order1, type, custom
xlDialogSortSpecial	sort_by, method, key1, order1, key2, order2, key3, order3, header, order, case
xlDialogSplit	col_split, row_split
xlDialogStandardFont	name_text, size_num, bold, italic, underline, strike, color, outline, shadow
xlDialogStandardWidth	standard_num
xlDialogStyle	bold, italic
xlDialogSubscribeTo	file_text, format_num
xlDialogSubtotalCreate	at_change_in, function_num, total, replace, pagebreaks, summary_below
xlDialogSummaryInfo	title, subject, author, keywords, comments
xlDialogTable	row_ref, column_ref
xlDialogTabOrder	
xlDialogTextToColumns	destination_ref, data_type, text_delim, consecutive_delim, tab, semicolon, comma, space, other, other_char, field_info
xlDialogUnhide	window_text
xlDialogUpdateLink	link_text, type_of_link

Tab. A2.1 (Fortsetzung)

Dialogfeldkonstante	Argumentliste(n)
xlDialogVbaInsertFile	filename_text
xlDialogVbaMakeAddIn	
xlDialogVbaProcedureDefinition	
xlDialogView3d	elevation, perspective, rotation, axes, height%, autoscale
xlDialogWebOptionsEncoding	
xlDialogWebOptionsFiles	
xlDialogWebOptionsFonts	
xlDialogWebOptionsGeneral	
xlDialogWebOptionsPictures	
xlDialogWindowMove	x_pos, y_pos, window_text
xlDialogWindowSize	width, height, window_text
xlDialogWorkbookAdd	name_array, dest_book, position_num
xlDialogWorkbookCopy	name_array, dest_book, position_num
xlDialogWorkbookInsert	type_num
xlDialogWorkbookMove	name_array, dest_book, position_num
xlDialogWorkbookName	oldname_text, newname_text
xlDialogWorkbookNew	
xlDialogWorkbookOptions	sheet_name, bound_logical, new_name
xlDialogWorkbookProtect	structure, windows, password
xlDialogWorkbookTabSplit	ratio_num
xlDialogWorkbookUnhide	sheet_text
xlDialogWorkgroup	name_array
xlDialogWorkspace	fixed, decimals, r1c1, scroll, status, formula, menu_key, remote, entermove, underlines, tools, notes, nav_keys, menu_key_action, drag_drop, show_info
xlDialogZoom	magnification

Tab. A2.2 Abgeleitete mathematische Funktionen

Name	Formel
Sekans	Sekans(X) = 1 / Cos(X)
Kosekans	Kosekans(X) = 1 / Sin(X)
Kotangens	Kotangens(X) = 1 / Tan(X)
Arkussinus	Arkussinus(X) = Atn(X / Sqr(−X * X + 1))
Arkuskosinus	Arkuskosinus(X) = Atn(−X / Sqr(−X * X + 1)) + 2 * Atn(1)
Arkussekans	Arkussekans(X) = Atn(X / Sqr(X * X − 1)) + Sgn((X) − 1) * (2 * Atn(1))
Arkuskosekans	Arkuskosekans(X) = Atn(X / Sqr(X * X − 1)) + (Sgn(X) − 1) * (2 * Atn(1))
Arkuskotangens	Arkuskotangens(X) = Atn(X) + 2 * Atn(1)
Hyperb. Sinus	HSin(X) = (Exp(X) − Exp(−X)) / 2
Hyperb. Kosinus	HCos(X) = (Exp(X) + Exp(−X)) / 2
Hyperb. Tangens	HTan(X) = (Exp(X) − Exp(−X)) / (Exp(X) + Exp(−X))
Hyperb. Sekans	HSekans(X) = 2 / (Exp(X) + Exp(−X))
Hyperb. Kosekans	HKosekans(X) = 2 / (Exp(X) − Exp(−X))
Hyperb. Kotangens	HKotangens(X) = (Exp(X) + Exp(−X)) / (Exp(X) − Exp(−X))
Hyperb. Arkussinus	HArkussinus(X) = Log(X + Sqr(X * X + 1))
Hyperb. Arkuskosinus	HArkuskosinus(X) = Log(X + Sqr(X * X − 1))
Hyperb. Arkustangens	HArkustangens(X) = Log((1 + X) / (1 − X)) / 2
Hyperb. Arkussekans	HArkussekans(X) = Log((Sqr(−X * X + 1) + 1) / X)
Hyperb. Arkuskosekans	HArkuskosekans(X) = Log((Sgn(X) * Sqr(X * X + 1) + 1) / X)
Hyperb. Arkuskotangens	HArkuskotangens(X) = Log((X + 1) / (X − 1)) / 2
Logarithmus zur Basis N	LogN(X) = Log(X) / Log(N)

Anhang 3

Tab. A3.1 Auflistung aller vordefinierten BuiltInDocumentProperties

Index	WdBuiltInProperty-Konstante	Bezeichnung der Eigenschaft	Format
1	wdPropertyTitle	Titel	Text
2	wdPropertySubject	Thema	Text
3	wdPropertyAuthor	Autor	Text
4	wdPropertyKeywords	Stichwörter	Text
5	wdPropertyComments	Kommentare	Text
6	wdPropertyTemplate	(Dokument)-Vorlage	Text
7	wdPropertyLastAuthor	letzter Autor	Text
8	wdPropertyRevision	Versionsnummer	Zahl
9	wdProperty AppName	Programmname	Text
10	wdPropertyTimeLastPrinted	zuletzt gedruckt am	Datum
11	wdPropertyTimeCreated	erstellt am	Datum
12	wdPropertyTimeLastSave	zuletzt gespeichert am	Datum
13	wdPropertyVBATotalEdit	Gesamtbearbeitungszeit	Zahl
14	wdPropertyPages	Anzahl der Seiten	Zahl
15	wdPropertyWords	Anzahl der Wörter	Zahl
16	wdPropertyCharacters	Anzahl der Zeichen ohne Leerzeichen	Zahl
17	wdPropertySecurity	Sicherheit (?)	Zahl
18	wdPropertyCategory	Kategorie	Text
19	wdPropertyFormat	Präsentationsformat	Text
20	wdPropertyManager	Manager	Text
21	wdPropertyCompany	Firma	Text
22	wdPropertyBytes	Anzahl der Bytes	Zahl
23	wdPropertyLines	Anzahl der Zeilen	Zahl
24	wdPropertyParas	Anzahl der Absätze	Zahl

Tab. A3.1 Auflistung aller vordefinierten BuiltInDocumentProperties

Index	WdBuiltInProperty-Konstante	Bezeichnung der Eigenschaft	Format
25	wdPropertySlides	Anzahl der Folien	Zahl
26	wdPropertyNnotes	Anzahl Notizen	Zahl
27	wdPropertyHiddenSlides	Anzahl der ausgeblendeten Folien	Zahl
28	wdProperty MMclips	Anzahl der Multimediaclips	Zahl
29	wdPropertyHyperlinkBase	Hyperlinkbasis	Text
30	wdPropertyCharsWSpaces	Anzahl Wörter mit Leerzeichen	Zahl

Tab. A3.2 What-Parameter

What-Parameter	Wert	Objekte
wdGoToBookmark	−1	Textmarken
wdGoToSection	0	Abschnitte
wdGoToLine	3	Zeilen
wdGoToObject	9	Objekte
wdGoToField	7	Felder
wdGoToPage	1	Seiten
wdGoToHeading	11	Überschriften
wdGoToFootNote	4	Fußnoten
wdGoToTable	2	Tabellen
wdGoToGraphic	8	Grafiken

Tab. A3.3 Which-Parameter

Which-Parameter	Wert	Position
wdGoToFirst	1	Erste
wdGoToLast	−1	Letzte
wdGoToNext	2	Nächste
wdGoToPrevious	3	Vorherige

Tab. A3.4 Unit-Parameter

Unit-Parameter	Wert	Objekt
wdStory	6	Dokument
wdLine	5	Zeile
wdColumn	9	Tabellenspalte
wdRow	10	Tabellenzeile

Tab. A3.5 Unit-Parameter für die Methoden MoveUp und MoveDown

Unit-Parameter	Wert	Objekt
wdLine	5	Zeile
wdParagraph	4	Absatz
wdWindow	11	Rand von ActiveWindow
wdScreen	7	Bildschirmseite

Tab. A3.6 Unit-Parameter für die Methoden MoveLeft und MoveRight

Unit-Parameter	Wert	Objekt
wdCharacter	1	Zeichen
wdWord	2	Wort
wdSentence	3	Satz
wdCell	12	Zelle der Tabelle

Tab. A3.7 Methoden zur Umrechnung von Maßeinheiten

Methode	Umrechnung nach Punkten
MillimetersToPoints	Millimeter in Points (1 mm = 2,84 pts)
CentimetersToPoints	Zentimeter in Points (1 cm = 28,35 pts)
LinesToPoints	Zeilen in Points (1 Zeile = 12 pts)
InchesToPoints	Zoll in Points (1 Zoll = 72 pts)

Tab. A3.8 Tabulatorarten

Name	Wert	Beschreibung
wdAlignTabBar	4	An Leiste ausgerichtet
wdAlignTabCenter	1	Zentriert
wdAlignTabDecimal	3	Dezimal ausgerichtet
wdAlignTabLeft	0	Linksbündig
wdAlignTabList	6	An Liste ausgerichtet
wdAlignTabRight	2	Rechtsbündig

Tab. A3.9 Formatvorlagetypen

Formatvorlagetyp	wdStyleType	Wert	Anwendung
Absatzformatvorlagen	wdStyleTypeParagraph	1	Absatzformate, zusätzlich Zeilenabstände und Textausrichtung
Zeichenformatvorlagen	wdStyleTypeCharacter	2	Zeichenformate, die Formatvorlage enthält Schriftart, Schriftgröße, Schriftform, etc.
Tabellenformatvorlagen	wdStyleTypeTable	3	Tabellenformate, Textausrichtungen und Tabellengestaltung
Listenformatvorlagen	wdStyleTypeList	4	Listenformate für Nummerierungen und Aufzählformen

Tab. A3.10 Einige Eigenschaften einer Formatvorlage

Eigenschaft	Auswirkung
BaseStyle	Gibt die Basis-Formatvorlage an, auf die diese Formatvorlage basiert
NameLocal	Name der Formatvorlage in der Landessprache
AutomaticallyUpdate	Ist die automatische Aktualisierung eingestellt
Font	Zeichenformatierung als Font-Objekt
InUse	Wird die Formatvorlage verwendet
ParagraphFormat:	Absatzformatierung als ParagraphFormat-Objekt
LanguageID	Bezeichnung nach der Sprach-ID (wdGerman = 1031)

Tab. A3.11 Parameter für die Eigenschaft DefaultFilePath

DefaultFilePath Parameter	Wert	Anwendung
wdAutoRecoverPath	5	Pfad für AutoWiederherstellen-Dateien
wdBorderArtPath	19	Pfad für Zierrahmen
wdCurrentFolderPath	14	Pfad für aktuellen Ordner
wdDocumentsPath	0	Pfad für Dokumente
wdGraphicsFiltersPath	10	Pfad für Grafikfilter
wdPicturesPath	1	Pfad für Bilder
wdProgramPath	9	Pfad des Programms
wdProofingToolsPath	12	Pfad für Korrekturhilfen
wdStartupPath	8	Startpfad
wdStyleGalleryPath	15	Pfad für Formatvorlagenkatalog
wdTempFilePath	13	Pfad für temporäre Dateien
wdTextConvertersPath	11	Pfad für Textkonverter
wdToolsPath	6	Pfad für Tools
wdTutorialPath	7	Pfad für Lernprogramme
wdUserOptionsPath	4	Pfad für Benutzeroptionen
wdUserTemplatesPath	2	Pfad für Benutzervorlagen
wdWorkgroupTemplatesPath	3	Pfad für Arbeitsgruppenvorlagen

Anhang 4

Tab. A4.1 Konstanten der Eigenschaft WindowState

PPWindowKonstante	Wert	Beschreibung
ppWindowNormal	1	Normalform
ppWindowMinimized	2	Symbolgröße
ppWindowMaximized	3	Vollbildmodus

Tab. A4.2 Parameter der Methode Open

Parameter	Typ	Angabe	Beschreibung
FileName	String	Erforderlich	Dateiname
ReadOnly	msoTrue/msoFalse	Optinal	Lese-/Schreibzugriff
Untitled	msoTrue/msoFalse	Optinal	Datei besitzt Titel
WithWindow	msoTrue/msoFalse	Optinal	Datei sichtbar

Tab. A4.3 Wichtige ppSlideLayout-Konstante

Layout	Wert	Beschreibung
ppLayoutTitle	1	Titel- und Inhalt
ppLayoutTitleOnly	11	Titel
ppLayoutBlank	12	Leere Folie
ppLayoutText	2	Titel mit Textaufzählungen
ppLayoutTable	4	Titel mit Tabelle
ppLayoutChart	8	Titel mit Diagramm

Tab. A4.4 Parameter für Farbverläufe

Parameter	Wert	Beschreibung der Farbverläufe
msoGradientDiagonalDown	4	diagonaler Farbverlauf von oben nach unten
msoGradientDiagonalUp	3	diagonaler Farbverlauf von unten nach oben
msoGradientFromCenter	7	von der Mitte nach außen
msoGradientFromCorner	5	von einer Ecke zu den anderen Ecken
msoGradientFromTitle	6	vom Titel nach außen
msoGradientHorizontal	1	horizontal über die Form
msoGradientMixed	−2	gemischter Farbverlauf
msoGradientVertical	2	vertikal über die Form

Tab. A4.5 ShowType-Parameter

Parameter	Wert	Beschreibung
ppShowTypeKiosk	3	Einzelvollbild
ppShowTypeSpeaker	1	Standard, Vollbild mit Durchlauf
ppShowTypeWindow	2	Ausführung in einem Fenster

Tab. A4.6 Konstante zur Folienumschaltung

Parameter	Wert	Beschreibung
ppSlideShowManualAdvance	1	manuelles Umschalten
ppSlideShowRehearseNewTimings	3	getestete Anzeigedauer
ppSlideShowUseSlideTimings	2	angegebene Anzeigedauer für jede Folie

Tab. A4.7 ShowPointerType Konstante

Parameter	Wert	Beschreibung
ppSlideShowPointerAlwaysHidden	3	immer ausgeblendet
ppSlideShowPointerArrow	1	Pfeil
ppSlideShowPointerAutoArrow	4	Auto Pfeil
ppSlideShowPointerNone	0	kein Zeiger
ppSlideShowPointerPen	2	Stift

Tab. A4.8 PrintColorType Konstante

Parameter	Wert	Beschreibung
ppBlackAndWhite	2	druckt in Graustufen
ppPrintColor	1	druckt in Farbe
ppPrintPureBlackAndWhite	3	druckt nur in Schwarz und Weiß

Tab. A4.9 PrintOutputType Konstante

Parameter	Wert	Beschreibung
ppPrintOutputBuildSlides	7	druckt Folien
ppPrintOutputNotesPages	5	druckt Notizen
ppPrintOutputOutline	6	druckt Gliederung
ppPrintOutputSlides	1	Standarddruck Folien
ppPrintOutputOneSlideHandouts	10	druckt Handzettel mit 1 Folie je Seite
ppPrintOutputTwoSlideHandouts	2	druckt Handzettel mit 2 Folien je Seite
ppPrintOutputThreeSlideHandouts	3	druckt Handzettel mit 3 Folien je Seite
ppPrintOutputSixSlideHandouts	4	druckt Handzettel mit 6 Folien je Seite
ppPrintOutputNineSlideHandouts	9	druckt Handzettel mit 9 Folien je Seite

Anhang 5

Tab. 5.1 Outlook Elementtyp Konstante

Name	Wert	Objekt
olAppointmentItem	1	Termineintrag
olContactItem	2	Kontakteintrag
olDistributionListItem	7	Verteilerliste
olJournalItem	4	Journaleintrag
olMailItem	0	Mail
olMobileItemMMS	9	Mail als MMS-Nachricht (Multimedia Messaging Service)
olMobileItemSMS	8	Mail als SMS-Nachricht (Short Message Service)
olNoteItem	5	Notizeintrag
olPostItem	6	Öffentlicher Ordner
olTaskItem	3	Aufgabeneintrag

Tab. 5.2 NameSpace Get-Methoden

NameSpace Methoden	Beschreibung
GetAddressEntryFromID	Liefert AdressEntry Objekt
GetDefaultFolder	Liefert Folder Objekt nach FolderType-Angabe
GetFolderFromID	Liefert Folder Objekt nach EntryID
GetGlobalAddressList	Liefert AddressList Objekt
GetItemFromID	Liefert Outlook Element nach EntryID
GetRecipientFromID	Liefert Recipient Objekt nach EntryID
GetSelectNamesDialog	Liefert SelectNamesDialog Objekt
GetSharedDefaultFolder	Liefert Folder Objekt
GetStoreFromId	Liefert Store nach ID

Tab. 5.3 Syntax für Zeitabfragen

Filter auf	Syntax
Heute	%today("*SchemaName*")%
Morgen	%tomorrow("*SchemaName*")%
Gestern	%yesterday("*SchemaName*")%
Nächsten 7 Tage	%next7days("*SchemaName*")%
Letzten 7 Tage	%last7days("*SchemaName*")%
Nächste Woche	%nextweek("*SchemaName*")%
Aktuelle Woche	%thisweek("*SchemaName*")%
Letzte Woche	%lastweek("*SchemaName*")%
Nächsten Monat	%nextmonth("*SchemaName*")%
Aktuellen Monat	%thismonth("*SchemaName*")%
Letzten Monat	%lastmonth("*SchemaName*")%

Tab. 5.4 Konstante für den CalendarViewMode

Name	Wert	Beschreibung
olCalendarView5DayWeek	4	Zeigt eine 5-Tage-Woche an.
olCalendarViewDay	0	Zeigt einen einzelnen Tag an.
olCalendarViewMonth	2	Zeigt einen Monat an.
olCalendarViewMultiDay	3	Zeigt eine Anzahl von Tagen an, die dem DaysInMultiDayMode-Eigenschaftswert des CalendarView-Objekts entspricht.
olCalendarViewWeek	1	Zeigt eine 7-Tage-Woche an.

Tab. 5.5 Ordnertyp-Konstante

Name	Wert	Ordner
olFolderCalendar	9	Kalender
olFolderContacts	10	Kontakte
olFolderDeletedItems	3	Gelöschte Objekte
olFolderDrafts	16	Entwürfe
olFolderInbox	6	Posteingang
olFolderJournal	11	Journal
olFolderJunk	23	Junk-E-Mail
olFolderNotes	12	Notizen
olFolderOutbox	4	Postausgang
olFolderSentMail	5	Gesendete Elemente
olFolderSuggestedContacts	30	Vorgeschlagene Kontakte
olFolderTasks	13	Aufgaben
olFolderToDo	28	Aufgaben
olFolderRssFeeds	25	RSS-Feeds

Anhang 6

Tab. A6.1 Datentypen in SQL

Datentyp	Byte(s)	Beschreibung
BINARY	1	Ein beliebiges Zeichen
BIT	1	Ja/Nein
TINYINT	1	Ganze Zahl 0 bis 255
SMALLINT	2	Short Integer Datentyp
TEXT	2/Zeichen	0 bis max. 2,14 GByte
CHARACTER	2/Zeichen	0 bis 255 / Zeichen
REAL	4	Gleitkommazahl, einfache Genauigkeit
INTEGER	4	Ganze Zahl, long integer
MONEY	8	Währung
DATETIME	8	Datum/Uhrzeit
FLOAT	8	Gleitkommazahl, doppelte Genauigkeit
UNIQUEIDENTIFIER	16	Eindeutige ID Nummer
DESIMAL	17	Numerischer Wert 1028 −1 bis −1028 −1
IMAGE	Wie erforderlich	0 bis max. 2,14 GByte für OLE Objekte

Tab. A6.2 Methoden zu Objektdarstellungen

Methode	Objekt
OpenDiagramm	Diagramm
OpenForm	Formular
OpenModule	Modul
OpenQuery	Abfrage
OpenReport	Report
OpenStoredProcedure	Gespeicherte Prozedur
OpenTable	Tabelle
OpenView	Ansicht

Tab. A6.3 Konstante zu Formularansichten

Name	Wert	Ansicht
acDesign	1	Entwurfsansicht
acFormDS	3	Datenblattansicht
acFormPivotChart	5	PivotChart-Ansicht
acFormPivotTable	4	PivotTable-Ansicht
acLayout	6	Layoutansicht
acNormal	0	Formularansicht (Standard)
acPreview	2	Seitenansicht

Tab. A6.4 Die wichtigsten Methoden des DoCmd-Objekts

DoCmd	Aktion
.OpenForm	Formular öffnen
.OpenReport	Bericht öffnen
.SetFilter	Filter in Datenblatt/Formular/Bericht/Tabelle setzen
.Close	aktives Fenster schließen
.RunCommand	Menübefehl ausführen
.CancelEvent	Ereignis abbrechen
.GoToControl	zu Steuerelement gehen
.GoToRecord	zu Datensatz gehen
.FindRecord	Datensatz suchen
.FindNext	weiter suchen
.SelectObject	Objekt auswählen
.RunMacro	Makro ausführen
.Quit	beendet Access (mit Optionen)
.SetWarnings	Systemmeldungen aktivieren/deaktivieren
.Beep	Signalton erzeugen

Tab. A6.5 Zugriff auf Objektlisten über die CurrentData-Eigenschaft

Objektliste	Inhalt
AllTables	Tabellen
AllViews	Ansichten (nur Access Objekte)
AllQueries	Abfragen (nur Access Datenbanken)
AllStoredProcedures	Prozeduren (nur Access Projekte)
AllDatabaseDiagrams	Datenbankdiagramme (nur Access Projekte)

Tab. A6.6 Zugriff auf Objektlisten über die CurrentProject-Eigenschaft

Objektliste	Inhalt
AllForms	Formulare
AllReports	Berichte
AllMacros	Makros
AllModules	Module
AllDataAccessPages	Datenzugriffsseiten

Tab. A6.7 Suchfunktionen

Funktion	Beschreibung
DMax	Maximalwert
DM;in	Minimalwert
DAvg	Mittelwert
DSum	Summe
DCount	Anzahl
DFirst	Erster Datensatz
DLast	Letzter Datensatz
dLookup	Datensatzkriterium

Tab. A6.8 Die Grenzwerte des Datensatzzeigers

Konstante	Beschreibung
BOF	Ist TRUE, wenn der Datensatzzeiger vor den ersten Datensatz zeigt, sonst FALSE.
EOF	Ist TRUE, wenn der Datensatzzeiger hinter den letzten Datensatz zeigt, sonst FALSE.

Tab. A6.9 Methoden zur Positionierung des Datensatzzeigers

Methode	Setzt den Datensatzzeiger …
Move	um die angegebene Anzahl Datensätze auf (+) oder ab (−).
MoveFirst	auf den ersten Datensatz.
MoveLast	auf den letzten Datensatz.
MovePrevious	auf den vorherigen Datensatz.
MoveNext	auf den nachfolgenden Datensatz.

Tab. A6.10 Recordset Typen

Konstante	Beschreibung
dbOpenTable	Instanziiert Tabellen-Recordset. Änderungen möglich.
dbOpenDynaset	Instanziiert Abfrage-Recordset. Änderungen evtl. möglich.
dbOpenSnapshot	Instanziiert Snapshot-Recordset über Datenzustand zur Instanziierung. Änderung nicht möglich.
dbOpenForwardOnly	Instanziiert Vorwärts-Recordset. Änderung nicht möglich.

Tab. A6.11 Recordset Optionen

Konstante	Wert	Beschreibung
dbDenyWrite	1	Datensätze änderbar
dbDenyRead	2	Datensätze lesbar
dbReadOnly	4	Schreibgeschützt
dbAppendOnly	8	Datensätze hinzufügbar, vorhandene nicht lesbar
dnInconsistent	16	Aktualisierung für alle Dynaset-Felder
dbConsistent	32	Aktualisierung für alle Felder ohne Auswirkung auf andere
dbSQLPassThrough	64	Sendet SQL-Anweisung an ODBC-Datenbank
dbFailOnError	128	Rollback für Aktualisierungen wenn Fehler auftritt
dbForwardOnly	256	Vorwärts Recordset vom Typ Snapshot
dbSeeChanges	512	Generiert Laufzeitfehler wenn andere User Daten ändern
dbRunAsync	1024	Führt Abfrage asynchron aus
dbExecDirect	2048	Führt Abfrage aus ohne ODBC-Funktion SQLPrepare

Tab. A6.12 Recordset Modus

Konstante	Wert	Beschreibung
dbOptimisticValue	1	Vollständige Parallelität basierend auf Datenwerten
dbPessimistic	2	Eingeschränkte Parallelität
dbOptimistic	3	Vollständige Parallelität basierend auf Datensatz-ID
dbOptimisticBatch	5	Vollständige Batchaktualisierungen (nur ODBC)

Tab. A6.13 Methoden zur Positionierung des Datensatzzeigers

Methode	Sucht mit den angegebenen Kriterien den …
FindFirst	ersten Datensatz
FindLast	letzten Datensatz
FindPrevious	vorherigen Datensatz
FindNext	nächsten Datensatz

Tab. A6.14 Parameter der Methode Execute des Command-Objekts

Parameter (alle optional)	Beschreibung
RecordsAffected	Ein Long-Type der die Anzahl der Datensätze ausgibt, auf die sich die Operation ausgewirkt hat
Parameters	Parameterwerte in einer SQL-Anweisung
Options	Ein Long-Type der die Art der Auswertung vorschreibt. Siehe Konstante CommandTypeEnum im Objektkatalog.

Tab. A6.15 Parameter der Methode Open des Recordset-Objekts

Parameter (alle optional)	Beschreibung
Source	Ein Variant-Datentyp, der ein gültiges Command-Objekt, eine SQL-Anweisung, einen Tabellennamen, den Aufruf einer gespeicherten Prozedur, eine URL oder den Namen einer Datei oder eines Stream-Objekts ergibt
ActiveConnection	Ein Variant-Datentyp, der Name einer gültigen Connection-Objektvariablen ist, ein String-Wert, oder der ConnectionString-Parameter enthält
CursorType	Ein CursorTypeEnum-Wert, bestimmt den Cursortyp, der zum Öffnen des Recordset-Objekts verwenden soll. Der Standardwert ist adOpenForwardOnly.
LockType	Ein LockTypeEnum-Wert, bestimmt den Sperrtyp (vollständige Parallelität), der zum Öffnen des Recordset-Objekts verwenden sollte. Der Standardwert lautet adLockReadOnly.
Options	Ein Long-Wert, der angibt, wie das Argument Source auszuwerten ist, wenn es ein anderes Element als ein Command-Objekt darstellt, oder dass das Recordset-Objekt aus einer Datei wiederhergestellt werden sollte, in der es zuvor gespeichert wurde. Es kann sich um einen oder mehrere CommandTypeEnum-Werte oder ExecuteOptionEnum-Werte handeln, die mit einem bitweisen AND-Operator kombiniert werden können.

Tab. A6.16 Parameter der Methode Open des Recordset-Objekts

Parameter	Beschreibung
Criteria	Ein String-Typ der den Spaltennamen, den Vergleichsoperator und den für die Suche zu verwendenden Wert angibt.
SkipRows (optinal)	Ein Long-Typ mit Standardwert 0, der den Zeilenversatz von der aktuellen Zeile oder Start-Textmarke aus gesehen angibt, bei der mit der Suche begonnen werden soll. Ohne Angabe startet die Suche bei der aktuellen Zeile.
SearchDirection (optinal)	Der *SearchDirectionEnum*-Wert gibt an, ob die Suche bei der aktuellen Zeile oder der nächsten verfügbaren Zeile in der Suchrichtung begonnen werden soll. Bei nicht erfolgreicher Suche wird die Suche am Ende des Recordset-Objekts abgebrochen, wenn der Wert *adSearchForward* ist. Bei nicht erfolgreichen Suche wird die Suche am Anfang des Recordset-Objekts abgebrochen, wenn der Wert *adSearchBackward* ist.
Start (optional)	Eine Variant-Textmarke, die als Startposition für die Suche fungiert.

Tab. A6.17 Bereichskonstante

Bereichskonstante	Index	Beschreibung
acDetail	0	Detailbereich
acHeader	1	Kopfbereich
acFooter	2	Fußbereich

Tab. A6.18 Wichtige Filter- und Datenoperations-Ereignisse eines Formulars

Ereignis	Auslösung
AfterBeginTransaction	nach Transaktionsbeginn
AfterDelConfirm	nach Löschbestätigung
AfterInsert	nach Eingabe
AfterUpdate	nach Aktualisierung
ApplyFilter	bei Filteranwendung
BeforeBeginTransaction	vor Transaktionsbeginn
BeforeDelConfirm	vor Löschbestätigung
BeforeInsert	vor Eingabe
BeforeUpdate	vor Aktualisierung
Current	bei Anzeige
DataChange	bei Datenänderung
Dirty	bei Änderung
Delete	bei Löschung
Filter	bei Filter
OnConnect	bei Verbindung
OnDisconnect	bei Trennung

Tab. A6.19 Konstante zur Startansicht eines Formulars

Konstante	Wert	Darstellungsform
acDefViewContinuous	1	Endlosformular
acDefViewDatasheet	2	Tabellenblatt
acDefViewPivotChart	4	PivotChart
acDefViewPivotTable	3	PivotTable
acDefViewSingle	0	Datensatz
acDefViewSplitForm	5	geteiltes Formular

Tab. A6.20 Konstante für den View-Parameter

Konstante	Wert	Beschreibung
acViewNormal	0	Druckansicht
acViewDesign	1	Entwurfsansicht
acViewPreview	2	Seitenansicht
acViewPivotTable	3	PivotTable-Ansicht
acViewPivotChart	4	PivotChart-Ansicht
acViewReport	5	Berichtansicht
acViewLayout	6	Layoutansicht

Tab. A6.21 Konstante für den TransferType

Konstante	Wert	Beschreibung
acImport	0	Die Daten werden importiert.
acExport	1	Die Daten werden exportiert.
acLink	2	Die Datenbank wird mit der angegebenen Datenquelle verknüpft.

Tab. A6.22 Einige Konstante für den SpreadsheetType

Konstante	Wert	Beschreibung
acSpreadsheetTypeExcel9	8	Excel xls-Format
acSpreadsheetTypeExcel12	9	?
acSpreadsheetTypeExcel12Xml	10	Excel xlsx-Format
acSpreadsheetTypeLotusWK3	3	Lotus 1-2-3 WK3-Format
acSpreadsheetTypeLotusWK4	7	Lotus 1-2-3 WK4-Format

Tab. A6.23 Konstante für die Eigenschaft Orientation in einer Pivot-Tabelle

Konstante	Wert	Beschreibung
xlRowField	1	Zeilenbelegung
xlColumnField	2	Spaltenbelegung
xlPageField	3	Seitenbelegung
xlDataField	4	Datenfeldbelegung

Tab. A6.24 Die wichtigsten Objekttypen für die Methode OutputTo

Konstante	Wert	Beschreibung
acOutputTable	0	Tabelle
acOutputQuery	1	Abfrage
acOutputForm	2	Formular
acOutputReport	3	Bericht

Anhang 7

Tab. A7.1 Verschiedene Print-Methoden des ActiveDocuments

Parameter	Beschreibung
PrintCenteredH	Horizontal zentriert
PrintCenteredV	Vertikal zentriert
PrintFitOnPages	An Seite anpassen
PrintPagesAcross	Über Seiten hinweg
PrintPagesDown	Von oben nach unter drucken
PrintLandscape	Im Querformat
PrintScale	Druckskala

Tab. A7.2 Eigenschaften der Druckränder

Eigenschaft	Beschreibung
LeftMargin	Linker Rand
RightMargin	Rechter Rand
TopMargin	Oberer Rand
BottomMargin	Unterer Rand

Tab. A7.3 Einige wichtige Maßkonstante

Maßkonstante	Wert	Beschreibung
visCentimeters	69	Zentimeter
visMillimeters	70	Millimeter
visMeters	71	Meter
visPoints	50	Punkte

Tab. A7.4 Parameter der DrawLine-Methode

Parameter	Beschreibung
xBegin	x-Koordinate des Anfangspunktes
yBegin	y-Koordinate des Anfangspunktes
xEnd	x-Koordinate des Endpunktes
yEnd	y-Koordinate des Endpunktes

Tab. A7.5 Parameter der DrawRextangle-Methode

Parameter	Beschreibung
xBegin	x-Koordinate der linken oberen Ecke
yBegin	y-Koordinate der linken oberen Ecke
xEnd	x-Koordinate der rechten unteren Ecke
yEnd	y-Koordinate der rechten unteren Ecke

Tab. A7.6 Draw Methoden zur Kurvendarstellung

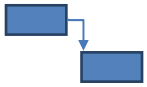
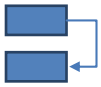
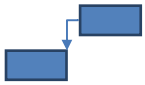
Draw Methode	Beschreibung
DrawBezier	Linie durch vorgegebene Bezier-Kontrollpunkte
DrawCircularArc	Kreisbogen, festgelegt durch Mittelpunkt, Radius, Anfangs- und Endpunkt
DrawNURBS	Bogen der aus einem Nonuniform Rational B-Spline besteht
DrawPolyline	Verbindungslinie durch Punkte
DrawQuarterArc	Elliptischer Bogen durch zwei Punkte
DrawSpline	Approximationslinie durch Punkte

Tab. A7.7 VisSelectionTypes-Konstante

Konstante	Wert	Beschreibung
visSelTypeEmpty	0	Enthält keine Shapes
visSelTypeAll	1	Alle Shapes
visSelTypeSingle	2	Ein Shape
visSelTypeByLayer	3	Alle Shapes einer Schicht
visSelTypeByType	4	Alle Shapes eines Types
visSelTypeByMaster	5	Aller Shapes eines Masters
visSelTypeByDataGraphics	6	Alle Shapes mit Datagrafik
visSelTypeByRole	7	Alle Shapes einer Rolle

Anhang 8

Tab. A8.1 Verknüpfungsarten zwischen Vorgängen

Verknüpfungsart	Beschreibung	Darstellung im Gantt-Diagramm
EA Ende-Anfang	Vorgang beginnt wenn Vorgänger endet	
AA Anfang-Anfang	Vorgang beginnt nur wenn Vorgänger ebenfalls beginnt	
EE Ende-Ende	Vorgang endet wenn Vorgänger endet	
AE Anfang-Ende	Vorgang endet erst wenn Vorgänger beginnt	

Literatur

1. Balzert, Heide, UML 2 kompakt mit Checklisten, 2. Auflage, Spektrum Akademischer Verlag, 2005
2. Weilkins, Tim, System Engineering mit SysMl/UML, 2. Auflage, dpunkt.verlag, 2008
3. Nahrstedt, Harald, Excel + VBA für Maschinenbauer, 4. Auflage, Springer Vieweg, 2013
4. Freßdorf, Christian / Gahler, Thomas / Meister, Cindy, Mikrosoft Word Programmierung, 4. Auflage, Microsoft Press, 2014
5. Mansfield, Richard, Mastering VBA for Office 2010, 2. Auflage, John Wiley & Sons, 2010
6. Slovak, Ken, Outlook 2007 Programming, John Wiley & Sons, 2007
7. Hölscher, Lorenz, Access 2010 VBA-Programming, Microsoft Press, 2010
8. Roth, Chris, Using Microsoft Visio 2010, Pearson Education, 2011
9. Gill, Rod, VBA Programming for Microsoft Project ’98 Through 2010 with an Introduction to VSTO, Msprojectexperts, 2011

Index Programmierung

Index Anwendungen