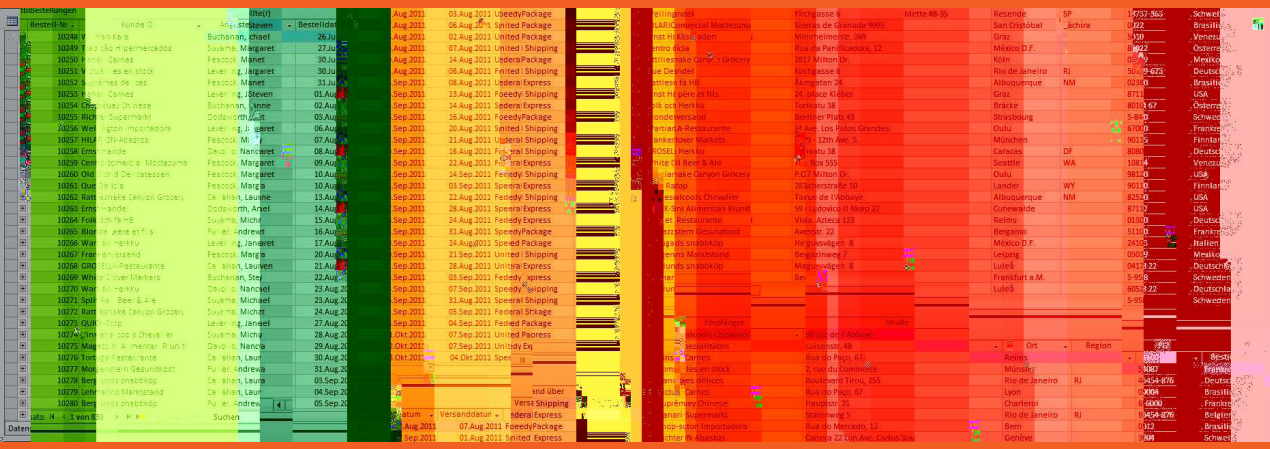
Leseprobe Stand 13.2.2015

ACCESS UND OFFICE



INTERAKTION ZWISCHEN ACCESS UND OFFICE-ANWENDUNGEN WIE EXCEL, OUTLOOK UND WORD

ANDRÉ MINHORST

André Minhorst Verlag

Leseprobe Stand 13.2.2015

Leseprobe Stand 13.2.2015

André Minhorst

Access und Office

**Interaktion zwischen
Access und Office-Anwendungen
wie Word, Excel, Outlook und PowerPoint**

Leseprobe Stand 13.2.2015

André Minhorst – Access und Office

ISBN 978-3-944216-03-4

1. Auflage 2015

Lektorat André Minhorst
Cover/Titelbild André Minhorst
Typographie, Layout und Satz André Minhorst
Herstellung André Minhorst
Druck und Bindung Kösel, Krugzell (www.koeselbuch.de)

Bibliografische Information der Deutschen Nationalbibliothek

Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliographie. Detaillierte bibliografische Daten finden Sie im Internet unter http://dnb.d-nb.de.

Leseprobe Stand 13.2.2015

Zeit ohne Zeit.

Leseprobe Stand 13.2.2015

Inhalt

Vorwort **15**

1 Datenaustausch zwischen Access und den Office-Anwendungen 19

1.1 Export nach Excel 19
1.1.1 Export-Einstellungen 19
1.1.2 Exportschritte speichern 22
1.1.3 Export nach Excel per Kontextmenü 25
1.1.4 Formulare nach Excel exportieren 27
1.1.5 Berichte nach Excel exportieren 27
1.1.6 Daten in Excel einbinden 29
1.2 Export nach Word 34
1.3 Export nach Word: Seriendruck 37
1.4 Import von Excel 47
1.4.1 Import in eine neue Tabelle 48
1.4.2 Import in eine bestehende Tabelle 51
1.4.3 Erstellen einer Verknüpfung 52
1.4.4 Excel-Daten als Abfrage verknüpfen 53
1.5 Import von Outlook 56
1.5.1 E-Mails importieren 58
1.5.2 Kontakte importieren 60
1.5.3 Termine importieren 61
1.5.4 E-Mails, Kontakte und Termine verknüpfen 62
1.5.5 Unterschiede je nach Installation 62

2 Datenaustausch mit VBA-Bordmitteln 65

2.1 Assistenten aufrufen 65
2.1.1 Excel-Import 66
2.1.2 Excel-Export 66
2.1.3 Word-Export 66
2.1.4 Outlook-Import 66
2.1.5 Word-Seriendruck 66
2.2 DoCmd.TransferSpreadsheet 67
2.2.1 Export nach Excel 69
2.2.2 Import von Excel 69
2.2.3 Verknüpfen von Excel-Daten 70
2.2.4 Bereiche importieren 71
2.3 DoCmd.TransferText 73
2.4 Zugriff per SQL 75
2.4.1 Excel-Tabelle per DAO durchlaufen 76
2.4.2 Aktionsabfragen auf Excel-Tabellen 77
2.4.3 Daten kopieren per AddNew 77
2.5 Access-Daten als Verknüpfung in Excel anzeigen 79

3 VBA-Zugriff auf Office 85
3.1 Early Binding 85
3.1.1 Vorteil: Zugriff per Objektkatalog 85
3.1.2 Vorteil: Nutzung von IntelliSense 87
3.1.3 Vorteil: Nutzung von Konstanten statt von Zahlenwerten 87
3.1.4 Vorteil: Ereignisse implementierbar 88
3.1.5 Vorteil: Performance 89
3.1.6 Nachteil: Verschiedene Versionen 89
3.2 Late Binding 91
3.2.1 Nachteile: Vorteile von Early Binding 91
3.2.2 Vorteil: Keine Probleme mit Verweisen 91
3.3 Stabile Verweise dank ACCDE 91
3.4 Von Early Binding zu Late Binding 92

4 Excel programmieren 95
4.1 Warum Excel programmieren? 95
4.2 Zugriff auf Excel 96
4.3 Das Application-Objekt 96
4.3.1 Excel-Instanz erstellen per Early Binding 97
4.3.2 Mehrere Instanzen 97
4.3.3 Bestehende Instanz referenzieren 99
4.3.4 Funktion zum Erzeugen einer Excel-Instanz 101
4.3.5 Excel sichtbar machen 102
4.4 Mit Excel-Dateien arbeiten 102
4.4.1 Excel-Datei erzeugen 102
4.4.2 Excel-Datei öffnen 104
4.4.3 Auf geöffnete Excel-Datei zugreifen 104
4.4.4 Funktion zum einfachen Öffnen einer Excel-Datei 104
4.4.5 Speichern einer Excel-Datei 105
4.4.6 Ein oder mehrere Workbooks? 106
4.4.7 Zugriff auf Tabellenblätter 106
4.4.8 Auf ein Sheet zugreifen 108
4.4.9 Ein neues Sheet anlegen 108
4.4.10 Ein Sheet löschen 109
4.5 Excel-Worksheets bearbeiten 110
4.5.1 Zellen lesen und beschreiben 110
4.5.2 Die Cells-Funktion 111
4.5.3 Die Range-Funktion 113
4.5.4 Experimentierhilfe 114
4.5.5 Range numerisch aufziehen 115
4.5.6 Zahlen statt Buchstaben 115
4.5.7 Spezielle Bereiche mit Range festlegen 117
4.5.8 Range-Bereich ausgeben 118
4.5.9 Range durchlaufen 118
4.5.10 Zeilen und Spalten im Range-Objekt 119

4.5.11 Eine Zelle mit einem Wert füllen 120
4.5.12 Alternative Notation für Bezüge 120
4.6 Gefüllte Bereiche ermitteln 122
4.6.1 Spalte mit dem letzten Wert 122
4.6.2 Zeile mit dem letzten Wert 122
4.7 Format des Zelleninhalts 122
4.8 Formatierung von Zellen 124
4.8.1 Schriftart 124
4.8.2 Rahmen 125
4.9 Formeln in Zellen 127
4.9.1 Formeln mit Funktionen 128
4.9.2 Bezüge in Formeln 130
4.10 Arrays und Excel 131
4.10.1 Array in Excel-Range schreiben 131
4.10.2 Excel-Range in Array schreiben 133
4.10.3 Einzelne Zeilen in Array einlesen 134
4.11 Excel-Ereignisse 135
4.11.1 Beispiel-Ereignisse 136
4.11.2 Beispiel: Berechnung nach Excel auslagern 137
4.12 Zusammenfassung 140

5 Daten in Excel-Tabellen schreiben 141

5.1 Einfache Tabellen oder Abfrage exportieren 141
5.1.1 Daten datensatzweise übertragen 141
5.1.2 Zeilenweises Übertragen mit CopyFromRecordset 143
5.1.3 Daten über ein Array exportieren 144
5.1.4 Array schneller füllen 147
5.1.5 Performance-Vergleich 148
5.1.6 Optimale Spaltenbreiten 148
5.2 Hierarchische Daten ausgeben 149
5.2.1 Beispieltabellen 149
5.2.2 Ausgabe der Daten 150
5.2.3 Gruppieren von Daten 152
5.3 Access-Daten in Excel als Chart darstellen 156
5.3.1 Daten übertragen 157
5.3.2 Chart erstellen 158
5.3.3 Verschiedene Diagrammtypen 160
5.3.4 Diagramm direkt aus Access-Daten erstellen 163
5.3.5 Weitere Möglichkeiten 166

6 Daten aus Excel-Tabellen lesen 167

6.1 Kontakte mit Anrede einlesen 167
6.1.1 Import programmieren 168
6.1.2 Import per DAO 168
6.1.3 Änderung für vorhandene Daten in der Zieltabelle 173

7 Word programmieren 175

7.1 Verweis auf die Word-Objektbibliothek 175
7.2 Eine Word-Instanz erstellen 175
7.2.1 Mehrere Instanzen 178
7.2.2 Bestehende Instanz nutzen 178
7.2.3 Word-Instanz sichtbar machen 179
7.3 Mit Dokumenten arbeiten 180
7.3.1 Word instanzieren und Dokument erstellen 180
7.3.2 Dokument mit Vorlage erstellen 182
7.3.3 Word direkt mit neuem Dokument öffnen 182
7.3.4 Word mit vorhandenem Dokument öffnen 182
7.3.5 Geöffnetes Dokument referenzieren 183
7.3.6 Word-Dokument implizit speichern 183
7.3.7 Word-Dokument explizit speichern 186
7.4 Makro-Recorder von Word nutzen 189
7.5 Word-Dokument bearbeiten 193
7.5.1 Schnellzugriff auf das aktuelle Word-Dokument 194
7.5.2 Kompletten Inhalt überschreiben 195
7.5.3 Text vor oder hinter der Markierung einfügen 196
7.5.4 Das Range-Objekt 197
7.5.5 Range-Objekte für einen Absatz 197
7.5.6 Absatz ändern 199
7.5.7 Absätze löschen 200
7.5.8 Absatz hinzufügen 201
7.6 Mit Textmarken arbeiten 202
7.6.1 Textmarke über die Benutzeroberfläche einfügen 203
7.6.2 Textmarken anzeigen 204
7.6.3 Offene und geschlossene Textmarken 204
7.6.4 Offene Textmarke per VBA anlegen 205
7.6.5 Text für eine offene Textmarke einfügen 206
7.6.6 Offene Textmarke einlesen 207
7.6.7 Geschlossene Textmarke anlegen 207
7.6.8 Inhalt einer geschlossenen Textmarke ändern 208
7.7 Word-Ereignisse nutzen 209
7.8 Dokumente in Anlage-Feldern speichern 211

8 Word-Dokumente mit Daten füllen 215

8.1 Kontaktliste in Word erstellen 215
8.1.1 Performance 217
8.1.2 Zeichenformate festlegen 218
8.1.3 Mögliche Formatierungen 221
8.1.4 Zeichenformate nutzen 221
8.1.5 Absatzformatvorlagen 223
8.2 Tabellen in Word erstellen 224
8.3 Suchen und Ersetzen 225

8.4 Bilder von Access nach Word 227
8.5 Katalog aus der Datenbank erstellen 229
8.5.1 Absatzformatvorlagen erstellen 234
8.5.2 Bilder einfügen 235
8.5.3 Beschreibung als Rich-Text einfügen 238
8.6 Speichern als PDF 240

9 Informationen aus Word-Dokumenten 241
9.1 Word-Dokumente auslesen 241
9.1.1 Absatz für Absatz 241
9.1.2 Absatzformat ermitteln 243
9.2 Bilder aus dem Dokument auslesen 244
9.2.1 Bilder per VBA extrahieren 245
9.2.2 Bild aus XML-Dokument beziehen 248
9.3 Tabellen einlesen 250

10 Word-Vorlage aus Access füllen 253
10.1 Word-Vorlage erstellen 253
10.1.1 Seitenränder einstellen 254
10.1.2 Briefkopf hinzufügen 255
10.1.3 Adressblock und Datum hinzufügen 257
10.1.4 Empfängeradresse 258
10.1.5 Textmarke für den Betreff 260
10.1.6 Textmarke für den Inhalt 261
10.1.7 Markierungslinien 261
10.1.8 Seitenzahlen 264
10.2 Textmarken füllen 265
10.2.1 Vorlage auswählen 266
10.2.2 Adressen per Kombinationsfeld 268
10.2.3 Dokument erstellen 268
10.2.4 Adresse zusammenstellen 270
10.3 Dokument nach dem Anlegen speichern 272
10.4 Daten aus einem Rich-Text-Feld einfügen 273
10.5 Quellen 278

11 Word-Rechnungen 279
11.1 Herausforderung 279
11.2 Beispieldatenbank 280
11.2.1 Rechnungsdaten speichern 280
11.2.2 Rechnungsempfänger speichern 281
11.2.3 Anreden speichern 282
11.2.4 Rechnungspositionen speichern 282
11.2.5 Mehrwertsteuersätze speichern 282
11.2.6 Einheiten speichern 283
11.3 Formulare der Rechnungserstellung 284
11.3.1 Unterformular zur Anzeige der Rechnungspositionen 284

11.3.2 Hauptformular zur Anzeige der Rechnungen 285
11.4 Neue Rechnungen und Rechnungspositionen hinzufügen 287
11.5 Word-Rechnung erstellen 289
11.5.1 Word-Vorlage erstellen 289
11.5.2 Erstellung der Word-Rechnung auslösen 290
11.5.3 Platzhalter ersetzen 294
11.5.4 Einfügen der Rechnungspositionen 294
11.5.5 Die Prozedur SpaltenbreitenEinstellen 302
11.5.6 Die Prozedur SpalteninhalteAusrichten 303
11.5.7 Die Prozedur SpaltenueberschriftenEinstellen 303
11.5.8 Die Prozedur RechnungspositionenEintragen 304
11.5.9 Die Prozedur ZwischensummeEintragen 304
11.5.10 Die Prozedur UebertragEintragen 304
11.5.11 Die Prozedur NettosummeEintragen 305
11.5.12 Die Prozedur MehrwertsteuerEintragen 305
11.5.13 Die Prozedur RechnungssummeEintragen 305

12 Word-Seriendruck per VBA 307

12.1 Datenquelle auswählen 307
12.2 Datensätze auswählen 309
12.2.1 Auswählen und Anzeige vereinfachen 314
12.2.2 Datensätze nach Status anzeigen 316
12.3 Seriendruck-Dokument auswählen 318
12.3.1 Bestehendes Dokument öffnen 318
12.3.2 Neues Dokument ohne Vorlage öffnen 319
12.3.3 Neues Dokument mit Vorlage 320
12.3.4 Dokument bearbeiten 321
12.4 Platzhalter einfügen 321
12.4.1 Verweis auf die Office-Bibliothek 322
12.4.2 Klassenmodule implementieren 322
12.4.3 Kontextmenü-Klassen 325
12.5 Seriendruck-Dokument erstellen 330
12.5.1 Seriendruck starten 330
12.6 Restarbeiten 335

13 Outlook programmieren 339

13.1 Verweis auf die Outlook-Bibliothek 339
13.2 Outlook-Instanz 339
13.2.1 Outlook-Objekt einfach verfügbar machen 340
13.2.2 Namespace-Objekt einfach verfügbar machen 341
13.3 Outlook-Accounts 342
13.4 Outlook-Kategorien 346
13.5 Outlook-Folder 347
13.5.1 Ordner auswählen 349
13.5.2 Auf Standardordner zugreifen 350
13.5.3 Folder-Objekt per Name finden 351

13.5.4 Folder-Objekt per Pfad finden 351
13.5.5 Ordner per Pfad oder EntryID ermitteln 353
13.6 Folder-Objekt nutzen 354
13.6.1 Bestimmten Folder anzeigen 354
13.6.2 Auf Elemente zugreifen 355
13.6.3 Alle Elemente eines Ordners löschen 355
13.7 Per GetTable auf Elemente zugreifen 357
13.7.1 Felder des Table-Objekts 357
13.7.2 Table-Objekt durchlaufen 359
13.7.3 Table durchsuchen 360
13.7.4 Table in Array speichern 361
13.7.5 Table direkt filtern 362
13.7.6 Funktion zum Einlesen von Table-Objekten 362
13.7.7 Einträge ausschließen 364
13.7.8 Folder hinzufügen 365
13.7.9 Folder entfernen 365
13.8 Interessante Ereignisse 366
13.8.1 Ereignisse des Folders-Objekts 366
13.8.2 Ereignisse des Folder-Objekts 368

14 Mails mit Outlook senden, empfangen und verarbeiten 371

14.1 Mails verschicken 371
14.1.1 Neue Mail in Outlook öffnen 372
14.1.2 Einfache Mail erstellen und anzeigen 373
14.1.3 Einfache Mail erstellen und direkt senden 375
14.1.4 Account zum Versenden von Mails abrufen und einstellen 376
14.2 To, Cc und Bcc 384
14.2.1 Adressaten hinzufügen 384
14.2.2 Mails per Cc: 387
14.2.3 Mails per Bcc: 389
14.3 Weitere Eigenschaften 389
14.4 Anlagen hinzufügen 393
14.5 HTML-Darstellung 396
14.5.1 HTML-Code übergeben 396
14.5.2 Rich-Text 397
14.6 Gesendete Elemente verschieben 401
14.7 Änderungen nach dem Anzeigen erfassen 402
14.8 E-Mails einlesen 406
14.8.1 Eine Mail öffnen 407
14.9 Anlagen einlesen und speichern 409
14.10 Sicherheitsabfragen 413
14.11 Berichte per Mail verschicken 415
14.11.1 Rechnungsformular 416
14.11.2 Rechnungsbericht 416
14.11.3 Rechnungsmail erstellen 419

14.12 Serienmails 422
14.13 Word-Dokument per Mail verschicken 423
14.14 Mail beim Senden direkt in Access speichern 423

15 Outlook-Kontakte im- und exportieren 425
15.1 Die EntryID 425
15.2 Struktur eines Kontakts 426
15.3 Datenmodell für die Kontakte 428
15.4 Outlook-Kontakte einlesen 428
15.4.1 Kontakt einlesen 428
15.4.2 Datenmodell für die Kontakte 429
15.4.3 Basisdaten schreiben 433
15.4.4 Kommunikationsdaten schreiben 435
15.4.5 Bild einlesen 437
15.4.6 EntryID schon vorhanden? 438
15.5 Outlook-Kontakte schreiben 439
15.5.1 Kontakte aus der Datenbank nach Outlook 439
15.5.2 Automatisierter Import 440
15.6 Synchronisieren von Kontakten, die auf beiden Seiten vorhanden sind 443
15.6.1 Abgleich von Outlook nach Access 444
15.6.2 Abgleich von Access nach Outlook 447
15.6.3 Konflikte auf Basis gleicher Daten 450

16 Mit Outlook-Terminen arbeiten 451
16.1 Termine auslesen 451
16.1.1 Aktuellste Termine einlesen 453
16.1.2 Termine nach Datum filtern 454
16.1.3 Filtern per Restrict 456
16.2 Termine in eine Tabelle importieren 458
16.2.1 Termine in die Tabelle schreiben 460
16.2.2 Performance-Optimierung 463
16.3 Serientermine auslesen 463
16.3.1 Terminserien unter VBA 465
16.3.2 Serieneigenschaften importieren 466
16.3.3 Formular zum Bearbeiten der Terminserien 470
16.4 Termin in Outlook anlegen 473
16.5 Termin in Outlook anzeigen 475
16.6 Geburtstage 476
16.6.1 Automatischer Serientermin 476
16.6.2 Geburtstage anlegen 477
16.6.3 Geburtstag von Access aus anlegen 477

17 PowerPoint programmieren 481
17.1 Vorbereitungen 481
17.2 Aufbau von PowerPoint-Dokumenten 481
17.3 PowerPoint programmieren 483

17.3.1 PowerPoint starten 483
17.3.2 Präsentation erstellen 484
17.3.3 Zugriff auf geöffnete Präsentation 484
17.4 Neue Folie einfügen 485
17.5 Design-Elemente ansprechen 487
17.6 Shape-Elemente füllen 488
17.6.1 Shape-Element mit Text füllen 488
17.6.2 Bilder hinzufügen 491
17.7 Katalog erstellen 492
17.7.1 Quelldaten 492
17.7.2 Formular zum Erstellen der Präsentation 493
17.7.3 Neues Layout definieren 494
17.7.4 Vorlage auswählen 497
17.7.5 Erstellung der Präsentation 498
17.7.6 Erstellen des Presentation-Objekts 501
17.7.7 Speichern der Bilder auf der Festplatte 501

18 Office-Dokumente im Access-Formular 503

18.1 DSOFramer-Steuerelement installieren 503
18.2 DSOFramer zum Formular hinzufügen 505
18.3 DSOFramer programmieren 508
18.3.1 Eigenschaften des DSOFramer-Objekts 508
18.3.2 Größe an das Formular anpassen 511
18.3.3 Menübefehle nachbilden 511
18.3.4 Neues Dokument erstellen 512
18.3.5 Dokument öffnen 513
18.3.6 Dokument mit vorgegebenem Pfad laden 514
18.3.7 Seite einrichten 514
18.3.8 Drucken eines Dokuments 515
18.3.9 Anzeigen der Eigenschaften eines Dokuments 515
18.3.10 Datei speichern 516
18.3.11 Manuell drucken 517
18.3.12 Dokument schließen 518
18.3.13 Druckvorschau anzeigen 518
18.4 Ereignisse des DSOFramer-Steuerelements 518
18.4.1 Beim Öffnen eines Dokuments 519
18.4.2 Vor dem Schließen eines Dokuments 519
18.4.3 Nach dem Schließen eines Dokuments 520
18.4.4 Bei Datei-Befehl 521
18.4.5 Beim Beenden der Druckvorschau 521
18.5 Zusammenfassung 521

Inhalt

1 Datenaustausch zwischen Access und den Office-Anwendungen

Die meisten Kapitel dieses Buchs werden sich mit der Programmierung von Lösungen für den Datenaustausch zwischen Access und den Office-Anwendungen sowie mit der Fernsteuerung der Office-Anwendungen für verschiedene Zwecke beschäftigen. Oftmals ist aber gar kein VBA-Einsatz nötig: Access bietet einige Möglichkeiten, um Daten an andere Anwendungen zu schicken und umgekehrt. Auch die Office-Anwendungen erlauben es, Daten einer Access-Anwendung zu nutzen. Dieses Kapitel kümmert sich um diese Möglichkeiten, also quasi um die Bordmittel.

Die meisten Befehle für den Import und Export von Daten finden Sie im Ribbon-Tab *Externe Daten* (siehe Abbildung 1.1). Die meisten Elemente der Gruppe *Exportieren* sind standardmäßig ausgeblendet und werden erst aktiviert, wenn Sie im Navigationsbereich eine Tabelle oder Abfrage, teilweise aber auch ein Formular oder einen Bericht markieren.

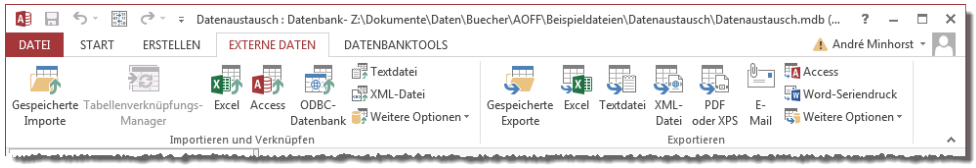


Abbildung 1.1: Ribbon-Tab für den Import und Export von Daten

In den folgenden Abschnitten schauen wir uns zuerst die Export-Möglichkeiten an, also die im rechten Bereich des Ribbons, also unter *Externe Daten|Export*. Danach folgen die verschiedenen Import-Möglichkeiten der Gruppe *Externe Daten|Import*.

1.1 Export nach Excel

Den Export des Inhalts einer Tabelle oder Abfrage nach Excel starten Sie mit dem Ribbon-Eintrag *Externe Daten|Exportieren|Excel*. Diese Befehl öffnet den Dialog *Exportieren - Excel-Tabelle* aus Abbildung 1.2.

1.1.1 Export-Einstellungen

Hier wählen Sie zunächst die Zieldatei für den Export aus. Dies erledigen Sie über die Schaltfläche *Durchsuchen* rechts neben dem Feld zur Eingabe des Pfades zur Zieldatei. Sie können bereits vorher mit dem darunter befindlichen Kombinationsfeld das Dateiformat festlegen, dass dann beim Öffnen des *Datei speichern*-Dialogs übernommen wird. Mit dem *Datei speichern*-Dialog wählen

Sie das Verzeichnis aus und geben den Namen der Zieldatei ein. Der Dialog schlägt bereits einen Namen für die Datei vor, der aus dem Namen der Quelle (also der Abfrage oder Tabelle) sowie der entsprechenden Dateiendung besteht (also *.xls* oder *.xlsx*, je nach gewählten Dateiformat).

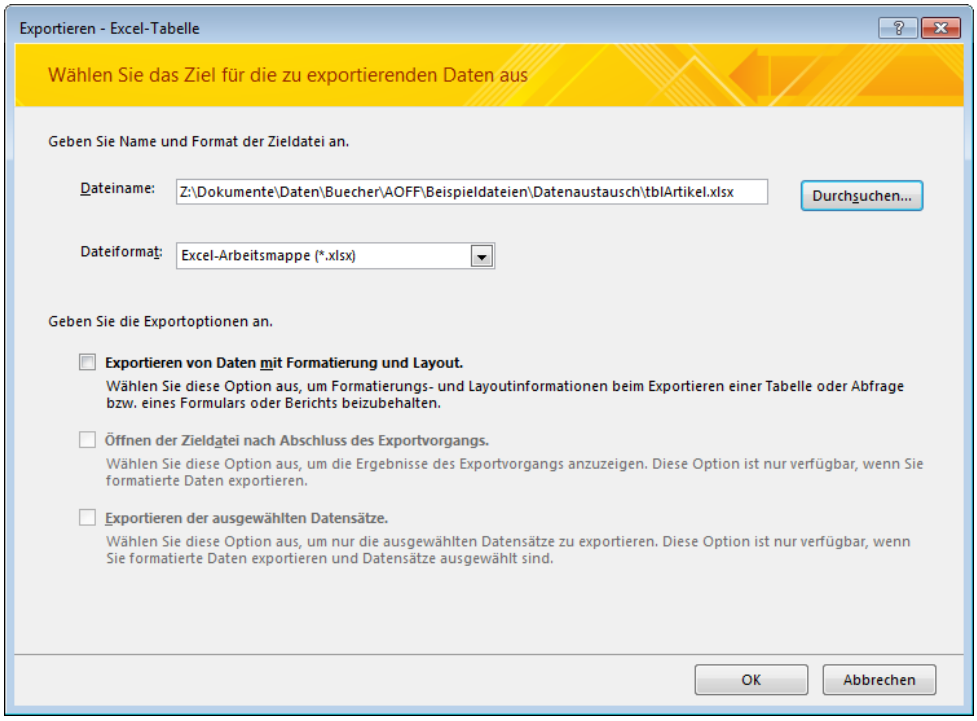


Abbildung 1.2: Einstellungen für den Export einer Tabelle oder Abfrage nach Excel

Der Dialog *Exportieren - Excel-Tabelle* bietet noch weitere Optionen, die wir uns gleich im Anschluss anschauen:

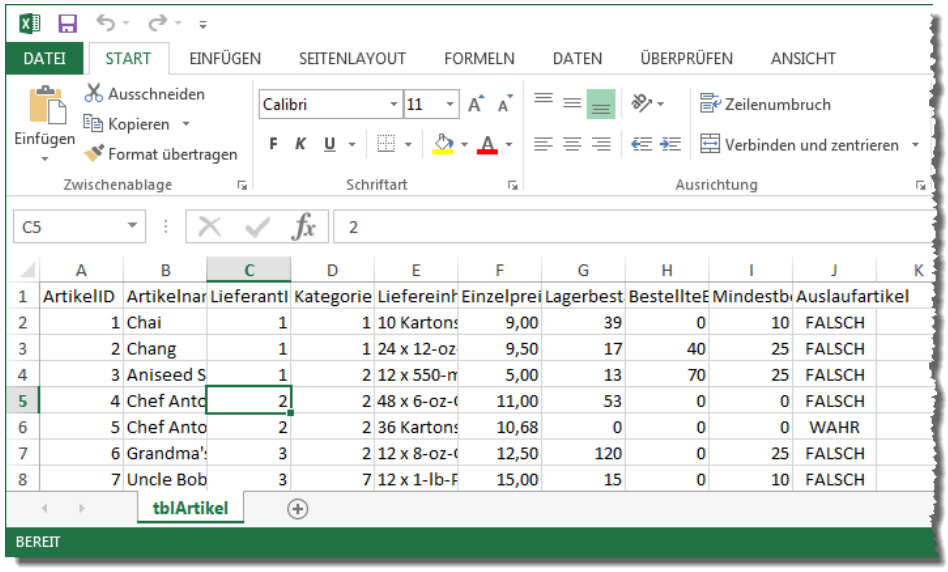
» *Exportieren von Daten mit Formatierung und Layout*

» *Öffnen der Zieldatei nach Abschluss des Exportvorgangs*

» *Exportieren der ausgewählten Datensätze*

Exportieren von Daten mit Formatierung und Layout

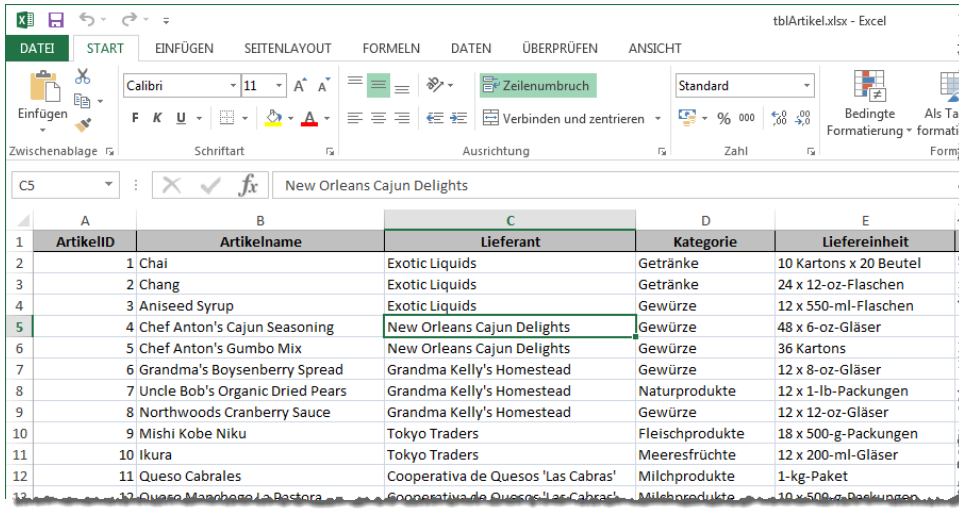
Wenn Sie diese Option nicht aktivieren, exportiert Access die Daten einfach in eine neue Tabelle hinein und passt weder die Spaltenbreiten an noch markiert es die Überschriftenzeile (siehe Abbildung 1.3).



	A	B	C	D	E	F	G	H	I	J	K
1	ArtikelID	Artikelnar	Lieferantl	Kategorie	Liefereinh	Einzelprei	Lagerbest	BestellteE	Mindestbe	Auslaufartikel	
2	1	Chai	1	1	10 Kartons	9,00	39	0	10	FALSCH	
3	2	Chang	1	1	24 x 12-oz	9,50	17	40	25	FALSCH	
4	3	Aniseed S	1	2	12 x 550-m	5,00	13	70	25	FALSCH	
5	4	Chef Anto	2	2	48 x 6-oz-(	11,00	53	0	0	FALSCH	
6	5	Chef Anto	2	2	36 Kartons	10,68	0	0	0	WAHR	
7	6	Grandma's	3	2	12 x 8-oz-(	12,50	120	0	25	FALSCH	
8	7	Uncle Bob	3	7	12 x 1-lb-F	15,00	15	0	10	FALSCH	

Abbildung 1.3: Export nach Excel ohne Formatierung

Aktivieren Sie diese Option hingegen, ermittelt Access beim Export die optimale Spaltenbreite für jede Spalte – das heißt, dass alle Inhalt vollständig sichtbar sind, ohne dass der Benutzer die Spalten nachträglich ändern muss. Außerdem markiert Access die Spaltenüberschriften wie in Abbildung 1.4 mit einer alternativen Hintergrundfarbe und in fetter Schrift.



	A	B	C	D	E
1	ArtikelID	Artikelname	Lieferant	Kategorie	Liefereinheit
2	1	Chai	Exotic Liquids	Getränke	10 Kartons x 20 Beutel
3	2	Chang	Exotic Liquids	Getränke	24 x 12-oz-Flaschen
4	3	Aniseed Syrup	Exotic Liquids	Gewürze	12 x 550-ml-Flaschen
5	4	Chef Anton's Cajun Seasoning	New Orleans Cajun Delights	Gewürze	48 x 6-oz-Gläser
6	5	Chef Anton's Gumbo Mix	New Orleans Cajun Delights	Gewürze	36 Kartons
7	6	Grandma's Boysenberry Spread	Grandma Kelly's Homestead	Gewürze	12 x 8-oz-Gläser
8	7	Uncle Bob's Organic Dried Pears	Grandma Kelly's Homestead	Naturprodukte	12 x 1-lb-Packungen
9	8	Northwoods Cranberry Sauce	Grandma Kelly's Homestead	Gewürze	12 x 12-oz-Gläser
10	9	Mishi Kobe Niku	Tokyo Traders	Fleischprodukte	18 x 500-g-Packungen
11	10	Ikura	Tokyo Traders	Meeresfrüchte	12 x 200-ml-Gläser
12	11	Queso Cabrales	Cooperativa de Quesos 'Las Cabras'	Milchprodukte	1-kg-Paket

Abbildung 1.4: Export nach Excel mit Formatierung

Öffnen der Zieldatei nach Abschluss des Exportvorgangs

Diese Option sorgt schlicht und einfach dafür, dass die erstellte Excel-Datei nach dem Export geöffnet wird.

Exportieren der ausgewählten Datensätze

Wenn Sie die zu exportierende Tabelle oder Abfrage zuvor öffnen und nur bestimmte Datensätze nach Excel übertragen wollen, wählen Sie diese Option. Allerdings müssen Sie die betroffenen Datensätze zuvor auch in Access markieren – siehe Abbildung 1.5.

Dies gelingt leider nur bei zusammenhängenden Datensätzen.

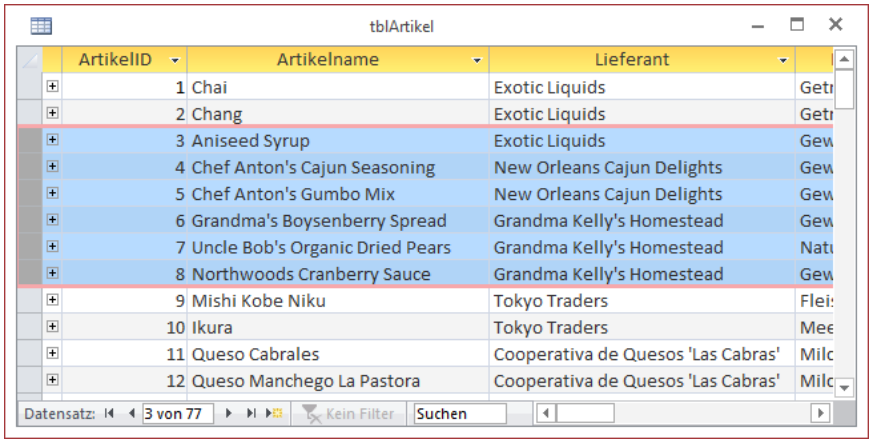


Abbildung 1.5: Markieren der zu exportierenden Datensätze

Wenn Sie also nur eine bestimmte Auswahl von Artikeln exportieren möchten, haben Sie zum Beispiel die folgenden Möglichkeiten:

» Filtern Sie die Datensätze der Tabelle mit den dafür vorgesehenen Elementen, die etwa über die Spaltenköpfe verfügbar sind, markieren Sie alle Datensätze des Ergebnisses und exportieren Sie dann die Auswahl.

» Oder erstellen Sie eine Abfrage, welche nur die zu exportierenden Datensätze enthält, und exportieren diese.

1.1.2 Exportschritte speichern

Nach dem Export erscheint noch ein weiterer Dialog, der die Möglichkeit bietet, die Exportschritte zu speichern (siehe Abbildung 1.6).

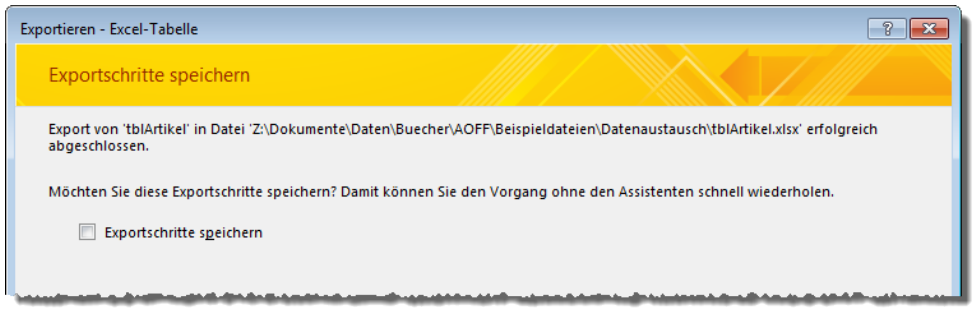


Abbildung 1.6: Option zum Speichern der Exportschritte

Aktivieren Sie diese Option, erscheint eine Reihe weiterer Einstellungen (siehe Abbildung 1.7).

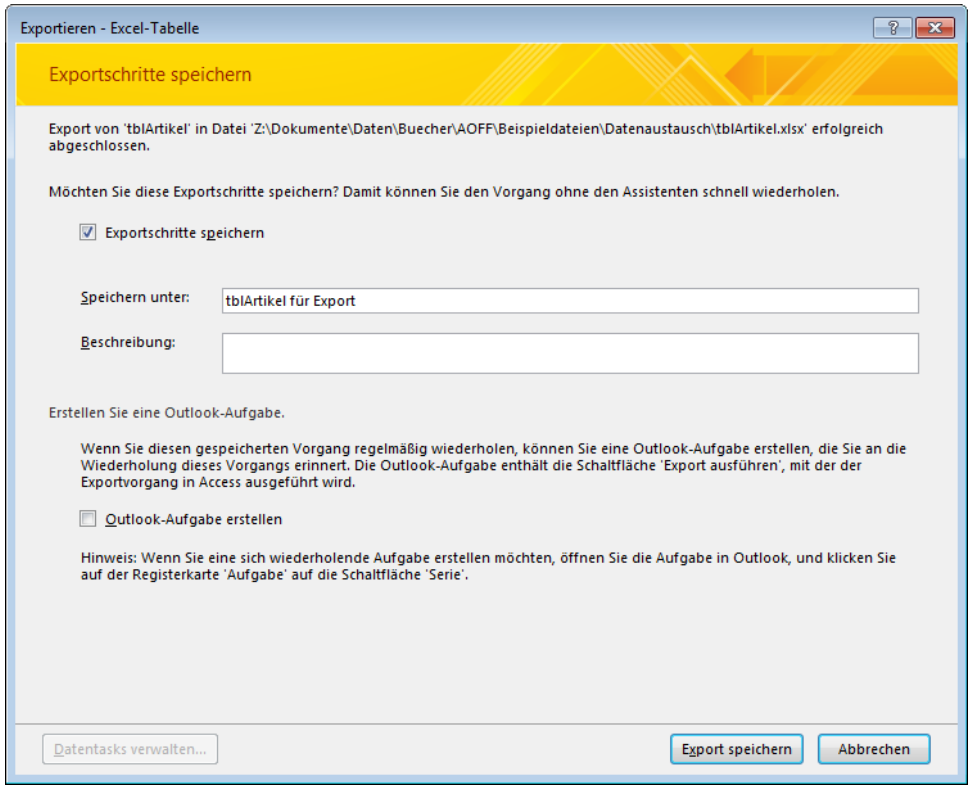


Abbildung 1.7: Weitere Optionen zum Speichern der Exportschritte

Hier können Sie etwa einen Namen festlegen, unter dem der Export gespeichert werden soll oder eine Beschreibung. Außerdem können Sie für diesen Export eine Outlook-Aufgabe erstellen.

Outlook-Aufgabe erstellen

Wenn Sie diese Option gewählt haben, erscheint nach der Bestätigung mit *Export speichern* eine neue Outlook-Aufgabe. Diese sieht wie in Abbildung 1.8 aus.

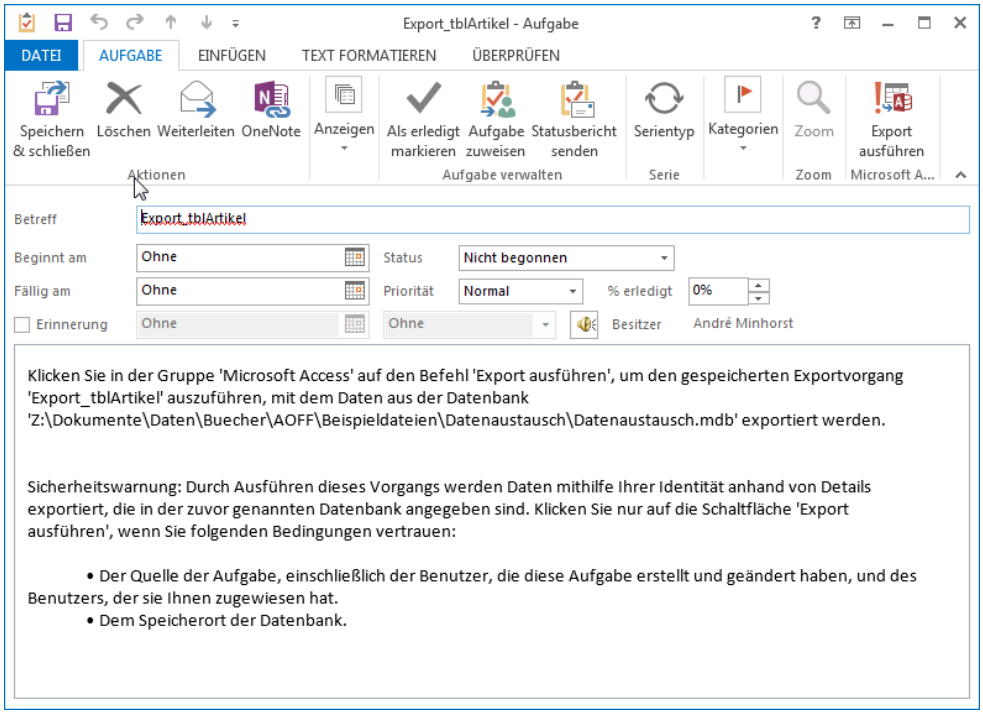


Abbildung 1.8: Outlook-Aufgabe mit der Möglichkeit, einen Export zu starten

Die Aufgabe können Sie nun terminieren und mit einer Erinnerung versehen. Der Clou dieser Aufgabe ist aber, dass diese in der Ribbon-Leiste rechts den Eintrag *Aufgabe|Microsoft Access|Export ausführen* enthält. Bevor der Export nach dem Anklicken dieses Befehls ausgeführt wird, erscheint noch eine Meldung mit einer Rückfrage, ob dies auch wirklich geschehen soll (siehe Abbildung 1.9).

Abbildung 1.9: Rückfrage vor der Ausführung eines Exports

Die Bestätigung mit *OK* brachte jedoch nicht das gewünschte Ergebnis, sondern die Fehlermeldung aus Abbildung 1.10. Die erste Vermutung, dass die möglicherweise bereits existierende Export-Datei den Fehler verursacht, weil diese nicht einfach überschrieben werden konnte, bestätigte sich nicht – auch nach dem Löschen der Zieldatei gleichen Namens erschien die Fehlermeldung noch.

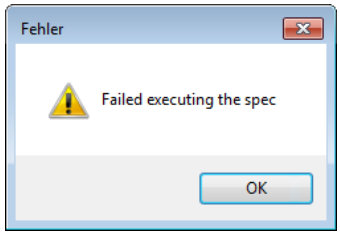


Abbildung 1.10: Fehler beim Ausführen des Exports per Outlook-Aufgabe

Speicherort der Export-Spezifikation

Nun stellt sich noch die Frage, wo Access diese Spezifikation überhaupt gespeichert hat und wie wir diese von Access aus erneut aufrufen können, wenn dies von Outlook aus offensichtlich schon nicht funktioniert.

Dieser Ort ist schnell gefunden: Sie können die gespeicherten Exporte über den Ribbon-Eintrag *Externe Daten|Exportieren|Gespeicherte Exporte* aufrufen (siehe Abbildung 1.11).

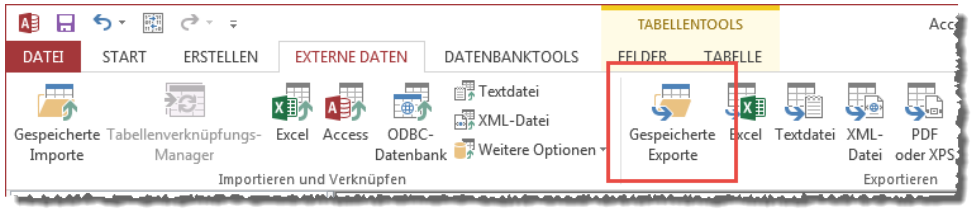


Abbildung 1.11: Aufruf der gespeicherten Exporte

Klicken Sie diesen Befehl an, erscheint auch gleich der Dialog aus Abbildung 1.12 mit der soeben gespeicherten Exportspezifikation.

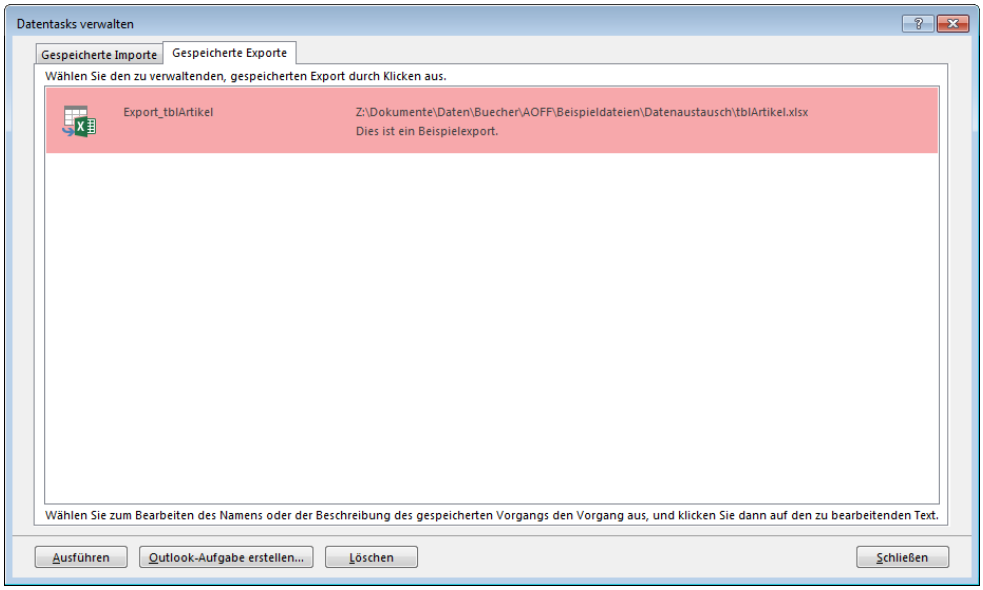


Abbildung 1.12: Dialog zur Verwaltung der gespeicherten Importe und Exporte

Hier ergeben sich wiederum verschiedene Möglichkeiten: Sie können den Export erneut ausführen, erneut die soeben bereits angelegte Outlook-Aufgabe erstellen oder den Export löschen. Ein Bearbeiten des Vorgangs ist allerdings nicht möglich. Immerhin: Im Gegensatz zum Aufruf von Outlook aus klappt der erneute Export über die Schaltfläche *Ausführen* reibungslos – bei bereits vorhandener Datei übrigens mir der Rückfrage, ob diese überschrieben werden soll.

1.1.3 Export nach Excel per Kontextmenü

Es gibt noch eine Alternative zum Aufruf des Export-Assistenten über das Ribbon. Wenn Sie mit der rechten Maustaste auf das zu exportierende Element klicken und dort den Eintrag *Exportieren|Excel* auswählen, erscheint der gleiche Dialog und bietet die Möglichkeit zum Exportieren des Objekts nach Excel an (siehe Abbildung 1.13).

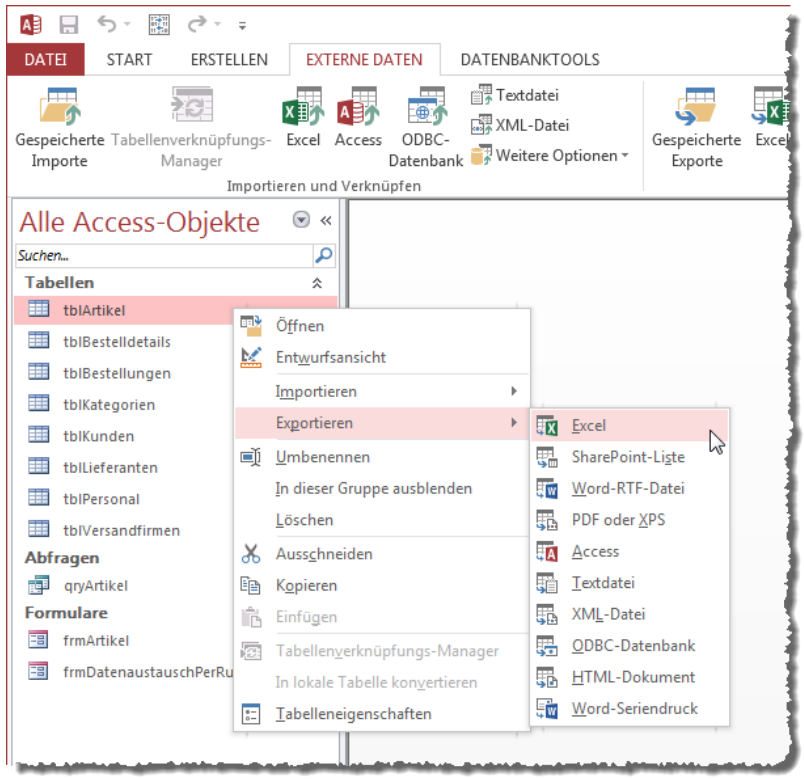


Abbildung 1.13: Export eines Objekts per Kontextmenü

1.1.4 Formulare nach Excel exportieren

Wenn Sie ein Formular nach Excel exportieren, erzeugt Access eine ähnliche Excel-Tabelle wie beim Export einer Tabelle oder einer Abfrage. Es landen jedoch lediglich die Felder in der Excel-Tabelle, die auch im Formular in Form gebundener Steuerelemente angelegt wurden.

Der Export etwa eines Formulars samt Unterformular zur Anzeige von Daten in einer 1:n-Beziehung liefert ebenfalls nur die Daten des Hauptformulars.

1.1.5 Berichte nach Excel exportieren

Wenn Sie Daten strukturiert nach Excel ausgeben möchten, müssen Sie einen Bericht anlegen. Strukturiert bedeutet dabei, dass Sie etwa die Bestellungen samt Bestellpositionen in einer Excel-Tabelle anzeigen möchten – oder Artikelkategorien und die dazu gehörenden Artikel. Als Beispiel haben wir einen Bericht erstellt, der in der Datenherkunft die beiden Tabellen *tblArtikel* und *tblKategorien* enthält. Diesem haben wir eine Gruppierung über das Feld *KategorieID* hin-

zufügt. Der Kopfbereich für diese Kategorie nimmt die Felder *KategorieID* und *Kategoriename* auf, einige Felder der Tabelle *tblArtikel* landen im Detailbereich.

Der Bericht sieht in der Seitenansicht wie in Abbildung 1.14 aus.

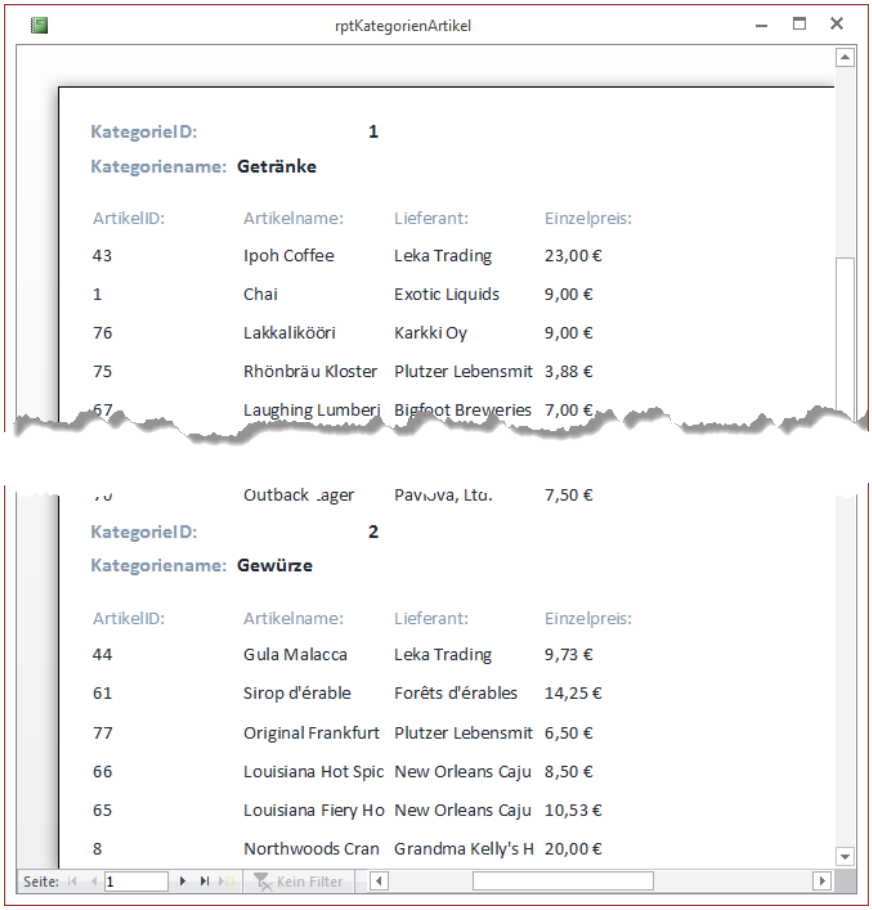


Abbildung 1.14: Bericht als Vorlage für einen strukturierten Export zweier Tabellen nach Excel

Wenn Sie für diesen Bericht den Kontextmenübefehl *Export|Excel* auswählen und die Zieldatei angegeben haben, arbeitet Access den Bericht offenbar seitenweise ab. Darauf deutet zumindest das nun erscheinende Fenster hin (siehe Abbildung 1.15).

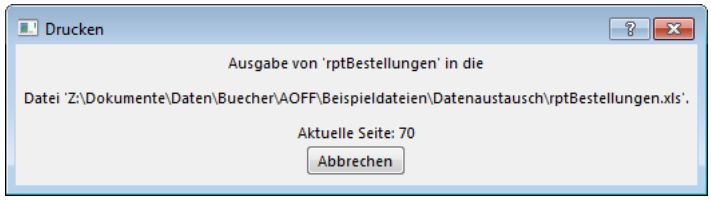
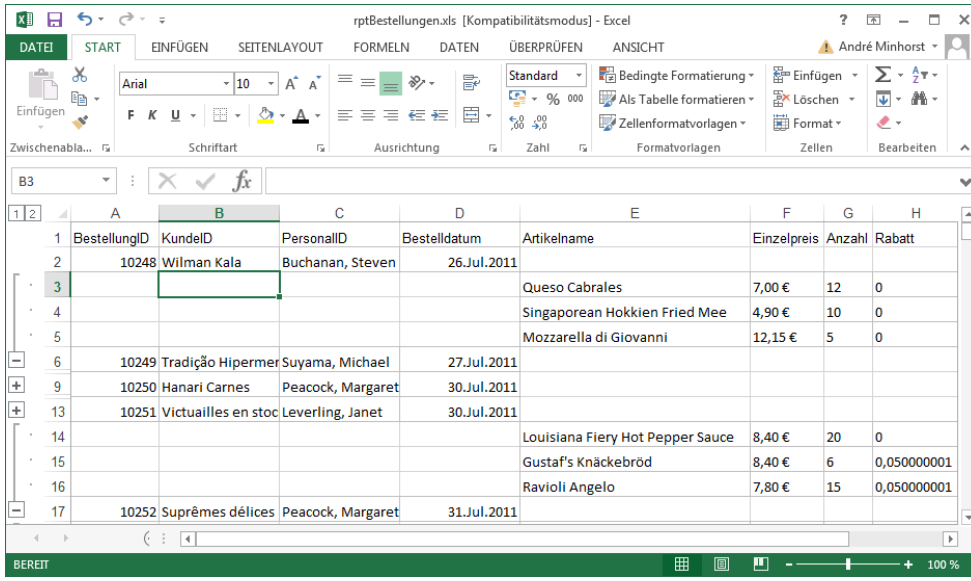


Abbildung 1.15: Fortschrittsanzeige beim Export eines etwas umfangreicheren Berichts

Das Ergebnis sieht interessant aus: Die erste Zeile enthält die Bestelldaten, die folgenden die einzelnen Bestellpositionen – danach folgen weitere Bestellungen mit den entsprechenden Positionen. Praktisch ist, dass man die Ebene mit den Bestellpositionen minimieren kann. Für eine einzelne Ebene gelingt dies mit den Plus/Minus-Zeichen am linken Rand, wie es in Abbildung 1.16 etwa in Zeile *6* und *9* der Fall ist.



	A	B	C	D	E	F	G	H
1	BestellungID	KundeID	PersonalID	Bestelldatum	Artikelname	Einzelpreis	Anzahl	Rabatt
2	10248	Wilman Kala	Buchanan, Steven	26.Jul.2011				
3					Queso Cabrales	7,00 €	12	0
4					Singaporean Hokkien Fried Mee	4,90 €	10	0
5					Mozzarella di Giovanni	12,15 €	5	0
6	10249	Tradição Hipermer	Suyama, Michael	27.Jul.2011				
9	10250	Hanari Carnes	Peacock, Margaret	30.Jul.2011				
13	10251	Victuailles en stoc	Leverling, Janet	30.Jul.2011				
14					Louisiana Fiery Hot Pepper Sauce	8,40 €	20	0
15					Gustaf's Knäckebröd	8,40 €	6	0,050000001
16					Ravioli Angelo	7,80 €	15	0,050000001
17	10252	Suprêmes délices	Peacock, Margaret	31.Jul.2011				

Abbildung 1.16: Ergebnis des Exports eines Berichts mit einer Gruppierungsebene

Mit einem Klick auf die kleine Schaltfläche mit der Nummer *1* am linken oberen Rand des Tabellenblatts können Sie sogar alle untergeordneten Ebenen ausblenden, mit einem Klick auf die *2* blenden Sie alle Ebenen wieder ein (siehe Abbildung 1.17).

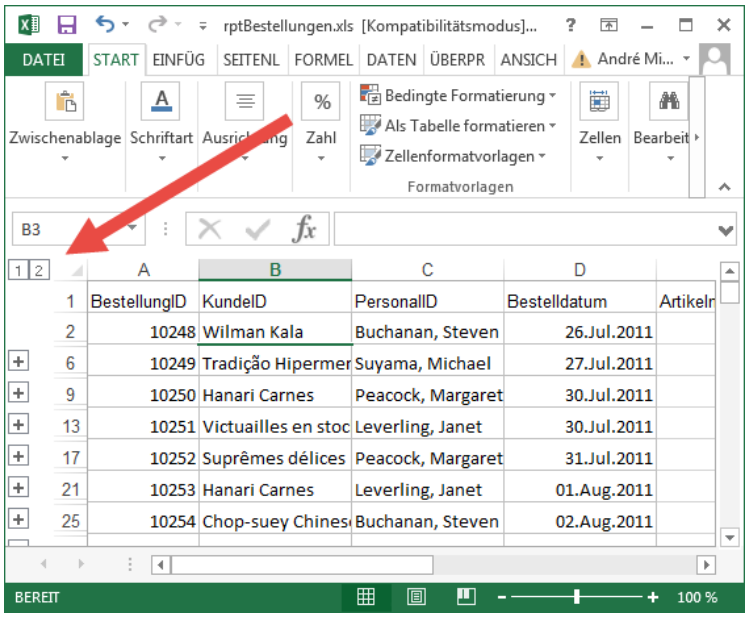


Abbildung 1.17: Minimieren aller untergeordneten Ebenen

1.1.6 Daten in Excel einbinden

Relativ unbekannt ist die Möglichkeit, Daten von Access in eine Excel-Datei zu übertragen und dabei eine Verknüpfung aufrecht zu erhalten – also so, dass die Daten der Access-Tabelle in Excel verfügbar sind, um beispielsweise Auswertungen über diese Daten durchzuführen. Dabei werden die Daten in der Excel-Datei jedoch bei Änderungen der zugrunde liegenden Daten in der Access-Datenbank aktualisiert. Dies gelingt jedoch über die Benutzeroberfläche nur von Excel aus – schauen wir uns an, wie es funktioniert.

Dazu erstellen wir zunächst eine neue, leere Excel-Datei namens *VerknuepfungMitAccess.xlsx*. Dann rufen Sie den Ribbon-Befehl *Daten|Externe Daten abrufen|Aus Access* auf (siehe Abbildung 1.18).

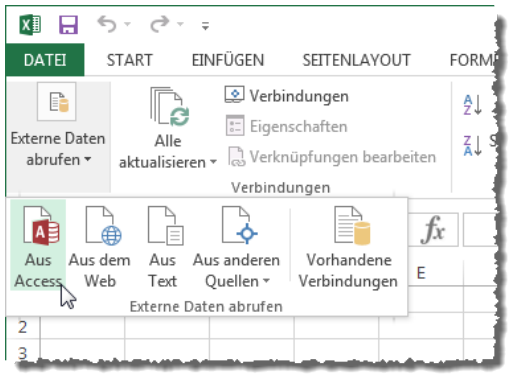


Abbildung 1.18: Externe Daten aus Access abrufen

Dies öffnet einen *Datei öffnen*-Dialog, mit dem Sie die Access-Datenbank auswählen, welche die Daten liefern soll. Nach der Auswahl der Access-Datenbank erscheint ein weiterer Dialog, mit dem Sie die Tabelle oder Abfrage auswählen können (siehe Abbildung 1.19). Dieser Dialog bietet außerdem die Möglichkeit, die Option *Auswahl mehrerer Tabelle aktivieren* zu setzen.

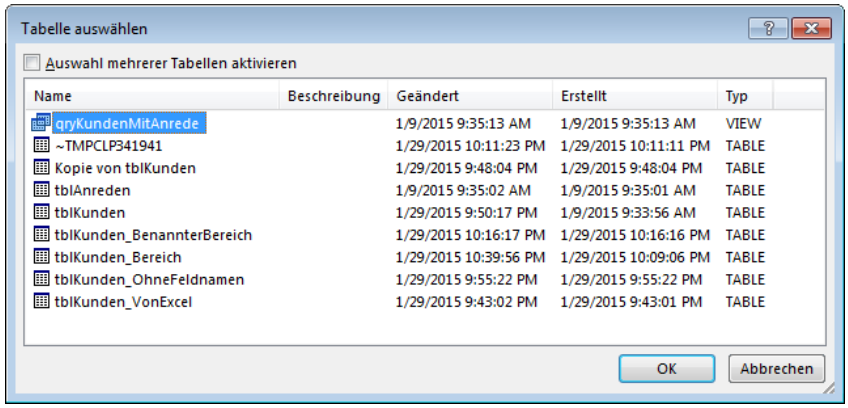


Abbildung 1.19: Auswahl der Datenquelle(n)

Wir wählen zunächst die Abfrage *qryKundenMitAnreden* aus. Nach einem Klick auf *OK* erscheint ein weiterer Dialog, mit dem Sie weitere Optionen einstellen können (siehe Abbildung 1.20).

Hier können Sie beispielsweise angeben, ob die Daten in einer einfachen Tabelle angezeigt werden sollen, als PivotTable-Bericht oder als Pivot-Chart.

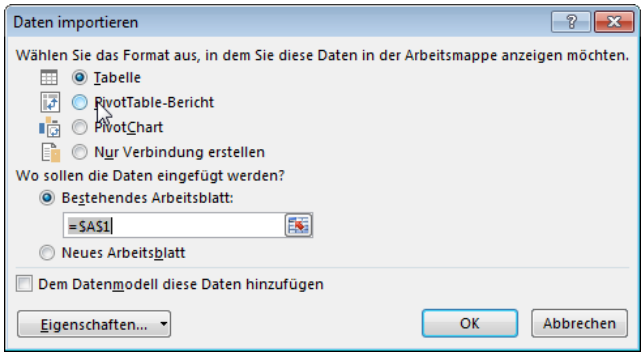
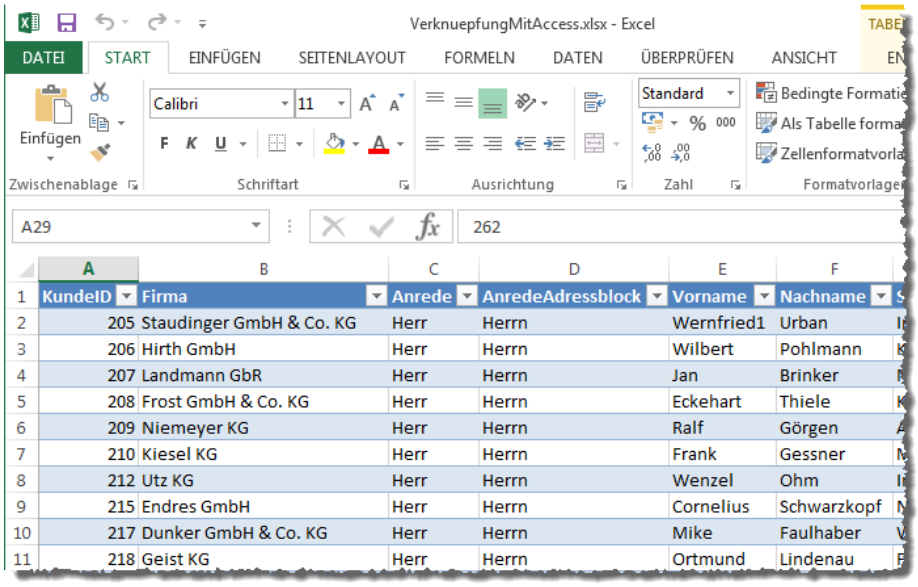


Abbildung 1.20: Einstellungen für den Import der Daten

Dessen Einstellungen behalten wir zunächst bei und klicken auf *OK*, um den Import der Daten durchzuführen. Das Ergebnis sieht zumindest schon mal hübsch aus (siehe Abbildung 1.21).



	A	B	C	D	E	F
1	KundeID	Firma	Anrede	AnredeAdressblock	Vorname	Nachname
2	205	Staudinger GmbH & Co. KG	Herr	Herrn	Wernfried1	Urban
3	206	Hirth GmbH	Herr	Herrn	Wilbert	Pohlmann
4	207	Landmann GbR	Herr	Herrn	Jan	Brinker
5	208	Frost GmbH & Co. KG	Herr	Herrn	Eckehart	Thiele
6	209	Niemeyer KG	Herr	Herrn	Ralf	Görgen
7	210	Kiesel KG	Herr	Herrn	Frank	Gessner
8	212	Utz KG	Herr	Herrn	Wenzel	Ohm
9	215	Endres GmbH	Herr	Herrn	Cornelius	Schwarzkopf
10	217	Dunker GmbH & Co. KG	Herr	Herrn	Mike	Faulhaber
11	218	Geist KG	Herr	Herrn	Ortmund	Lindenau

Abbildung 1.21: Mit Access verknüpfte Daten

Nun wollen wir natürlich wissen, was es mit der Aktualisierung der Daten auf sich hat. Werden die Daten nur von einer in die andere Richtung synchronisiert oder sogar in beide Richtungen? Also ändern wir den Vornamen des ersten Datensatzes. Das Schließen und erneute Öffnen der Excel-Datei liefert den geänderten Vornamen. In der Access-Datenbank liegt hingegen noch der alte Wert vor. Eine Aktualisierung von Excel in Richtung Access scheint also nicht zu funktionieren. Aber warum zeigt Excel nach dem Schließen und Öffnen der Tabelle nicht wieder den tat-

sächlich in der verknüpften Tabelle der Datenbank enthaltenen Wert an, sondern den manuell geänderten Wert? Anscheinend müssen wir das mit den aktuellen Einstellungen selbst anstoßen. Dazu verwenden Sie den Ribbon-Eintrag *Daten|Verbindungen|Alle aktualisieren|Aktualisieren* (siehe Abbildung 1.22). Dies liefert dann wieder den Wert aus der Access-Tabelle.

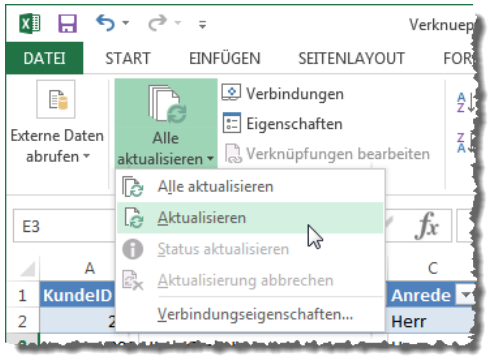


Abbildung 1.22: Aktualisieren der Daten

Wie aber können wir die Aktualisierung beeinflussen? Auf den ersten Blick finden wir den Ribbon-Eintrag *Daten|Verbindungen|Eigenschaften*. Dieser öffnet den Dialog *Externe Dateneigenschaften* (siehe Abbildung 1.23).

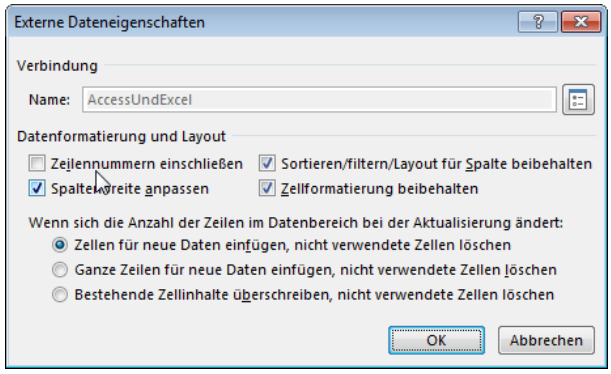


Abbildung 1.23: Externe Dateneigenschaften

Mit der Option *Zeilennummer einschließen* können Sie beispielsweise festlegen, dass die erste Spalte der Excel-Tabelle noch einen *0*-basierten Index für die Zeilen anzeigt. Das Aktualisierungsintervall und ähnliche Einstellungen lassen sich hier allerdings nicht vornehmen.

Dies gelingt im Dialog *Verbindungseigenschaften*. Diesen hätten Sie soeben beim Durchführen des Imports im Dialog *Daten importieren* öffnen können, aber auch mit einem Klick auf die Schaltfläche *Verbindungseigenschaften* im Dialog *Externe Dateneigenschaften*. Sie können das

aber auch nachträglich erledigen, und zwar mit dem Ribbon-Befehl *Daten|Verbindungen|Alle aktualisieren|Verbindungseigenschaften*.

Der Dialog *Verbindungseigenschaften* bietet auf der ersten Registerseite einen Bereich namens *Aktulisierungssteuerung* an, mit dem Sie beispielsweise das Aktualisierungsintervall für eine automatische Aktualisierung einstellen können. Die kleinste Einheit ist eine Minute. Das ist aber auch nicht schlimm, denn Sie können ja manuell aktualisieren. Außerdem werden Sie die Daten wohl auch nur nach Excel verknüpfen, um die dortigen Auswertungsmöglichkeiten zu nutzen – und wenn etwa eine umfangreiche Pivot-Tabelle ständig aktualisiert wird, macht dies auch keinen Sinn. Sie können in diesem Dialog außerdem festlegen, dass die Inhalte beim Öffnen der Datei neu eingelesen werden (siehe Abbildung 1.24).

Auf der zweiten Registerseite (siehe Abbildung 1.25) finden Sie beispielsweise die Möglichkeit, den Speicherort der Quelldatei anzupassen. Außerdem können Sie dort die Verbindungszeichenfolge entnehmen, mit der wir später eine solche Verknüpfung per VBA anlegen werden.

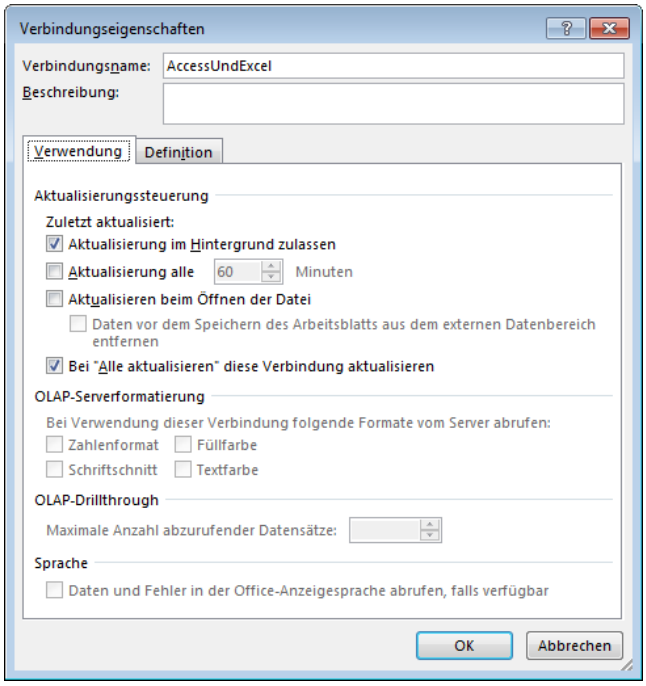


Abbildung 1.24: Eigenschaften der Verbindung, hier unter anderem für die Aktualisierung

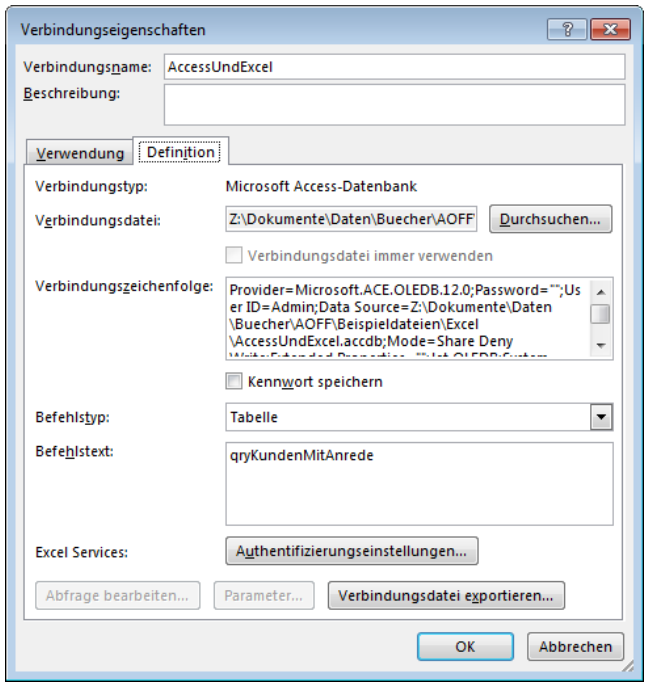


Abbildung 1.25: Weitere Verbindungseigenschaften, zum Beispiel mit der Verbindungszeichenfolge

1.2 Export nach Word

Für den Export von Daten nach Word gibt es verschiedene Möglichkeiten. Die erste sieht ähnlich aus wie beim einfachen Export nach Excel: Sie markieren dabei schlicht die zu exportierende Datenquelle, zum Beispiel eine Tabelle oder Abfrage, und wählen dann den entsprechenden Ribbon-Eintrag aus. Dieser heißt in diesem Fall *Externe Daten|Exportieren|Weitere Optionen|Word* (siehe Abbildung 1.26).

2 Datenaustausch mit VBA-Bordmitteln

Im vorherigen Kapitel haben Sie die Möglichkeiten der Benutzeroberfläche für den Datenaustauschen zwischen den Office-Programmen kennen gelernt – wobei wir uns auf die Wege beschränkt haben, die wir von Access aus angesteuert haben. In diesem Kapitel nun wollen wir uns ansehen, wie wir diese recht einfach Möglichkeiten per VBA nachbilden können. „Bordmittel" bezieht sich dabei darauf, dass wir nur solche Befehle verwenden wollen, die in der Access-Bibliothek enthalten sind und somit keine weiteren Bibliotheken erfordern

Später werden wir die Bibliotheken der jeweiligen Anwendungen, also Word, Excel und Outlook, nutzen, um diese Programme fernzusteuern und beispielsweise Dokumente zu erstellen, mit Inhalten zu füllen oder auch auszulesen.

2.1 Assistenten aufrufen

Zunächst einmal gibt es einige Befehle, mit denen Sie die Befehle, die Sie sonst über die entsprechenden Ribbon-Einträge oder die Kontextmenübefehle der Objekte im Navigationsbereich aufrufen, ausführen können. Damit starten Sie dann den jeweiligen Assistenten. Wozu soll das gut sein? Vielleicht möchten Sie eine Access-Anwendung entwickeln, die keine der eingebauten Elemente der Benutzeroberfläche mehr enthält und nur mit Ihren eigenen Ribbon-Definitionen und Formularen ausgestattet ist. Wenn Sie dem Benutzer dann beispielsweise den Assistenten für den Import von Excel-Daten verfügbar machen wollen, können Sie den entsprechenden Befehl verwenden.

In der Beispieldatenbank finden Sie ein Formular, dass einige Schaltflächen zum Aufruf der bereits beschriebenen Assistenten enthält (siehe Abbildung 2.1).

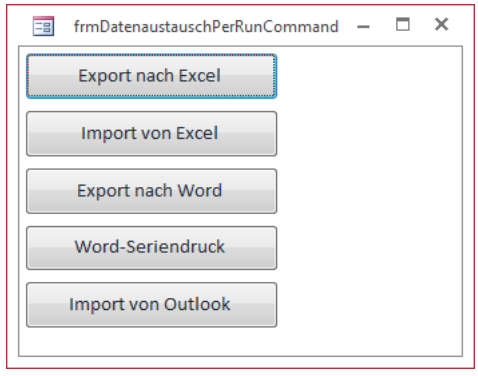


Abbildung 2.1: Formular zum Testen der Aufrufe der verschiedenen Import- und Export-Assistenten

2.1.1 Excel-Import

Den Assistenten für den Import oder die Verknüpfung von Daten aus Excel rufen Sie mit der folgenden Anweisung auf:

```
RunCommand acCmdImportAttachExcel
```

Dies öffnet den bereits bekannten Assistenten.

2.1.2 Excel-Export

Um den Export-Assistenten zu öffnen, verwenden Sie diese Anweisung:

```
RunCommand acCmdExportExcel
```

2.1.3 Word-Export

Die *RunCommand*-Konstanten für den Aufruf der Export-Assistenten beginnen alle mit *acCmdExport...* – wir auch dieser:

```
RunCommand acCmdExportRTF
```

2.1.4 Outlook-Import

Der Vollständigkeit halber der Aufruf des Outlook-Imports:

```
RunCommand acCmdImportAttachOutlook
```

Bei Verwendung des Imports mit einigen Ordnern ergeben sich Probleme, die bei Verwendung des von der Benutzeroberfläche gestarteten Import-Assistenten in einer entsprechenden Fehlermeldung resultieren. Wenn Sie den Assistenten mit der *RunCommand*-Methode aufrufen, wird die Funktion nicht nur nicht ausgeführt, es erscheint auch keine Fehlermeldung.

2.1.5 Word-Seriendruck

Den Word-Seriendruck-Assistenten starten Sie mit dieser Anweisung:

```
RunCommand acCmdWordMailMerge
```

An dieser Stelle sei erwähnt, dass vor dem Aufrufen des Seriendruck-Assistenten über den Ribbon-Eintrag oder das Kontextmenü des jeweiligen Elements die Datenquelle im Navigationsbereich markiert sein muss. Wenn Sie die *RunCommand acCmdWordMailMerge*-Methode über die Schaltfläche eines Formulars aufrufen, reicht dies nicht aus: Access meldet dann den Fehler aus Abbildung 2.2.

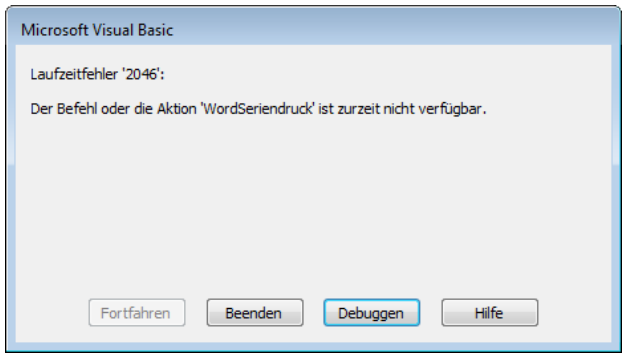


Abbildung 2.2: Fehler beim Starten des Seriendruck-Assistenten, wenn keine Datenquelle markiert oder geöffnet ist

Wenn Sie dies per Schaltfläche erledigen möchten, müssen Sie vorab die Tabelle oder Abfrage in der Datenblattansicht öffnen. Dazu gestalten wir die Prozedur, die durch das Ereignis *Beim Klicken* der Schaltfläche *cmdSeriendruck* ausgelöst wird, wie folgt:

```
Private Sub cmdSeriendruck_Click()
    DoCmd.OpenTable "tblArtikel"
    RunCommand acCmdWordMailMerge
End Sub
```

2.2 DoCmd.TransferSpreadsheet

Weiter oben im Kapitel ****Datenaustausch zwischen Access und den Office-Anwendungen haben Sie ja die verschiedenen Assistenten für den Export und Import von Daten kennengelernt. Gerade eben haben Sie erfahren, wie Sie diese Assistenten per VBA-Anweisung aufrufen. Sie können die mit den Assisten durchgeführten Exporte aber auch direkt durchführen. Dazu verwenden Sie verschiedene Methoden des *DoCmd*-Objekts.

Um diese aufzurufen, wenden Sie einfach die entsprechende Methode des *DoCmd*-Objekts an und übergeben die benötigten Parameter. Die Methoden finden Sie per IntelliSense ganz am Ende der Liste der möglichen Befehle (siehe Abbildung 2.3).

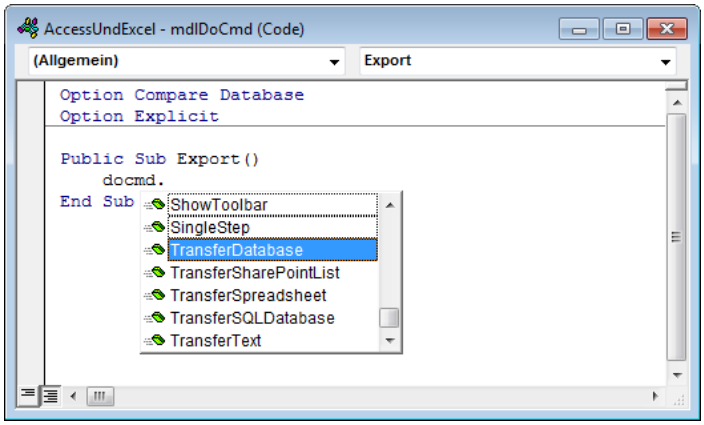


Abbildung 2.3: Verschiedene *DoCmd*-Methoden

Wenn Excel der Partner für den Import/Export sein soll, müssen Sie die Methode *TransferSpreadsheet* verwenden. Diese erwartet die folgenden Parameter (siehe auch Abbildung 2.4) – der letzte Parameter *UseOA* wird nicht unterstützt:

» *TransferType*: Ein Wert der Auflistung *acDataTransferType*. *acExport* (*1*) bedeutet, dass die Daten exportiert werden, bei *acImport* (*0*, Standardwert) werden diese importiert und bei *acLink* (*2*) stellt Access eine Verknüpfung zu der externen Datenquelle her.

» *SpreadsheetType*: Ein Wert der Auflistung *acSpreadSheetType*. Damit legen Sie fest, welche Excel-Variante verwendet wird: *acSpreadsheetTypeExcel12Xml* (*10*) ist das XML-Format von Excel 2010/2013, *acSpreadsheetTypeExcel12* (*9*) ist das normale Excel 2010/2013-Format, *acSpreadsheetTypeExcel9* (*8*) entspricht Excel 97/2000/2003.

» *TableName*: Name der Tabelle der Access-Datenbank, die an dem Import/Export beteiligt ist.

» *FileName*: Pfad der Excel-Tabelle, welche die Daten liefert oder mit Daten gefüllt werden soll.

» *HasFieldNames*: Gibt an, ob die erste Zeile der Excel-Tabelle Feldnamen enthält oder nicht. Bei Export werden die Feldnamen automatisch in die erste Zeile geschrieben.

» *Range*: Beim Importieren von Excel können Sie hier mit einen Bereich im Excel-Tabellenblatt festlegen, der importiert werden soll. Sie können auch einen sogenannten benannten Bereich angeben.

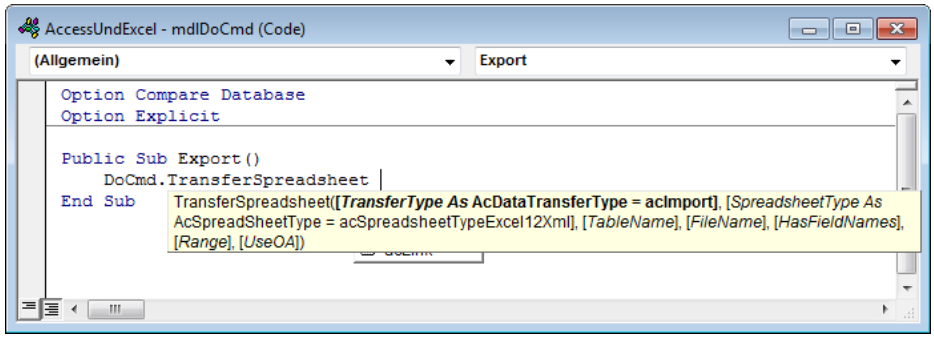


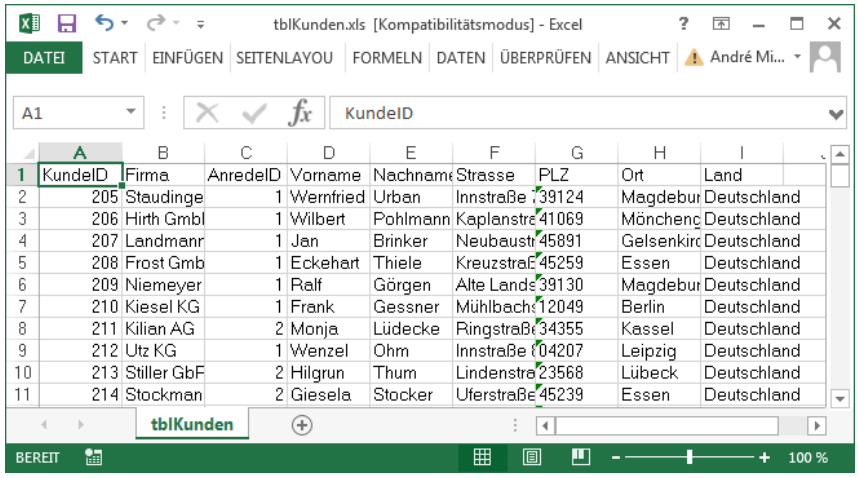
Abbildung 2.4: Parameter der *TransferSpreadsheet*-Methode per IntelliSense

2.2.1 Export nach Excel

Im ersten Beispiel wollen wir die Daten der Tabelle *tblKunden* in eine neue Excel-Datei namens *tblKunden.xls* exportieren. Wir nehmen an, dass der Kunde, der diese Tabelle erhält, mit einer älteren Excel-Version arbeitet, und legen für den Export den Typ *acSpreadsheetTypeExcel8* fest:

```
Public Sub Export()
    DoCmd.TransferSpreadsheet acExport, acSpreadsheetTypeExcel8, "tblKunden", _
        CurrentProject.Path & "\tblKunden.xls"
End Sub
```

Dieser Export liefert genau das erwartete Ergebnis (siehe Abbildung 2.5).



KundeID	Firma	AnredeID	Vorname	Nachname	Strasse	PLZ	Ort	Land
205	Staudinge	1	Wernfried	Urban	Innstraße	39124	Magdebur	Deutschland
206	Hirth GmbI	1	Wilbert	Pohlmann	Kaplanstra	41069	Mönchenc	Deutschland
207	Landmanr	1	Jan	Brinker	Neubaustr	45891	Gelsenkirc	Deutschland
208	Frost Gmb	1	Eckehart	Thiele	KreuzstraE	45259	Essen	Deutschland
209	Niemeyer	1	Ralf	Görgen	Alte Lands	39130	Magdebur	Deutschland
210	Kiesel KG	1	Frank	Gessner	Mühlbach:	12049	Berlin	Deutschland
211	Kilian AG	2	Monja	Lüdecke	Ringstraße	34355	Kassel	Deutschland
212	Utz KG	1	Wenzel	Ohm	Innstraße	04207	Leipzig	Deutschland
213	Stiller GbF	2	Hilgrun	Thum	Lindenstra	23568	Lübeck	Deutschland
214	Stockman	2	Giesela	Stocker	UferstraBe	45239	Essen	Deutschland

Abbildung 2.5: Export per *TransferSpreadsheet* nach Excel

2.2.2 Import von Excel

Nun wollen wir die Daten aus dieser Excel-Tabelle wieder in eine Access-Tabelle zurückholen. Dies können wir auf zwei Arten erledigen – mit dem Import in eine neue oder in eine bestehende Tabelle. Wir probieren zuerst den Import in eine neue Tabelle aus.

Dazu wählen wir nun als ersten Parameter die Konstante *acImport*, geben für den dritten Parameter den Namen einer neuen Tabelle namens *tblKunden_VonExcel* an und legen für den Parameter *HasFieldNames* den Wert *True* fest:

```
Public Sub Import()
    DoCmd.TransferSpreadsheet acImport, acSpreadsheetTypeExcel8, "tblKunden_VonExcel", _
        CurrentProject.Path & "\tblKunden.xls", True
End Sub
```

Damit erhalten wir fast wieder die Ausgangstabelle zurück – mit dem kleinen Unterschied, dass das Feld *KundeID* nicht als Primärschlüsselfeld des Typs *Autowert* festgelegt wurde (siehe Abbildung 2.6).

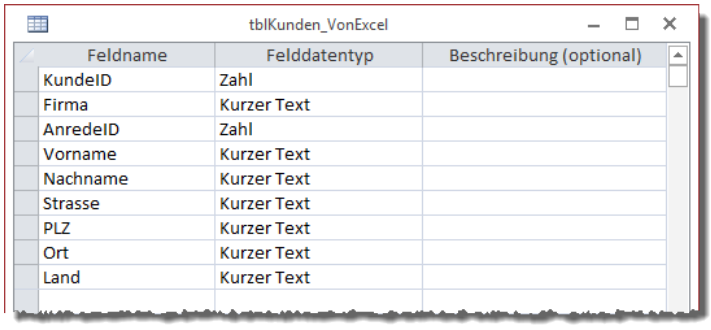


Abbildung 2.6: Tabellenentwurf der soeben importierten Tabelle

Daten anhängen

Wenn Sie versuchen, die Daten aus der Excel-Tabelle an die bestehende Tabelle *tblKunden* anzuhängen, erhalten Sie in diesem Fall eine Fehlermeldung, da die Werte des Feldes *KundeID* der Excel-Tabelle ja alle bereits im Primärschlüsselfeld *KundeID* der Access-Tabelle enthalten sind.

Sollten Sie jedoch Daten aus einer identisch aufgebauten Excel-Tabelle einlesen, deren Primärschlüsselwerte noch nicht in der Zieltabelle enthalten sind, dann fügt Access die neuen Datensätze ohne Probleme ein.

Import ohne Feldnamen

Wenn wir bei obiger Anweisung den letzten Parameter *HasFieldNames* von *True* auf *False* ändern, erhalten wir übrigens das gleiche Import-Ergebnis wie zuvor:

```
DoCmd.TransferSpreadsheet acImport, acSpreadsheetTypeExcel8, "tblKunden_OhneFeldnamen",
CurrentProject.Path & "\tblKunden.xls", False
```

2.2.3 Verknüpfen von Excel-Daten

Nun wollen wir die soeben erzeugte Excel-Tabelle als verknüpfte Tabelle in die aktuelle Access-Datenbank einbetten. Dazu brauchen wir im Vergleich um vorherigen Aufruf nur den ersten Parameter von *acImport* auf *acLink* zu ändern:

```
Public Sub Verknuepfen()
    DoCmd.TransferSpreadsheet acLink, acSpreadsheetTypeExcel8, "tblKunden",
CurrentProject.Path & "\tblKunden.xls", True
End Sub
```

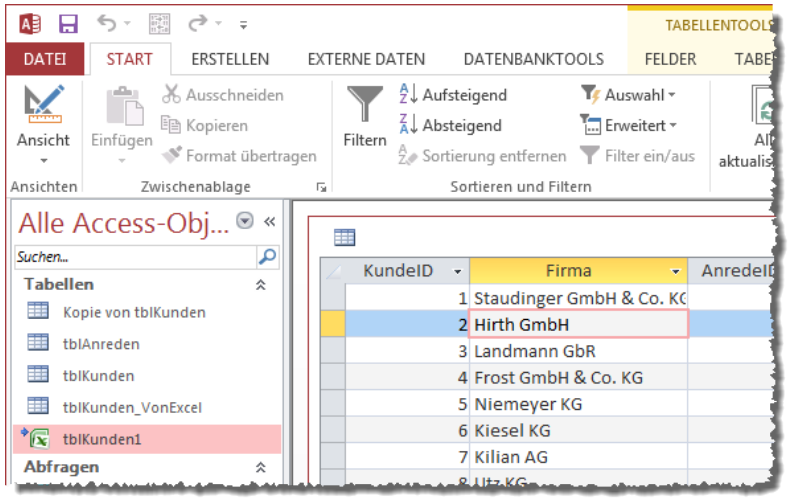


Abbildung 2.7: Eine verknüpfte Excel-Tabelle

Die Datensätze der verknüpften Tabelle sind allerdings nicht aktualisierbar.

2.2.4 Bereiche importieren

Werfen wir noch einen Blick auf den bisher noch nicht eingesetzten Parameter *Range*. Diesem können wir einen Bereich als Zeichenfolge angeben oder aber den Namen eines benannten Bereichs.

Wie wollen zunächst einmal probieren, nur die ersten drei Zeilen der Excel-Tabelle zu importieren (also vom Element *B1* bis zum Element *I4*). Geben wir diesen Bereich also als letzten Parameter an:

```
Public Sub ImportBereich()
    DoCmd.TransferSpreadsheet acImport, acSpreadsheetTypeExcel8, "tblKunden_Bereich", _
        CurrentProject.Path & "\tblKunden.xls", False, "A2:I4"
End Sub
```

Das Ergebnis sieht dann wie in Abbildung 2.8 aus. Logischerweise landen hier keine Spaltennamen in der importierten Tabelle, sodass Access die Felder mit *F1* bis *F8* benennt.

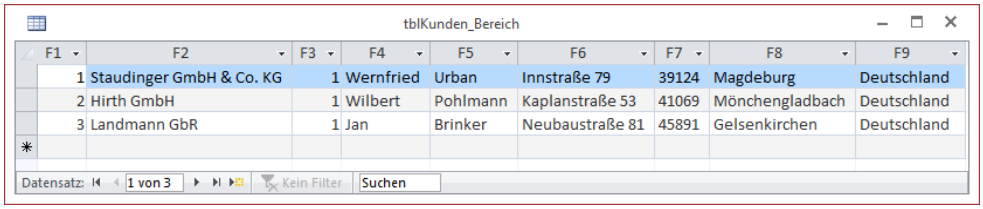


Abbildung 2.8: Ein importierter Bereich, allerdings ohne Feldnamen

Benannter Bereich

Nun können Sie in Access ein oder mehrere zusammenhängende Zellen mit einem Bereichsnamen versehen, unter dem Sie die vorgenommene Markierung später auch wieder abrufen können. Dazu markieren Sie den gewünschten Bereich und geben oben links, wo sonst der Name der aktuellen Zelle angezeigt wird, den Namen für den benannten Bereich ein (siehe Abbildung 2.9).

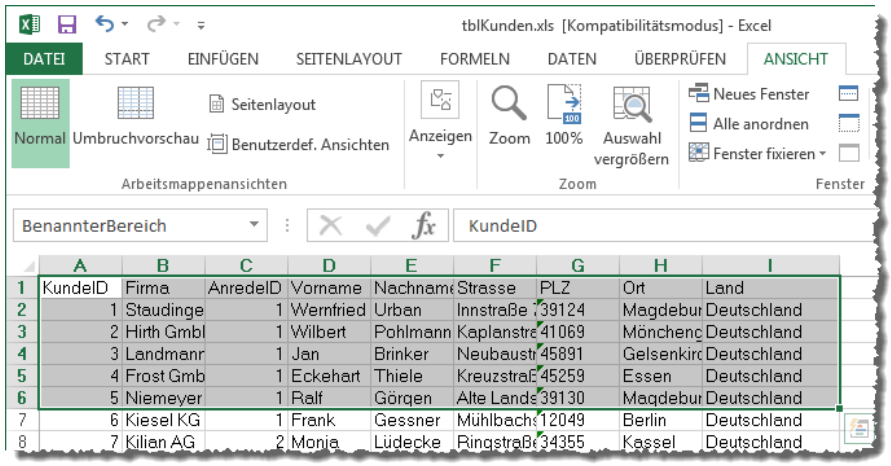


Abbildung 2.9: Benennen eines Bereichs

Diesen Bereich können Sie dann in Excel wieder aktivieren, indem Sie den entsprechenden Eintrag aus dem Kombinationsfeld wie in Abbildung 2.10 auswählen.

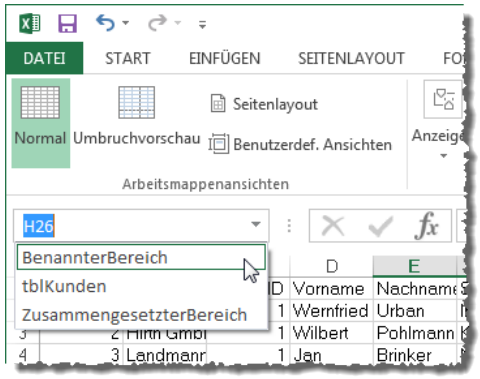


Abbildung 2.10: Auswählen eines benannten Bereichs

Wenn Sie einen solchen Bereich importieren möchten, müssen Sie einfach den Namen des benannten Bereichs als letzten Parameter der *TransferSpreadsheet*-Methode angeben:

```
Public Sub ImportBenannterBereich()
    DoCmd.TransferSpreadsheet acImport, acSpreadsheetTypeExcel8, _
        "tblKunden_BenannterBereich", CurrentProject.Path & "\tblKunden.xls", _
        True, "BenannterBereich"
End Sub
```

#ToDo: DoCmd.TransferSpreadsheet parametrisieren

2.3 DoCmd.TransferText

Es gibt noch eine weitere Möglichkeit, um Daten von Access nach Excel und zurück zu transportieren – und zwar mit der Methode *TransferText* des *DoCmd*-Objekts. Dieses ist zwar eigentlich für Textdateien gedacht, aber letztlich können Sie damit auch *.csv*-Dateien importieren und exportieren, was ja auch als Austauschformat für Excel verwendet werden kann.

#ToDo: Docmd.TransferText

» zum Beispiel Export der Daten an Firstinkasso automatisieren

» Export-Einstellungen speichern

2.4 Zugriff per SQL

Auf Excel-Dateien greift man am einfachsten durch einen entsprechenden Import oder eine Verknüpfung zu. Gelegentlich macht es auch Sinn, per Automation durch die Zellen zu huschen, um die gewünschten Informationen zu lesen oder zu schreiben. Eine wenig beachtete Möglichkeit ist die, per SQL oder DAO auf die in einer Excel-Tabelle gespeicherte Tabelle zuzugreifen.

Die Beispiele finden Sie im Formular *frmSQLDAOZugriff* der Beispieldatenbank. Das Formular enthält drei Textfelder zur Eingabe der Excel-Datei, des Tabellennamens sowie des Bereichs, auf den die Prozeduren zugreifen sollen (siehe Abbildung 2.11).

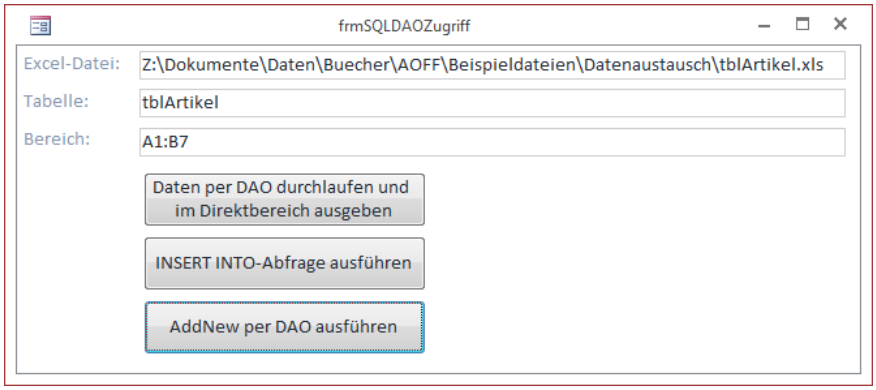


Abbildung 2.11: Formular zum Ausprobieren der folgenden Beispiele

3 VBA-Zugriff auf Office

Richtig interessant wird die Interaktion zwischen Access und den übrigen Office-Produkten, wenn Sie diese per VBA für die Zwecke Ihrer Datenbank-Anwendung einsetzen können. Das bedeutet beispielsweise, dass Sie Outlook nutzen, um eine E-Mail von Access aus zu verschicken, ein Word-Dokument mit den Daten aus eine Adressdatensatz füllen oder auch eine komplette Tabelle nach Ihren Wünschen formatiert in eine Excel-Tabelle schreiben, damit diese etwa von einer anderen Anwendung weitergenutzt werden kann.

Das A und O ist dabei der Zugriff auf die entsprechenden Typbibliotheken der verschiedenen Office-Anwendungen. Die erste Voraussetzung dabei ist, dass die gewünschte Anwendung überhaupt auf dem Rechner installiert ist. Die zweite ist, dass die jeweilige Zielanwendung in einer Version vorliegt, die alle von Ihnen genutzten Objekte, Methoden und Eigenschaften sowie Ereignisse unterstützt – was hilft es, wenn Sie eine Access-Anwendung, die Outlook 2007-Features nutzen will, auf einem Rechner installieren, der nur Outlook 2003 enthält?

Um den Zugriff auf die Typbibliothek und somit auch auf die jeweilige Anwendung zu ermöglichen, gibt es zwei Möglichkeiten: Sie binden die Bibliothek mithilfe eines Verweises fest in das VBA-Projekt der Access-Anwendung ein. Dann können Sie ganz einfach über das Objektmodell auf die Elemente der Bibliothek zugreifen. Oder Sie erstellen den Verweis auf die benötigten Objekte erst zur Laufzeit. Beides hat Vor- und Nachteile – welche, erfahren Sie im Anschluss.

3.1 Early Binding

Wenn Sie per Early Binding auf die VBA-Bibliothek einer der anderen Office-Anwendungen zugreifen wollen, müssen Sie zunächst einen Verweis auf diese Bibliothek erstellen. Dies erledigen Sie, indem Sie den VBA-Editor öffnen (wahlweise mit der Tastenkombination *Strg + G* oder *Alt + F11*) und dort den Menüeintrag *Extras|Verweise* aufrufen.

Hier finden Sie in einer jungfräulichen Access-Datenbank nur einige wenige aktivierte Verweise vor. Wenn Sie nun nach unten scrollen, stoßen Sie früher oder später auf den Eintrag *Microsoft Excel x.0 Object Library* (das x steht für die interne Anwendungs-Version, also *15* für 2013, *14* für 2010 und *12* für 2007 – die *13* hat Microsoft ausgelassen ...). Diesen Eintrag fügen Sie durch Anklicken des Kontrollkästchens zum aktuellen VBA-Projekt hinzu (siehe Abbildung 3.1).

3.1.1 Vorteil: Zugriff per Objektkatalog

Die erste interessante Änderung ist nun, dass Sie im Objektkatalog (*F2*), der ja Informationen über die Objekte und ihre Methoden, Eigenschaften und Ereignisse liefert, einen Eintrag na-

mens *Excel* zur Auswahl vorfinden (siehe Abbildung 3.2). Damit können Sie nun schon einmal die einzelnen Objekte ansehen.

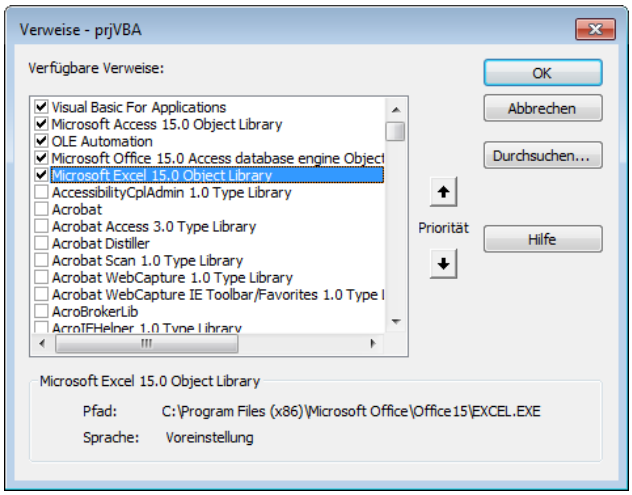


Abbildung 3.1: Einfügen eines Verweises auf die Objektbibliothek von Excel

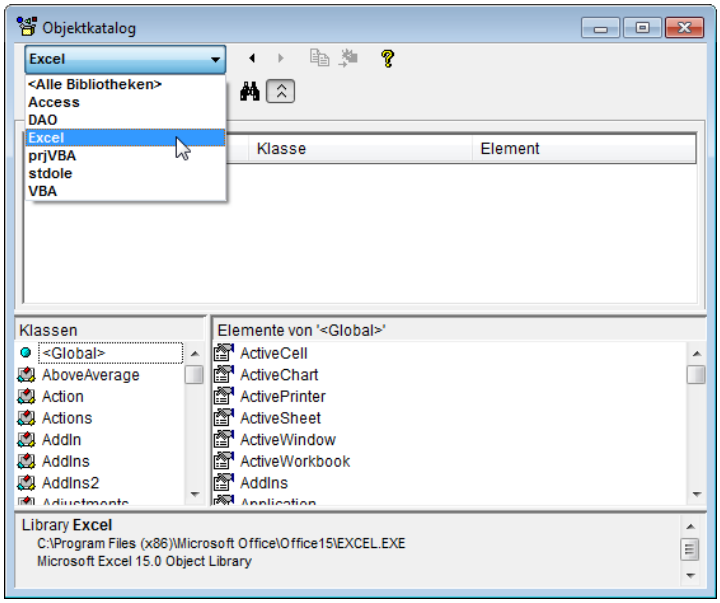


Abbildung 3.2: Excel im Objekt-Katalog

Auf die gleiche Weise fügen Sie die Objektverweise für die Bibliotheken von *Outlook*, *Word* oder *PowerPoint* hinzu. Gelegentlich werden Sie auch die Bibliothek *Microsoft Office x.0 Object*

Library benötigen – etwa, wenn Sie die Kontextmenüs einiger Anwendungen oder das Ribbon anpassen möchten.

3.1.2 Vorteil: Nutzung von IntelliSense

Den ersten auf Anhieb sichtbaren Vorteil bei der Nutzung von Early Binding erkennen Sie, wenn Sie ein Objekt dieser Bibliothek deklarieren möchten.

Wenn Sie nun den Namen der Bibliothek, also Excel, in den VBA-Editor eingeben und dann einen Punkt hinzufügen, blendet der VBA-Editor alle Objekte, Eigenschaften und Methoden dieses Objekts ein (siehe Abbildung 3.3).

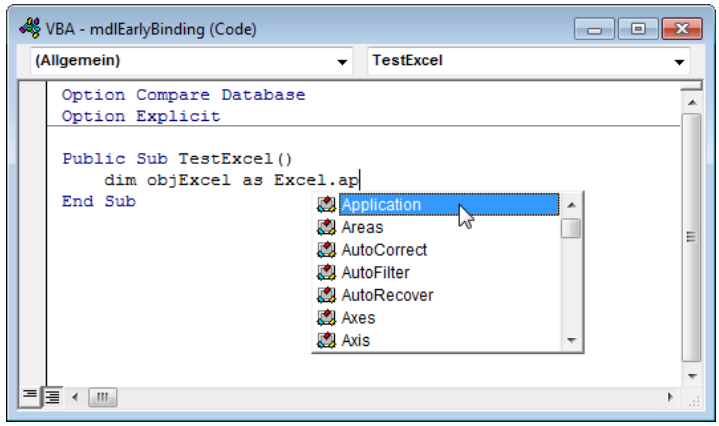


Abbildung 3.3: Nutzung von IntelliSense

3.1.3 Vorteil: Nutzung von Konstanten statt von Zahlenwerten

Ein weiterer Vorteil ist, dass Sie für Parameter von Funktionen und Methoden, die ihre Werte aus einer der eingebauten Enumerationen beziehen, die Konstanten verwenden können statt der reinen Zahlenwerte (siehe Abbildung 3.4).

Das bedeutet, dass Sie zum Beispiel bei der Zuweisung der Rahmenart in Excel erstens den Rahmen mit *xlEdgeTop* (oberer Rahmen) und die Rahmenart mit *xlContinuous* (durchgehender Rahmen) definieren können:

```
objWorksheet.Range("A1").Borders(xlEdgeTop).LineStyle = xlContinuous
```

Jede der mit *xl...* beginnenden Konstanten entspricht einem Zahlenwert, den Sie im Direktfenster mit folgender Anweisung ermitteln können:

```
? xlContinuous
 1
```

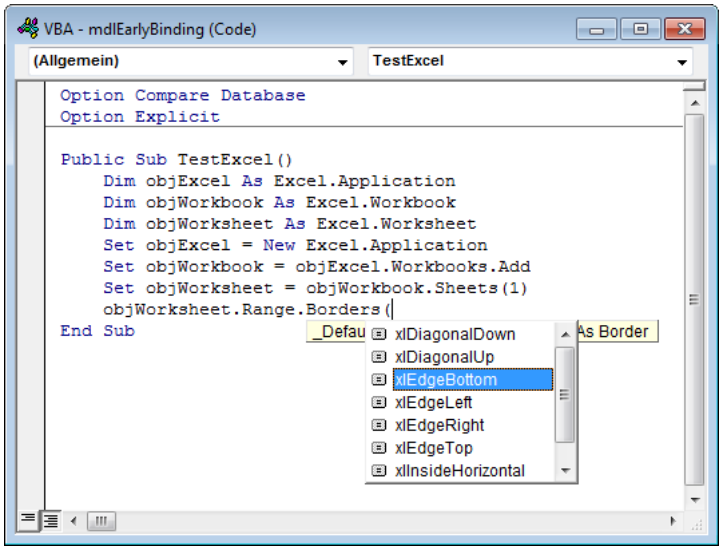


Abbildung 3.4: Verwendung von Konstanten statt Zahlenwerten als Parameter

Sie könnten der Anweisung also auch die Zahlenwerte statt der Konstanten zuweisen, was den Code aber nicht besonders lesbar macht.

Die Konstanten für die einzelnen Bibliotheken haben übrigens jeweils ein eigenes Präfix:

» *xl*: Excel

» *ac*: Access

» *wd*: Word

» *ol*: Outlook

» *pp*: PowerPoint

3.1.4 Vorteil: Ereignisse implementierbar

Der nächste Vorteil von Early Binding ist, dass Sie nur auf diese Weise Ereignisse von Objekten implementieren können, die solche anbieten. Ein Beispiel ist das *Application*-Objekt von Excel. Um dessen Ereignisse implementieren zu können, wie Sie etwa die Ereignisse eines Formulars oder eines Steuerelements nutzen, müssen Sie zunächst eine entsprechende Variable deklarieren und mit dem Schlüsselwort *WithEvents* ausstatten.

Voraussetzung dafür ist, dass Sie die Variable in einem Klassenmodul deklarieren, also entweder in einem einfachen Klassenmodul, dass Sie über *Einfügen|Klassenmodul* zum Projekt hinzufügen oder über das Klassenmodul etwa eines Formulars.

4 Excel programmieren

Die Elemente eines Excel-Dokument lassen sich wunderbar referenzieren – teilweise sogar über verschiedene Methoden. Dieses Kapitel stellt die grundlegenden Objekte unter Excel vor und zeigt, wie Sie mit VBA auf diese zugreifen. Dazu gehören die Excel-Anwendung selbst, die darin geöffneten Dateien und die darin enthaltenen Tabellenblätter.

Wenn Sie Daten aus einer Excel-Datei lesen oder in eine Excel-Datei schreiben möchten, müssen Sie genau wissen, wie Sie auf die verschiedenen Bereiche der Tabelle zugreifen – egal, ob es sich nun um einzelne Zellen oder komplette Bereiche handelt.

Dieses Kapitel stellt die grundlegenden Objekte von Excel und Excel-Tabellen vor und zeigt, wie Sie per VBA von Access aus auf diese zugreifen.

4.1 Warum Excel programmieren?

Wenn Sie per VBA auf die Inhalte von Excel-Dokumenten zugreifen wollen, kann das verschiedene Gründe haben. Im Grunde lassen sich diese jedoch auf die folgenden beiden reduzieren:

» Sie möchten Daten aus einem Excel-Dokument auslesen.

» Sie möchten Daten in ein Excel-Dokument eintragen.

Hinzu kommt gegebenenfalls noch eine Erweiterung des zweiten Punktes, nämlich das Erstellen kompletter Excel-Dokument, die nicht nur Daten enthalten, sondern gegebenenfalls auch noch Formeln für die Durchführung von Berechnungen aufgrund der enthaltenen Daten – deren Ergebnisse später an Access zurückgegeben werden.

Programmierung von Excel aus oder von außerhalb?

Wenn Sie im VBA-Editor von Excel programmieren, stehen Ihnen die Objekte von Excel direkt zur Verfügung. Sie müssen beispielsweise nicht erst eine Excel-Instanz erstellen, um auf eine zuzugreifen, denn der Code befindet sich ja schon in einer. Sie können dort einfach mit *Application* auf die Instanz zugreifen. Wenn Sie beispielsweise von Access aus auf Excel oder ein darin angezeigtes Dokument zugreifen möchten, können Sie das Objekt *Application* natürlich nicht für den Zugriff auf Excel nutzen – dieses ist ja das Access-*Application*-Objekt. Als müssen Sie, wie die folgenden Abschnitte ausführlich erläutern, zuerst eine Excel-Instanz erstellen und diese oder eine bestehende Instanz referenzieren.

Die folgenden Beispiele gehen sämtlich davon aus, dass sie vom VBA-Projekt einer Access-Datenbank aus ausgeführt werden.

4.2 Zugriff auf Excel

Was auch immer Sie mit Excel anstellen möchten: Sie benötigen zunächst eine Excel-Instanz und müssen dann das zu lesende oder zu bearbeitende Dokument öffnen oder erstellen. Dies können Sie teilweise in einem Schritt erledigen, teilweise sind mehrere Schritte nötig. Die einzelnen Varianten schauen wir uns gleich an.

Wie üblich, können Sie vor dem VBA-Zugriff auf Excel einen Verweis zum VBA-Projekt hinzufügen (Early Binding) oder die Objekte mit dem Datentyp *Object* deklarieren und diese erst zur Laufzeit erstellen (Late Binding). Die beiden Vorgehensweisen werden ausführlich im Kapitel ****VBA-Zugriff auf Office* erläutert.

Im Rahmen dieses Kapitels nutzen wir ausschließlich Early Binding, das heißt, dass wir zunächst einen Verweis auf die Excel-Objektbibliothek erstellen. Dies erledigen Sie, indem Sie vom Access-Fenster aus mit *Strg + G* den VBA-Editor öffnen und dort den Menübefehl *Extras|Verweise* aufrufen. Dies zeigt den *Verweise*-Dialog an, mit dem Sie einen Verweis auf die Bibliothek *Microsoft Excel x.0 Object Library* hinzufügen – das x steht dabei für die verfügbare Version (siehe Abbildung 4.1).

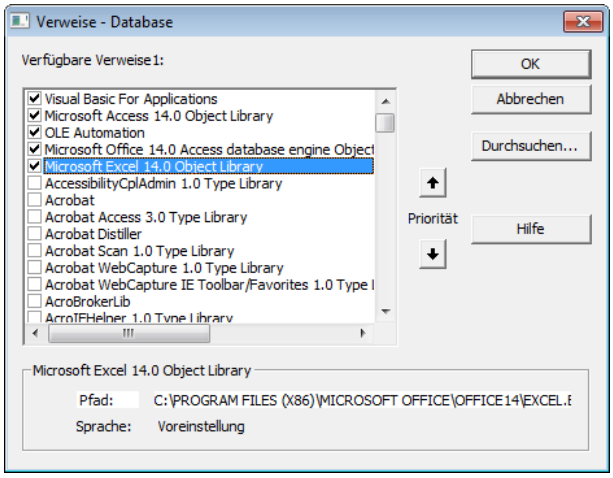


Abbildung 4.1: Einbinden eines Verweises auf die Excel-Objektbibliothek

4.3 Das Application-Objekt

Auf eine Excel-Instanz greifen Sie wie beiden übrigen Office-Anwendungen über das *Application*-Objekt zu.

4.3.1 Excel-Instanz erstellen per Early Binding

Wenn Sie eine Excel-Instanz erstellen, werden Sie den Verweis darauf in aller Regel in einer Variablen speichern. Diese Variable deklarieren Sie in allen folgenden Beispielen wie folgt:

```
Dim objExcel As Excel.Application
```

Wenn Sie mit Early Binding arbeiten, erstellen Sie mit der folgenden Anweisung die neue Excel-Instanz und speichern einen Verweis darauf in der Variablen *objExcel*:

```
Set objExcel = New Excel.Application
```

Wenn Sie die Excel-Instanz nicht mehr benötigen, können Sie einfach die Objektvariable auf *Nothing* setzen:

```
Set objExcel = New Excel.Application
```

Um zu sehen, ob die Excel-Instanz wirklich erstellt und wieder beendet wird, können Sie einmal die folgende Prozedur aus dem Modul *mdlAccessUndExcel* der Beispieldatenbank ausführen:

```
Public Sub ExcelInstanz()
    Dim objExcel As Excel.Application
    Stop 'keine Excel-Instanz
    Set objExcel = New Excel.Application
    Stop 'Excel-Instanz vorhanden
    Debug.Print objExcel.Name
    Set objExcel = Nothing
    Stop 'Keine Excel-Instanz
End Sub
```

Neben dem VBA-Editor öffnen Sie dabei den Taskmanager und beobachten, ob und wann dort ein Eintrag namens *EXCEL.EXE *32* erscheint (für die 32bit-Version). Dies sollte, wenn an der ersten Haltemarke (*Stop*) noch kein solcher Eintrag vorliegt, an der zweiten Haltemarke wie in Abbildung 4.2 aussehen. Später beim Erreichen der dritten Haltemarke verschwindet dieser Eintrag wieder.

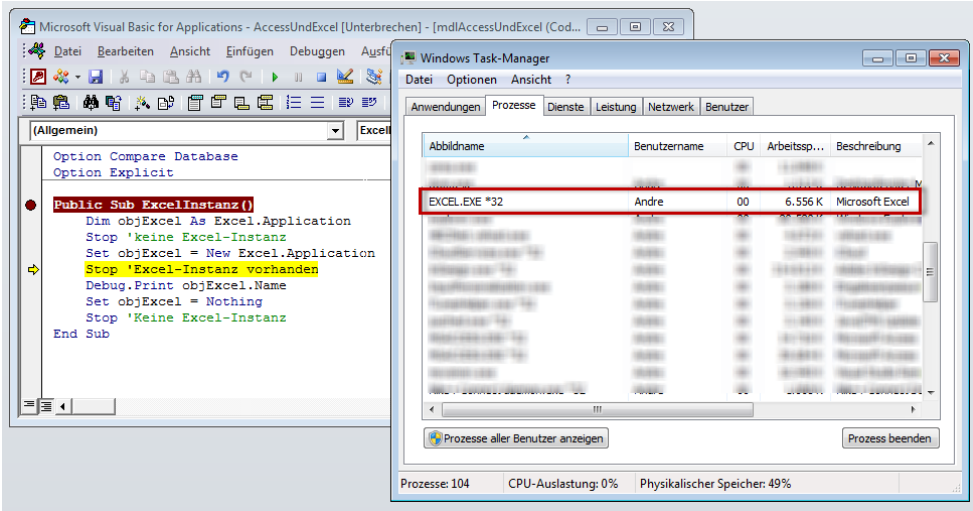


Abbildung 4.2: Excel-Instanz im Task-Manager

4.3.2 Mehrere Instanzen

Auf einem Rechner können mehrere Instanzen von Excel gleichzeitig laufen. Dies schauen wir uns wie folgt an. Zunächst deklarieren wir eine globale Variable, die einen Verweis auf eine Excel-Instanz aufnimmt:

```
Public objExcelGlobal As Excel.Application
```

Dann erstellen wir eine neue Excel-Instanz und referenzieren diese mithilfe der folgenden Prozedur:

```
Public Sub ExcelGlobalInstanzieren()
    Set objExcelGlobal = New Excel.Application
End Sub
```

Danach finden Sie im Task-Manager eine einzelne Excel-Instanz vor. Nun führen Sie die weiter oben vorgestellte Prozedur *ExcelInstanz* bis zur zweiten *Stop*-Anweisung aus. Ein Blick in den Task-Manager liefert nun zwei Einträge namens *EXCEL.EXE *32*.

Wenn Sie die Prozedur nun bis zum Ende durchlaufen lassen, verschwindet ein Eintrag wieder. Den zweiten Eintrag entfernen Sie, indem Sie die entsprechende Objektvariable auf den Wert *Nothing* einstellen – und zwar mit der folgenden Prozedur:

```
Public Sub ExcelGlobalZerstoeren()
    Set objExcelGlobal = Nothing
End Sub
```

Dieses Experiment liefert eine wichtige Erkenntnis: Wenn Sie von VBA aus eine neue Excel-Instanz erzeugen, handelt es sich um eine eigenständige Instanz. In dieser können Sie tun und lassen, was Sie möchten – Dokumente anlegen, ändern, löschen, die Instanz wieder zerstören. Das ist wichtig, weil es ja sein kann, dass der Benutzer bereits eine (sichtbar) Excel-Instanz auf dem Rechner gestartet hat. Würden Sie diese referenzieren, müssten Sie vor dem Beenden der Instanz beispielsweise prüfen, ob nicht eventuell noch Dokumente geöffnet sind, die der Benutzer noch benötigt.

4.3.3 Bestehende Instanz referenzieren

Dennoch schauen wir uns an, wie Sie eine bereits geöffnete Excel-Instanz referenzieren und darauf zugreifen.

```
Public Sub BestehendeExcelInstanz()
    'Setzt voraus, dass eine Excel-Instanz geöffnet ist
    Dim objExcel As Excel.Application
    Set objExcel = GetObject(, "Excel.Application")
    Debug.Print objExcel.Name
End Sub
```

Das funktioniert aber nur, wenn auch eine Excel-Instanz vorliegt. Wenn nicht, erhalten Sie eine Fehlermeldung mit der Nummer *429* und dem Text *Objekterstellung durch ActiveX-Komponente nicht möglich* (siehe Abbildung 4.3).

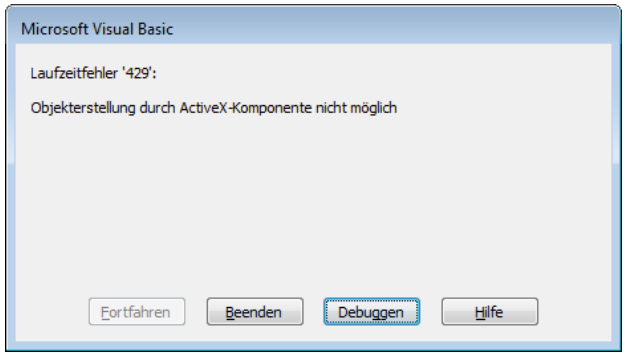


Abbildung 4.3: Diese Fehlermeldung erscheint beim Versuch, eine nicht vorhandene Excel-Instanz zu referenzieren.

Diesen Fehler können Sie im Code beispielsweise so abfangen:

```
Public Sub BestehendeExcelInstanzMitMeldung()
    Dim objExcel As Excel.Application
    On Error Resume Next
```

```
    Set objExcel = GetObject(, "Excel.Application")
    If Err.Number = 429 Then
        MsgBox "Excel ist nicht geöffnet."
        Exit Sub
    End If
    MsgBox "Excel ist geöffnet."
End Sub
```

Wenn bei nicht vorhandener Excel-Instanz eine neue geöffnet werden soll, erweitern Sie den Code wie folgt:

```
Public Sub BestehendeOderNeueExcelInstanz()
    Dim objExcel As Excel.Application
    On Error Resume Next
    Set objExcel = GetObject(, "Excel.Application")
    If Err.Number = 429 Then
        Set objExcel = CreateObject("Excel.Application")
    End If
    MsgBox "Excel ist geöffnet."
End Sub
```

objExcel enthält danach auf jeden Fall einen Verweis auf eine Excel-Instanz - vorausgesetzt, dass Excel überhaupt auf dem Rechner installiert ist. Nun haben Sie allerdings zwei verschiedene Methoden gesehen, um eine frische Excel-Instanz zu erzeugen:

```
Set objExcel = New Excel.Application
```

und

```
Set objExcel = CreateObject("Excel.Application")
```

Wo ist hier der Unterschied? Beide Anweisungen erledigen die gleiche Aufgabe, allerdings funktioniert die erste nur, wenn Sie mit Early Binding arbeiten, also ein Verweis auf die Excel-Bibliothek vorhanden ist.

An dieser Stelle sei auch auch noch etwas zum Übernehmen einer bestehenden Excel-Instanz angemerkt: Sie sollten dies nur tun, wenn es die Situation unbedingt erfordert, wofür es kaum Beispiele gibt. Gegebenenfalls könnte man es mit Performance-Gründen rechtfertigen, aber bei der heutigen Rechenleistung sollte ein Rechner zwei oder mehr parallel geöffnete Excel-Instanzen verkraften können.

Wenn Sie für Ihre Aufgabe eine eigene Excel-Instanz öffnen und diese nicht sichtbar machen, können Sie davon ausgehen, dass der Benutzer nicht anderweitig auf diese zugreift. Daher können Sie diese Instanz auch ohne Gewissensbisse wieder schließen - Sie nehmen sie niemandem weg.

Und wenn Sie sich davon überzeugen möchten, dass Windows tatsächlich eine neue Excel-Instanz erzeugt und keine vorhandene Instanz verwendet, wenn Sie per Code eine neue Instanz erzeugen, können Sie das Experiment von oben in abgewandelter Form ausführen und sich im Task-Manager davon überzeugen, dass mehrere Excel-Instanzen existieren. Sie können aber auch einfach eine Excel-Instanz von Hand öffnen und dann eine mit der New-Anweisung – Sie werden schnell erkennen, dass die bereits bestehende Instanz davon unberührt bleibt.

Hier brauchen Sie im Übrigen nur die Excel-Objektvariablen durch Setzen auf *Nothing* zu leeren, um die Excel-Instanzen zu beenden - später erfahren Sie, dass dies nicht immer ausreicht.

4.3.4 Funktion zum Erzeugen einer Excel-Instanz

Manchmal brauchen Sie eine Excel-Instanz nur ganz kurz, um etwa ein Excel-Workbook zu öffnen, manchmal arbeiten Sie auch länger mit dieser Instanz.

In beiden Fällen kann es nicht schaden, den Code zum Erzeugen der Excel-Instanz in eine kleine Funktion auszulagern. Die Excel-Instanz selbst speichern wir in einer Objektvariablen in einem Standardmodul, dass beispielsweise *mdlExcel* heißen könnte:

```
Private m_Excel As Excel.Application
```

Dazu bauen wir uns eine kleine Funktion, die schaut, ob bereits eine Excel-Instanz in *mExcel* gespeichert wurde und dies gegebenenfalls nachholt. Der in *mExcel* gespeicherte Verweis wir in jedem Fall als Übergabewert der Funktion *GetExcel* zurückgeliefert:

```
Public Function GetExcel() As _
    Excel.Application
    If m_Excel Is Nothing Then
        Set m_Excel = _
        New Excel.Application
    End If
    Set GetExcel = m_Excel
End Function
```

Wenn Sie nun mit der Anweisung *GetExcel* eine neue Instanz holen wollen, um damit eine bestimmte Aufgabe zu erledigen, gelingt dies beispielsweise wie folgt:

```
Public Sub TestExcelfunktion()
    Debug.Print GetExcel.Name
End Sub
```

Führen Sie diese Prozedur im Einzelschrittmodus aus (Einfügemarke in die Prozedur setzen und dann mit *F8* schrittweise durch die Anweisungen laufen). Im ersten Durchlauf liefert *m_Excel Is Nothing* den Wert *True* und die Prozedur durchläuft den inneren Teil der *If...Then*-Bedingung (siehe Abbildung 4.4). Der Verweis auf die Excel-Instanz landet in der Variablen *m_Excel*.

Durchlaufen Sie die Prozedur nun ein zweites Mal, hat *m_Excel Is Nothing* den Wert *False* und die Funktion liefert einfach den Wert aus der Variablen *m_Excel* als Ergebnis zurück.

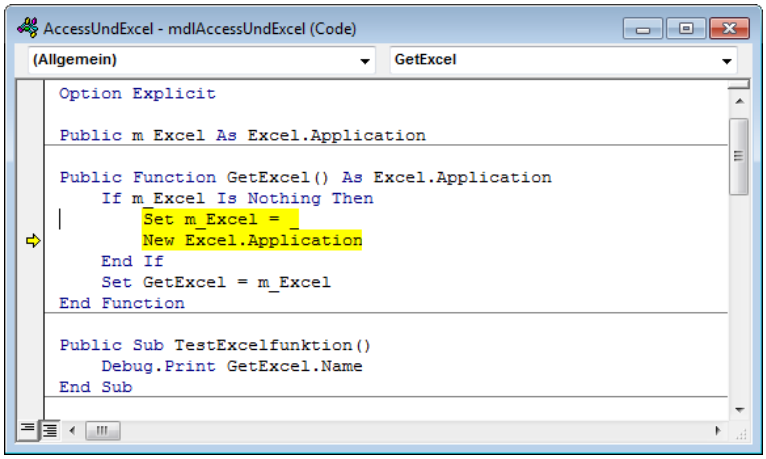


Abbildung 4.4: Funktion zum Holen einer Excel-Instanz

Was bringt diese Technik nun? Sie sparen damit, wenn Sie in mehreren Prozeduren auf eine Excel-Instanz zugreifen wollen, einige Zeilen Code.

4.3.5 Excel sichtbar machen

Alles, was wir in den bisherigen Beispielen mit Excel erledigt haben (was ja nicht viel war), geschah im Hintergrund, also ohne Anzeigen des Excel-Fensters.

Bevor wir gleich in den Zugriff auf Excel-Dateien einsteigen, erfahren Sie noch, wie Sie die Excel-Instanz sichtbar machen – Sie wollen ja auch beobachten, was die beschriebenen Codezeilen bewirken. Dies erledigt die folgende Anweisung:

```
objExcel.Visible = True
```

4.4 Mit Excel-Dateien arbeiten

Nun, da Sie wissen, wie Sie eine Excel-Instanz erstellen oder auf eine bestehende Instanz zugreifen, wenden wir uns den Excel-Dateien zu.

4.4.1 Excel-Datei erzeugen

Wenn Sie in *objExcel* einen Verweis auf eine Excel-Instanz gespeichert haben, können Sie damit ein neues Dokument erzeugen. Dafür deklarieren Sie zunächst eine entsprechende Variable:

```
Dim objWorkbook As Excel.Workbook
```

Danach können Sie diese Variable durch Verwenden der *Add*-Anweisung der *Workbooks*-Auflistung des *Application*-Objekts erstellen:

```
Set objWorkbook = objExcel.Workbooks.Add
```

Nun können Sie die Excel-Datei mit den weiter unten beschriebenen Methoden bearbeiten und diese dann schließen:

```
objWorkbook.Close
```

Danach schließen Sie die Excel-Instanz und leeren dann die Objektvariable *objExcel*:

```
objExcel.Quit
Set objExcel = Nothing
```

Es gibt aber auch noch eine wesentlich direktere Methode, die Sie vor allem dann nutzen sollten, wenn Sie nur auf das Workbook und nicht direkt auf die Excel-Instanz zugreifen möchten:

```
Set objWorkbook = CreateObject("Excel.Sheet")
```

Ein vollständiges Beispiel, dass Sie im Einzelschrittmodus durchlaufen können, sieht wie folgt aus:

```
Public Sub WorkbookErstellen()
    Dim objExcel As Excel.Application
    Dim objWorkbook As Excel.Workbook
    Set objExcel = New Excel.Application
    objExcel.Visible = True
    Set objWorkbook = objExcel.Workbooks.Add
    objWorkbook.Sheets(1).Range("A1") = "bla"
    On Error Resume Next
    Kill CurrentProject.Path & "\Leer.xlsx"
    On Error GoTo 0
    objWorkbook.Close True, CurrentProject.Path & "\Leer.xlsx"
    objExcel.Quit
    Set objExcel = Nothing
End Sub
```

Die Prozedur erstellt zunächst eine neue Excel-Instanz und referenziert diese mit *objExcel*. Diese blendet sie ein und fügt der Auflistung *Workbooks* der Instanz mit der *Add*-Methode ein neues Objekt hinzu. Um das Objekt zu bearbeiten, damit es gleich gespeichert wird, fügt die Prozedur für die Zelle *A1* einen Beispieltext ein. Dann löscht sie eine eventuell bereits vorhandene Datei namens *Leer.xls*. Die folgende *Close*-Methode ist so eingestellt, dass sie eine geänderte Datei automatisch speichert – und zwar unter dem mit dem zweiten Parameter angegebenen

Dateinamen. Schließlich beendet die Prozedur die Excel-Instanz und leert die Objektvariable *objExcel*.

4.4.2 Excel-Datei öffnen

Für den Zugriff auf eine bestehende Excel-Datei verwenden Sie die folgende Anweisung, die auf der oben genannten Vorgehensweise zum Erzeugen oder Holen einer Excel-Instanz basiert.

Mit dieser Instanz können Sie dann Folgendes tun:

```
Set objWorkbook = objExcel.Workbooks.Open(CurrentProject.Path & "\Leer.xls")
```

Auch hier gibt es eine zweite Variante. Dabei übergeben Sie einfach den Dateinamen an die *CreateObject*-Methode:

```
Set objWorkbook = CreateObject(CurrentProject.Path & "\Leer.xls")
```

Hier ist zu beachten, dass wir direkt auf die gespeicherte Datei zugreifen. Das bedeutet, dass wir kein Objekt des Typs *Excel.Application* benötigen. Was aber, wenn wir das Excel-Fenster nun einblenden möchten? Kein Problem: Das *Application*-Objekt ist dem Workbook direkt übergeordnete, sodass wir das Excel-Fenster wie folgt einblenden können:

```
objWorkbook.Parent.Visible = True
```

4.4.3 Auf geöffnete Excel-Datei zugreifen

Richtig interessant für die nachfolgenden Beispiele ist die Tatsache, dass Sie auch eine bereits geöffnete Excel-Datei referenzieren können – egal ob diese per VBA oder manuell geöffnet wurde. Dazu verwenden wir ganz einfach die *GetObject*-Methode und übergeben dieser Den Namen der zu öffnenden Datei.

4.4.4 Funktion zum einfachen Öffnen einer Excel-Datei

Genau wie für das Erzeugen einer Excel-Instanz wollen wir auch eine einfache Funktion zum Öffnen eines Excel-Workbooks programmieren.

Diese setzt auf der oben bereits vorgestellten Funktion *GetExcel* auf und und öffnet die als Parameter angegebene Datei in der damit verfügbar gemachten Excel-Instanz:

```
Public Function GetWorkbook(strPath As String) As Excel.Workbook
    Dim objWorkbook As Excel.Workbook
    Set objWorkbook = GetExcel.Workbooks.Open(strPath)
    Set GetWorkbook = objWorkbook
End Function
```

Wenn Sie ein Workbook öffnen und in einer Prozedur damit arbeiten möchten, brauchen Sie also nur eine entsprechende Objektvariable zu deklarieren und diese mit Hilfe der Funktion *GetWorkbook* zu füllen:

```
Dim objWorkbook As Excel.Workbook
Set objWorkbook = GetWorkbook(CurrentProject.Path & "Leer.xls")
```

Um die nachfolgenden Beispiele kurz zu halten, greifen wir immer wieder auf diese beiden Zeilen zurück.

4.4.5 Speichern einer Excel-Datei

Weiter oben haben wir die frisch erstellte Excel-Datei *Leer.xlsx* implizit gespeichert, indem wir mit den Parametern der *Close*-Methode festgelegt haben, dass das Workbook gespeichert werden soll, wenn es geändert wurde – und unter welchem Dateinamen. In einem vorherigen Versuch hatte ich dort noch die Dateiendung *.xls* verwendet. Dies führt beim nächsten Öffnen dieses neu erstellten Workbooks zu einem Fehler, denn diese Excel-Version verwendet zum Speichern mit *Close* automatisch das der aktuellen Version entsprechende Dateiformat. Das heißt, dass das Workbook zwar mit einer älteren Dateiendung, aber im neuen Format gespeichert wurde – keine gute Kombination, wenn man das Workbook später nochmals öffnen möchte.

Also speichert man Workbooks am besten direkt explizit mit der *SaveAs*-Methode des *Workbook*-Objekts. Diese erwartet eine ganze Reihe Parameter, von denen uns aktuell nur die ersten beiden interessieren. Der erste erwartet den Namen der Zieldatei, der zweite die Excel-Version, für die das Workbook gespeichert werden soll.

Dummerweise liefert der VBA-Editor zwar die Liste der Parameter, aber nicht die für den zweiten Parameter *FileFormat* verfügbaren Werte. Wie so oft in einer solchen Situation hilft der Objektkatalog des VBA-Editors weiter. Wenn Sie dort nach *FileFormat* suchen, gelangen Sie schnell zum Eintrag *XLFileFormat*, der alle möglichen Werte liefert (siehe Abbildung 4.5).

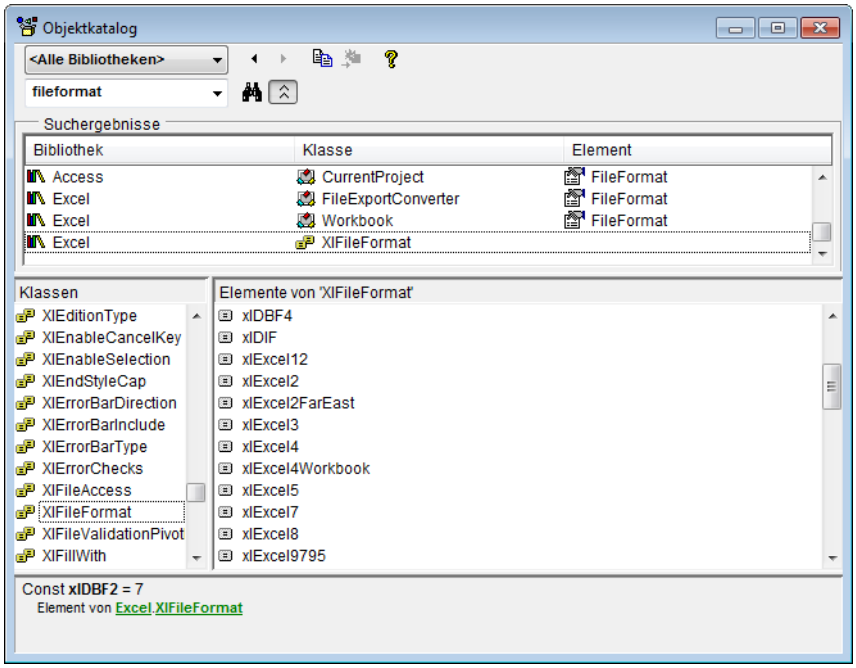


Abbildung 4.5: Ermitteln der Konstanten für die verschiedenen Excel-Dateiformate

Interessant sind hier die folgenden Einträge:

» *xlExcel8*: Excel-Workbook im Format von Excel 97-2003

» *xlExcel12*: Excel-Workbook (binär) im Format von Excel 2007-2013

» *xlOpenXMLWorkbook*: Excel-Workbook (XML) im Format von Excel 2007-2013

» *xlOpenXMLWorkbookMacroEnabled*: Excel-Workbook (XML) mit Makros im Format von Excel 2007-2013

Für *xlExcel8* verwenden Sie die Dateiendung *.xls*, für die aktuelle binäre Variante die *.xlsb*, für die XML-Variante ohne Makros *.xlsx* und für die mit Makros *.xlsm*.

4.4.6 Ein oder mehrere Workbooks?

Eine Excel-Instanz kann ein oder mehrere Workbooks enthalten. Diese lassen sich mit der *Workbooks*-Auflistung des *Application*-Objekts referenzieren.

In den folgenden Beispielen gehen wir jedoch zunächst davon aus, dass wir immer nur ein Workbook in einer frischen Excel-Instanz öffnen und lesend oder schreibend darauf zugreifen.

5 Daten in Excel-Tabellen schreiben

Wenn Sie Daten aus einer Access-Tabelle in eine Excel-Tabelle übertragen wollen, stehen Ihnen einige Techniken zur Verfügung, mit denen Sie das Ziel schnell erreichen – möglicherweise erfüllt dieses aber dann nicht Ihre optischen Ansprüche. Diese Techniken haben wir bereits beschrieben: Es handelt sich um die eingebauten Export-Funktionen sowie die VBA-Befehle *DoCmd.TransferSpreadsheet* et cetera.

Wenn Sie in einem Zuge Daten übertragen und auch eine genau Ihren Wünschen entsprechende Formatierung des Excel-Tabellenblatts erhalten möchten, bietet sich eine weitere Möglichkeit: Die Programmierung des kompletten Exports per VBA. Sie verwenden dann das Excel-Objektmodell, um auf die zu füllende Tabelle zuzugreifen und können dementsprechend alle dort verfügbaren Funktionen nutzen.

Wir dürfen nicht unerwähnt lassen, dass die Performance bei den zuerst genannten, eingebauten Funktionen zum Export von Access-Daten nach Excel in den meisten Fällen viel besser ist, als wenn Sie die Daten per VBA einzeln nach Excel rüberschaufeln.

Ein weiterer Grund, zu der OLE-Automation genannten Technik für den Zugriff auf Excel-Tabellen zu greifen, könnte sein, dass die Daten in Excel so strukturiert werden sollen, wie Sie es mit einem einfachen Export nicht realisieren können.

5.1 Einfache Tabellen oder Abfrage exportieren

Wenn Sie einfach nur die Inhalte des Feldes einer Tabelle oder Abfrage in eine neue Excel-Datei exportieren möchten, dies aber aus irgendeinem Grund nicht per *DoCmd.TransferSpreadsheet* möglich ist, gibt es eine Reihe von Varianten auf Basis von Recordsets.

5.1.1 Daten datensatzweise übertragen

Wenn Sie gleich Formatierungen et cetera zuweisen möchten, können sie beispielsweise eine Prozedur wie die folgende verwenden:

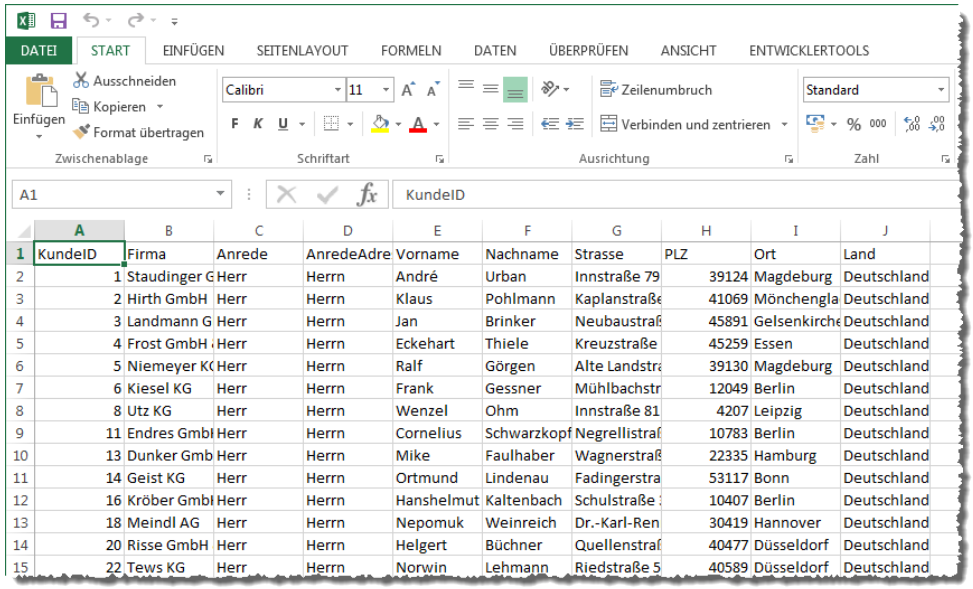
```
Public Sub TabelleExportieren()
    Dim db As DAO.Database
    Dim rst As DAO.Recordset
    Dim fld As DAO.Field
    Dim objExcel As Excel.Application
    Dim objWorkbook As Excel.Workbook
    Dim objWorksheet As Excel.Worksheet
    Dim i As Integer
```

```
    Set db = CurrentDb
    Set rst = db.OpenRecordset("SELECT * FROM qryKundenMitAnrede", dbOpenDynaset)
    Set objExcel = New Excel.Application
    Set objWorkbook = objExcel.Workbooks.Add
    Set objWorksheet = objWorkbook.Sheets(1)
    objWorkbook.Parent.Visible = True
    For i = 0 To rst.Fields.Count - 1
        Set fld = rst.Fields(i)
        objWorksheet.Cells(1, i + 1) = fld.Name
    Next i
    Do While Not rst.EOF
        For i = 0 To rst.Fields.Count - 1
            Set fld = rst.Fields(i)
            objWorksheet.Cells(rst.AbsolutePosition + 2, i + 1) = fld.Value
        Next i
        rst.MoveNext
    Loop
End Sub
```

Die Prozedur erstellt ein neues Excel-Workbook und öffnet ein Recordset auf Basis der Abfrage *qryKundenMitAnrede*. Nach dem Einblenden des Excel-Fensters durchläuft die Prozedur zunächst einmal alle Felder des Recordsets und trägt die Feldnamen als Überschriften in das Excel-Sheet ein. Dabei verwendet sie die *Cells*-Methode, um die jeweilige Ziel-Zelle des *Worksheet*-Objekts zu ermitteln. Der erste Wert ist immer 1, da wir ja ausschließlich Einträge in der ersten Zeile vornehmen möchten. Für die Spalte, also den zweiten Wert der *Cells*-Auflistung, verwenden wir den Wert der Laufvariablen *i* und addieren den Wert *1* hinzu, da der Index der *Fields*-Auflistung bei *0* beginnt.

Anschließend führt die Prozedur noch einmal fast die gleiche *For...Next*-Schleife aus – diesmal allerdings eingerahmt von einer *Do While*-Schleife über alle Datensätze des Recordsets. Darin durchläuft die Prozedur auch für alle Datensätze alle Felder und trägt die jeweiligen Werte in die entsprechenden Zellen ein. Hier müssen wir nun allerdings den Index für die Zeile jeweils um eins erhöhen. Um eine Zählervariable zu sparen, greifen wir dazu auf die Eigenschaft *AbsolutePosition* des *Recordset*-Objekts zu. Dieses ist ebenfalls *0*-basiert. Da außerdem die erste Zeile bereits mit den Spaltenüberschriften gefüllt ist, addieren wir hier noch den Wert *2* hinzu.

Das Ergebnis sieht ohne Formatierungen und ohne Anpassung der Spaltenbreiten noch etwas wirr aus (siehe Abbildung 5.1).



	A	B	C	D	E	F	G	H	I	J
1	KundeID	Firma	Anrede	AnredeAdre	Vorname	Nachname	Strasse	PLZ	Ort	Land
2	1	Staudinger G	Herr	Herrn	André	Urban	Innstraße 79	39124	Magdeburg	Deutschland
3	2	Hirth GmbH	Herr	Herrn	Klaus	Pohlmann	Kaplanstraße	41069	Mönchengla	Deutschland
4	3	Landmann G	Herr	Herrn	Jan	Brinker	Neubaustraß	45891	Gelsenkirche	Deutschland
5	4	Frost GmbH	Herr	Herrn	Eckehart	Thiele	Kreuzstraße	45259	Essen	Deutschland
6	5	Niemeyer KG	Herr	Herrn	Ralf	Görgen	Alte Landstra	39130	Magdeburg	Deutschland
7	6	Kiesel KG	Herr	Herrn	Frank	Gessner	Mühlbachstr	12049	Berlin	Deutschland
8	8	Utz KG	Herr	Herrn	Wenzel	Ohm	Innstraße 81	4207	Leipzig	Deutschland
9	11	Endres GmbH	Herr	Herrn	Cornelius	Schwarzkopf	Negrellistraß	10783	Berlin	Deutschland
10	13	Dunker Gmb	Herr	Herrn	Mike	Faulhaber	Wagnerstraß	22335	Hamburg	Deutschland
11	14	Geist KG	Herr	Herrn	Ortmund	Lindenau	Fadingerstra	53117	Bonn	Deutschland
12	16	Kröber GmbH	Herr	Herrn	Hanshelmut	Kaltenbach	Schulstraße	10407	Berlin	Deutschland
13	18	Meindl AG	Herr	Herrn	Nepomuk	Weinreich	Dr.-Karl-Ren	30419	Hannover	Deutschland
14	20	Risse GmbH	Herr	Herrn	Helgert	Büchner	Quellenstraß	40477	Düsseldorf	Deutschland
15	22	Tews KG	Herr	Herrn	Norwin	Lehmann	Riedstraße 5	40589	Düsseldorf	Deutschland

Abbildung 5.1: Frisch nach Excel exportierte Daten

5.1.2 Zeilenweises Übertragen mit CopyFromRecordset

Eine schnellere Variante verwendet die Methode *CopyFromRecordset* des *Range*-Objekts von Excel. Dafür erzeugen wir wie gehabt ein *Workbook*-Objekt und zeigen dieses an. Außerdem öffnen wir das gleiche Recordset wie zuvor. Da wir für die Methode *CopyFromRecordset* die zu übertragende Anzahl der Datensätze ermitteln müssen, bewegen wir den Datensatzzeiger mit *MoveLast* zuerst zum letzten und dann mit *MoveFirst* wieder zum ersten Datensatz der Datensatzgruppe – nur so funktioniert die Eigenschaft *RecordCount* später zuverlässig.

Damit können wir die Anzahl der Zeilen mit der *RecordCount*-Eigenschaft ermitteln und in die Variable *lngZeilen* eintragen. Die Anzahl der Spalten ermitteln wir aus der Anzahl der Felder des Recordsets und speichern diese in *lngSpalten*. Nun definieren wir das *Range*-Objekt mit der gewünschten Größe. Dieses legen wir mit der *Range*-Funktion des *Worksheet*-Objekts fest, das zwei Zellen zur Definition benötigt – die linke, obere und die rechte, untere Zelle. Die linke, obere erhalten wir mit *objWorksheet.Cells(1, 1)*, die rechte, untere mit *objWorksheet.Cells(lngZeilen, lngSpalten)*. Das so ermittelte *Range*-Objekt bietet nun die *CopyFromRecordset*-Methode an, der Sie das Recordset sowie nochmals die Anzahl der Zeilen und Spalten übergeben:

```
Public Sub RecordsetExportieren()
    Dim db As DAO.Database
    Dim rst As DAO.Recordset
    Dim objExcel As Excel.Application
```

```
    Dim objWorkbook As Excel.Workbook
    Dim objWorksheet As Excel.Worksheet
    Dim objRange As Excel.Range
    Dim lngZeilen As Long
    Dim lngSpalten As Long
    Set db = CurrentDb
    Set rst = db.OpenRecordset("SELECT * FROM qryKundenMitAnrede", dbOpenDynaset)
    rst.MoveLast
    rst.MoveFirst
    Set objExcel = New Excel.Application
    Set objWorkbook = objExcel.Workbooks.Add
    Set objWorksheet = objWorkbook.Sheets(1)
    objWorkbook.Parent.Visible = True
    lngZeilen = rst.RecordCount
    lngSpalten = rst.Fields.Count
    Set objRange = objWorksheet.Range(objWorksheet.Cells(1, 1), _
        objWorksheet.Cells(lngZeilen, lngSpalten))
    objRange.CopyFromRecordset rst, lngZeilen, lngSpalten
End Sub
```

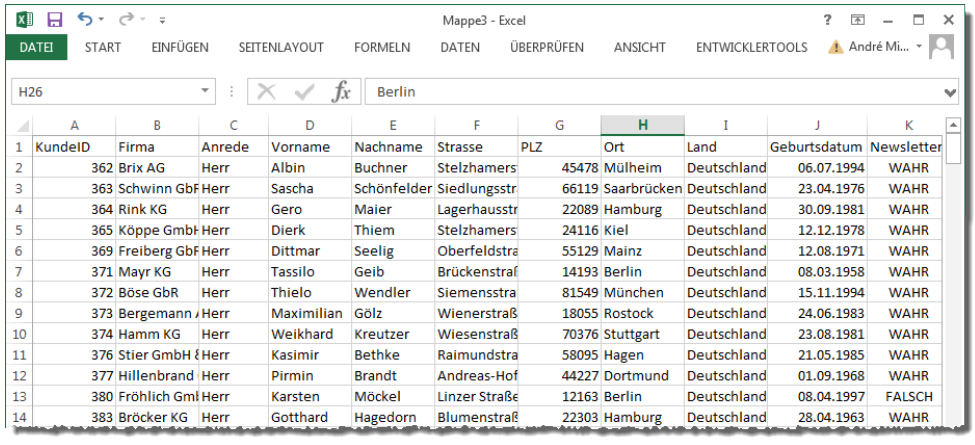
6 Daten aus Excel-Tabellen lesen

Das Lesen von Daten aus Excel-Tabellen steht meist an, wenn ein Kunde die Migration seiner Daten aus einer oder mehreren Excel-Sheets in ein Datenmodell einer Access-Datenbank wünscht. Dabei können mehr oder weniger komplizierte Fälle auftreten – das größte Problem ist wohl die von Dateninkonsistenzen innerhalb einer Spalte.

Das bedeutet, dass beispielsweise die Daten der Spalte *Anrede* nicht nur einheitliche Werte wie *Herr* oder *Frau* enthalten, sondern auch noch *Herrn*, *Sehr geehrter Herr*, *Lieber* et cetera. Excel lädt auch dazu ein, nicht nur die eigentlichen Werte, sondern Zusatzinformationen einzugeben. Wenn ein Feld das Geburtsdatum aufnehmen soll, trägt der gemeine Nutzer gern auch mal Informationen wie *1.2.1971 (wird dieses Jahr 50)* ein. Solche Fälle sind optimalerweise im Vorfeld zu klären, da solche Werte natürlich nicht in ein Feld des Datentyps *Datum* eingetragen werden können. Wir schauen uns in diesem Kapitel einige einfache Fälle für den Import von Daten aus Excel-Tabellen in Access-Tabelle an, damit Sie die grundlegenden Techniken kennen lernen.

6.1 Kontakte mit Anrede einlesen

Wer unter Excel eine Kontakt- oder Adress-Tabelle pflegt, wird dort vermutlich kein Auswahlfeld für die verschiedenen Werte der Spalte *Anrede* unterbringen, sondern die Werte einfach von Hand eintragen. Mit etwas Glück landen dort dann auch nur gültige Werte wie *Herr*, *Frau* oder *Firma*. Normalerweise finden sich dort aber, wie oben bereits angesprochen, noch einige weitere Werte (siehe Abbildung 6.1).



	A	B	C	D	E	F	G	H	I	J	K
1	KundeID	Firma	Anrede	Vorname	Nachname	Strasse	PLZ	Ort	Land	Geburtsdatum	Newsletter
2	362	Brix AG	Herr	Albin	Buchner	Stelzhamers	45478	Mülheim	Deutschland	06.07.1994	WAHR
3	363	Schwinn GbF	Herr	Sascha	Schönfelder	Siedlungsstr	66119	Saarbrücken	Deutschland	23.04.1976	WAHR
4	364	Rink KG	Herr	Gero	Maier	Lagerhausstr	22089	Hamburg	Deutschland	30.09.1981	WAHR
5	365	Köppe GmbH	Herr	Dierk	Thiem	Stelzhamers	24116	Kiel	Deutschland	12.12.1978	WAHR
6	369	Freiberg GbF	Herr	Dittmar	Seelig	Oberfeldstra	55129	Mainz	Deutschland	12.08.1971	WAHR
7	371	Mayr KG	Herr	Tassilo	Geib	Brückenstraß	14193	Berlin	Deutschland	08.03.1958	WAHR
8	372	Böse GbR	Herr	Thielo	Wendler	Siemensstra	81549	München	Deutschland	15.11.1994	WAHR
9	373	Bergemann /	Herr	Maximilian	Gölz	Wienerstraß	18055	Rostock	Deutschland	24.06.1983	WAHR
10	374	Hamm KG	Herr	Weikhard	Kreutzer	Wiesenstraß	70376	Stuttgart	Deutschland	23.08.1981	WAHR
11	376	Stier GmbH &	Herr	Kasimir	Bethke	Raimundstra	58095	Hagen	Deutschland	21.05.1985	WAHR
12	377	Hillenbrand	Herr	Pirmin	Brandt	Andreas-Hof	44227	Dortmund	Deutschland	01.09.1968	WAHR
13	380	Fröhlich Gml	Herr	Karsten	Möckel	Linzer Straße	12163	Berlin	Deutschland	08.04.1997	FALSCH
14	383	Bröcker KG	Herr	Gotthard	Hagedorn	Blumenstraß	22303	Hamburg	Deutschland	28.04.1963	WAHR

Abbildung 6.1: Zu importierende Excel-Tabelle

Wir gehen davon aus, dass die Werte einigermaßen okay sind und sich nur der eine oder andere Ausreißer im Feld *Anrede* findet.

6.1.1 Import programmieren

Wir wollen nun eine Routine programmieren, welche uns den Import der oben dargestellten Excel-Datei abnimmt. Dabei ist zu beachten, dass die Anreden nicht in ein Feld *Anrede* der Zieltabelle gelangen sollen, sondern dass die Werte dieses Feldes in einer gesonderten Tabelle namens *tblAnreden* gespeichert werden sollen.

Dafür enthält die Zieltabelle dann ein Feld namens *AnredeID*, das mit dem Primärschlüsselfeld der Tabelle *tblAnreden* verknüpft ist. Außerdem müssen die Werte des Feldes *Newsletter* von *WAHR* und *FALSCH* in *-1* und *0* umgewandelt werden.

Je nachdem, ob das Feld *Anrede* der Excel-Tabelle außer den Werten *Herr*, *Frau* oder *Firma* noch andere Werte enthält, müssen wir hier nachher noch nachbessern.

Da wir Access-Programmierer sind, möchten wir möglichst wenig mit der externen Anwendung zu tun haben, aus der wir die Daten beziehen, sondern soweit es geht mit reinen Access-Objekten arbeiten.

Dementsprechend gehen wir in diesem Fall nicht wie im Kapitel »Excel programmieren« ab Seite 95 vor und lesen die Daten direkt aus der geöffneten Excel-Tabelle ein, sondern wir machen uns die Daten in Form eines Access-Objekts verfügbar. Das bedeutet, dass wir die Daten entweder als verknüpfte Tabelle ins Boot holen oder gleich die komplette Tabelle importieren.

Performanter dürfte es sein, wenn wir die Daten importieren, dann stehen diese als echte Access-Tabelle zur Verfügung.

6.1.2 Import per DAO

Am einfachsten dürfte es für die meisten Access-Entwickler sein, die Routine für den Import der Daten komplett mit DAO zu erstellen und dabei die Recordsets datensatzweise zu durchlaufen und Feld für Feld zuzuweisen.

Dies ist vermutlich die langsamste Variante, aber da Sie einen solchen Import vermutlich nicht regelmäßig durchführen, sondern nur einmal bei der Migration der bestehenden Daten, ist dies kein Problem.

Wir sollen nun die Daten des ersten Tabellenblatts der Excel-Datei *KundenFuerImport.xls* in die beiden Tabellen *tblKundenVonExcel* und *tblAnredenVonExcel* importieren (wir nehmen Kopien der eigentlichen Tabellen *tblKunden* und *tblAnreden*, um die Originaldaten nicht zu zerschießen).

Diese Tabellen sind ganz einfach aufgebaut (siehe Abbildung 6.2).

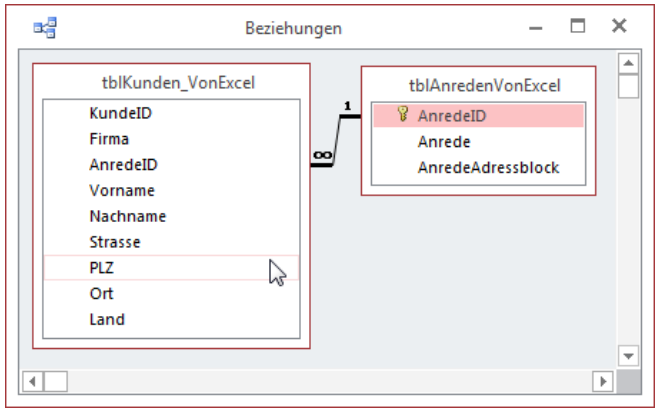


Abbildung 6.2: Zieltabellen des Imports

Die folgende Prozedur deklariert zunächst die benötigten Variablen:

```
Public Sub ImportVonExcel()
    Dim db As DAO.Database
    Dim rstQuelle As DAO.Recordset
    Dim rstKunden As DAO.Recordset
    Dim rstAnreden As DAO.Recordset
    Dim lngAnredeID As Long
```

Danach füllt sie die Variable *db* mit einem Verweis auf das *Database*-Objekt der aktuellen Datenbank:

```
    Set db = CurrentDb
```

Dieses verwendet sie gleich, um die Inhalt der beiden Zieltabellen *tblKundenVonExcel* und *tblAnredenVonExcel* zu leeren.

Dies werden Sie bei der Programmierung eines solchen Imports vermutlich auch tun, wenn Sie nach dem Aufdecken eines Fehlers noch einmal neu beginnen wollen.

Auf diese Weise sparen Sie sich das manuelle Löschen der Daten der Zieltabellen:

```
    db.Execute "DELETE FROM tblKundenVonExcel", dbFailOnError
    db.Execute "DELETE FROM tblAnredenVonExcel", dbFailOnError
```

Schließlich löschen wir auch die zu importierende Excel-Tabelle, um diese anschließend neu zu importieren:

```
    On Error Resume Next
    DoCmd.DeleteObject acTable, "tblImportVonExcel"
    On Error GoTo 0
```

Den Import führen wir mit der *TransferSpreadsheet*-Methode des *DoCmd*-Objekts durch. Dabei geben wir die Import-Art *acImport*, das entsprechende Format der Quelldatei, den Namen der Zieltabelle sowie den Pfad der Quelldatei an.

Außerdem stellen wir ein, dass die Quelltabelle in der ersten Spalten Feldnamen enthält. Danach aktualisieren wir den Navigationsbereich von Access, damit die neue Tabelle dort auch angezeigt wird:

```
DoCmd.TransferSpreadsheet acImport, acSpreadsheetTypeExcel12Xml, _
    "tblImportVonExcel", CurrentProject.Path & "\KundenFuerImport.xlsx", True
Application.RefreshDatabaseWindow
```

Nun erstellen wir drei Recordsets auf Basis der Quell- und Zieltabellen:

```
Set rstQuelle = db.OpenRecordset("tblImportVonExcel")
Set rstKunden = db.OpenRecordset("tblKundenVonExcel")
Set rstAnreden = db.OpenRecordset("tblAnredenVonExcel")
```

Und damit steigen wir auch schon in den Import ein. Diesen führen wir innerhalb einer *Do While*-Schleife über alle zu importierenden Datensätze aus.

Dabei legen wir zunächst einen neuen Datensatz in der Tabelle *tblKundenVonExcel* an und weisen dem Primärschlüsselfeld den Wert der Spalte *KundeID* zu. Auch die *Firma* übernehmen wir einfach vom gleichnamigen Feld der Quelltabelle:

```
Do While Not rstQuelle.EOF
    rstKunden.AddNew
    rstKunden!KundeID = rstQuelle!KundeID
    rstKunden!Firma = rstQuelle!Firma
```

Danach wird es bereits interessant – wir müssen uns um die Anrede kümmern. Diese liegt ja in der Quelltabelle nur in Textform vor, wir wollen der Zieltabelle jedoch einen Zahlenwert zuweisen, welcher der entsprechenden Anrede der Tabelle *tblAnreden* entspricht.

Wir müssen also zunächst prüfen, ob der Wert der Spalte *Anrede*, zum Beispiel *Herr*, bereits in der Tabelle *tblAnredenVonExcel* gespeichert ist.

Dazu verwenden wir den Aufruf der folgenden *DLookup*-Funktion. Diese ist noch in die *Nz*-Funktion eingefasst, die den Wert *0* zurückliefert, sollte noch kein entsprechender Eintrag in der Tabelle *tblAnredenVonExcel* vorliegen:

```
lngAnredeID = Nz(DLookup("AnredeID", "tblAnredenVonExcel", _
    "Anrede = '" & rstQuelle!Anrede & "'"), 0)
```

Hat *lngAnredeID* nun den Wert *0*, legt die Prozedur einen neuen Datensatz in der Tabelle *tblAnredenVonExcel* an und weist dem Feld *Anrede* den Wert der Spalte *Anrede* zu. Bevor die Prozedur den mit *AddNew* angelegten Datensatz mit der *Update*-Methode speichert, liest sie

7 Word programmieren

Word und Excel haben so viele Gemeinsamkeiten – vom Nutzer aus betrachtet, aber auch vom Entwickler aus. Beide lassen sich in einer oder mehreren Instanzen starten, beide öffnen Dokumente und stellen diese zur Bearbeitung bereit. Die Dokumente lassen sich speichern und drucken. Aber wenn es um die Bearbeitung per VBA kommt, hören die Gemeinsamkeiten schnell auf: Das *Application*-Objekt als oberstes Element der Hierarchie haben beide noch gemein, und während Word das *Document*-Objekt als Dokument verwendet, ist es unter Excel das *Workbook*-Objekt. Danach geht es bei Word direkt mit dem Inhalt des Dokuments los, während ein Excel-*Workbook* noch ein oder mehrere *Worksheet*-Objekte, also Tabellen, enthalten kann. Erst danach folgen mit dem Zellen (*Cells*) die eigentlichen Objekte, die Daten aufnehmen.

Das Schöne bei der Programmierung von Excel ist für den Access-Anwender die Struktur der Inhalte. Ein Workbook enthält ein oder mehrere Worksheets, ein Worksheet enthält x mal y Zellen mit Daten. Die Zellen können direkt referenziert werden.

Unter Word sieht das ganz ganz anders aus: Man könnte naiv an die Programmierung herangehen und annehmen, dass man erst eine Seite referenziert, dann eine Zeile und schließlich ein Wort. Aber weit gefehlt: Word arbeitet ja mit Fließtext, der sich je nach Ausgestaltung von Texten und Seiten mal auf drei, mal auf acht Seiten verteilen könnte. Das heißt also, dass eine Seite als Startpunkt für das Referenzieren etwa eines bestimmten Wortes ein schlechter Ausgangspunkt ist.

#ToDo: noch ergänzen

7.1 Verweis auf die Word-Objektbibliothek

Wenn Sie mit Early Binding arbeiten möchten, was zumindest in solchen Fällen verpflichtend ist, wenn Sie auf Ereignisse der Word-Objekte wie etwas das Öffnen oder Schließen eines Dokuments reagieren möchten, benötigen Sie zunächst eine Verweis auf die Bibliothek *Microsoft Word x.0 Object Library* (siehe Abbildung 7.1). Die Vor- und Nachteile von Early Binding und Late Binding haben wir bereits in Kapitel *****VBAZugriffAufOffice* erläutert.

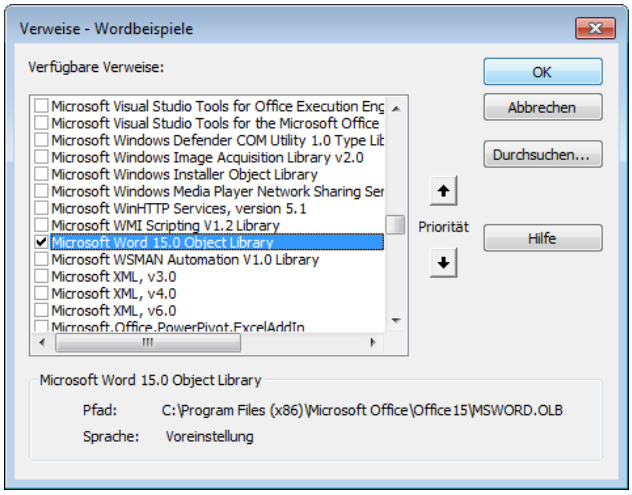


Abbildung 7.1: Verweis auf die Bibliothek *Microsoft Word x.0 Object Library*

7.2 Eine Word-Instanz erstellen

Wenn Sie eine Word-Instanz mit Early Binding erstellen möchten, wovon wir im Folgenden ausgehen, verwenden Sie die *New*-Anweisung unter Angabe des gewünschten Objekttyps, hier *Word.Application*. Das Ergebnis dieser Operation speichern Sie in einer Objektvariablen des gleichen Typs:

```
Dim objWord As Word.Application
Set objWord = New Word.Application
```

Alternativ verwenden Sie für Late Binding (kein Verweis nötig) die folgenden Codezeilen:

```
Dim objWord As Object
Set objWord = CreateObject("Word.Application")
```

Wenn Sie dies in einer Prozedur ausprobieren, geschieht natürlich nicht viel – die Word-Instanz wird noch nicht mal angezeigt. Noch dazu bleibt diese auch noch unsichtbar im Speicher hängen – die Variable *objWord* hat zwar nach Beenden der Prozedur keine Gültigkeit mehr, aber wenn Sie die Word nicht explizit beenden, bleibt es dennoch im Speicher erhalten.

Um uns dies anzusehen, ohne die Word-Instanz sichtbar zu machen, verwenden wir eine kleine Testprozedur. Als Vorbereitung legen Sie die folgende API-Funktion im Kopf des Moduls an (in der Beispieldatenbank im Modul *mdlBeispiel* zu finden):

```
Declare Sub Sleep Lib "kernel32" (ByVal dwMilliseconds As Long)
```

Diese Funktion hält den aufrufenden Code für die angegebene Anzahl Millisekunden an. Die Beispielprozedur sieht so aus:

```
Public Sub WordInstanzieren()
    Dim objWord As Word.Application
    Set objWord = New Word.Application
    'Wichtige Aufgaben mit Word erledigen
    Sleep 3000
    objWord.Quit
    Set objWord = Nothing
End Sub
```

Sie startet eine Word-Instanz, wartet drei Sekunden (3.000 Millisekunden) und schließt diese dann wieder. Genug Zeit, um sich im Task-Manager von Windows (rechte Maustaste auf die Taskleiste, Eintrag *Task-Manager*) davon zu überzeugen, dass Word tatsächlich geöffnet und auch wieder geschlossen wird. Den passenden Eintrag finden Sie übrigens nicht unter dem Registerreiter *Anwendungen*, sondern unter *Prozesse* (siehe Abbildung 7.2). Sollten dort von vorherigen Experimenten noch weitere Einträge mit der Bezeichnung *WINWORD.EXE *32* vorliegen, können Sie diese ebenfalls beenden. Aber Vorsicht: Wenn Sie noch eine Word-Instanz geöffnet haben, in der Sie gerade ein nicht gespeichertes Dokument bearbeiten, sollten Sie dieses erst sichern, bevor Sie alle Word-Instanzen mit Gewalt beenden.

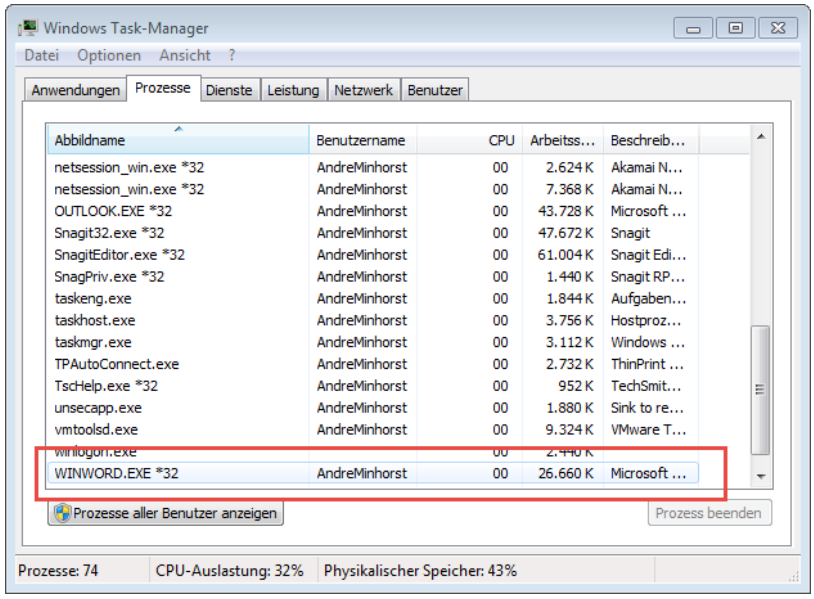


Abbildung 7.2: Word-Instanz im Task-Manager

Ein paar Hinweise zu dieser Prozedur:

» Wenn Sie die *Quit*-Methode des Objekts *objWord* weglassen, verbleibt die Word-Instanz im Speicher.

» Wenn Sie eine Word-Instanz mit der obigen Prozedur erstellen, die *Quit*-Methode weglassen (was dazu führt, dass die Instanz im Speicher bleibt) und Word dann manuell öffnen, finden Sie nachher immer noch nur eine Instanz im Task-Manager vor. Schließen Sie die sichtbar Word-Instanz dann, verschwindet die aktuell vorliegende Instanz auch aus dem Task-Manager. Die neue, manuell gestartete Instanz scheint also die nicht sichtbare Instanz zu übernehmen. Die Tatsache, dass sich der Speicherbedarf der Instanz nach dem Öffnen der sichtbaren Instanz etwa verdoppelt, spricht für die Annahme, dass die manuell geöffnete Instanz die nicht sichtbare übernimmt.

» Öffnen wir dann erneut mit der obigen Prozedur (ohne *Quit*) eine weitere Instanz, finden wir allerdings, wie zu erwarten, zwei Instanzen im Task-Manager vor.

» Öffnen wir dann auch noch eine zweite sichtbare Instanz (wiederum manuell), bleiben natürlich beide Instanzen im Task-Manager erhalten. Allerdings wächst der Speicherbedarf der ersten Instanz weiter, während die der neu hinzugefügten konstant bleibt.

» Dass nach dem Schließen der sichtbaren Instanz nun ein Exemplar des Eintrags *WINWORD.EXE 32** im Task-Manager verbleibt, spricht dafür, dass sichtbare Word-Fenster immer einer geöffneten Instanz zugeordnet werden.

» Diese Annahme können wir unterlegen, indem wir nun zunächst alle *WINWORD.EXE 32**-Einträge beenden. Dann öffnen wir manuell eine erste Word-Instanz und dann noch ein zweites Word-Fenster. Es bleibt bei einem Eintrag im Task-Manager, obwohl wir zwei Fenster geöffnet haben.

7.2.1 Mehrere Instanzen

Hier erkennen Sie bereits: Auch von Word können Sie, genau wie von Excel, mehrere Instanzen gleichzeitig betreiben. Deshalb stellt sich auch hier vor der Verwendung von Word per VBA, ob Sie eine bestehende Instanz verwenden oder eine neue Instanz erstellen wollen.

Es kommt dabei wiederum auf den Anwendungsfall an. Wenn Sie etwa von Access aus ein Dokument unter Word erstellen möchten, dass lediglich mit dem gewünschte Text gefüllt und dann gespeichert werden soll, empfiehlt sich auf jeden Fall der Einsatz einer eigenen Instanz. So kommen Sie nicht mit eventuell geöffneten anderen Instanzen ins Gehege.

Es gibt jedoch auch Fälle, in denen Sie vielleicht auf das aktuell geöffnete Word-Dokument zugreifen wollen – beispielsweise, um Text aus diesem Dokument auszulesen oder welchen hinzuzufügen. Dann sollten Sie die bestehende Word-Instanz nutzen, genau gesagt sogar das derzeit geöffnete Dokument.

7.2.2 Bestehende Instanz nutzen

Wenn Sie eine bestehende Instanz nutzen wollen, kann es sich um eine sichtbare (also eine manuell geöffnete Instanz oder eine, die per VBA erstellt und sichtbar gemacht wurde) oder eine nicht sichtbare Instanz handeln. Wenn eine sichtbare Instanz vorliegt, dann wird die folgende Prozedur auf diese zugreifen. Sie ermittelt mit der *GetObject*-Methode eine bestehende Instanz. Sollte keine vorhanden sei, löst dies den Fehler mit der Nummer *429* aus (*Objekterstellung durch ActiveX-Komponente nicht möglich*). In diesem Fall gibt die Prozedur eine entsprechende Meldung aus. Anderenfalls liefert sie die Anzahl der enthaltenen Dokumente mit der *Count*-Methode der *Documents*-Auflistung und macht die Instanz sichtbar, sofern dies noch nicht der Fall ist.

```
Public Sub WordBestehendeInstanz()
    Dim objWord As Object
    On Error Resume Next
    Set objWord = GetObject(, "Word.Application")
    If Err.Number = 429 Then
        MsgBox "Keine Word-Instanz vorhanden."
    Else
        MsgBox "Anzahl Dokumente: " & objWord.Documents.Count
        objWord.Visible = True
    End If
End Sub
```

Wir haben diese Prozedur mit verschiedenen Mengen geöffneter Instanzen durchgeführt. Es scheint jedoch so zu sein, dass *GetObject* immer auf die sichtbare Instanz zugreift.

Wie oben erwähnt, macht der Zugriff auf eine bestehende Word-Instanz eigentlich nur Sinn, wenn Sie gezielt auf ein darin enthaltenes Dokument zugreifen möchten. Wie das gelingt, erfahren Sie weiter unten. In allen anderen Fällen sollten Sie eine eigene Instanz erzeugen und referenzieren, um nicht mit vorhandenen Instanzen und den darin angezeigten und möglicherweise sogar in Bearbeitung befindlichen Dokumenten zusammenzustoßen.

7.2.3 Word-Instanz sichtbar machen

Wie Sie die Word-Instanz sichtbar machen, haben wir oben schon angerissen. Sie stellen dazu die Eigenschaft Visible auf den Wert True ein. Damit erscheint die Word-Instanz zwar, aber in der Regel hinter dem Fenster, von dem aus seine Erstellung angestoßen wurde – also etwa hinter dem Access-Fenster oder dem VBA-Editor.

Hier benötigen wir eine weitere Funktion namens *AppActivate*. Diese erwartet als Parameter den Titel des anzuzeigenden Fensters. Im Falle einer frisch instanzierten und sichtbar gemach-

ten Word-Instanz handelt es sich um den Titel *Word*. Die folgende Prozedur erzeugt eine Word-Instanz und bringt diese direkt in den Vordergrund:

```
Public Sub WordSichtbarMachen()
    Dim objWord As Word.Application
    Set objWord = New Word.Application
    objWord.Visible = True
    AppActivate "Word"
End Sub
```

Achtung: *AppActivate* setzt voraus, dass es ein Fenster mit dem angegebenen Titel gibt. Wenn Sie eine neue Word-Instanz erzeugt, diese aber noch nicht mit *Visible = True* eingeblendet haben, löst der Aufruf von *AppActivate* einen Fehler aus, weil das Fenster nicht gefunden werden konnte.

7.3 Mit Dokumenten arbeiten

Nun wollen wir ja nicht immer nur leere Word-Instanzen erstellen, sondern auch einmal ein neues Dokument darin anzeigen oder auf ein bestehendes Dokument zugreifen. Wie dies gelingt, zeigen die folgenden Abschnitte.

7.3.1 Word instanzieren und Dokument erstellen

Wenn Sie einen Verweis per Objekt-Variable auf die Word-Instanz benötigen und darin ein Dokument erstellen wollen, gehen Sie wie folgt vor:

```
Public Sub WordStartenDokumentOeffnen()
    Dim objWord As Word.Application
    Dim objDocument As Word.Document
    Set objWord = New Word.Application
    objWord.Visible = True
    AppActivate "Word"
    Set objDocument = objWord.Documents.Add
End Sub
```

Die Prozedur erzeugt eine Word-Instanz und fügt dann der *Documents*-Auflistung mit der *Add*-Methode ein neues Dokument hinzu. Das Ergebnis sieht dann wie in Abbildung 7.3 aus.

Interessant ist an dieser Stelle die Reihenfolge, in der das Word-Fenster mit *AppActivate* in den Vordergrund geholt und das Dokument geöffnet wird. Bei dieser Variante reicht es, *AppActivate* den Namen des leeren Word-Fensters zu übergeben, also *Word*.

Sie können natürlich auch zuerst das Word-Dokument anlegen und dann das *AppActivate* aufrufen. In diesen Fall müssen Sie allerdings als Parameter den Namen des Word-Dokuments übergeben. Den ermitteln wir allerdings dynamisch aus der Eigenschaft *Name* des Objekts *objDocument*:

```
Public Sub WordStartenDokumentOeffnen_II()
    ...
    Set objDocument = objWord.Documents.Add
    AppActivate objDocument.Name
End Sub
```

Nun ergibt sich hier ein kleines Problem: Auf dem ersten Testsystem funktioniert dies, weil Word den Namen der Datei inklusive Dateiendung anzeigte (*Beispiel.docx*). Auf einem anderen Testsystem gelang die Aktivierung des Fensters nicht, da hier nur der Name ohne Dateiendung in der Titelleiste erschien. Wir müssen also eine eigene Funktion zum Aktivieren des Fensters programmieren, welche die verschiedenen Varianten durchprobiert.

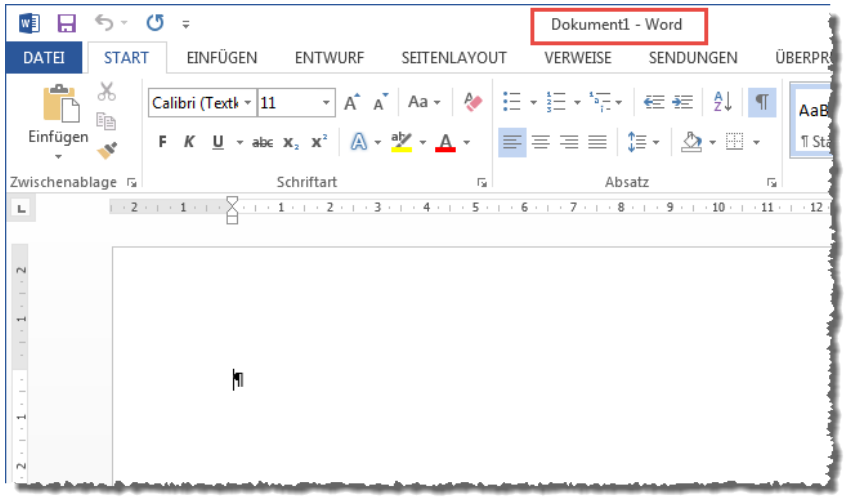


Abbildung 7.3: Ein frisch erzeugtes Word-Dokument

ActivateApp statt AppActivate

Also bauen wir eine kleine Prozedur, die nicht *AppActivate* heißt, sondern *ActivateApp*. Sie nimmt auch nicht den Titel des Fensters entgegen, sondern einen Verweis auf das *Document*-Objekt, das aktuell angezeigt wird.

```
Public Sub ActivateApp(objDocument As Word.Document)
    Dim strCaption As String
    strCaption = objDocument.Parent.ActiveWindow.Caption
```

```
    AppActivate strCaption
End Sub
```

Die Prozedur ermittelt aus der *Caption*-Eigenschaft des aktiven Fensters der Word-Instanz den Titel und übergibt diesen dann an die Methode *AppActivate*. Somit ist gesichert, dass immer der angezeigte Titel des aktuellen Fensters verwendet wird, wenn das Fenster in den Vordergrund geholt werden soll. Diese ist in der Beispieldatenbank überall dort eingebaut, wo das Fenster mit einem bestimmten Dokument den Fokus erhalten soll.

7.3.2 Dokument mit Vorlage erstellen

Wenn Sie ein neues Dokument auf Basis einer Vorlage erstellen möchten, müssen Sie nur den Aufruf der *Add*-Methode der *Documents*-Auflistung anpassen – und zwar, indem Sie für den einzigen Parameter dieser Methode den Pfad zu der zu verwendenden Vorlage übergeben:

```
Set objDocument = objWord.Documents.Add "<Name der Vorlage>"
```

Auf die Verwendung von Vorlagen kommen wir später zurück.

7.3.3 Word direkt mit neuem Dokument öffnen

Sollten Sie die Word-Instanz nicht unbedingt mit einer Objektvariablen referenzieren wollen, können Sie auf diese auch verzichten. Sie können nämlich auch eine Word-Instanz erstellen, indem Sie direkt ein neues Objekt des Typs *Word.Document* instanzieren. Das sieht dann wie folgt aus:

```
Public Sub DokumentDirektErstellen()
    Dim objDocument As Word.Document
    Set objDocument = New Word.Document
    objDocument.Parent.Visible = True
    AppActivate objDocument.Name
End Sub
```

Auch hier aktivieren wir das Word-Fenster direkt wieder mit *ActivateApp*.

7.3.4 Word mit vorhandenem Dokument öffnen

Wenn Sie ein auf der Festplatte vorhandenes Dokument mit Word öffnen möchten, gibt es ebenfalls verschiedene Wege. Der erste verwendet die Version mit der vorherigen Instanzierung von Word:

```
Public Sub DokumentOeffnen()
    Dim objWord As Word.Application
    Dim objDocument As Word.Document
```

```
    Dim strDocument As String
    Set objWord = New Word.Application
    objWord.Visible = True
    strDocument = CurrentProject.Path & "\Beispiel.docx"
    Set objDocument = objWord.Documents.Open(strDocument)
    AppActivate objDocument.Name
End Sub
```

Die Prozedur erstellt mit *New* eine neue Word-Instanz und blendet diese mit der *Visible*-Eigenschaft ein. Dann speichert sie den Pfad des zu öffnenden Dokuments in der Variablen *strDocument* und verwendet diese als Parameter der *Open*-Methode der *Documents*-Auflistung von Word.

7.3.5 Geöffnetes Dokument referenzieren

Für den oben bereits erwähnten Fall, dass Sie etwas mit einem aktuell bereits geöffneten Word-Dokument anstellen möchten, benötigen Sie eine Objektvariable, die auf dieses Word-Dokument verweist. Diese füllen wir beispielsweise mit der folgenden Prozedur:

```
Public Sub DokumentReferenzieren()
    Dim objDocument As Word.Document
    Dim strDocument As String
    strDocument = CurrentProject.Path & "\Beispiel.docx"
    Set objDocument = GetObject(strDocument)
    AppActivate objDocument.Name
End Sub
```

Die Prozedur verwendet die *GetObject*-Methode, um das für den ersten Parameter per Pfad angegebene Word-Dokument zu referenzieren.

Auch für geschlossene Dokumente

Mit der gleichen Prozedur können Sie übrigens auch noch nicht geöffnete Dokumente in Word anzeigen.

7.3.6 Word-Dokument impizit speichern

Wenn Sie ein Word-Dokument per VBA-Code erzeugen, wollen Sie dieses in der Regel auch anschließend speichern. Oft speichern Sie Word-Dokumente, ohne dies explizit anzustoßen. Dies geschieht etwa, wenn Sie Word schließen wollen, ohne zuvor die Änderungen gespeichert zu haben. Oder Sie erstellen ein neues Word-Dokument per VBA, bearbeiten es und schließen Word, bevor Sie das Dokument gespeichert haben. Dies geschieht in der folgenden Prozedur:

```
Public Sub DokumentErstellenUndAendern()
```

```
    Dim objWord As Word.Application
    Dim objDocument As Word.Document
    Set objWord = New Word.Application
    Set objDocument = objWord.Documents.Add
    objWord.Visible = True
    AppActivate objDocument.Name
    objDocument.Range = "Beispieltext"
    objWord.Quit
End Sub
```

Resultat dieses Vorgangs ist das Erscheinen eines Dialogs, der fragt, ob Sie das Dokument speichern, die Änderungen verwerfen oder gar das Schließen abbrechen wollen (siehe Abbildung 7.4).

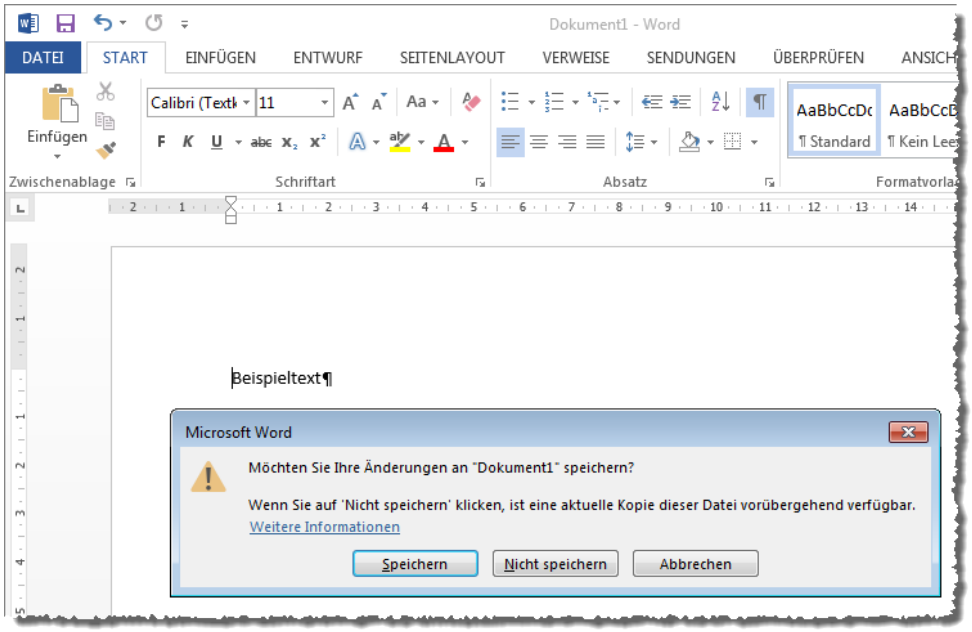


Abbildung 7.4: Rückfrage, ob das Dokument gespeichert werden soll oder nicht

Wenn Sie hingegen das Dokument selbst schließen, bevor Sie Word beenden, erfolgt erst gar keine Rückfrage, sondern Word zeigt gleichen einen *Datei speichern*-Dialog an (siehe Abbildung 7.5):

```
Public Sub DokumentErstellenAendernDokumentSchliessen()
    ...
    objDocument.Close
```

```
End Sub
```

Wenn Sie das Dokument über diesen Dialog speichern, wird die Prozedur problemlos beendet. Anderenfalls löst dies einen Fehler aus, weil die *Close*-Methode des *Document*-Objekts fehlgeschlagen ist (siehe Abbildung 7.6).

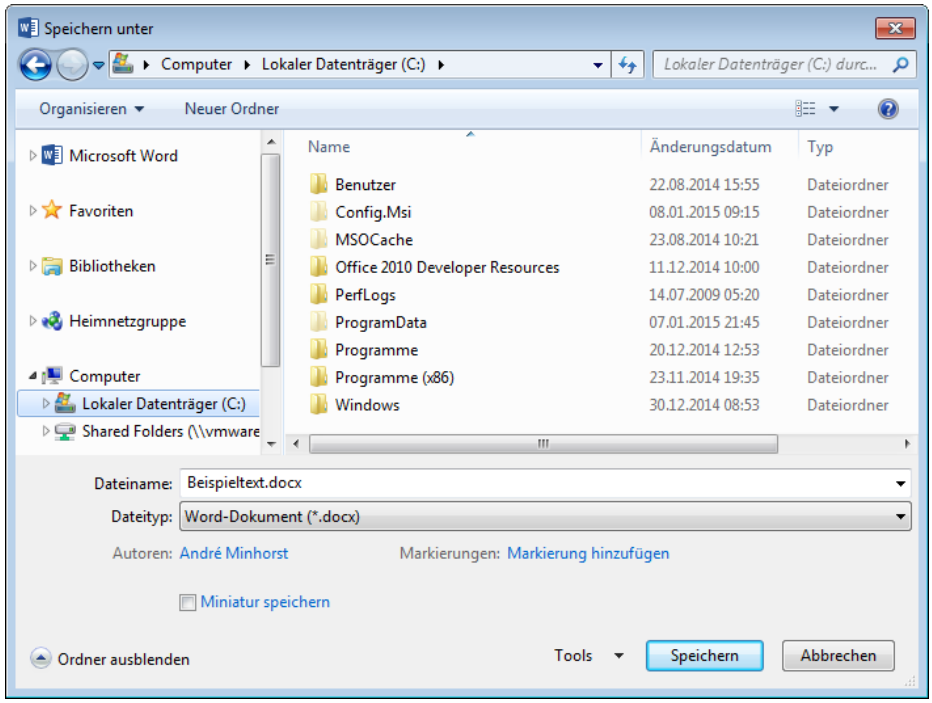


Abbildung 7.5: *Datei speichern*-Dialog

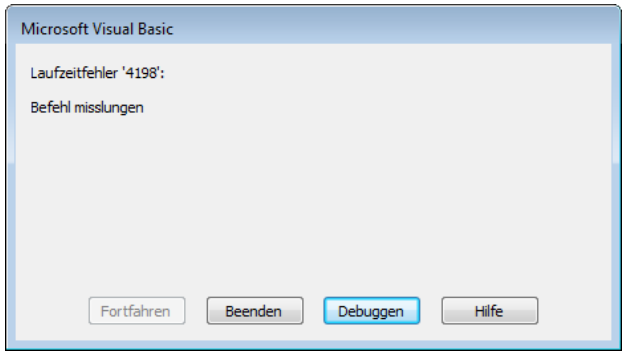


Abbildung 7.6: Fehler, wenn der *Datei speichern*-Dialog ohne Speichern geschlossen wurde

Wenn Sie die obige Prozedur wie folgt erweitern, erhalten Sie dann zumindest eine aussagekräftige Meldung:

```
Public Sub DokumentErstellenAendernDokumentSchliessen()
    ...
    On Error Resume Next
    objDocument.Close
    Select Case Err.Number
        Case 0
        Case Else
            MsgBox "Das Dokument konnte nicht gespeichert werden, da der Benutzer 7
                                                    den Speichervorgang abgebrochen hat."
    End Select
End Sub
```

Grundsätzlich können Sie aber bei Verwendung der *Close*-Methode *theoretisch* festlegen, ob Word rückfragen soll, wie und ob die Datei zu speichern ist. Dazu legen Sie für den ersten Parameter diese Methode einen der folgenden Werte fest:

» *wdDoNotSaveChanges* (*0*): Schließt das Dokument, ohne die Änderungen zu speichern. Bei einem neuen Dokument bedeutet dies, dass es erst gar nicht auf der Festplatte gespeichert wird.

» *wdPromptToSaveChanges* (*-2*): Standardeinstellung. Hier erscheint bei einem neuen Dokument direkt der *Datei speichern*-Dialog.

» *wdSaveChanges* (*-1*): Hier erscheint bei einem neuen Dokument ebenfalls der *Datei speichern*-Dialog.

Warum theoretisch? Weil die Konstante *wdPromptToSaveChanges* zumindest unter Word 2010 und Word 2013 einen Bug hat und genau das gleiche Verhalten liefert wie die Konstante *wdSaveChanges*. Wir wollen uns an dieser Stelle nicht damit beschäftigen, warum diese Bug existiert und nicht behoben wird, sondern einfach eine zuverlässige Methode verwenden.

7.3.7 Word-Dokument explizit speichern

Dazu speichern wir das Word-Dokument einfach, bevor es per Code geschlossen wird und damit die nicht sauber kontrollierbaren Speicher-Automatismen ausgelöst werden.

Dazu liefert das *Document*-Objekt die Methode *SaveAs2*. Diese Methode erwartet eine ganze Reihe Parameter, von denen die ersten beiden am wichtigsten sind. Der erste heißt *Filename*, der zweite *FileFormat*. Der zweite erwartet eine Konstante, die das Dateiformat repräsentiert. Während es in der Excel-Bibliothek noch einfach war, die passende Enumeration zu finden (*xlFileFormat*), muss man in Word etwas länger im Objektkatalog suchen. Dann stößt man früher oder später auf die Auflistung *wdSaveFormat* (siehe Abbildung 7.7).

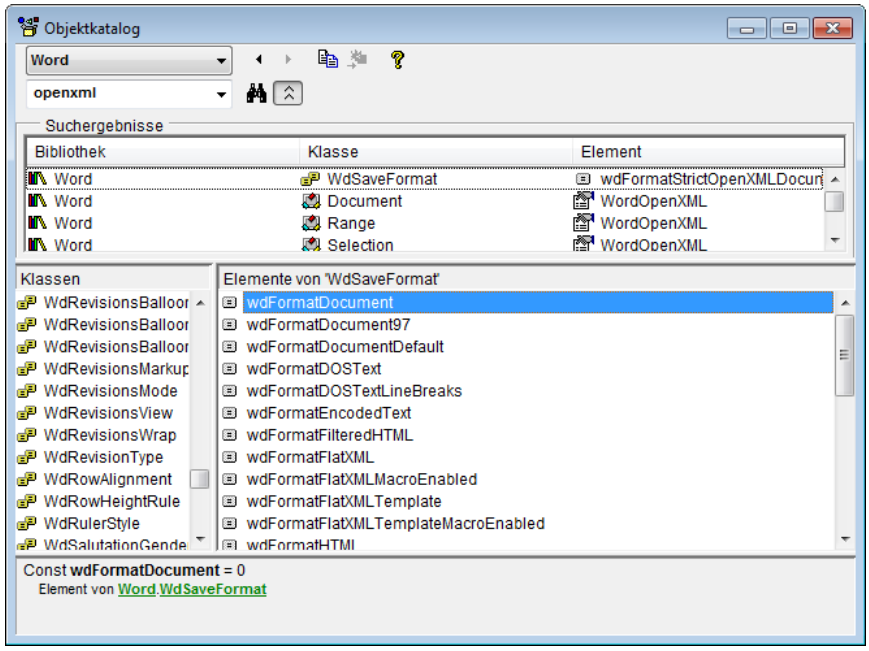


Abbildung 7.7: Konstanten für das Dateiformat im Objektkatalog

Aber welches der vielen Formate verwenden wir für welchen Zweck? Das finden wir am schnellsten heraus, indem wir uns die Format der gängigen Word-Dokumente ansehen. Legen wir einfach ein neues Dokument an und speichern von Word aus manuel es unter dem Namen *Test_Format.docx* mit dem Dateityp *Word-Dokument (*.doc)*. Nun verwenden wir die folgende Funktion, um das aktuell geöffnete Word-Dokument zu referenzieren und seinen Dateityp zu ermitteln:

```
Public Sub FormatAktuellesDokument()
    Dim objWord As Word.Application
    Dim objDocument As Word.Document
    Set objWord = GetObject(, "Word.Application")
    Set objDocument = objWord.ActiveDocument
    Debug.Print objDocument.SaveFormat
End Sub
```

Für die *.docx*-Datei liefert dies den Wert *12*. Nun schauen wir im Objektkatalog nach, welcher Dateityp diesem Wert entspricht. Dazu durchlaufen wir die einzelnen Elemente der Auflistung *wdSaveFormat*. Damit finden wir schnell heraus, dass die passende Konstante *wdFormatXMLDocument* heißt (nicht, dass wir diese Konstante unbedingt benötigen – wir könnten in *SaveAs2* auch einfach den Wert *12* nutzen). Auf diese Weise finden wir einige Dateiformate heraus:

- *.docx*: *wdFormatXMLDocument* (*12*) – Word 2007 und neuer
- *.docm*: *wdFormatXMLDocumentMacroEnabled* (*13*) – Word 2007 und neuer, mit aktivierten Makro
- *.doc: wdFormatDocument* (*0*) – Word 97-2003
- *.dotx*: *wdFormatXMLTemplate* (*14*) – Vorlage Word 2007 und neuer
- *.dot*/*.dotm*: *wdFormatXMLTemplateMacroEnabled* (*15*) – Vorlage Word 2007 und neuer mit Makros
- *.pdf*: *wdFormatPDF* (*17*) – PDF-Dokument

Diese Werte nutzen Sie beim Speichern, wenn Sie explizit das Dateiformat vorgeben möchten. Normalerweise speichert Word die Dokumente im Format *wdFormatXMLDocument*.

Die folgende Prozedur erstellt ein neues Dokument und speichert es unter einem bestimmten Namen – hier mit der Endung *.doc*. Ein anschließender Test mit der obigen Funktion ergibt jedoch, dass das Dokument im Format *wdFormatXMLDocument* gespeichert wurde. Wollen Sie also ein Dokument im Format *wdFormatDocument* (Access 97-2003) speichern, geben Sie die entsprechende Konstante zusätzlich zum Dateinamen an:

```
Public Sub DokumentErstellenUndSpeichern()
    Dim objWord As Word.Application
    Dim objDocument As Word.Document
    Set objWord = New Word.Application
    objWord.Visible = True
    Set objDocument = objWord.Documents.Add
    ActivateApp objDocument
    objDocument.Range = objDocument.Range & "bla"
    On Error Resume Next
    Kill CurrentProject.Path & "\SaveAs2.doc"
    On Error GoTo 0
    objDocument.SaveAs2 CurrentProject.Path & "\SaveAs2.doc" ', wdFormatXMLDocument
    objDocument.Close
    objWord.Quit
End Sub
```

Übrigens: Wenn Sie kein Verzeichnis angeben, speichert Word die Datei in dem Ordner, den Sie mit der Funktion *CurDir* ermitteln können – zum Beispiel in *C:\Users\<Benutzername>\Documents*. Dieses Verzeichnis können Sie mit der Funktion ChDir ändern. Wenn Sie beispielsweise das aktuelle Datenbankverzeichnis als aktuelles Verzeichnis verwenden möchten, setzen Sie die folgende Anweisung im Direktfenster (oder in einer Prozedur) ab:

```
ChDir CurrentProject.Path
```

8 Word-Dokumente mit Daten füllen

Interessant wird Word für uns natürlich, wenn wir die Daten aus einer Access-Datenbank hineinschreiben können. Schauen wir uns also an, wie dies gelingt – zunächst anhand einer kleinen Artikelliste. Später wollen wir dann auch noch beispielsweise Bilddateien aus einer Datenbank in das Word-Dokument einfügen.

8.1 Kontaktliste in Word erstellen

Zum Start wollen wir die Daten der Tabelle *tblKontakte* aus der Access-Datenbank in eine Word-Datei schreiben. Diese müssen wir zunächst erstellen, danach fügen wir die Daten in das Dokument ein.

Das Einfügen der Daten erfolgt durch das Markieren des letzten vorliegenden Absatzes (*Paragraphs.Last.Range*) und dem Einfügen der neuen Zeile in den dahinter liegenden Bereich (*InsertAfter*). Da wir in diesem Fall gleich mit dem ersten Absatz beginnen, hängen wir den Zeilenumbruch (*vbCr*) hinten an den anzufügenden Ausdruck an.

Die notwendigen Anweisungen legen wir für eine Ereignisprozedur an, die durch die Schaltfläche *cmdAlleKontakteNachWord* im Formular *frmKontakteNachWord* ausgelöst wird:

```
Private Sub cmdAlleKontakteNachWord_Click()
    Dim objDocument As Word.Document
    Dim objRange As Word.Range
    Dim db As DAO.Database
    Dim rst As DAO.Recordset
    Set objDocument = New Word.Document
    objParent.Visible = True
    ActivateApp objDocument
    Set db = CurrentDb
    Set rst = db.OpenRecordset("SELECT Vorname, Nachname, Strasse, PLZ, Ort ⤵
                                                    FROM tblKontakte", dbOpenDynaset)
    Do While Not rst.EOF
        Set objRange = objDocument.Paragraphs.Last.Range
        objRange.InsertAfter rst!Vorname & vbTab & rst!Nachname & vbTab _
            & rst!Strasse & vbTab & rst!PLZ & vbTab & rst!Ort & vbCr
        rst.MoveNext
    Loop
End Sub
```

Das Ergebnis überzeugt noch nicht komplett (siehe Abbildung 8.1) – aber dies nur, weil die Tabulatoren nicht so eingerichtet sind, dass alle Elemente genau untereinander stehen.

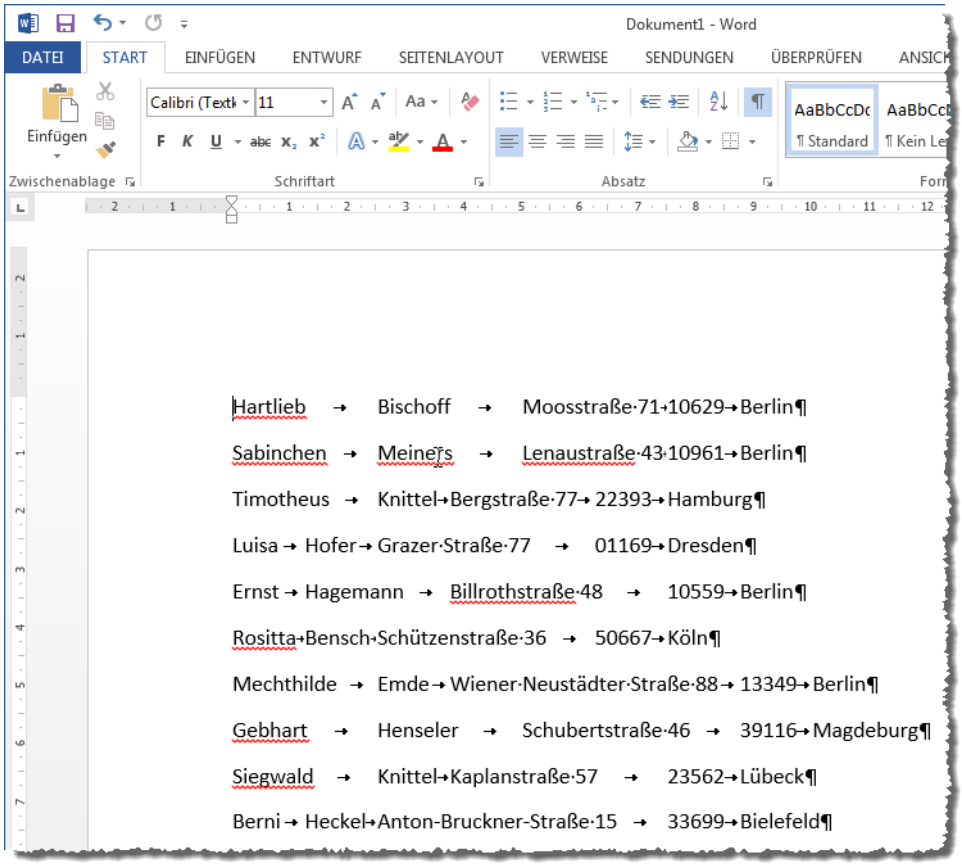


Abbildung 8.1: Ergebnis des ersten Exports der Daten in ein Word-Dokument

Das ist aber kein Problem – wir passen einfach die Tabulator-Positionen ein wenig an. Dazu nutzen wir die *TabStops*-Auflistung des ersten Absatzes. Die übrigen Absätze übernehmen die hier vorgenommenen Einstellungen, sodass das Ergebnis nun wie in Abbildung 8.2 aussieht. Außerdem haben wir noch eine erste Zeile mit den Spaltenüberschriften hinzugefügt:

```
Private Sub cmdAlleKontakteNachWord_Click()
    ...
    With objDocument.Paragraphs(1)
       .TabStops(1).Position = 0
       .TabStops(2).Position = 80
       .TabStops(3).Position = 160
```

```
        .TabStops(4).Position = 350
    End With
    Set objRange = objDocument.Paragraphs.Last.Range
    objRange.InsertAfter "Vorname" & vbTab & "Nachname" & vbTab & "Straße" _
        & vbTab & "PLZ und Ort" & vbCr
    ...
End Sub
```

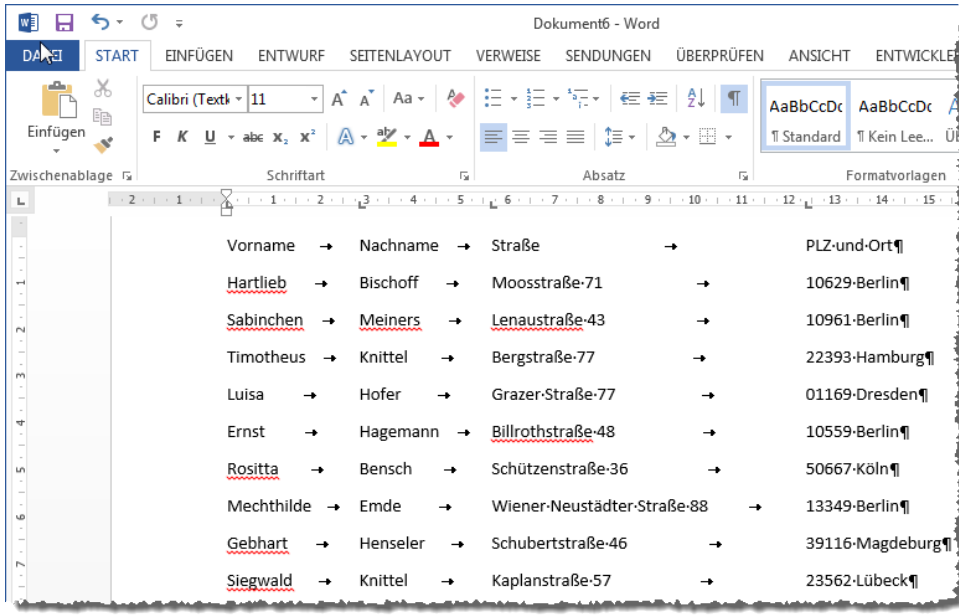


Abbildung 8.2: Ausgabe mit korrekten Tabulatoren

8.1.1 Performance

So richtig schnell überträgt die Prozedur die Daten nicht. Dabei kommt es allerdings auch darauf an, was Sie später noch für Operationen mit den eingefügten Texten durchführen wollen. Gleich im Anschluss zeigen wir beispielsweise, wie Sie die Inhalte einzelner Spalten der eingefügte Daten einheitlich formatieren können – beispielsweise fett oder kursiv. Wenn solche eine Formatierung nicht gewünscht ist, geht es mit der folgenden Variante wesentlich schneller:

```
Private Sub cmdKontakteNachWordSchnell_Click()
    ...
    Dim strWord As String
    ...
    strWord = "Vorname" & vbTab & "Nachname" & vbTab & "Straße" & vbTab _
```

```
        & "PLZ und Ort" & vbCr
    Do While Not rst.EOF
        strWord = strWord & rst!Vorname & vbTab & rst!Nachname & vbTab & rst!Strasse _
            & vbTab & rst!PLZ & " " & rst!Ort & vbCr
        rst.MoveNext
    Loop
    Set objRange = objDocument.Paragraphs.Last.Range
    objRange.InsertAfter strWord
End Sub
```

Diese Prozedur fügt die Zeichenkette aus allen Datensätzen zunächst in einer Variablen namens *strWord* zusammen und übergibt diese dann in einem Rutsch an das Word-Dokument.

8.1.2 Zeichenformate festlegen

Nun möchten wir vielleicht jeweils die Nachnamen fett drucken. Außerdem soll die Zeile mit den Spaltenköpfen ebenfalls fett erscheinen. Dazu liefert das *Range*-Objekt einige interessante Eigenschaften – mehr dazu weiter unten.

Wir könnten nun direkt beim Anlegen die Zeichenformate für die einzelnen Elemente festlegen. Dies gelingt am einfachsten, indem wir beim Anlegen zunächst die einzelnen Elemente mit jeweils einem Bookmark markieren und nachher beim Formatieren auf die jeweiligen Bookmarks zugreifen.

Die Vornamen sollen dabei jeweils eine Textmarke mit dem Namen *Vorname1*, *Vorname2*, *Vorname3* und so weiter heißen, die Nachnamen erhalten die Bezeichnunge *Nachname1*, *Nachname2*, *Nachname3* und die Textmarken der übrigen Elemente erhalten entsprechende Namen.

Damit dies gelingt, erweitern wir die Prozedur von oben ein wenig. Wir fügen nur eine einzige Objektvariable des Typs *Bookmark* hinzu und nennen diese *objBookmark*. Die Prozedur, die durch die neue Schaltfläche *cmdKontakteMitMarkierung* ausgelöst wird, startet wie die vorheriger, daher hier in gekürzter Fassung (wir beschränken die auszugebenden Daten auf zehn Datensätze):

```
Private Sub cmdKontakteMitMarkierung_Click()
    ...
    Dim objBookmark As Word.Bookmark
    Set rst = db.OpenRecordset("SELECT TOP 10 KontaktID, Vorname, Nachname, Strasse, 7
                                            PLZ, Ort FROM tblKontakte", dbOpenDynaset)
    ...
    Set objRange = objDocument.Paragraphs.Last.Range
    objRange.InsertAfter "Vorname" & vbTab & "Nachname" & vbTab & "Straße" & vbTab _
        & "PLZ und Ort" & vbCr
```

Nach dem Anlegen der ersten Zeile mit den Spaltenüberschriften definieren wir das erste Bookmark – und zwar über den Inhalt des aktuelle *Range*-Objekts:

```
Set objBookmark = objDocument.Bookmarks.Add("Spaltenkoepfe", objRange)
```

Danach beginnt der Durchlauf der *Do While*-Schleife, wobei wir diesmal zunächst den Beginn des mit dem Objekt *objRange* referenzierten *Range*-Elements mit der *Start*-Eigenschaft auf das Ende des bis dahin gültigen *Range* einstellen. Dann fügt die Prozedur den Vornamen ein. Das *Range*-Objekt wird somit auf den Vornamen erweitert. Damit können wir nun ein *Bookmark*-Objekt für den Vornamen festlegen. Dieses erhält eine Bezeichnung bestehend aus der Zeichenkette *Vorname* und dem Primärschlüsselwert des aktuellen Datensatzes:

```
Do While Not rst.EOF
    objRange.Start = objRange.End
    objRange.InsertAfter rst!Vorname
    Set objBookmark = objDocument.Bookmarks.Add("Vorname" & rst!KontaktID, objRange)
```

Danach fügt die Prozedur hinter der aktuellen Markierung das Tabulator-Element ein und markiert erneut den Bereich ab der *End*-Position des vorherigen Ranges als *Start* des neuen Ranges. Nun folgt der Nachname, der wiederum durch ein Bookmark-Objekt eingefasst wird:

```
    objRange.InsertAfter vbTab
    objRange.Start = objRange.End
    objRange.InsertAfter rst!Nachname
    Set objBookmark = objDocument.Bookmarks.Add("Nachname" & rst!KontaktID, objRange)
```

Auf die gleiche Weise geht es mit den übrigen Elementen weiter:

```
    objRange.InsertAfter vbTab
    objRange.Start = objRange.End
    objRange.InsertAfter rst!Strasse
    Set objBookmark = objDocument.Bookmarks.Add("Strasse" & rst!KontaktID, objRange)
    objRange.InsertAfter vbTab
    objRange.Start = objRange.End
    objRange.InsertAfter rst!PLZ & " " & rst!Ort
    Set objBookmark = objDocument.Bookmarks.Add("PLZOrt" & rst!KontaktID, objRange)
    objRange.InsertAfter vbCr
    rst.MoveNext
Loop
```

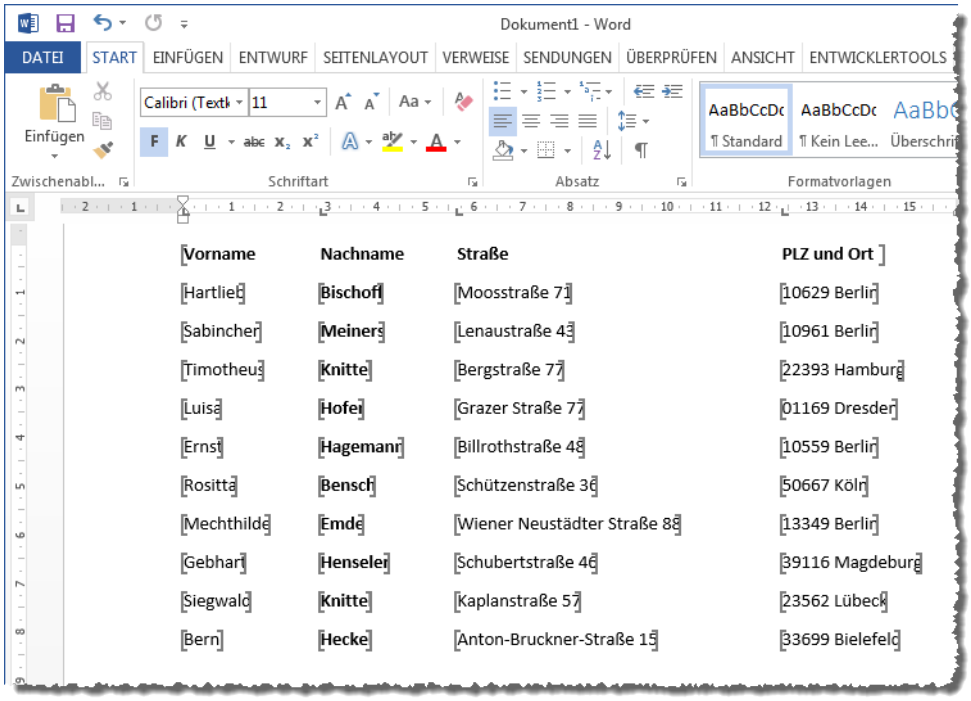
Nun folgt die eigentliche Formatierung. Dabei stellt die Prozedur zunächst die Eigenschaft *Font.Bold* für den Inhalt des Bookmarks *Spaltenkoepfe* auf den Wert *True* ein:

```
objDocument.Bookmarks("Spaltenkoepfe").Range.Font.Bold = True
```

Danach bewegt sie den Datensatzzeiger wieder auf die erste Position des Recordsets und durchläuft die Datensätze von vorn. Dabei markiert sie jeweils den Nachnamen der aktuellen Zeile fett:

```
        rst.MoveFirst
        Do While Not rst.EOF
            objDocument.Bookmarks("Nachname" & rst!KontaktID).Range.Font.Bold = True
            rst.MoveNext
        Loop
    End Sub
```

Das Ergebnis sieht nun wie in Abbildung 8.3 aus. Wie Sie sehen, wurde dort tatsächlich für jedes Element eine Textmarke festgelegt. Leider ist das Einfügen der Texte plus das Hinzufügen der Textmarken doch sehr langsam, sodass diese Vorgehensweise doch eher für weniger umfangreiche Datenmengen geeignet ist, nicht jedoch für größere Tabellen.



Vorname	Nachname	Straße	PLZ und Ort
Hartliet	Bischoff	Moosstraße 71	10629 Berlin
Sabincher	Meiners	Lenaustraße 43	10961 Berlin
Timotheus	Knitte	Bergstraße 77	22393 Hamburg
Luisa	Hofer	Grazer Straße 77	01169 Dresden
Ernst	Hagemann	Billrothstraße 48	10559 Berlin
Rositta	Bensch	Schützenstraße 36	50667 Köln
Mechthilde	Emde	Wiener Neustädter Straße 88	13349 Berlin
Gebhart	Henseler	Schubertstraße 46	39116 Magdeburg
Siegwald	Knitte	Kaplanstraße 57	23562 Lübeck
Bern	Hecke	Anton-Bruckner-Straße 15	33699 Bielefeld

Abbildung 8.3: Mit Bookmarks versehene und fett gedruckte Elemente

8.1.3 Mögliche Formatierungen

Auf die soeben vorgestellte Weise können Sie einige Textformatierungen vornehmen. Hier sind die üblichen Arten:

» *.Font.Bold*: Fett

» *.Font.Italic*: Kursiv

» *.Font.Name*: Name der Schriftart

» *.Font.Size*: Größe der Schriftart

8.1.4 Zeichenformate nutzen

Im folgenden Beispiel wollen wir wiederum ein Dokument mit dem gleichen Inhalt erzeugen. Diesmal wollen wir die *Format*-Eigenschaften aber nicht direkt zuweisen, sondern über Zeichenformat. Zeichenformate legen Sie normalerweise über eine Dokumentvorlage an und nutzen diese dann in den darauf basierenden Dokumenten. Wir wollen nun zwei Zeichenformate definieren (*Hervorhebung1* und *Hervorhebung2*) und diese für die Formatierung von fett gedruckten und kursiv gedruckten Texten nutzen.

Zunächst wollen wir den Verweis auf das erstellte Dokument nicht in der erstellenden Prozedur unterbringen, sondern als modulweit deklarierte Variable im Kopf des Klassenmoduls:

```
Dim objDocumentHervorhebungen As Word.Document
```

Damit stellen wir sicher, dass wir auch nach dem Verlassen der Prozedur zum Erstellen des Dokuments noch auf dieses zugreifen können.

Die Schaltfläche *cmdKontakteMitZeichenformat* löst die folgende Prozedur aus, die weitgehend wie die vorhergehenden aussieht, aber erstens das Dokument in der extern deklarierten Variable speichert und dann eine weitere Prozedur namens *ZeichenformatDefinieren* aufruft. Diese soll zwei Zeichenformatvorlagen namens *Hervorhebung1* und *Hervorhebung2* anlegen und mit den entsprechenden Formateigenschaften versehen (siehe weiter unten). Danach können wir diese Zeichenformatvorlagen der Eigenschaft *Style* des *Range*-Bereichs zuweisen, um die in der Zeichenformatvorlage festgelegten Eigenschaften auf die Zeichen des *Range*-Bereichs anzupassen. Dies erledigen wir einmal mit *Hervorhebung1* für die Spaltenköpfe und mit *Hervorhebung2* für die Spalte mit den Nachnamen:

```
Private Sub cmdKontakteMitZeichenformat_Click()
    ...
    Set objDocumentHervorhebungen = New Word.Document
    ...
    ZeichenformatDefinieren objDocumentHervorhebungen
    ...
```

```
    objDocumentHervorhebungen.Bookmarks("Spaltenkoepfe").Range.Style = "Hervorhebung1"
    rst.MoveFirst
    Do While Not rst.EOF
        objDocumentHervorhebungen.Bookmarks("Nachname" & rst!KontaktID).Range.Style _
            = "Hervorhebung2"
        rst.MoveNext
    Loop
End Sub
```

Dies führt dazu, dass das Dokument wie zuvor erstellt wird – mit fett gedruckter Schrift in der Titelzeile und kursiver Schrift in der Spalte *Nachname*. Die Definition der Zeichenformate erledigt die folgende Prozedur:

```
Private Sub ZeichenformatDefinieren(objDocument As Word.Document)
    Dim objStyle As Word.Style
    Set objStyle = objDocument.Styles.Add("Hervorhebung1", wdStyleTypeCharacter)
    With objStyle
        .Font.Bold = True
        .QuickStyle = True
    End With
    Set objStyle = objDocument.Styles.Add("Hervorhebung2", wdStyleTypeCharacter)
    With objStyle
        .Font.Italic = True
        .QuickStyle = True
    End With
End Sub
```

Die Prozedur nimmt das per Parameter übergebene *Document*-Objekt auf und erstellt dafür zwei neue Zeichenformatvorlagen. Dafür nutzt es die *Add*-Methode der *Styles*-Auflistung des Dokuments und weist dieser den Namen der Formatvorlage sowie den Typ zu – *wdStyleTypeCharacter* steht dabei für Zeichenformatvorlage. Die neue Vorlage landet in der Variablen *objStyle*, die wir anschließend dazu nutzen, erstens die Schriftart einzustellen und diese zweitens den Schnellformatierungen hinzuzufügen (*QuickStyles = True*).

Der Vorteil bei der Verwendung von Zeichenformaten ist, dass Sie nun Änderungen an den damit formatierten Elementen nur noch an einer Stelle erledigen müssen – nämlich an den Zeichenformatvorlagen. Dazu haben wir dem Formular *frmKontakteNachWord* eine weitere Schaltfläche hinzufügt, welche auf das in der Variablen *objDocumentHervorhebungen* gespeicherte Dokument zugreift und die Eigenschaften der Zeichenformatvorlagen *Hervorhebung1* und *Hervorhebung2* vertauscht (aus fett wird kursiv und umgekehrt):

```
Private Sub cmdZeichenformatAendern_Click()
    With objDocumentHervorhebungen.Styles("Hervorhebung1")
        .Font.Bold = Not .Font.Bold
```

```
        .Font.Italic = Not .Font.Italic
    End With
    With objDocumentHervorhebungen.Styles("Hervorhebung2")
        .Font.Italic = Not .Font.Italic
        .Font.Bold = Not .Font.Bold
    End With
End Sub
```

Die Eigenschaft *Quickstyle* legt fest, dass die Formatvorlage in den Schnellvorlagen angezeigt wird. Diese können Sie dann auch über die Benutzeroberfläche nutzen (siehe Abbildung 8.4).

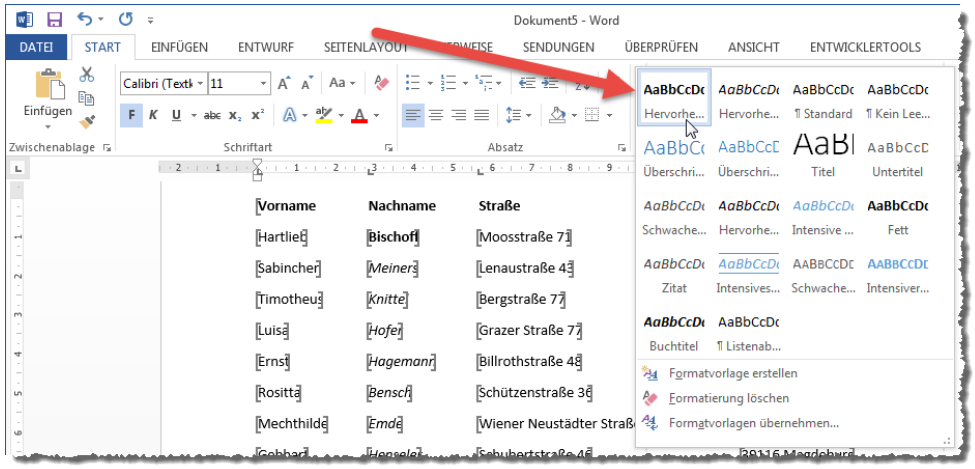


Abbildung 8.4: Schnellformatvorlagen

8.1.5 Absatzformatvorlagen

Auf ähnliche Weise definieren Sie Absatzformatvorlagen und weisen diese einem Dokument zu, sofern dies noch nicht durch die Dokumentvorlage geschieht (alternativ können Sie auch eine Dokumentvorlage damit ausstatten).

Der erste Unterschied ist, dass Sie der *Add*-Methode als zweiten Parameter den Wert *wdStyleTypeParagraph* übergeben. Darüber hinaus besitzen Absatzformatvorlagen natürlich einige Eigenschaften mehr – etwa zur Einstellung der Abstände des Absatzes nach oben oder unten, zur Einrichtung einer Einrückung oder zum Festlegen von Nummerierungen, Aufzählungszeichen et cetera.

8.2 Tabellen in Word erstellen

Da die in ein Word-Dokument zu exportierenden Daten ja aus Tabellen stammen, kann es natürlich sein, dass diese auch in Tabellenform dargestellt werden sollen.

Die Prozedur durchläuft ein Recordset auf Basis der Tabelle *tblKontakte*. Vor dem ersten Durchlauf erstellt die Prozedur eine Tabelle mit zwei Zeilen und sieben Spalten. Die erste Zeile nimmt die Spaltenüberschriften auf, hier *KontaktID*, *Vorname*, *Nachname*, *Straße*, *PLZ*, *Ort* und *Land*. Die Überschriften werden fett formatiert.

Die zweite Zeile bleibt leer. Warum? Weil beim nachfolgenden Anlegen neuer Zeilen mit der *Add*-Methode der *Rows*-Auflistung jeweils die Formatierungen der vorhergehenden Zeile übernommen werden. Da die Zeilen mit den eigentlichen Daten aber normal und nicht fett dargestellt werden sollen, haben wir eine Pufferzeile ohne Formatierungen dazwischengeschaltet.

Beim Durchlaufen der Schleife fügt die Prozedur jeweils eine neue Zeile pro Datensatz hinzu und trägt die Werte der Felder *KontaktID*, *Vorname*, *Nachname*, *Strasse*, *PLZ*, *Ort* und *Land* in die mit den entsprechenden Koordinaten versehenen *Cell*-Objekte ein.

Die *Autofit*-Methode sorgt schließlich dafür, dass beide Zeilen in der optimalen Spaltenbreite angezeigt werden:

```
Private Sub cmdTabelleErstellen_Click()
    Dim objDocument As Word.Document
    Dim objRange As Word.Range
    Dim db As DAO.Database
    Dim rst As DAO.Recordset
    Dim tbl As Word.Table
    Dim i As Integer
    Set objDocument = New Word.Document
    objDocument.Parent.Visible = True
    AppActivate objDocument.Name
    Set db = CurrentDb
    Set rst = db.OpenRecordset("SELECT TOP 10 * FROM tblKontakte", dbOpenSnapshot)
    Set objRange = objDocument.Range
    Set tbl = objRange.Tables.Add(objRange, 2, 7)
    tbl.Cell(1, 1).Range = "KontaktID"
    tbl.Cell(1, 2).Range = "Nachname"
    tbl.Cell(1, 3).Range = "Vorname"
    tbl.Cell(1, 4).Range = "Straße"
    tbl.Cell(1, 5).Range = "PLZ"
    tbl.Cell(1, 6).Range = "Ort"
    tbl.Cell(1, 7).Range = "Land"
```

9 Informationen aus Word-Dokumenten

Gelegentlich werden Sie die Inhalte von Word-Dokumenten auslesen wollen – beispielsweise, weil Ihnen jemand Artikeldaten als Katalog geschickt hat, die Sie gern in den eigenen Shop übernehmen möchten oder weil diese bestimmte Daten, die interessant für Sie sind, in einer Tabelle enthält.

Vielleicht wollen Sie auch einfach nur den kompletten Inhalt Absatz für Absatz aus dem Dokument heraus in eine Tabelle schreiben, um den Text weiterzuverarbeiten – etwa, weil Sie die Daten in einem Blog bereitstellen möchten.

9.1 Word-Dokumente auslesen

Das Auslesen eines Word-Dokuments läuft normalerweise immer nach dem gleichen Schema ab – je nachdem, für welche Inhaltstypen Sie sich interessieren. Wenn Sie die reinen Texte einlesen möchten, durchlaufen Sie die *Paragraphs*-Auflistung, wenn Sie die enthaltenen Bilder weiterverwenden möchten, greifen Sie auf die *Shapes*- oder die *InlineShapes*-Auflistung zu.

Vielleicht möchten Sie auch nur auf ein bestimmtes Element wie eine Tabelle zugreifen, dann würden Sie dieses Element referenzieren und dann die darin enthaltenen Daten verarbeiten. Die folgenden Beispiele zeigen, wie dies für verschiedene Fälle gelingt.

9.1.1 Absatz für Absatz

Zunächst werden Sie die einzelnen Absätze des Dokuments durchlaufen wollen. Als Beispiel dazu verwenden wir das Dokument, das wir im Beispiel aus »Katalog aus der Datenbank erstellen« ab Seite 229 angelegt haben. Die folgende Prozedur rufen Sie auf, indem Sie den Prozedurnamen samt Pfad zur auszulesenden Word-Datei im Direktfenster absetzen, also etwa so:

```
DokumentEinlesen CurrentProject.Path & "\Artikelkatalog.docx"
```

Die Prozedur erstellt eine Word-Instanz und öffnet das angegebene Dokument. Dazu müssen Sie den Dokumentnamen nochmals in Anführungszeichen einfassen, war wir hier elegant mit der *Chr*-Funktion erledigen. Dann durchlaufen wir alle Elemente der *Paragraphs*-Auflistung des Dokuments in einer *For Each*-Schleife. Innerhalb der Schleife geben wir die enthaltenen Texte einfach im Direktfenster aus:

```
Public Sub DokumentEinlesen(strDokument As String)
    Dim objWord As Word.Application
    Dim objDocument As Word.Document
    Dim objParagraph As Word.Paragraph
```

```
    Set objWord = New Word.Application
    Set objDocument = objWord.Documents.Open(Chr(34) & strDokument & Chr(34))
    For Each objParagraph In objDocument.Paragraphs
        Debug.Print objParagraph.Range.Text
    Next objParagraph
    objDocument.Close
    Set objDocument = Nothing
    objWord.Quit
    Set objWord = Nothing
End Sub
```

Wie das Ergebnis aus Abbildung 9.1 zeigt, gibt die Prozedur die Absätze nicht einfach untereinander aus, sondern fügt noch jeweils einen Zeilenumbruch ein. Das liegt daran, dass einerseits jeder Absatz noch einen Zeilenumbruch enthält und andererseits natürlich auch die *Debug.Print*-Anweisung jede Ausgabe mit einem Zeilenumbruch abschließt.

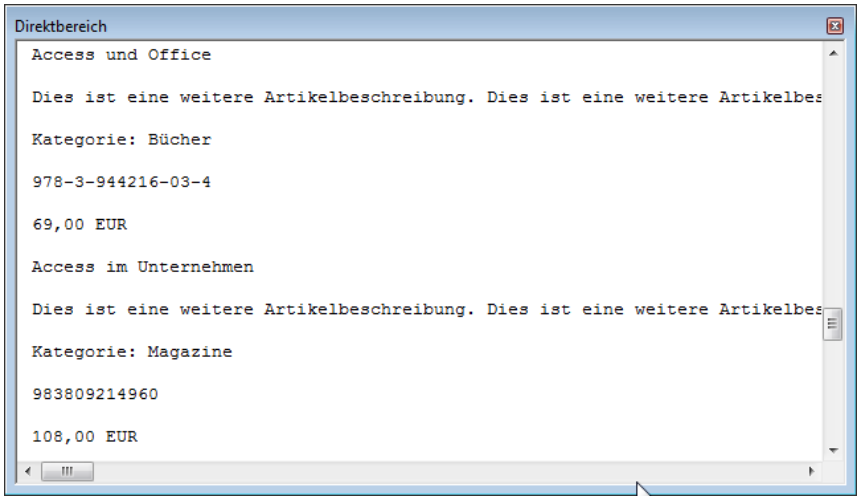


Abbildung 9.1: Ausgabe der Absätze eines Dokuments im Direktfenster

Um die Absätze ohne Zeilenumbruch zu ermitteln, passen wir die Prozedur etwas an. Dabei speichern wir den kompletten Absatz zunächst in der Variablen *strAbsatz* und prüfen dann, ob das letzte Zeichen ein Zeilenumbruch ist (*Chr(13)* oder *vbCr*) und entfernen diesen gegebenenfalls.

Damit erhalten wir nun die reinen Inhalt des Absatzes ohne abschließen Zeilenumbruch:

```
Public Sub DokumentEinlesen(strDokument As String)
    ...
    Dim strAbsatz As String
    ...
```

```
    For Each objParagraph In objDocument.Paragraphs
        strAbsatz = objParagraph.Range.Text
        If Asc(Right(strAbsatz, 1)) = 13 Then
            strAbsatz = Left(strAbsatz, Len(strAbsatz) - 1)
        End If
        Debug.Print strAbsatz
    Next objParagraph
    ...
End Sub
```

9.1.2 Absatzformat ermitteln

Nun wissen wir allerdings nicht, wie der Absatz formatiert ist – die Variable *strAbsatz* enthält lediglich den reinen Text. Optimalerweise hat der Ersteller des Word-Dokuments Absatzformatvorlagen verwendet.

Diese können Sie durch eine Erweiterung der *Debug.Print*-Anweisung der vorherigen Prozedur ganz leicht mit ausgeben:

```
Debug.Print objParagraph.Range.ParagraphStyle, strAbsatz
```

Die Zuordnung einer Absatzformatvorlage liefert wichtige Informationen über die Strukturierung des Dokuments, denn Überschriftenebenen, Fließtexte, Aufzählungen und so weiter sind dann natürlich an der Absatzformatvorlage erkennbar.

Wenn Sie beispielsweise einen Katalog wie in diesem Beispiel einlesen wollen, erkennen Sie an der Formatvorlage für den Titel beispielsweise, wann ein neuer Katalogeintrag beginnt.

Gibt es keine Absatzformatvorlagen, können Sie gegebenenfalls die Formatierungen ermitteln und darüber auf die verschiedenen Strukturelemente schließen.

Die Formatierungen eines Absatzes erhalten sie über verschiedene Eigenschaften, die Schriftart etwa über das *Font*-Objekt des *Range*-Elements (siehe Abbildung 9.2).

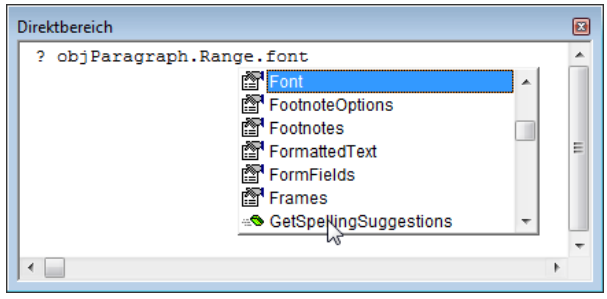


Abbildung 9.2: Ermitteln von Format-Eigenschaften

Ein Tipp hierzu: Wenn Sie IntelliSense für tiefere Verschachtelungen benötigen, legen Sie entsprechende Objektvariablen für die übergeordneten Ebenen fest, also zum Beispiel so wie in Abbildung 9.3.

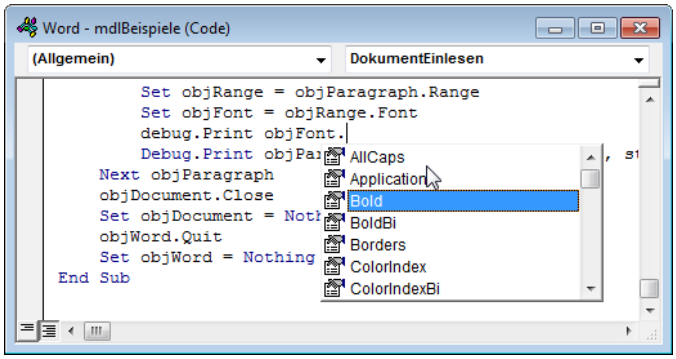


Abbildung 9.3: Zugriff auf IntelliSense in tieferen Ebenen per zusätzlicher Objektvariable

9.2 Bilder aus dem Dokument auslesen

Bilder kommen in Word-Dokumenten in zwei Ausprägungen vor: in einem *Shape*- oder in einem *InlineShape*-Objekt. *Shape*-Objekt können Sie frei im Dokument platzieren (sofern die Verankerung sich auf der gleichen Seite befindet), *InlineShape*-Objekte befinden sich mit dem Text in einer Zeile.

Bei Einfügen eines Bildes über den entsprechenden Ribbon-Eintrag wird ein Bild zunächst als *InlineShape* im Fließtext platziert.

In Word 2013 können Sie daraus ganz einfach ein *Shape*-Objekt machen: Dazu klicken Sie auf das Symbol rechts vom Bild und wählen im nun erscheinenden Popup-Menü einen der Einträge unterhalb von *Mit Textumbruch* aus (siehe Abbildung 9.4).

Die schlechte Nachricht ist: So leicht Sie ein Bild per VBA in einem Word-Dokument platzieren können, so kompliziert ist der Weg, um dieses aus dem Dokument heraus zu exportieren. Die gute Nachricht lautet: Die Methode, die wir nachfolgend vorstellen, behandelt alle Bilder, ob in *Shapes* oder in *InlineShapes*, gleich.

Allerdings funktioniert sie auch nur für aktuelle Word-Dokumente, also solche, die im XML-Format vorliegen und die somit mit Word 2007 oder höher erstellt wurden.

Diese Dokumente haben gemein, dass ein Dokument nicht mehr ein Monolith ist, den man eigentlich nur über die Anwendung Word lesen oder ändern kann. Wenn Sie ein *.docx*-Dokument mit der Dateiendung *.zip* versehen und doppelt darauf klicken, sehen Sie, wie das gemeint ist – Sie erhalten dann eine Ordner-Struktur wie in Abbildung 9.5.

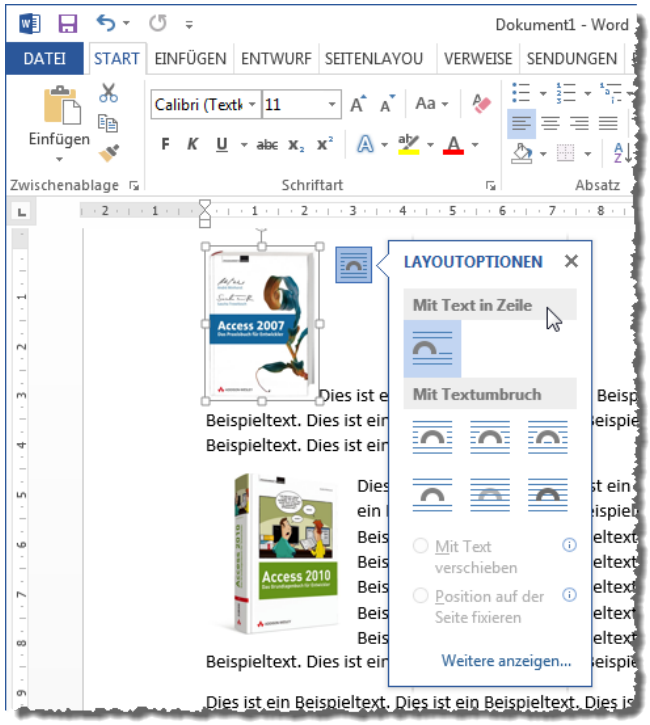


Abbildung 9.4: Beispiele für *Shape*- und *InlineShape*-Objekte

Wenn Sie hier ein wenig suchen, finden Sie im Unterordner *word/media* schnell die beiden im Beispieldokument vorhanden Bilddateien (siehe Abbildung 9.6).

Falls Sie nun denken, wir ändern die Dateiendung des Word-Dokument per VBA in *.zip* um und extrahieren die Bilder einfach aus diesem Archiv, machen Sie es sich zu einfach:

Dort finden sich ja beispielsweise keinerlei Informationen, welchem Absatz die Bilder zugeordnet sind. Diese Information wollen wir jedoch beim Exportieren der Bilder schon erhalten, da wir diese sonst ja nur lose neben dem Inhalt des Word-Dokuments in der Datenbank speichern könnten.

9.2.1 Bilder per VBA extrahieren

Wie aber können wir diese Aufgabe lösen? Hier hilft ein Blick in das Objektmodell von Word. Da wir es ja mit einem XML-Dokument zu tun haben, sollte man doch irgendwie auf den XML-Code der einzelnen Elemente des Dokuments herankommen können.

Und genau das ist der Fall: Beispielsweise das *Range*-Objekt liefert die Eigenschaft *XML* (siehe Abbildung 9.7).

10 Word-Vorlage aus Access füllen

In dieser Lösung möchten wir eine Word-Dokumentvorlage erstellen, die ein einfaches Anschreiben abbildet – also eines, dass Sie etwa für einen Brief, ein Kündigungsschreiben oder ähnliche Zwecke nutzen können.

Die Dokumentvorlage soll einige feste Elemente wie einen Briefkopf, einen Fußbereich und die Absenderadresse enthalten.

Außerdem soll sie einige Textmarken bereitstellen, die Informationen wie die Empfängeradresse, den Betreff, das Datum und den eigentlichen Inhalt aufnehmen sollen.

10.1 Word-Vorlage erstellen

Als Erstes erstellen wir eine geeignete Vorlage. Dies erledigen wir direkt in Word. Öffnen Sie also ein neues Dokument und speichern Sie es zunächst unter dem Dateityp *Word-Vorlage (*.dotx)*.

Dies ändert sofort das Zielverzeichnis in das Verzeichnis der benutzerdefinierten Office-Vorlagen (siehe Abbildung 10.1). Wir verwenden den Namen *Anschreiben.dotx*.

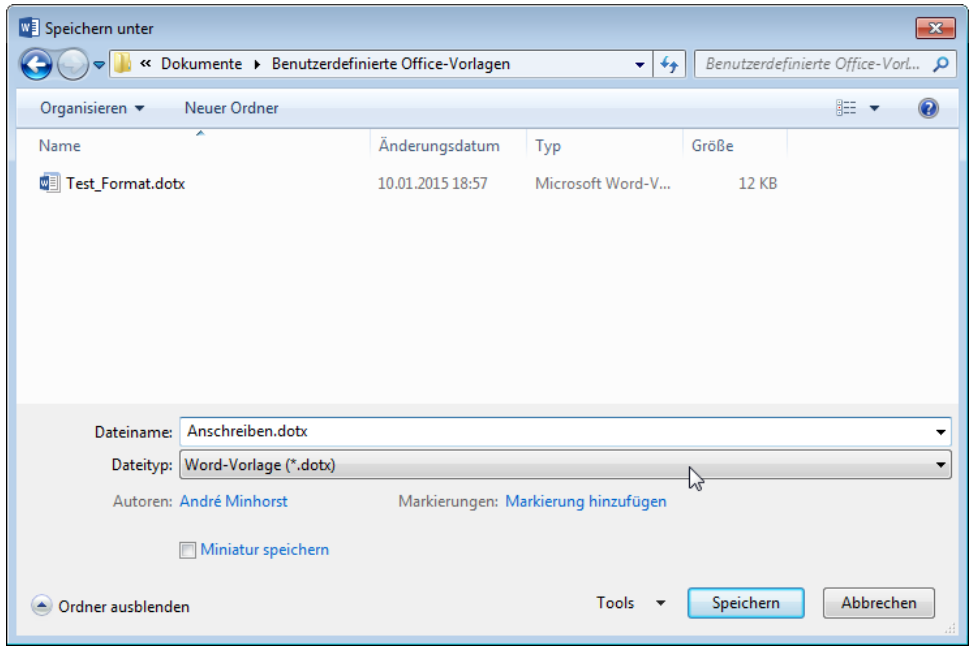


Abbildung 10.1: Festlegen des Dateityps und des Speicherorts

Wenn Sie das aktuelle Vorlagenverzeichnis für benutzerdefinierte Vorlagen unter Word ermitteln wollen, verwenden Sie die folgende Prozedur:

```
Public Sub WordVorlagenordner()
    Dim objWord As Word.Application
    Set objWord = New Word.Application
    Debug.Print objWord.Options.DefaultFilePath(wdUserTemplatesPath)
    objWord.Quit
    Set objWord = Nothing
End Sub
```

Die soeben erstellte Vorlage findet sich auf meinem Rechner etwa im Verzeichnis aus Abbildung 10.7.

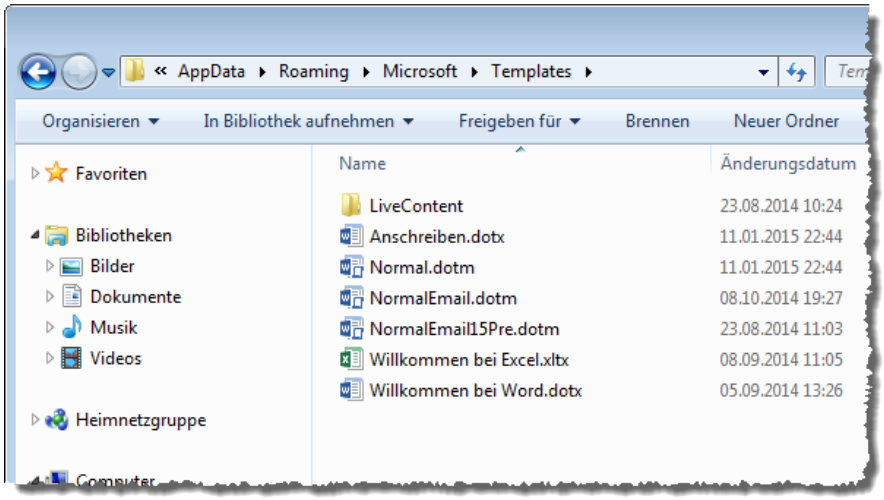


Abbildung 10.2: Vorlagen-Ordner von Word

10.1.1 Seitenränder einstellen

Danach beginnen wir, die Vorlage mit Informationen zu füllen. Als Erstes stellen wir die Spaltenränder auf gängige Werte ein. Dabei handelt es sich um benutzerdefinierte Seitenränder, die Sie über den Ribbonbefehl *Seitenlayout|Seite einrichten|Seitenränder|Benutzerdefinierte Seitenränder...* einstellen (siehe Abbildung 10.3).

Hier nehmen Sie die Einstellungen aus dem Dialog *Seite einrichten* vor (siehe Abbildung 10.4).

Nun müssen wir uns überlegen, wie wir die statischen und die dynamischen Inhalte im frisch erzeugten Word-Dokument unterbringen. Manch einer schwört auf Textfelder, es gibt jedoch keinen Grund, einfach eine Tabelle oder mehrere Tabellen zu nutzen.

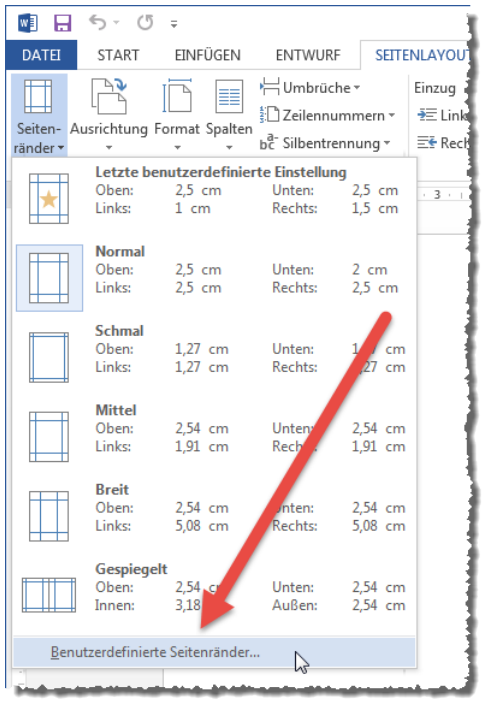


Abbildung 10.3: Öffnen des Dialog zum Einstellen der benutzerdefinierten Seitenränder

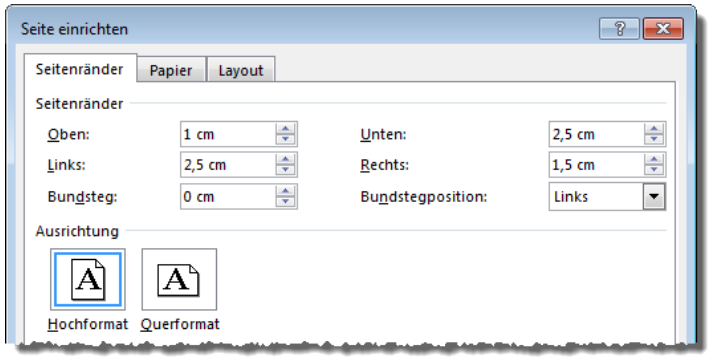


Abbildung 10.4: Einstellungen für die Seitenränder

10.1.2 Briefkopf hinzufügen

Je nachdem, wie Sie Ihren Briefkopf gestalten möchten, reicht möglicherweise schon eine kleine Tabelle mit zwei oder drei Spalten dafür aus. Diese legen Sie blitzschnell mit dem Ribbon-Eintrag

Einfügen|Tabellen|Tabelle und anschließender Auswahl der Dimension der Tabelle an (siehe Abbildung 10.5).

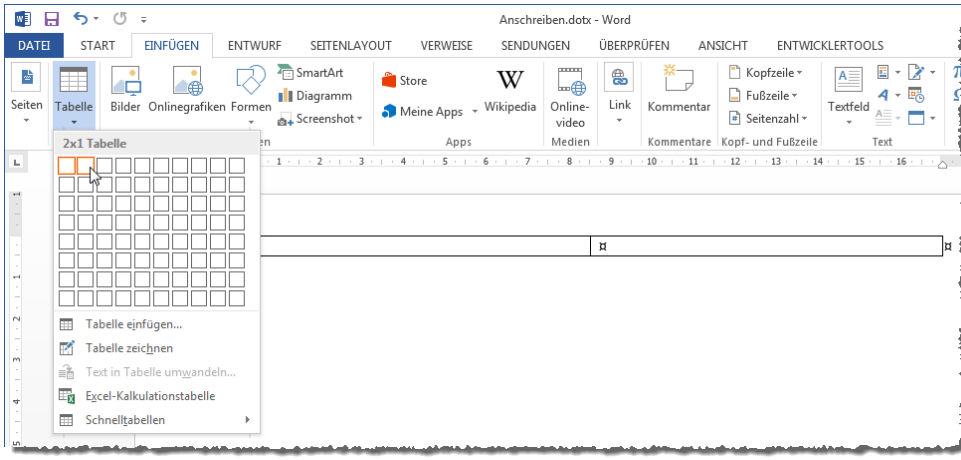


Abbildung 10.5: Hinzufügen einer Tabelle für den Briefkopf

Anschließend lassen wir den Rahmen der Tabelle verschwinden. Dazu klicken Sie mit der Maus auf das Fadenkreuz links über der Tabelle. Es erscheint nun ein Popup-Menü, das unter anderem die Möglichkeit zum Anpassen des Tabellenrahmens bietet.

Wählen Sie hier den Eintrag *Kein Rahmen* aus (siehe Abbildung 10.6).

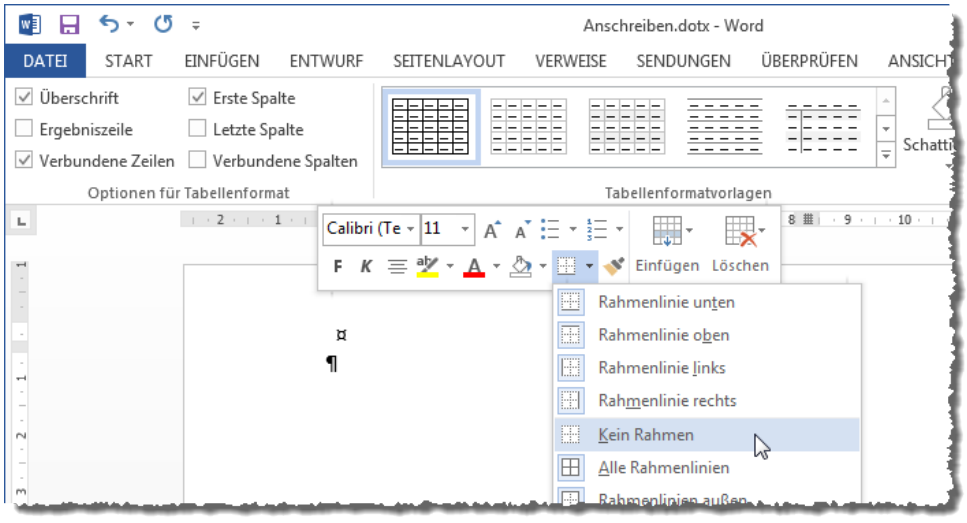


Abbildung 10.6: Entfernen des Tabellenrahmens

Anschließend blenden Sie das Tabellenraster ein, damit wir beim Entwurf des Dokuments die Rasterlinien der Tabelle sehen können. Außerdem fügen wir beispielsweise ein Logo und ein paar Adress-Informationen des Absenders zur oberen Tabelle hinzu (siehe Abbildung 10.7).

Das Logo fügen Sie am einfachsten ein, indem Sie es im Windows Explorer anzeigen und von dort an die gewünschte Stelle im Dokument ziehen.

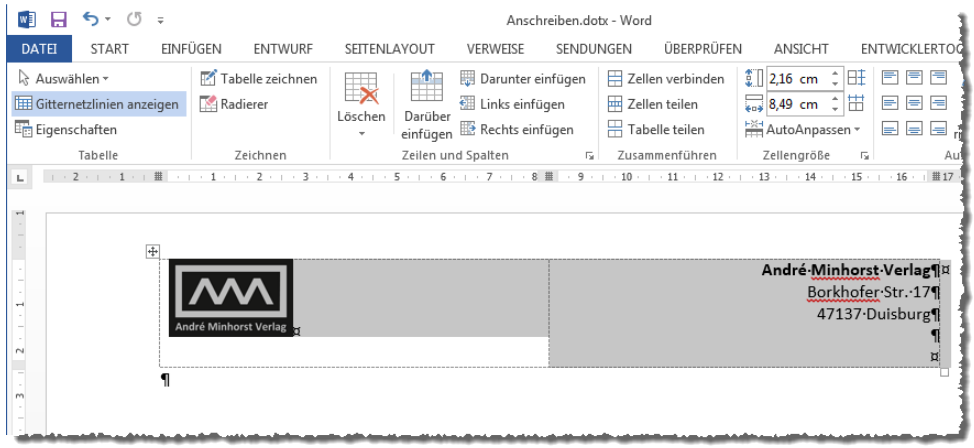


Abbildung 10.7: Einblenden des Tabellenrasters

10.1.3 Adressblock und Datum hinzufügen

Nun folgt der Adressblock. Heutzutage gibt es im geschäftlichen Geschäftsverkehr kaum noch Briefe, welche die Adresse auf dem Umschlag abbilden – meist besitzen die Umschläge ein Sichtfenster, sodass die auf dem Anschreiben selbst abgedruckte Adresse des Empfängers im Adressfenster erscheint.

Über der Adresse befindet sich eine Zeile mit der Absenderadresse, damit auch diese direkt auf das Anschreiben gedruckt werden kann. Diese Zeile können wir schon statisch im Dokument anbringen.

Dazu legen Sie eine weitere Tabelle an, die zweimal zwei Felder besitzt. Die Tabelle positionieren wir 5,2 cm vom oberen Seitenrand und 2,5 cm vom linken Seitenrand entfernt.

Bevor Sie die Tabelle einfügen, bringen Sie eine Leerzeile zwischen die obere Tabelle und die Zeile, in die Sie die weitere Tabelle einfügen möchten. Fügen Sie dann die Tabelle mit zwei mal zwei Feldern ein.

Nun müssen wir die Abstände einstellen. Um den Abstand vom linken Rand müssen wir uns wiederum nicht kümmern. Den Abstand von oben stellen wir mithilfe des Abstands nach oben für die Zeile zwischen den beiden Tabellen ein (Kontextmenü-Eintrag *Absatz*). Diesen vergrößern

Sie so lange, bis sich der obere Tabellenrand etwa auf Höhe der 5,25 cm-Markierung des linken Lineals befindet (siehe Abbildung 10.8).

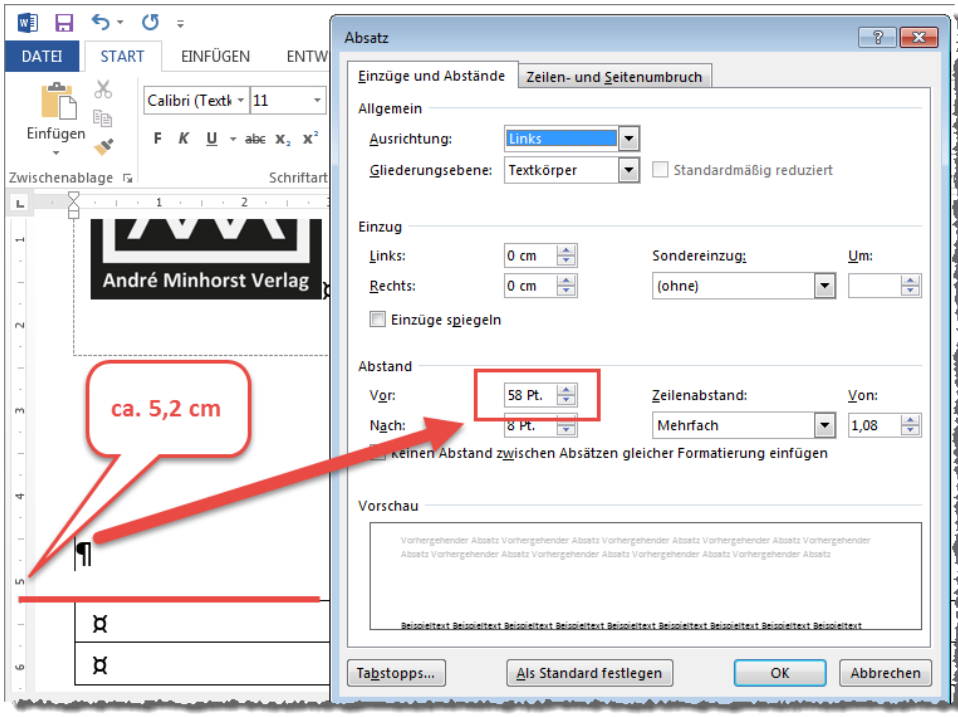


Abbildung 10.8: Einstellen des Randes

Danach entfernen Sie wieder die Rahmenlinien der Tabelle. Fügen Sie nun in kleiner Schriftart (8pt) die Absenderzeile hinzu. Sie können selbst entscheiden, ob Sie die Linie unter der Absenderadresse mit der Unterstreichen-Funktion hinzufügen möchten oder ob Sie die Linie zwischen der linken oberen Zelle und der linken unteren Zelle der Tabelle wieder aktivieren möchten. Die erste Variante liefert eine Linie, welche die unteren Teile einiger Buchstaben etwas schneidet (siehe Abbildung 10.9)

Und wenn ich mir einige Anschreiben auf meinem Schreibtisch ansehe, stelle ich fest, dass viele gar keine Linie zur Trennung von Absender- und Empfängeranschrift aufweisen.

10.1.4 Empfängeradresse

Nun fügen wir in die linke untere Tabellenzelle eine Textmarke für die Empfänger ein. Den entsprechenden Dialog öffnen Sie über den Ribbon-Befehl *Einfügen|Link|Textmarke*. Die neue Textmarke soll den Namen *Empfaengeradresse* erhalten (siehe Abbildung 10.10).

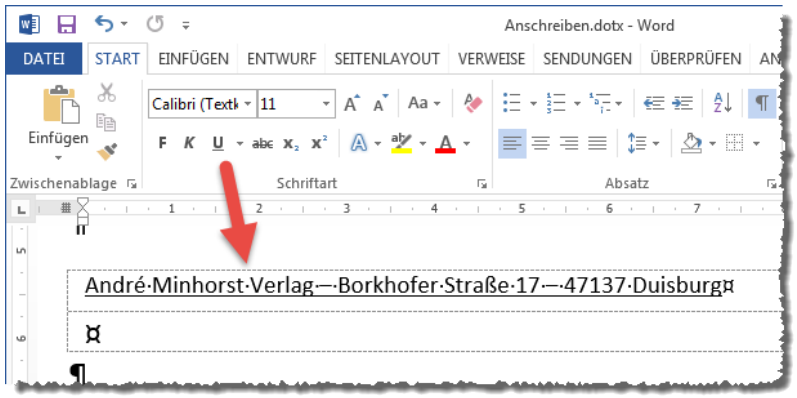


Abbildung 10.9: Unterstrichene Adresse

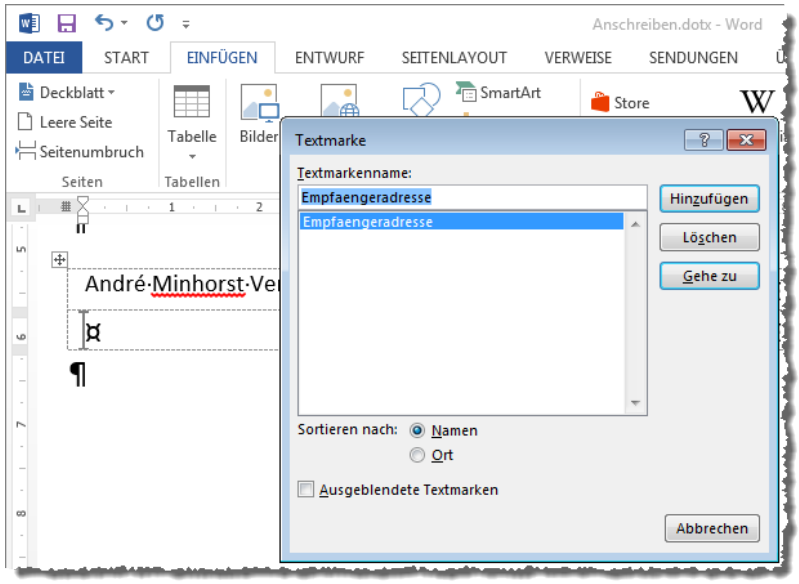


Abbildung 10.10: Hinzufügen einer Textmarke für die Empfängeradresse

Damit wir nun schon festlegen können, an welcher Höhe die folgende Zeile mit dem Betreff des Anschreibens beginnt, wollen wir die Höhe der Tabelle für die Absender- und Empfängeradresse auf ein Mindestmaß einstellen – genau genommen die der unteren Zeile der Tabelle.

Also klicken Sie in eine der beiden unteren Zellen und wählen Sie aus dem Kontextmenü den Eintrag *Tabelleneigenschaften* aus. Es erscheint der Dialog aus Abbildung 10.11, mit dem Sie unter anderem die Eigenschaft Höhe definieren festlegen können. Wir haben es so gemacht, dass

wir temporär einen fünfzeiligen Adressblock zu dieser Tabellenzelle hinzugefügt und dann die Höhe so eingestellt haben, dass auch noch Platz für eine sechste Zeile gewesen wäre.

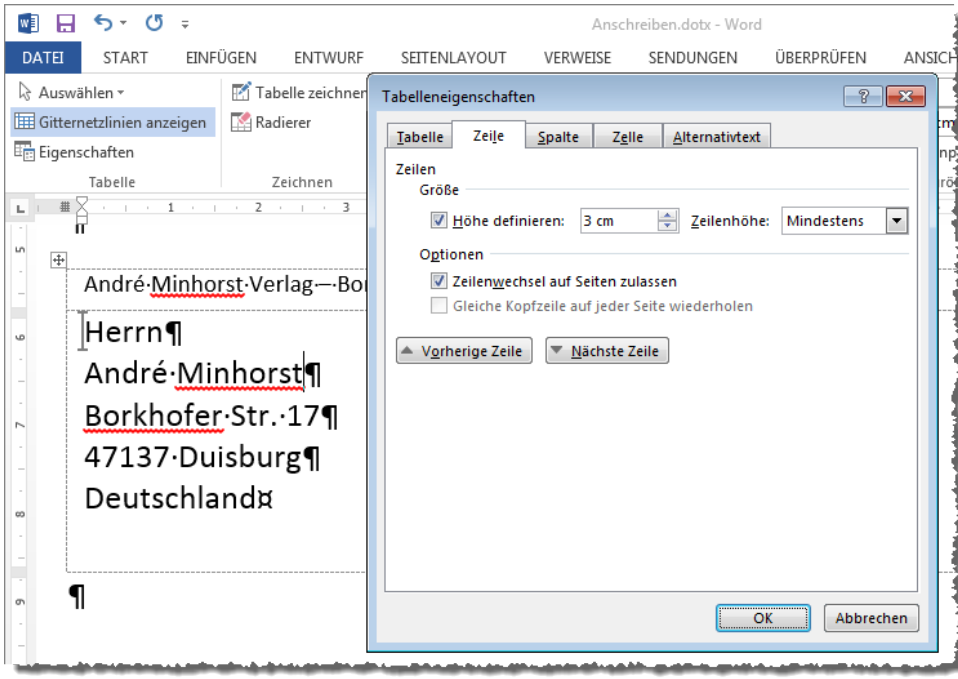


Abbildung 10.11: Höhe des Tabellenfeldes für die Empfängeradresse einstellen

Die temporäre Adresse können Sie nun wieder löschen und sich davon überzeugen, dass die eingestellte Höhe der Tabellenzelle erhalten bleibt. Nun fügen Sie in der rechten unteren Tabellenzelle eine weitere Textmarke ein, die Sie *OrtUndDatum* nennen. Diese soll nun erstens rechts und zweitens unten ausgerichtet werden, was wir etwa unter Word 2013 mit dem Ribbon-Befehl *Layout|Ausrichtung|Unten rechts ausrichten* (nur als Icon zu sehen) erledigen.

10.1.5 Textmarke für den Betreff

Damit kommen wir schon zur Betreffzeile. Gemäß dem Sinn von Eigenschaften wie *Abstand vor:* und *Abstand nach:* wollen wir die vertikale Position der Betreffzeile nicht durch stumpfsinniges Einfügen von Leerzeilen erhalten, sondern durch die Einstellung der Eigenschaft *Vor:* des Absatz-Dialogs (hier etwa auf *30pt*).

Der Abstand ist in Abbildung 10.12 durch die Markierung mit dem grauen Hintergrund des Zeileninhalts ebenso zu erkennen wie die Textmarke, die wir links neben dem temporär einge-

brachten und gleich wieder zu entfernenden Text *Betreff* eingefügt haben. Die Betreffzeile sollte etwa zu Beginn des zweiten Drittels des Anschreibens beginnen.

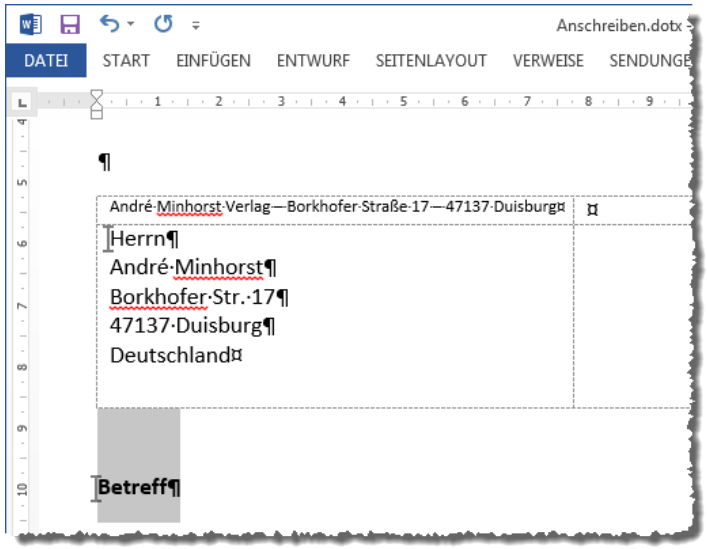


Abbildung 10.12: Einfügen und Anpassen der Betreffzeile

10.1.6 Textmarke für den Inhalt

Fehlt nur noch der Inhalt: Dieser soll gleich in der nächsten Zeile folgen. Wir fügen also eine weitere Zeile ein, die etwa einen Zentimeter Zwischenraum zur Betreffzeile lässt. Da Word immer die Einstellungen der aktuellen Zeile übernimmt, wenn Sie per Eingabetaste eine neue Zeile anlegen, sollten Sie die Abstandseinstellungen für den Abstand zwischen Betreffzeile und Inhalt des Anschreibens auf die Betreffzeile verschieben.

Außerdem fügen Sie der ersten Zeile des Inhalts eine Textmarke namens *Inhalt* hinzu.

10.1.7 Markierungslinien

Wenn wir schon einmal eine ordentliche Vorlage erstellen, wollen wir die fehlenden Elemente nicht vergessen. Als Erstes wollen wir Linien zum Falzen einbauen. Diese fügen wir aber nicht etwa ins Dokument selbst ein, sondern in die Kopfzeile.

Auf diese greifen Sie zu, indem Sie den Ribbon-Befehl *Einfügen|Kopfzeile|Kopfzeile bearbeiten* aufrufen (hier für Word 2013).

Nach dem Aktivieren des Bearbeitungsmodus fügen Sie eine Linie ein. Dazu wählen Sie den Ribbon-Eintrag *Einfügen|Formen|Linien* und wählen dort die einfache Linie aus.

Fügen Sie die Linie ohne Rücksicht auf die Position zum Kopfbereich hinzu, indem Sie bei gedrückter Umschalttaste einmal für den Startpunkt klicken, die Maus dann bei gedrückter Maustaste einen halben Zentimeter nach rechts bewegen und dann die Maustaste loslassen. Das Halten der Umschalttaste sorgt dafür, dass die Linie genau waagerecht angelegt wird.

Nun müssen Sie noch die Größe und die Position einstellen. Zuerst die Größe: Dazu klicken Sie auf das Symbol zum Öffnen der Layoutoptionen des *Linie*-Objekts.

Dies öffnet ein Popup-Element, in dem Sie unten auf den Text *Weitere anzeigen* klicken (siehe Abbildung 10.13).

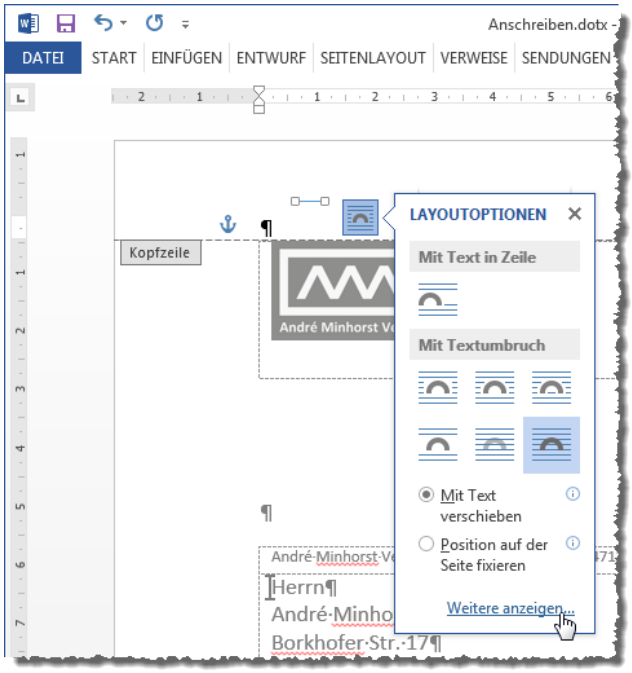


Abbildung 10.13: Aufrufen des Dialogs zum Einstellen von Größe und Position

Zum Anpassen der Länge auf 0,5 cm wechseln Sie zur Registerseite Größe und stellen dort den gewünschten Wert ein (siehe Abbildung 10.14).

Danach geht es mit der Registerseite *Position* weiter (siehe Abbildung 10.15). Hier legen Sie im Bereich *Horizontal* den Wert der Eigenschaft *rechts von* auf *Linker Rand* und dann *Absolute Position* auf *0,5 cm* fest. Dies in dieser Reihenfolge, da das Ändern des Bezugspunktes auch den Abstand ändern kann.

Im Bereich *Vertikal* verwenden Sie den Wert *Oberer Rand* für die Eigenschaft *unterhalb* und *10,5 cm* für *Absolute Position*.

11 Word-Rechnungen

Öfter mal taucht die Anforderung auf, Rechnungen nicht in einem Access-Bericht zu erfassen, sondern gleich ein Word-Dokument daraus zu erstellen und dort die Rechnungsdaten einzutragen. Das hat den Vorteil, dass der Benutzer noch manuell an der einen oder anderen Stelle eingreifen kann, wo es bei einem Access-Bericht nicht so einfach möglich ist – hier kann man, wenn überhaupt, nur an bereits dafür vorgesehenen Stellen Werte ändern oder aber man muss selbst in den Entwurf des Berichts eingreifen. Also zeigen wir in diesem Kapitel, wie Sie mit Access eine Rechnung als Word-Dokument erstellen und dieses speichern.

Dabei gibt es die eine oder andere Herausforderung, die auch unter Access die Rechnungserstellung zu einer interessanten Aufgabe machen – zum Beispiel bei Rechnungen, die nicht mehr auf eine Seite passen, die richtige Stelle für den Seitenumbruch zu finden und Zwischensumme und Übertrag entsprechend zu platzieren.

Leider kann man sich hier nicht einfach auf eine bestimmte Anzahl von Rechnungspositionen festlegen, die auf jede Seite passen – immerhin kann es ja auch einmal passieren, dass die Beschreibung einer Position mehrere Zeilen einnimmt und somit den Platz mehrerer Rechnungspositionen.

Schauen wir uns also einmal an, welche Möglichkeiten Word uns hier bietet.

Voraussetzung für die Änderung einer Rechnung sollte dabei jedoch sein, dass die in der Rechnung enthaltenen Zahlen nicht geändert werden dürfen. Dies würde keinen Sinn machen, da diese sonst von den in der Datenbank enthaltenen Zahlen abweichen würden – wenn Sie also etwa später die Rechnungseingänge mit dieser Datenbank prüfen wollten und die Rechnungsbeträge nur in den Word-Dokumenten geändert hätten, wäre das sicher kein Vergnügen.

11.1 Herausforderung

Die Herausforderung bei dieser Lösung besteht erstens daraus, die Informationen zur Rechnung in eine dafür geeignete Dokumentvorlage zu schreiben. Der schwierigste Teil ist die Erstellung einer Tabelle, welche die folgenden Bestandteile enthalten soll:

» die Spaltenüberschriften, also *Lfd. Nr.*, *Rechnungsposition*, *Einheit*, *Einzelpreis*, *Menge*, *Mwst.-Satz* und *Bruttopreis*,

» die einzelnen Rechnungspositionen mit den den Spaltenüberschriften entsprechenden Werten,

» gegebenenfalls eine Zwischensumme-Zeile, falls sich die Tabelle über zwei oder mehr Seiten erstreckt,

» desgleichen eine Übertrag-Zeile als erste Zeile der folgenden Seite der Rechnung,

» die Netto-Summe der Rechnungspositionen,

» die Anteile der Mehrwertsteuer an der Gesamtsumme, aufgeteilt auf die verschiedenen Steuersätze sowie

» die Bruttosumme der Rechnung.

Das alles wäre ja noch kein allzu großes Problem, wenn die Rechnungen immer nur über eine Seite gingen. Dann bräuchte man sich über Dinge wie Zwischensumme, Übertrag et cetera keine Sorgen machen.

Bei den aktuell verfügbaren Anwendungen zur Rechnungserstellung sollte man auch eigentlich auf solche Feinheiten wie Zwischensumme und Übertrag verzichten können, da die Anwendungen eigentlich zuverlässig arbeiten sollten. Aber da dürften dann die Anforderungen des einen oder anderen Kunden im Weg stehen, der auf solche Features besteht.

Aber selbst, wenn wir auf Zwischensumme und Übertrag verzichten würden, könnten wir nicht einfach eine Tabelle mit Spaltenüberschriften, Rechnungspositionen, Nettosumme, Mehrwertsteueranteilen und Bruttosumme erstellen und hoffen, dass die Seitenumbrüche uns gnädig gestimmt sein mögen. Es kann nämlich beispielsweise vorkommen, dass die Rechnung so viele Positionen enthält, dass gerade noch die Netto-Summe auf der ersten Seite erscheint, der Rest aber auf der folgenden Seite. Das sieht weder professionell aus noch ist es übersichtlich.

Wir wollen also folgende Dinge mit der nachfolgend beschriebenen Lösung erreichen:

» Wenn nicht alle Rechnungspositionen auf eine Seite passen, soll am Ende der ersten Seite die Zwischensumme und zu Beginn der Folgeseite der Übertrag ausgegeben werden.

» Der Block mit Nettosumme, Mehrwertsteueranteilen und Bruttosumme soll immer mit der letzten Rechnungsposition auf einer Seite landen.

Also, machen wir uns ans Werk!

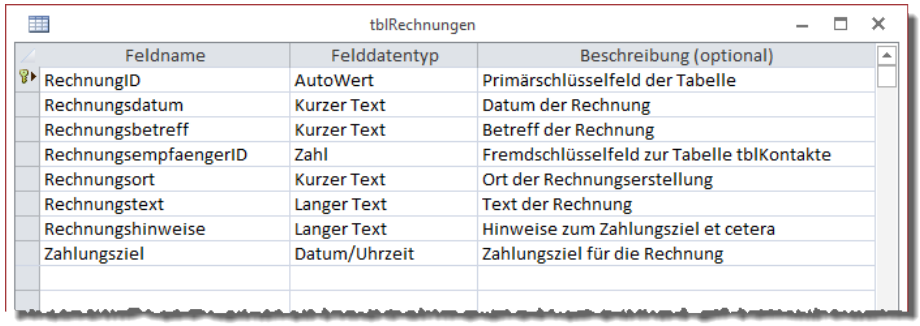
11.2 Beispieldatenbank

Die Datenbank dieses Beispiels heißt *Word.accdb*. Sie enthält einige Tabellen, ein Formular und ein Unterformular samt Klassenmodulen sowie einige Prozeduren in weiteren Modulen.

11.2.1 Rechnungsdaten speichern

Die erste Tabelle heißt *tblRechnungen* und nimmt die rechnungsbezogenen Daten wie Rechnungsempfänger, Rechnungsdatum, Zahlungsziel sowie einige Texte auf. Das Feld *Rechnungsbetreff* entspricht dem üblichen Betreff eines Anschreibens. *Rechnungstext* nimmt den Text auf, der

zwischen dem Anschreiben und der Auflistung der Positionen erscheinen soll. *Rechnungshinweis* enthält den Text, der unter der Auflistung der Positionen angezeigt werden soll. Hier bringen Sie beispielsweise Informationen wie das Zahlungsziel oder die Bankverbindung unter (siehe Abbildung 11.1).



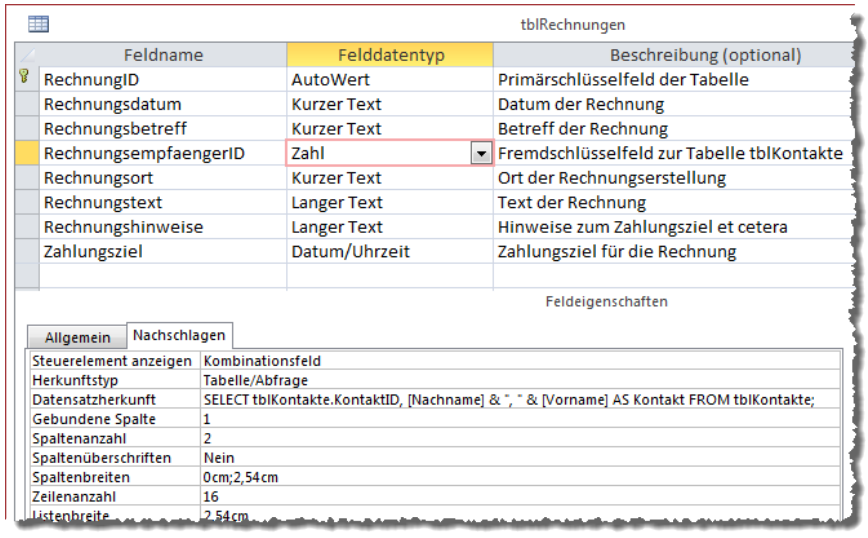
tblRechnungen

Feldname	Felddatentyp	Beschreibung (optional)
RechnungID	AutoWert	Primärschlüsselfeld der Tabelle
Rechnungsdatum	Kurzer Text	Datum der Rechnung
Rechnungsbetreff	Kurzer Text	Betreff der Rechnung
RechnungsempfaengerID	Zahl	Fremdschlüsselfeld zur Tabelle tblKontakte
Rechnungsort	Kurzer Text	Ort der Rechnungserstellung
Rechnungstext	Langer Text	Text der Rechnung
Rechnungshinweise	Langer Text	Hinweise zum Zahlungsziel et cetera
Zahlungsziel	Datum/Uhrzeit	Zahlungsziel für die Rechnung

Abbildung 11.1: Tabelle zum Speichern der Rechnungsdaten

11.2.2 Rechnungsempfänger speichern

Die Tabelle *tblRechnungen* hat ein Fremdschlüsselfeld namens *RechnungsempfaengerID*, welches mit dem Primärschlüsselfeld *KontaktID* der Tabelle *tblKontakte* verknüpft ist. Diese finden Sie in Abbildung 11.2. Die Tabelle *tblKontakte* enthält ihrerseits ein Fremdschlüsselfeld namens *AnredeID*, mit den zu jedem Kontakt ein Eintrag der Tabelle *tblAnreden* ausgewählt werden soll.



tblRechnungen

Feldname	Felddatentyp	Beschreibung (optional)
RechnungID	AutoWert	Primärschlüsselfeld der Tabelle
Rechnungsdatum	Kurzer Text	Datum der Rechnung
Rechnungsbetreff	Kurzer Text	Betreff der Rechnung
RechnungsempfaengerID	Zahl	Fremdschlüsselfeld zur Tabelle tblKontakte
Rechnungsort	Kurzer Text	Ort der Rechnungserstellung
Rechnungstext	Langer Text	Text der Rechnung
Rechnungshinweise	Langer Text	Hinweise zum Zahlungsziel et cetera
Zahlungsziel	Datum/Uhrzeit	Zahlungsziel für die Rechnung

Feldeigenschaften

Allgemein | Nachschlagen

Steuerelement anzeigen	Kombinationsfeld
Herkunftstyp	Tabelle/Abfrage
Datensatzherkunft	SELECT tblKontakte.KontaktID, [Nachname] & ", " & [Vorname] AS Kontakt FROM tblKontakte;
Gebundene Spalte	1
Spaltenanzahl	2
Spaltenüberschriften	Nein
Spaltenbreiten	0cm;2,54cm
Zeilenanzahl	16
Listenbreite	2,54cm

Abbildung 11.2: Kontaktdaten des Rechnungsempfängers

Dieses Fremdschlüsselfeld legen wir als Nachschlagefeld aus, wobei wir das anzuzeigende Feld aus dem Feld *Nachname*, einem Komma und dem Feld *Vorname* der Tabelle *tblKontakte* zusammensetzen:

```
SELECT tblKontakte.KontaktID, [Nachname] & ", " & [Vorname] AS Kontakt FROM tblKontakte
ORDER BY [Nachname] & ", " & [Vorname];
```

11.2.3 Anreden speichern

Die Tabelle *tblAnreden* liefert eine einfache Anrede sowie eine förmliche Anrede für die Grußformel und eine für das Adressfeld (siehe Abbildung 11.3).

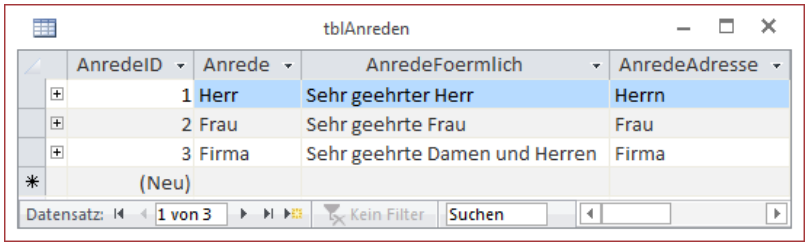


Abbildung 11.3: Die Tabelle mit den verschiedenen Anreden

11.2.4 Rechnungspositionen speichern

Mit der Tabelle *tblRechnungen* verknüpft ist die Tabelle *tblRechnungspositionen*. Diese nimmt eine oder mehrere Positionen je Rechnung auf. Um das Beispiel nicht zu kompliziert zu machen, beziehen wir die Rechnungspositionen nicht noch aus einer Artikeltabelle, sondern gehen davon aus, dass es sich um individuelle Leistungen handelt. Die Tabelle *tblRechnungspositionen* enthält ein Fremdschlüsselfeld namens *RechnungID*, mit dem wir die Rechnungsposition einem der Einträge der Tabelle *tblRechnungen* zuweisen können (siehe Abbildung 11.4).

Des Weiteren finden Sie dort zwei weitere Fremdschlüsselfelder, die zur Auswahl von Daten aus Lookuptabellen dienen: *MehrwertsteuersatzID* und *EinheitID*.

11.2.5 Mehrwertsteuersätze speichern

Die Tabelle *tblMehrwertsteuersaetze* speichert neben dem Primärschlüsselwert lediglich noch den *Mehrwertsteuersatz*. Hier gibt es eine Besonderheit: Als Datentyp für dieses Feld haben wir *Währung* festgelegt.

Währungsfelder speichern Festkommazahlen und erlauben somit die für Finanzanwendungen notwendige Genauigkeit. Da wir aber ein Prozentfeld brauchen, stellen wir einfach das Formular auf Prozentzahl ein (siehe Abbildung 11.5).

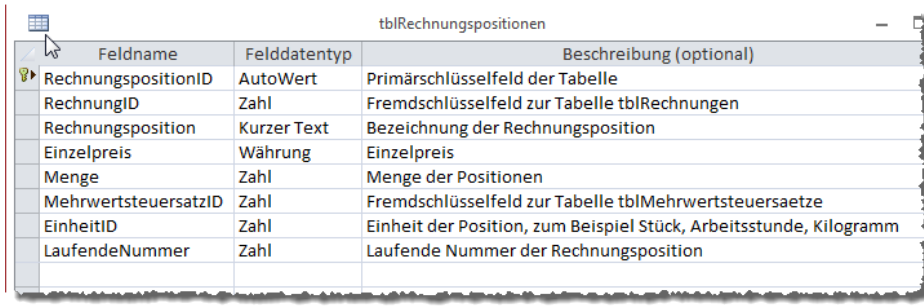


Abbildung 11.4: Die Tabelle *tblRechnungspositionen*

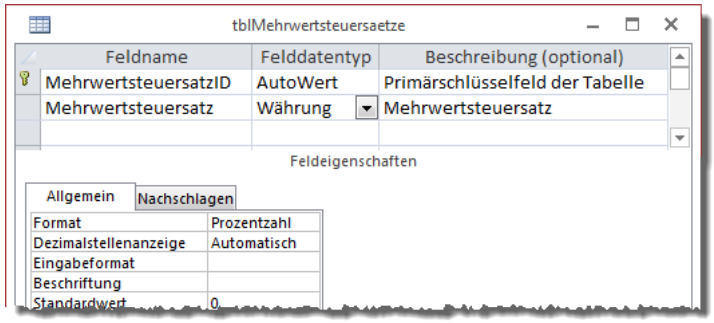


Abbildung 11.5: Die Tabelle zum Speichern der Mehrwertsteuersätze

11.2.6 Einheiten speichern

Fehlen noch die Einheiten für die Rechnungspositionen. Diese speichern wir in einer einfachen Lookup-Tabelle namens *tblEinheiten*, die gefüllt in der Datenblattansicht wie in Abbildung 11.6 aussieht. Dieser Tabelle können Sie nach Belieben weitere Einheiten hinzufügen.

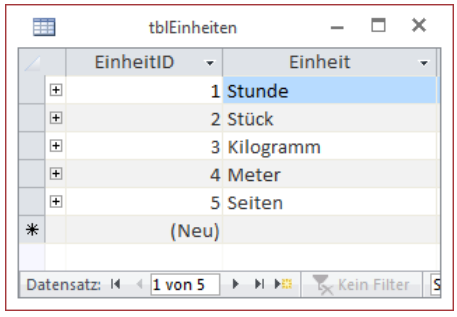


Abbildung 11.6: Einheiten für die Rechnungspositionen

11.3 Formulare der Rechnungserstellung

Wir verwenden zwei Formulare, von denen eines die Rechnungsdaten anzeigt. Das zweite liefert die Rechnungspositionen zum jeweils angezeigten Rechnungsdatensatz und erscheint als Unterformular im ersten Formular.

11.3.1 Unterformular zur Anzeige der Rechnungspositionen

Das Unterformular *sfmRechnungen* verwendet die Abfrage *qryRechnungspositionen* als Datenherkunft. Diese liefert alle Felder der Tabelle *tblRechnungspositionen* und ein weiteres, berechnetes Feld (siehe Abbildung 11.7).

Dieses dient der Sortierung der Datensätze im Formular. Warum benötigen wir dazu ein berechnetes Feld? Weil das Feld *LaufendeNummer* nicht automatisch gefüllt wird, sondern erst nach dem Speichern des jeweiligen Datensatzes.

Damit ein Datensatz, dessen Feld *LaufendeNummer* leer ist, dennoch als neuester Rechnungsposition ganz unten im Unterformular erscheint, fügen wir das berechnete Feld *LaufendeNummerNichtLeer* hinzu, das entweder den Wert des Feldes *LaufendeNummer* liefert oder, wenn dieses leer ist, den Wert *999* (wir gehen an dieser Stelle davon aus, dass eine Rechnung nicht mehr als 999 Positionen aufweist ...).

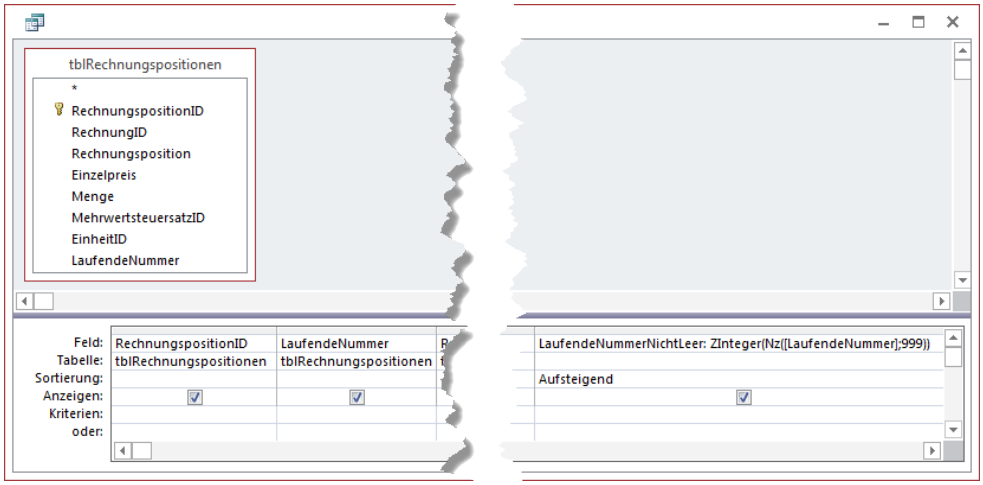


Abbildung 11.7: Datenherkunft des Unterformulars *sfmRechnungen*

Nachdem Sie die Datenherkunft erstellt haben, fügen Sie dem Unterformular die Felder *LaufendeNummer*, *Rechnungsposition*, *EinheitID*, *Menge* und *MehrwertsteuersatzID* hinzu (siehe Abbildung 11.8). Außerdem stellen Sie die Eigenschaft *Standardansicht* auf *Datenblatt* ein.

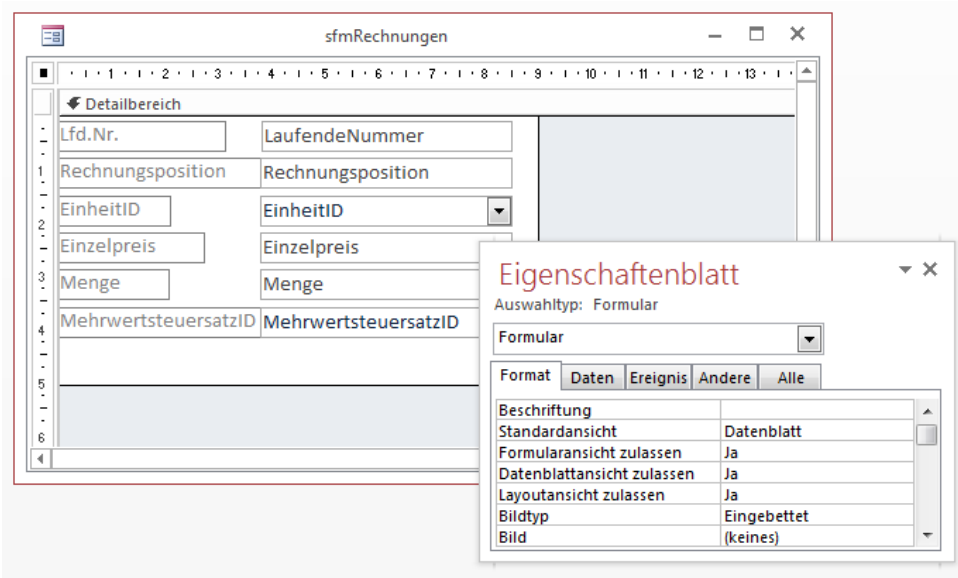


Abbildung 11.8: Das Unterformular *sfmRechnungen* in der Entwurfsansicht

11.3.2 Hauptformular zur Anzeige der Rechnungen

Das Hauptformular heißt *frmBestellungen* und verwendet schlicht und einfach die Tabelle *tbl-Rechnungen* als Datenherkunft.

Im Formular ordnen Sie nacheinander die Felder *RechnungID*, *Rechnungsdatum*, *Zahlungsziel*, *Rechnungsort*, *RechnungsempfaengerID*, *Rechnungsbetreff* und *Rechnungstext* an (siehe Abbildung 11.9).

Dadurch, dass wir weiter oben das Feld *RechnungsempfaengerID* als Nachschlagefeld ausgelegt haben, fügt Access dieses gleich als Kombinationsfeld mit den entsprechenden Einstellungen hinzu.

Darunter fügen Sie das Unterformular *sfmRechnungspositionen* ein. Wenn alles richtig läuft, erkennt Access, dass die Datenherkunft des Unterformulars ein Fremdschlüsselfeld enthält, das mit dem Primärschlüsselfeld der Datenherkunft des Hauptformulars verknüpft ist und stellt die beiden Eigenschaften *Verknüpfen von* und *Verknüpfen nach* jeweils auf den Wert *RechnungID* ein.

Dies prüfen Sie in den Eigenschaften für das Unterformular-Steuerelement (nicht für das Unterformular selbst!), die wie in Abbildung 11.10 aussehen sollten.

Unter dem Unterformular fügen Sie noch das Textfeld *Rechnungshinweise* hinzu sowie eine Schaltfläche zum Erstellen der Rechnung.

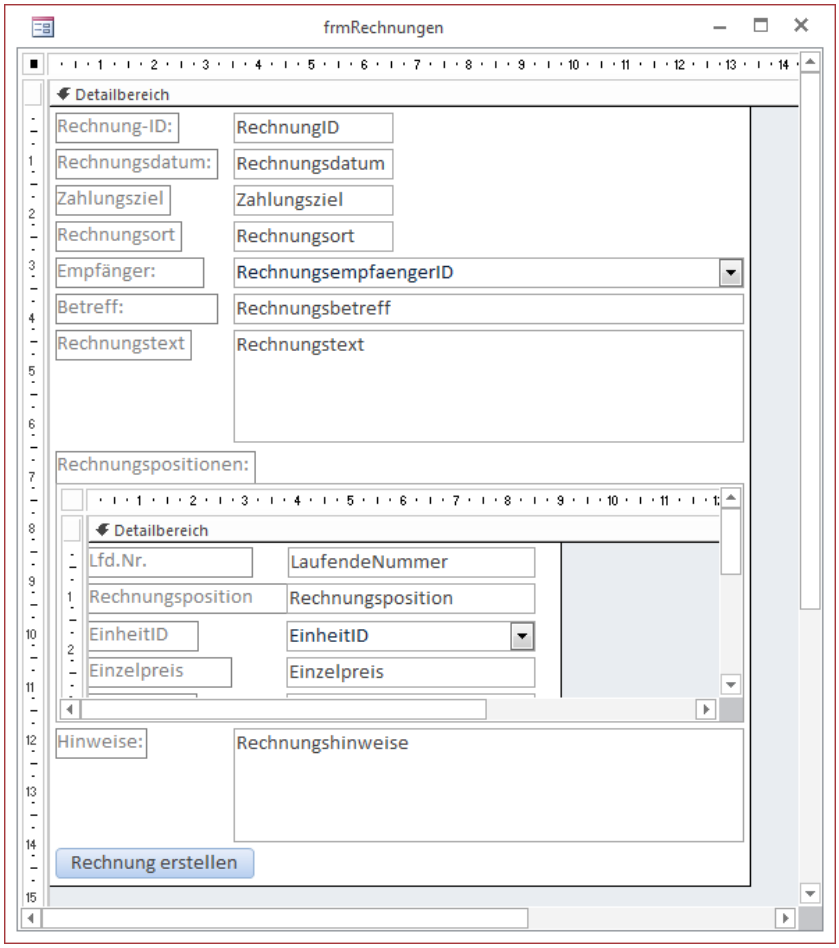


Abbildung 11.9: Das Hauptformular *frmRechnungen* in der Formularansicht

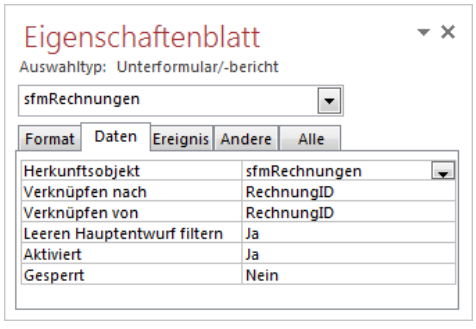


Abbildung 11.10: Einstellungen für die Verknüpfung zwischen Haupt- und Unterformular

11.4 Neue Rechnungen und Rechnungspositionen hinzufügen

Wenn der Benutzer eine neue Rechnung zum Hauptformular hinzufügt, bleibt das Unterformular zunächst leer. Logisch, es gibt ja noch keine zu der neuen Rechnung gehörenden Datensätze in der Tabelle *tblRechnungspositionen*.

Nun fügt der Benutzer die erste Rechnungsposition hinzu. Das hier nicht sichtbare, da nicht zum Formularentwurf hinzugefügte Feld *RechnungID* des Unterformulars erhält automatisch den Wert des Feldes *RechnungID* des Hauptformulars als Standardwert.

Allerdings bleibt das Feld *LaufendeNummer* zunächst leer (siehe Abbildung 11.11). Das ist durchaus beabsichtigt, da diese erst nach dem Speichern des Datensatzes automatisch hinzugefügt wird.

Dafür sind zwei Ereignisprozeduren des Unterformulars verantwortlich, die durch die Ereignisse *Nach Löschbestätigung* und *Nach Aktualisierung* ausgelöst werden:

```
Private Sub Form_AfterDelConfirm(Status As Integer)
    LaufendeNummerEinstellen
End Sub

Private Sub Form_AfterUpdate()
    LaufendeNummerEinstellen
End Sub
```

Um Missverständnissen vorzubeugen, zeigt Abbildung 11.12, wie Sie die beiden Prozeduren für die beiden Ereigniseigenschaften des Unterformulars *sfmRechnungen* hinterlegen. Beide Ereignisprozeduren rufen die folgende Prozedur auf:

```
Private Sub LaufendeNummerEinstellen()
    Dim db As DAO.Database
    Dim rst As DAO.Recordset
    Set db = CurrentDb
    Set rst = db.OpenRecordset("SELECT * FROM qryRechnungspositionen " _
        & "WHERE RechnungID = " & Me.Parent!RechnungID, dbOpenDynaset)
    Do While Not rst.EOF
        rst.Edit
        rst!LaufendeNummer = rst.AbsolutePosition + 1
        rst.Update
        rst.MoveNext
    Loop
End Sub
```

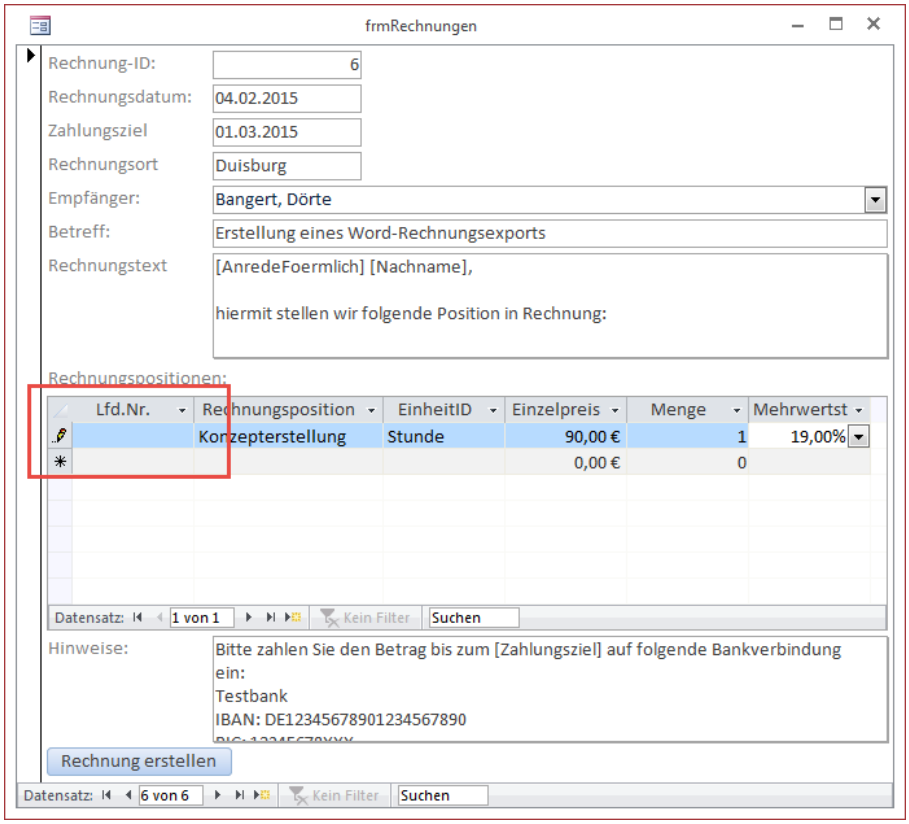


Abbildung 11.11: Anlegen einer ersten Rechnungsposition

Diese durchläuft alle Datensätze der auch dem Unterformular zugrunde liegenden Abfrage *qry-Rechnungspositionen* – und zwar gefiltert nach der Rechnung, die aktuell im Hauptformular angezeigt wird.

Die Datensätze dieser Abfrage sind ja, wie weiter oben bereits besprochen, nach dem berechneten Feld *LaufendeNummerOhneNull* sortiert, wobei ein neuer Datensatz mit noch leerem Feld *LaufendeNummer* ganz hinten landet, weil das Feld *LaufendeNummerOhneNull* diesem zu Sortierzwecken den Wert *999* zuweist.

Die Prozedur durchläuft die Datensätze dann in dieser Reihenfolge und versetzt den jeweils aktuellen Datensatz mit der *Edit*-Methode in den Bearbeitungsmodus. Dann schreibt sie einen neuen Wert in das Feld *LaufendeNummer*, welcher der Eigenschaft *AbsolutePosition* des aktuellen Datensatzes plus *1* entspricht.

Danach speichert die Prozedur die Änderung und ändert die Werte des Feldes *LaufendeNummer* für die übrigen Datensätze.

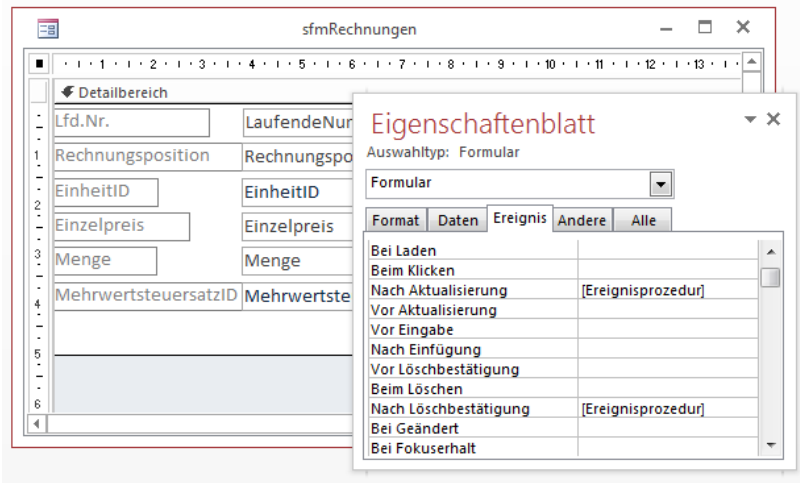


Abbildung 11.12: Anlegen der Ereignisprozeduren für das Unterformular *sfmBestellungen*

11.5 Word-Rechnung erstellen

Damit kommen wir zum aufwendigsten Teil dieser kleinen Lösung. Wir wollen die Daten, die der Benutzer wie in Abbildung 11.13 in das Formular eingegeben hat, in ein Word-Dokument übertragen.

In der Abbildung ist ein weiteres Feature zu erkennen, auf das wir später bei der Programmierung kommen werden: Sie können für die Felder *Rechnungstext* und *Rechnungshinweise* Platzhalter einfügen, die vor dem Übertragen in das Word-Dokument durch die entsprechenden Feldinhalte des aktuellen Datensatzes der Abfrage *qryRechnungen* ersetzt werden.

Die einzige Voraussetzung für die verwendeten Platzhalter ist, dass diese in der Abfrage *qryRechnungenMitKontakt* aufgeführt sind – dazu später mehr.

11.5.1 Word-Vorlage erstellen

Für das Erstellen der Rechnung verwenden wir eine Vorlage, die eng an die in *****Word-Vorlage erstellen* beschriebene Vorlage angelehnt ist. Abbildung 11.14 zeigt die Vorlage, wobei wir hinter die Textmarken die Namen der Textmarken eingetragen haben. Diese Texte sind in der Originalvorlage natürlich nicht vorhanden, da diese nicht automatisch durch Befüllen der Textmarken entfernt werden.

Gegenüber der im oben angegebenen Abschnitt beschriebenen Dokumentvorlage verwendet diese nur zwei Textmarken mehr, und zwar für die Rechnungspositionen und für die Rechnungshinweise (*Hinweise*).

12 Word-Seriendruck per VBA

Der Seriendruck unter Word ist wohl das Paradebeispiel für die Interaktion zwischen Access und Word (oder auch Excel und Word). Sowohl Access als auch Excel dienen als Datenlieferanten für einen mit Word zu erstellenden Seriendruck. Dieses Kapitel wird eine Variante des Seriendrucks mit Word von Access aus vorstellen, die für den Benutzer fast noch einfacher als die in Word eingebaute Funktion ist. Sie benötigen nur ein einziges Formular für die komplette Steuerung beim Erstellen des Seriendruck-Dokuments und später auch für die Erstellung des Seriendrucks.

Bei dieser Gelegenheit gleich zwei Definitionen: Das Dokument, das wir mit den Platzhaltern beziehungsweise Feldern ausstatten, die später mit den Daten aus der Datenbank gefüllt werden sollen, soll Seriendruck-Dokument heißen. Das Dokument mit den fertigen Dokumenten/Briefen heißt dann Serienbrief.

12.1 Datenquelle auswählen

Den oberen Bereich unseres Formulars namens *frmSeriendruck*, das Sie in der Beispieldatenbank finden, enthält ein Kombinationsfeld zur Auswahl der Tabelle oder Abfrage, welche die Daten für den Seriendruck liefern soll. Das Kombinationsfeld heißt *cboDatenquelle* und verwendet die Abfrage aus Abbildung 12.1 als Datensatzherkunft.

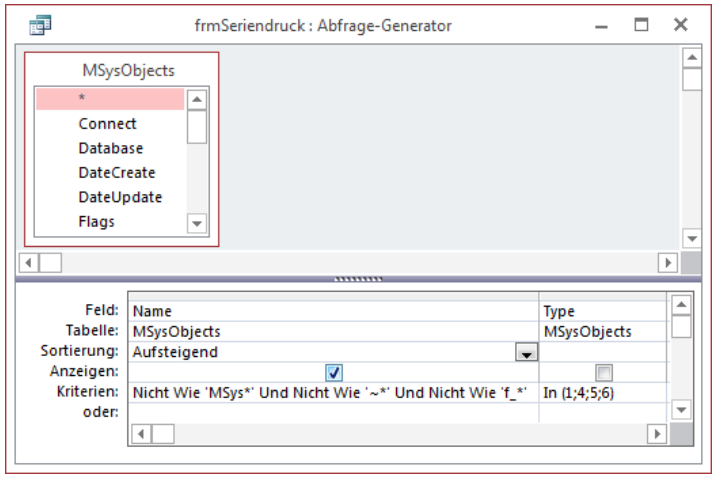


Abbildung 12.1: Datenherkunft des Kombinationsfeldes *cboDatenquelle*

Damit Sie die dort verwendete Tabelle *MSysObjects*, eine Systemtabelle von Access, überhaupt einfügen können, müssen Sie zuvor gegebenenfalls noch die Anzeige von Systemtabellen akti-

vieren. Dazu klicken Sie mit der rechten Maustaste auf den Titel des Navigationsbereichs und wählen aus dem dann erscheinenden Kontextmenü den Eintrag *Navigationsoptionen...* aus.

Es erscheint der Dialog *Navigationsoptionen* aus Abbildung 12.2. Hier aktivieren Sie die Option *Systemobjekte anzeigen*. Anschließend erscheinen im Navigationsbereich einige ausgegraute Einträge, darunter auch die Tabelle *MSysObjects*.

Diese enthält die Namen und weitere Informationen zu allen in der Datenbank enthaltenen Objekten.

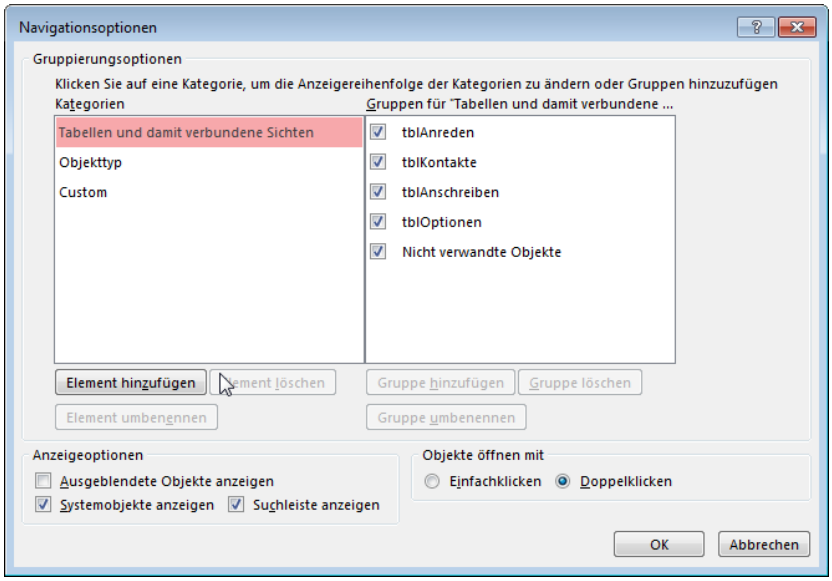


Abbildung 12.2: Aktivieren der Anzeige von Systemtabellen in Access

Das Feld *Type* dieser Tabelle gibt Auskunft über die Objektart. Wir wollen die folgenden Werte einbeziehen:

» *1*: Lokale Access-Tabellen

» *4*: ODBC-verknüpfte Tabellen (zum Beispiel MySQL oder SQL Server)

» *5*: Abfragen

» *6*: Jet-Verknüpfungen (zum Beispiel verknüpfte Access-Tabellen aus anderen Datenbanken)

Wir wollen auch einige Objekte weglassen – zum Beispiel die soeben eingeblendeten Systemtabellen, die alle mit *MSys* beginnen und somit leicht herauszufiltern sind.

Außerdem wollen wir die für Kombinationsfelder et cetera hinterlegten Abfragen, die nicht als eigene Objekte gespeichert sind, nicht anzeigen. Diese beginnen mit dem Tilde-Zeichen (~).

Schließlich gibt es noch Einträge, die mit *f_* beginnen – dies sind die Tabellen, die sich hinter den Anlage-Feldern verbergen.

Unsere Datensatzherkunft liefert ihre Daten dann etwa wie in Abbildung 12.3.

Abbildung 12.3: Kombinationsfeld zur Auswahl der Daten

12.2 Datensätze auswählen

Der zweite Schritt besteht darin, die Daten der gewählten Datenherkunft anzuzeigen und dem Benutzer die Möglichkeit zu bieten, daraus einen oder mehrere Datensätze für die Erstellung des Serienbriefs zu selektieren.

Um die Datensätze auszuwählen, müssen wir eine Möglichkeit wie etwa ein Kontrollkästchen zum Anhaken der zu berücksichtigen Datensätze schaffen. Aber können wir einfach ein entsprechendes Ja/Nein-Feld zu der Tabelle oder Abfrage hinzufügen, welche die Daten für den Serienbrief liefern soll? Nein, das wäre ein zu tiefer Eingriff in das Datenmodell, nur um die Serienbrief-Empfänger festzulegen.

Hinzu kommt ein zweiter Aspekt, nach dem die Serienbrief-Funktion von Word die einzufügenden Daten mitunter in merkwürdigen Formatierungen liefert. Es wäre also am besten, wenn wir die Datenquelle direkt durchgängig in Textform liefern statt als Zahlen, Datumsangaben oder Ja/Nein-Felder. Die logische Folge dieser zwei Forderungen ist, dass wir eine neue Tabelle auf Basis der zu übermittelnden Daten erstellen, welche erstens neben den Feldern der eigentlichen Datenquelle noch ein Ja/Nein-Feld zur Auswahl der zu berücksichtigenden Datensätze enthält und zweitens alle Daten in Textfeldern speichert.

Diese Tabelle müssen wir natürlich, da ja gegebenenfalls verschiedene Tabellen oder Abfragen als Datenquelle dienen sollen, jeweils dynamisch neu erstellen und mit den Daten der Tabelle oder Abfrage füllen.

Zu diesem Zweck legen wir eine eigene Prozedur namens *TempTabelleErstellen*, welche mit dem Parameter *strDatenquelle* den Namen der Tabelle oder Abfrage entgegennimmt, aus der die Daten stammen:

```
Public Sub TempTabelleErstellen(strDatenquelle As String)
    Dim tdf As DAO.TableDef
    Dim fld As DAO.Field
    Dim fldDatenquelle As DAO.Field
    Dim prp As DAO.Property
    Dim db As DAO.Database
    Dim rstDatenquelle As DAO.Recordset
    Dim strFelder As String
```

Die Prozedur füllt die Variable *db* mit einem Verweis auf das *Database*-Objekt der aktuellen Datenbank und löscht dann eine eventuell noch vorhandene Tabelle namens *tblSerienbrief_Temp*:

```
    Set db = CurrentDb
    On Error Resume Next
    db.Execute "DROP TABLE tblSerienbrief_Temp", dbFailOnError
    On Error GoTo 0
```

Dann erstellt sie die Tabelle *tblSerienbrief_Temp* neu und referenziert diese noch feldlose Tabelle mit der Varialben *tdf*:

```
    Set tdf = db.CreateTableDef("tblSerienbrief_Temp")
```

Danach liest sie ein leeres Recordset auf Basis der angegebenen Tabelle oder Abfrage in das Recordset-Objekt *rstDatenquelle* ein. Dieses benötigen wir nur, um die Felder zu durchlaufen und diese zum *TableDef*-Objekt *tdf* hinzuzufügen.

Warum verwenden wir dafür nicht einfach das *TableDef*-Objekt der Quelltabelle? Weil es sich ja auch um eine Abfrage handeln kann und wir dann das *QueryDef*-Objekt verwenden müssten – was eine vorherige Untersuchung des Typs des Quellobjekts erfordern würde.

Dies sparen wir uns und arbeiten direkt mit einem Recordset, welches sowohl Tabellen als auch Abfragen aufnimmt:

```
    Set rstDatenquelle = db.OpenRecordset("SELECT * FROM " & strDatenquelle _
        & " WHERE 1 = 2")
```

Danach durchläuft die Prozedur alle *Field*-Elemente des Recordsets, also alle Felder der Datenquelle, in einer *For Each*-Schleife. Dabei prüft sie zunächst, ob es sich bei dem Feld um ein Anlage-Feld oder ein mehrwertiges Feld handelt, denn diese Feldtypen werden beim Seriendruck nicht unterstützt:

```
    For Each fldDatenquelle In rstDatenquelle.Fields
        Select Case fldDatenquelle.type
            Case dbAttachment, dbComplexByte, dbComplexDecimal, dbComplexDouble, _
                dbComplexGUID, dbComplexInteger, dbComplexLong, dbComplexSingle, _
```

```
            dbComplexText
```

Für alle übrigen Felddatentypen soll ein Feld in der Tabelle *tblSerienbrief_Temp* angelegt werden. Dazu erstellt die Prozedur zunächst ein neues Feld mit der *CreateField*-Methode und übergibt dabei den Feldnamen und den Datentyp *dbText* als Parameter.

Sollte der Felddatentyp der Quelltabelle ebenfalls *dbText* sein, stellt die Prozedur noch die Eigenschaft *AllowZeroLength* auf den Wert *True* ein – somit darf das neue Feld auch leere Zeichenketten aufnehmen. Schließlich hängt die Prozedur das neue *Field*-Objekt an die *Fields*-Auflistung des *TableDef*-Objekts an:

```
        Case Else
            Set fld = tdf.CreateField(fldDatenquelle.Name, dbText)
            If fldDatenquelle.type = dbText Then
                fld.AllowZeroLength = True
            End If
            tdf.Fields.Append fld
            strFelder = strFelder & ", " & fldDatenquelle.Name
    End Select
Next fldDatenquelle
```

Außerdem stellt die Prozedur in der Schleife noch eine Liste der Feldnamen in der Variablen *strFelder* zusammen.

Danach fügt die Prozedur noch das *Boolean*-Feld *Serienbrief* zur Tabelle hinzu. Dieses dient später dem Selektieren der Datensätze für den Seriendruck:

```
Set fld = tdf.CreateField("Serienbrief", dbBoolean)
tdf.Fields.Append fld
```

Nun hängt die Prozedur noch das neue *TableDef*-Objekt an die *TableDefs*-Auflistung der Datenbank an:

```
db.TableDefs.Append tdf
```

Außerdem wird der Wert des *Boolean*-Feldes *Serienbrief* aktuell noch als Text angezeigt, also als *0* (*False*) oder *-1* (*True*). Dies ändern wir, indem wir die Eigenschaft *DisplayControl* zum Feld hinzufügen und dessen Wert auf *dbCheckBox* einstellen:

```
Set fld = tdf.Fields("Serienbrief")
Set prp = fld.CreateProperty("DisplayControl", dbInteger, acCheckBox, True)
```

Vorsichtshalber entfernen wir eine gleichnamige Eigenschaft, bevor wir diese an die Properties-Auflistung anhängen:

```
On Error Resume Next
fld.Properties.Delete "DisplayControl"
```

```
    On Error GoTo 0
    fld.Properties.Append prp
```

Das war es fast. Fehlen nur noch die Daten aus der Quelltabelle. Diese fügt die Prozedur mit einer geeigneten *INSERT INTO*-Anweisung zur Tabelle *tblSerienbrief_Temp* hinzu, wobei die *SELECT*-Anweisung zum Festlegen der zu berücksichtigenden Felder der Quelltabelle aus der Variablen *strFelder* entnommen wird, die weiter oben in einer Schleife gefüllt wurde und die jetzt noch um das führende Komma erleichtert wird:

```
    strFelder = Mid(strFelder, 3)
    db.Execute "INSERT INTO tblSerienbrief_Temp SELECT " & strFelder & " FROM " _
        & strDatenquelle, dbFailOnError
```

Die letzte Anweisung stellt nun noch den Wert des Feldes *Serienbrief* für alle Datensätze auf den Wert *True* ein:

```
    db.Execute "UPDATE tblSerienbrief_Temp SET Serienbrief = TRUE", dbFailOnError
End Sub
```

Als Beispiel wollen wir eine Abfrage verwenden, welche die beiden Tabellen *tblKontakte* und *tblAnreden* zusammenführt und zusätzlich zu den wichtigsten Feldern der Tabelle *tblKontakt* noch die drei Felder *Anrede*, *AnredeFoermlich* und *AnredeAdresse* der Tabelle *Anrede* hinzufügt. Wenn wir den Namen dieser Abfrage als Parameter an die Prozedur *TempTabelleErstellen* übergeben, erstellt diese die Tabelle aus Abbildung 12.4.

KontaktID	Vorname	Nachname	Anrede	AnredeFoermlich	AnredeAdresse	Serienbrief
3172	Betti	Schrade	Frau	Sehr geehrte Frau	Frau	☑
3173	Louis	Hopp	Herr	Sehr geehrter Herr	Herrn	☑
3174	Ireneus	Bühler	Herr	Sehr geehrter Herr	Herrn	☑
3175	Renilde	Lambrecht	Frau	Sehr geehrte Frau	Frau	☑
3176	Dorota	Nau	Frau	Sehr geehrte Frau	Frau	☑
3177	Irena	Kaya	Frau	Sehr geehrte Frau	Frau	☑
3178	Lindhilde	Reiner	Frau	Sehr geehrte Frau	Frau	☑
3179	Gottreich	Kügler	Herr	Sehr geehrter Herr	Herrn	☑
3180	Leonhardt	Giesen	Herr	Sehr geehrter Herr	Herrn	☑
3181	Grit	Cohrs	Frau	Sehr geehrte Frau	Frau	☑
3182	Gero	Gabler	Herr	Sehr geehrter Herr	Herrn	☑
3183	Mechthilde	Schütt	Frau	Sehr geehrte Frau	Frau	☑
3184	Gerald	Bräutigam	Herr	Sehr geehrter Herr	Herrn	☑
3185	Charlotte	Tremmel	Frau	Sehr geehrte Frau	Frau	☑
3186	Engelberta	Eckl	Frau	Sehr geehrte Frau	Frau	☑

Datensatz: 1 von 100 Kein Filter

Abbildung 12.4: Datenquelle für die Erstellung des Serienbriefs

Nun wollen wir diese Tabelle im Formular *frmSeriendruck* zugänglich machen, damit der Benutzer die zu druckenden Datensätze auswählen und den Vorgang starten kann. Dazu nutzen wir die seit Access 2007 bestehende Möglichkeit, in einem Unterformular-Steuerelement direkt

eine Tabelle anzuzeigen, ohne diese erst in einem Unterformular abbilden zu müssen. Erweitern wir also das Formular um ein Unterformular-Steuerelement namens *sfmSeriendruck* unter dem bereits vorhandenen Kombinationsfeld zur Auswahl der Datenquelle (siehe Abbildung 12.5).

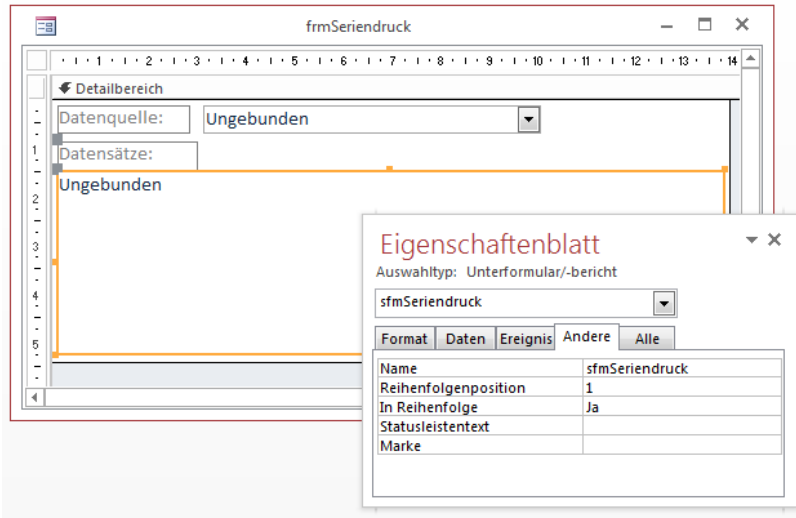


Abbildung 12.5: Unterformular zum Abbilden der Seriendruck-Datensätze

Damit das Unterformular nun nach der Auswahl eines Eintrags im Kombinationsfeld *cboDatenquelle* die gewünschten Daten anzeigt, fügen Sie dem Ereignis *Nach Aktualisierung* des Kombinationsfelds die folgende Ereignisprozedur hinzu:

```
Private Sub cboDatenquelle_AfterUpdate()
    Dim ctl As Control
    Me!sfmSeriendruck.SourceObject = ""
    If Not Len(Me!cboDatenquelle) = 0 Then
        TempTabelleErstellen (Me!cboDatenquelle)
        With Me!sfmSeriendruck
            .SourceObject = "Table.tblSerienbrief_Temp"
            .Form.Controls("Serienbrief").ColumnOrder = 0
            .Form.DatasheetFontHeight = 9
            For Each ctl In .Form.Controls
                If Not TypeName(ctl) = "Label" Then
                    ctl.ColumnWidth = -2
                End If
            Next ctl
        End With
    End If
```

```
End Sub
```

Die Prozedur leert zunächst das Unterformular, indem es die Eigenschaft *SourceObject* auf eine leere Zeichenkette einstellt. Dann prüft sie, ob das Kombinationsfeld *cboDatenquelle* tatsächlich einen Eintrag enthält.

In diesem Fall ruft sie zunächst die bereits beschriebene Prozedur *TempTabelleErstellen* auf, um die Tabelle *tblSerienbrief_Temp* anzulegen und mit Daten zu füllen. Danach bearbeitet sie die Eigenschaften des Unterformular-Steuerelements. Dabei füllt sie dieses mit den Daten der Tabelle *tblSerienbrief_Temp*, indem Sie die Eigenschaft *SourceObject* auf *Table.tblSerienbrief_Temp* einstellt.

Dann holt sie das letzte Feld der Tabelle, nämlich Serienbrief, durch Einstellen der Eigenschaft *ColumnOrder* für dieses Feld ganz nach links, damit der Benutzer es einfacher einstellen kann. Mit der Eigenschaft *DatasheetFontHeight* stellen wir die Schriftgröße im Unterformular auf *9* ein. Außerdem durchläuft die Prozedur alle Steuerelemente der im Unterformular angezeigten Tabelle und stellt für alle Steuerelemente, die kein Bezeichnungsfeld sind, den Wert der Eigenschaft *ColumnWidth* auf *-2* ein. Damit erhalten wir die optimale Spaltenbreite für die aktuell angezeigten Daten in der jeweiligen Spalte.

Das Ergebnis zeigt Abbildung 12.6. Die Spaltenbreiten stimmen, das Feld zum Selektieren der Einträge ist links und die Schriftgröße erlaubt die Anzeige vieler Informationen. Damit der Benutzer beim Aufziehen des Formulars noch mehr Daten sieht, haben wir die Eigenschaft *Horizontaler Anker* und *Vertikaler Anker* jeweils auf den Wert *Beide* eingestellt.

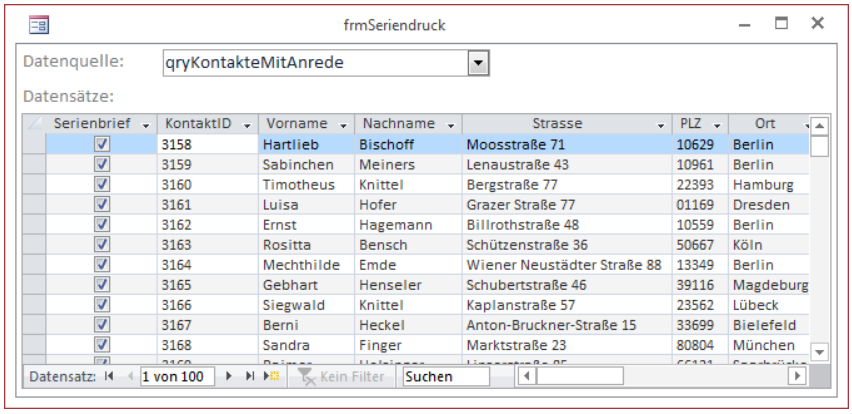


Abbildung 12.6: Auswahl der Datenquelle und Anzeige der Daten samt Auswahlmöglichkeit

12.2.1 Auswählen und Anzeige vereinfachen

Nun fügen wir ein paar Steuerelemente hinzu, mit denen wir die Auswahl und die Anzeige der gewählten Einträge des Unterformulars vereinfachen.

Alle auswählen

Die erste Schaltfläche heißt *cmdAlleAuswaehlen* und soll schlicht bei allen Einträgen einen Haken in die Spalte *Serienbrief* setzen. Die dadurch ausgelöste Ereignisprozedur sieht so aus:

```
Private Sub cmdAlleAuswaehlen_Click()
    Dim db As DAO.Database
    Set db = CurrentDb
    db.Execute "UPDATE tblSerienbrief_Temp SET Serienbrief = -1", dbFailOnError
    Me!sfmSeriendruck.Form.Requery
End Sub
```

Die Prozedur führt eine *UPDATE*-Aktionsabfrage aus, die das Feld *Serienbrief* ohne Einschränkung der betroffenen Datensätze auf den Wert *-1* (*True*) einstellt. Anschließend aktualisiert sie das Unterformular.

Alle Markierungen aufheben

Die zweite Schaltfläche heißt *cmdAlleAbwaehlen* und bewirkt das Gegenteil: Sie soll nämlich den Haken in der Spalte *Serienbrief* für alle Einträge des Unterformulars entfernen. Dies ist die entsprechende Ereignisprozedur:

```
Private Sub cmdAlleAbwaehlen_Click()
    Dim db As DAO.Database
    Set db = CurrentDb
    db.Execute "UPDATE tblSerienbrief_Temp SET Serienbrief = 0", dbFailOnError
    Me!sfmSeriendruck.Form.Requery
End Sub
```

Diese Prozedur stellt den Wert des Feldes *Serienbrief* per *UPDATE*-Abfrage auf den Wert *0* (*False*) ein. Auch hier aktualisiert die Prozedur anschließend die Daten im Unterformular.

Gefilterte Einträge markieren

Nun ist es ja über die Pfeile rechts neben den Spaltenbeschriftungen und auch über weitere Elemente der Benutzeroberfläche möglich, die Daten im Unterformular zu filtern. Dies schlägt sich erstens in der Menge der angezeigten Daten nieder, zweitens in der Eigenschaft *Filter*, die für das Unterformular in der Datenblattansicht festgelegt wird. Sie können auf diese Weise einen Filter anwenden (zum Beispiel *Alle Kontakte, deren Feld Vorname mit dem Buchstaben A beginnt*) und dann alle angezeigten Datensätze auswählen. Dazu klicken Sie auf die Schaltfläche *cmdAuswahlMarkieren*, welche die folgende Ereignisprozedur auslöst:

```
Private Sub cmdAuswahlMarkieren_Click()
    Dim db As DAO.Database
    Dim strFilter As String
```

```
    Set db = CurrentDb
    strFilter = Me!sfmSeriendruck.Form.Filter
    If Len(strFilter) > 0 Then
        strFilter = " WHERE " & strFilter
    End If
    db.Execute "UPDATE tblSerienbrief_Temp SET Serienbrief = -1 " & strFilter, _
        dbFailOnError
    Me!sfmSeriendruck.Form.Requery
End Sub
```

Die Prozedur liest zunächst den für das Unterformular festgelegten Filter ein und speichert diesen in der Variablen *strFilter*. Nun müssen wir damit eine *UPDATE*-Abfrage wie oben erzeugen, diesmal allerdings mit einem zusätzlichen *WHERE*-Kriterium, welches den Filterausdruck enthält. Dazu prüft die Prozedur zunächst, ob *strFilter* überhaupt einen Ausdruck enthält.

Falls ja, ergänzt sie diesen noch um das Schlüsselwort *WHERE*. Anschließend stellt sie die gleiche *UPDATE*-Abfrage wie weiter oben zusammen, allerdings erweitert um die *WHERE*-Bedingung.

Alle gefilterten Datensätze abwählen

Vielleicht möchten Sie dies auch genau umgekehrt erledigen und mit dem Filter alle Datensätze auswählen, die Sie nicht in die Auswahl für den Serienbrief aufnehmen möchten. In diesem Fall stellen Sie wie gehabt den Filter für die betroffenen Datensätze ein und klicken dann auf die Schaltfläche *cmdAusgewaehlteAbwaehlen*. Diese löst die folgende Prozedur aus und stellt genau für die per Filter festgelegten Datensätze den Wert des Feldes *Serienbrief* auf *0* ein:

```
Private Sub cmdAusgewaehlteAbwaehlen_Click()
    Dim db As DAO.Database
    Dim strFilter As String
    Set db = CurrentDb
    strFilter = Me!sfmSeriendruck.Form.Filter
    If Len(strFilter) > 0 Then
        strFilter = " WHERE " & strFilter
    End If
    db.Execute "UPDATE tblSerienbrief_Temp SET Serienbrief = 0 " & strFilter, _
        dbFailOnError
    Me!sfmSeriendruck.Form.Requery
End Sub
```

12.2.2 Datensätze nach Status anzeigen

Neben dem manuellen Aus- und Abwählen können Sie nun auch mit den vier oben vorgestellten Schaltflächen Datensätze für den Seriendruck selektieren. Nun wollen wir noch eine Möglichkeit

hinzufügen, mit der Sie erstens alle Datensätze anzeigen können, zweitens nur die ausgewählten Datensätze und drittens nur die nicht ausgewählten Datensätze.

Dazu fügen wir dem Formular eine Optionsgruppe namens *ogrAnzeigen* hinzu und legen darin drei Optionsfelder an. Diese sollen die Wert *1* (*Alle anzeigen*), *2* (*Ausgewählte anzeigen*) und *3* (*Nicht ausgewählte anzeigen*) aufnehmen. Das Formular sieht nun wie in Abbildung 12.7 aus.

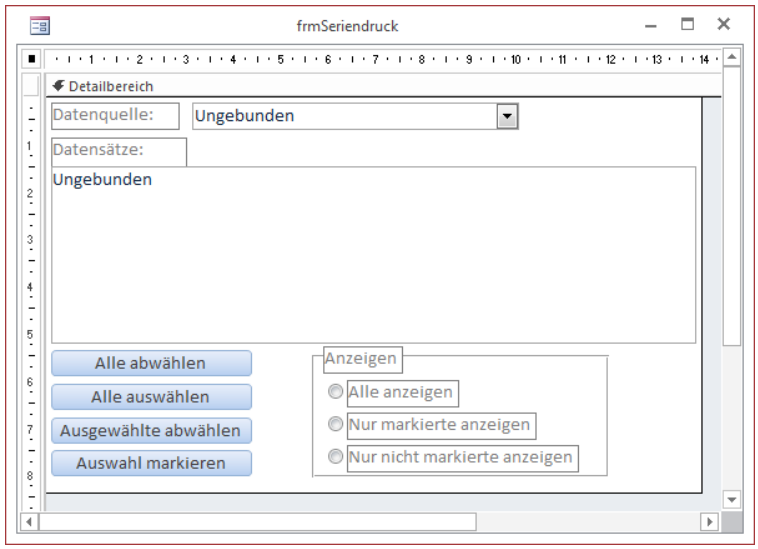


Abbildung 12.7: Formular mit Schaltflächen und Optionen für Auswahl und Anzeige der Datensätze

Für die Optionsgruppe legen Sie außerdem den Wert *1* für die Eigenschaft *Standardwert* fest. Nun soll diese nach der Auswahl eines der Optionswerte die Daten im Unterformular entsprechend filtern. Dies erledigen wir mit der folgenden Ereignisprozedur, die durch das Ereignis *Nach Aktualisierung* ausgelöst wird:

```
Private Sub ogrAnzeigen_AfterUpdate()
    With Me!sfmSeriendruck.Form
        Select Case Me!ogrAnzeige
            Case 1
                .Filter = ""
                .FilterOn = False
            Case 2
                .Filter = "Serienbrief = -1"
                .FilterOn = True
            Case 3
                .Filter = "Serienbrief = 0"
                .FilterOn = True
```

13 Outlook programmieren

Im Gegensatz zu Word und Excel verläuft die Programmierung von Outlook etwas anders. Das liegt in erster Linie daran, dass Outlook ja nicht zur Bearbeitung von Dokumenten dient wie Word und Excel. Stattdessen dient es der Kommunikation via E-Mail und verwaltet Objekte wie Kontakte, Termine oder Aufgaben. Um die Programmierung des Zugriffs auf all diese Objekte kümmern wir uns in weiteren Kapiteln. In diesem hier interessiert uns zunächst, wie wir grundsätzlich von Access aus per VBA auf Outlook zugreifen und welche Elemente für uns wichtig sind.

13.1 Verweis auf die Outlook-Bibliothek

Bevor wir beginnen, fügen Sie Ihrem VBA-Projekt über den Verweise-Dialog (VBA-Editor, Menüeintrag *Extras|Verweise*) einen Verweis auf die Bibliothek *Microsoft Outlook x.0 Object Library* hinzu (siehe Abbildung 13.1). Damit machen Sie die komplette Objektbibliothek von Outlook verfügbar – schauen wir, was sich damit erledigen lässt.

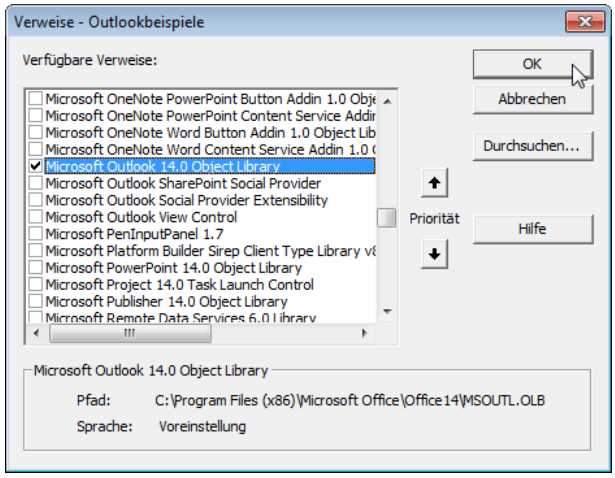


Abbildung 13.1: Herstellen eines Verweises auf die Outlook-Objektbibliothek

13.2 Outlook-Instanz

Vor dem Zugriff per VBA benötigen Sie einen Verweis auf eine Instanz von Outlook. Sie können diese, wie bei den übrigen Office-Anwendungen, per Late- oder Early Binding holen. Wir verwenden Early Binding, da wir ja bereits einen Verweis gesetzt haben.

Im Gegensatz zu Word und Outlook ist der Zugriff auf die Outlook-Instanz übrigens deutlich einfacher: Wenn noch keine Outlook-Instanz geöffnet ist, erstellt der New-Befehl eine neue Instanz, wenn bereits eine Instanz vorliegt, liefert der New-Befehl einen Verweis auf diese Instanz. Auch wenn Sie bereits eine Instanz per VBA erzeugt haben und dann eine weitere über den entsprechenden Windows-Befehl starten, wird nur eine Instanz geladen.

Eine Objektvariable zum Speichern eines Verweises auf die Outlook-Instanz deklarieren Sie so:

```
Dim objOutlook As Outlook.Application
```

Die Instanzierung und Zuweisung erledigt diese Anweisung:

```
Set objOutlook = New Outlook.Application
```

Das Outlook-Objekt bietet gar nicht mal so viele Optionen an. Für uns als VBA-Programmierer ist die interessanteste die Methode *GetNamespace*, mit der wir einen Verweis auf den so genannten MAPI-Namespace erhalten. *MAPI* heißt *Messaging Application Programming Interface* und bietet Zugriff auf fast alle Objekte in Outlook, die wir uns in diesem Buch ansehen.

Den MAPI-Namespace speichern wir in einer Variablen des Typs *Namespace*:

```
Dim objMAPI As Outlook.NameSpace
```

Es gibt nun nicht etwa eine *Namespaces*-Auflistung, sondern schlicht und einfach die Methode *GetNamespace*. Dieser übergeben wir den Parameter *MAPI*:

```
Set objMAPI = objOutlook.GetNamespace("MAPI")
```

13.2.1 Outlook-Objekt einfach verfügbar machen

In den nächsten Abschnitten werden wir noch einige Male auf ein *Outlook.Application*-Objekt zugreifen, und auch in den folgenden Kapiteln wird dies geschehen. Damit wir nicht in jeder Routine erneut die Deklaration und Instanzierung unterbringen müssen, erledigen wird dies in einer kleinen Funktion. Diese bezieht sich immer auf eine private Variable im gleichen Modul, die wir wie folgt deklarieren:

```
Private m_Outlook As Outlook.Application
```

Hinzu kommt die folgende Funktion namens *GetOutlook*. Diese prüft, ob *m_Outlook* bereits einen Verweis auf eine Outlook-Instanz enthält. Ist dies nicht der Fall (*m_Outlook Is Nothing*), füllt sie *m_Outlook* mit einem neuen Verweis auf die bestehende oder neu zu erzeugende Instanz. Vielleicht ist *m_Outlook* auch schon gefüllt. Dann versucht die Prozedur, die Länge des Wertes der Eigenschaft *Name* zu ermitteln.

Warum das? Eigentlich wollen wir nur prüfen, ob *m_Outlook* auch tatsächlich eine Outlook-Instanz referenziert. Es kann nämlich auch sein, dass wir *m_Outlook* gefüllt haben, die Outlook-Instanz aber zwischenzeitlich anderweitig beendet wurde – beispielsweise durch den Benutzer

über den Task Manager. Noch wahrscheinlicher ist der Fall, dass der Benutzer eine Outlook-Instanz über die Benutzeroberfläche von Windows öffnet, dann Access ebenfalls auf diese Instanz zugreift und der Benutzer Outlook über die Benutzeroberfläche wieder schließt. Auch dann ist *m_Outlook* noch gefüllt, die dahinter steckende Instanz existiert allerdings nicht mehr.

Der Zugriff auf eine der Eigenschaften von *m_Outlook* würde dann einen Fehler auslösen, den wir durch vorheriges Setzen der Anweisung *On Error Resume Next* unterbinden. Hat *m_Outlook.Name* die Länge *0*, soll die Instanzvariable *m_Outlook* ebenfalls neu gefüllt werden. Schließlich übergibt die Funktion den Inhalt von *m_Outlook* an die Rückgabevariable der Funktion namens *GetOutlook*:

```
Public Function GetOutlook() As Outlook.Application
    If m_Outlook Is Nothing Then
        Set m_Outlook = New Outlook.Application
    Else
        On Error Resume Next
        If Len(m_Outlook) = 0 Then
            Set m_Outlook = New Outlook.Application
        End If
        On Error GoTo 0
    End If
    Set GetOutlook = m_Outlook
End Function
```

Sie können also nun von überall aus (von anderen Routinen, vom Direktfenster) auf eine Outlook-Instanz zugreifen – zum Beispiel wie hier vom Direktfenster aus, wo wir den Namen der Outlook-Instanz ausgeben:

```
Debug.Print GetOutlook.Name
Outlook
```

Übrigens bleibt die Outlook-Instanz solange erhalten, bis Sie die Variable *m_Outlook* leeren. Auch dafür bauen wir eine kleine Prozedur:

```
Public Sub DestroyOutlook()
    Set m_Outlook = Nothing
End Sub
```

13.2.2 Namespace-Objekt einfach verfügbar machen

Da das MAPI-Namespace-Objekt so ziemlich das einzige Element in der ersten Hierarchie-Ebene unterhalb des *Outlook*-Objekts ist, das wir benutzen, wollen wir auch für dieses eine ähnliche Funktion einrichten. Deshalb legen wir auch für das *Namespace*-Objekt eine entsprechende Objektvariable an:

```
Private m_MAPI As Outlook.NameSpace
```

Die Funktion heißt *GetMAPI* und sieht ähnlich aus wie die Funktion *GetOutlook* – nur etwas einfacher. Sie prüft ebenfalls, ob die gefragte Objektvariable bereits gefüllt ist. Falls nicht, füllt sie die Variable über die Funktion *GetNamespace*. Da wir dazu eine Outlook-Instanz benötigen, holen wir diese mit der Funktion *GetOutlook* und führen die Funktion *GetNamespace* direkt auf diesem Objekt aus. Der Inhalt von *m_MAPI* landet schließlich als Rückgabewert in *GetMAPI*:

```
Public Function GetMAPI() As Outlook.NameSpace
    If m_MAPI Is Nothing Then
        Set m_MAPI = GetOutlook.GetNamespace("MAPI")
    End If
    Set GetMAPI = m_MAPI
End Function
```

Greifen wir dem folgenden Abschnitt vor und lassen uns mit dem MAPI-Namespace-Objekt die Anzahl der Konten der aktuellen Outlook-Instanz ausgeben:

```
Debug.Print GetMAPI.Accounts.Count
 1
```

Der Zugriff mit der Funktion *GetMAPI* gelingt also ohne Probleme.

13.3 Outlook-Accounts

In Outlook können Sie ein oder mehrere Benutzerkonten verwalten. Diese verwalten Sie in der Benutzeroberfläche, indem Sie etwa unter Outlook 2010 auf den Reiter *Datei* klicken und dann im Bereich *Informationen* auf die Schaltfläche *Kontoeinstellungen* und dann nochmals auf den nun erscheinenden Eintrag *Kontoeinstellungen* klicken. Dies öffnet den Dialog aus Abbildung 13.2. Hier sehen Sie einen Account des Typs *POP/SMTP*, der logischerweise als Standard-Eintrag festgelegt ist.

Die Anzahl der Accounts hatten wir bereits weiter oben ausgelesen, nun schauen wir uns den Account selbst mit seinen Eigenschaften an.

Dies erledigen wir mit einer kleinen Prozedur namens *Accounteigenschaften*:

```
Public Sub Accounteigenschaften()
    Dim objAccount As Outlook.Account
    For Each objAccount In GetMAPI.Accounts
        With objAccount
            Debug.Print "Account-Typ: " & .AccountType
            Debug.Print "DisplayName: " & .DisplayName
            Debug.Print "SmtpAddress: " & .SmtpAddress
```

```
            Debug.Print "UserName:     " & .UserName
        End With
    Next objAccount
End Sub
```

Die Prozedur deklariert ein Objekt des Typs *Account* und durchläuft dann in einer *For Each*-Schleife alle Elemente der *Accounts*-Auflistung des aktuellen MAPI-Namespaces. Dabei gibt es die Eigenschaften *AccountType*, *DisplayName*, *SmtpAddress* und *UserName* aus.

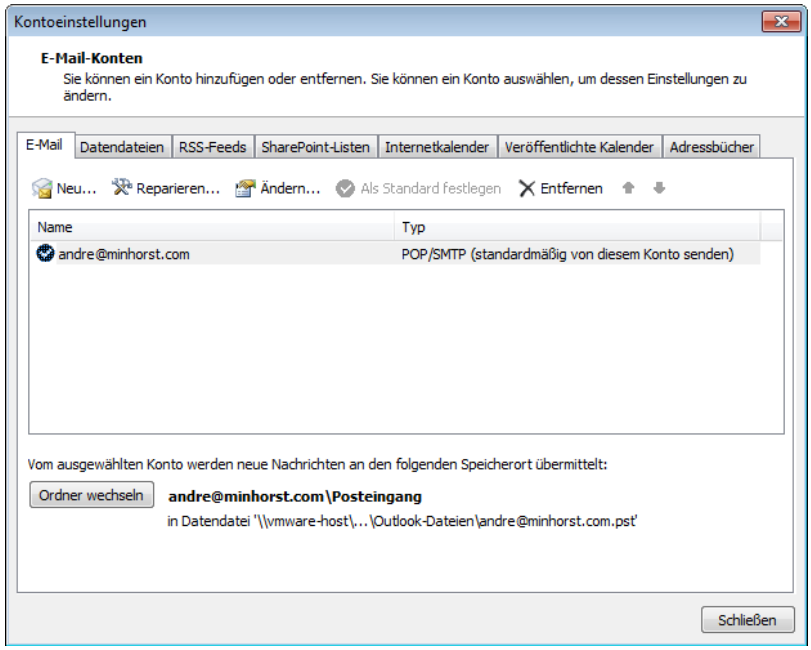


Abbildung 13.2: Dialog zum Verwalten von Outlook-Konten

Dies geschieht allerdings nur, wenn die Sicherheitseinstellungen dieser Outlook-Installation dies erlauben. Ansonsten kann es auch gut sein, dass die Meldung aus Abbildung 13.3 erscheint.

Ist dies der Fall, können Sie sich entscheiden: Entweder, Sie gewöhnen sich daran, bei jedem programmgesteuerten Aufruf über diese Meldung zu stolpern.

Oder Sie passen die Sicherheitseinstellungen von Outlook so an, dass diese Meldung nicht mehr erscheint.

Im letzten Fall klicken Sie etwa unter Outlook 2010 unter dem Reiter *Datei* auf den Befehl *Optionen*. Dort wechseln Sie zum Bereich *Sicherheitscenter* und klicken dort auf die Schaltfläche *Einstellungen für das Sicherheitscenter...*, welches daraufhin geöffnet wird (siehe Abbildung 13.4).

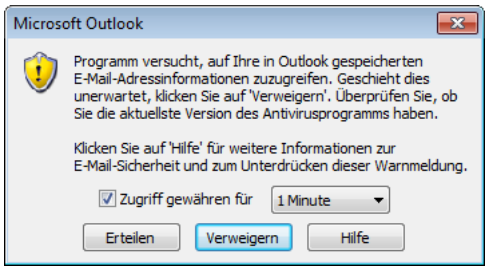


Abbildung 13.3: Sicherheitsmeldung beim programmgesteuerten Zugriff auf Outlook

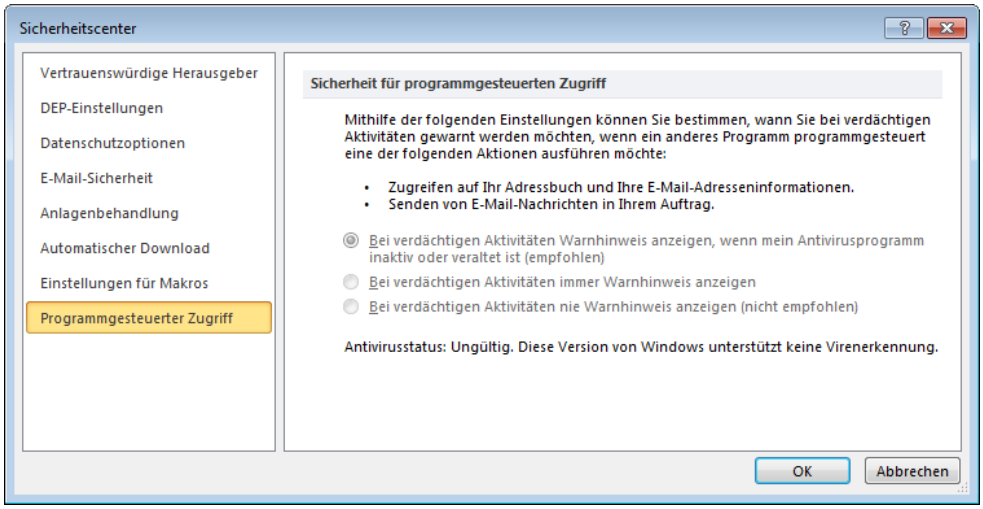


Abbildung 13.4: Kein Zugriff auf die Sicherheitseinstellungen für den programmgesteuerten Zugriff

Dort finden wir prinzipiell genau die Optionen, die wir brauchen. Das Ganze hat nur einen Nachteil: Die Optionen sind deaktiviert und lassen sich nicht einstellen. An dieser Stelle wäre ein Hinweis hilfreich gewesen, dass man diese Einstellung nur ändern kann, wenn man Outlook als Administrator gestartet hat, denn dies ist der Fall.

Also schließen wir Outlook und öffnen es als Administrator. Dazu müssen wir zuerst die Datei *Outlook.exe* suchen, die wir etwa unter Windows 7 in folgendem Ordner finden:

```
C:\Program Files (x86)\Microsoft Office\Office14
```

Dort klicken Sie mit der rechten Maustaste auf den Eintrag *Outlook.exe* und wählen den Menübefehl *Als Administrator ausführen* aus (siehe Abbildung 13.5).

Nun können Sie die Option *Bei verdächtigen Aktivitäten nie Warnhinweis anzeigen* aktivieren (siehe Abbildung 13.6).

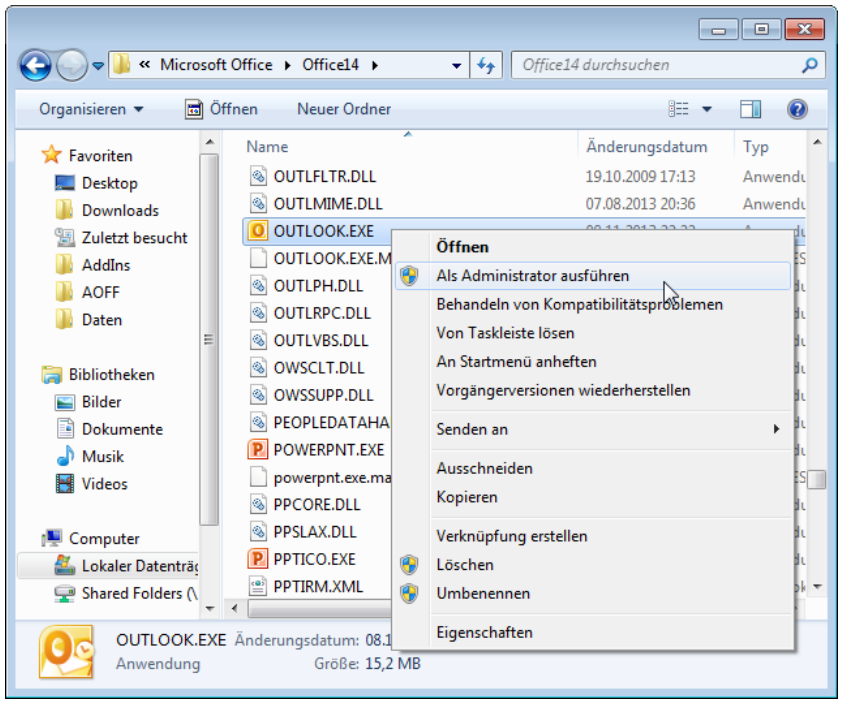


Abbildung 13.5: Outlook als Administrator öffnen

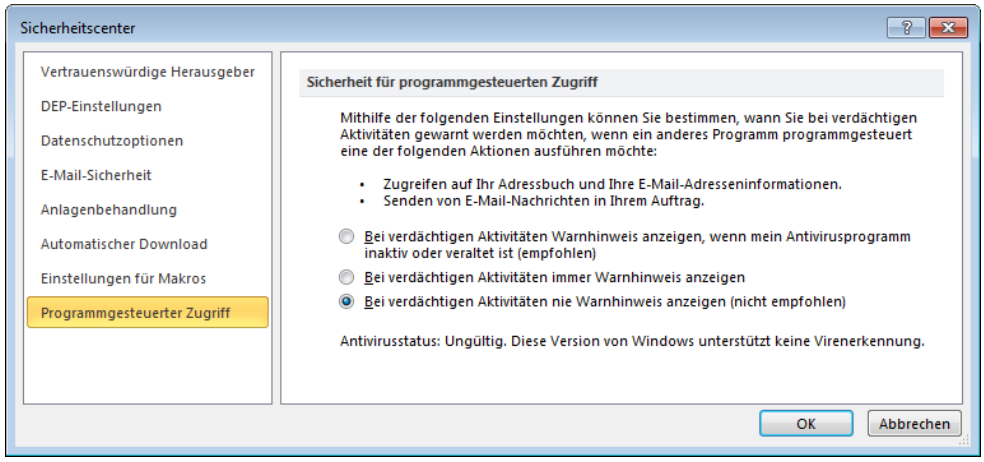


Abbildung 13.6: Freigabe des programmgesteuerten Zugriffs auf Outlook

Mit den *Account*-Elementen können wir arbeiten, wenn wir programmgesteuert ermitteln wollen, welcher der aktuelle Standardaccount ist. Gegebenenfalls möchten Sie diesen ja ändern,

wenn Sie etwa von Access aus eine Mail erstellen möchten, die über einen anderen als den Standardaccount verschickt werden soll.

13.4 Outlook-Kategorien

Für viele Fälle sind Kategorien nützlich. Sie können Elemente wie Mails, Aufgaben, Termine oder Kontakte einer Kategorie zuordnen oder auch mehreren. Die Kategorie stellen Sie über die Benutzeroberfläche ein, indem Sie das gewünschte Objekt öffnen und oben im Ribbon auf *Kategorisieren* klicken. Dort erscheint dann eine Liste der vorhandenen Kategorien (siehe Abbildung 13.7).

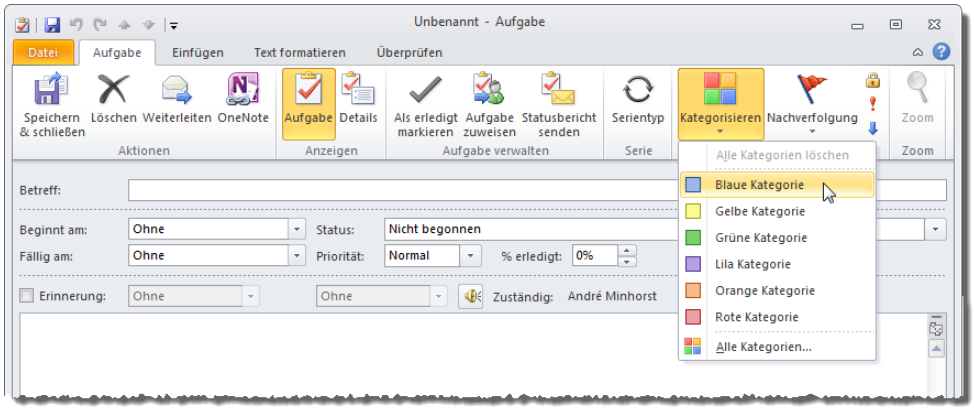


Abbildung 13.7: Einstellen einer Kategorie etwa für eine Aufgabe

Auf diese Kategorien können Sie auch per VBA zugreifen – und zwar über die Auflistung *Categories* des *Namespace*-Objekts. Hier geben wir den Namen der Kategorie sowie den Wert der Eigenschaft *Color* aus:

```
Public Sub Kategorien()
    Dim objCategory As Outlook.Category
    For Each objCategory In GetMAPI.Categories
        With objCategory
            Debug.Print .Name, .Color
        End With
    Next objCategory
End Sub
```

Die Ausgabe für die Kategorien aus der Abbildung sieht so aus:

```
Lila Kategorie              9
```

```
Gelbe Kategorie          4
Blaue Kategorie          8
Grüne Kategorie          5
Rote Kategorie           1
Orange Kategorie         2
```

Die Benutzeroberfläche bietet über den Eintrag *Alle Kategorien ...* des Ribbon-Menüs den Dialog aus Abbildung 13.8 an, mit dem Sie die Kategorien bearbeiten können.

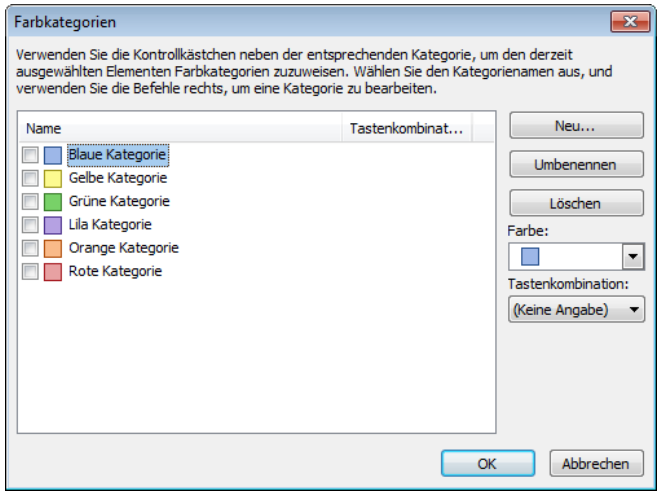


Abbildung 13.8: Dialog zum Bearbeiten der Kategorien

13.5 Outlook-Folder

Im Moment sind für uns die *Folder*-Objekte von Outlook viel interessanter. Darin befinden sich nämlich die eigentlichen Objekte, nämlich Mails, Adressen, Termine, Aufgaben et cetera.

Das *Namespace*-Objekt stellt auch für Outlook-Ordner eine Auflistung zur Verfügung, die logischerweise *Folders* heißt. Die darin enthaltenen Objekte haben den Typ *Folder*. Wir wollen uns einen Überblick verschaffen und schauen uns erstmal alle *Folder*-Objekte an. Dazu benötigen wir zwei Routinen, von denen die zweite rekursiv ist, sich also selbst aufruft. Die erste Routine sieht wie folgt aus:

```
Public Sub Outlookfolder()
    Dim objFolder As Outlook.Folder
    For Each objFolder In GetMAPI.Folders
        With objFolder
            Debug.Print .Name, .DefaultItemType
```

```
            Outlookfolder_Rek objFolder, 1
        End With
    Next objFolder
End Sub
```

Sie durchläuft alle Elemente der *Folders*-Auflistung in einer *For Each*-Schleife und gibt dabei jeweils den Namen des Ordners sowie den Zahlenwert für die Eigenschaft *DefaultItemType* aus.

DefaultItemType kann beispielsweise die folgenden Werte annehmen:

» *olMailItem* (*0*): Mail

» *olAppointmentItem* (*1*): Termin

» *olContactItem* (*2*): Kontakt

» *olTaskItem* (*3*): Aufgabe

» *olJournalItem* (*4*): Journal

» *olNoteItem* (*5*): Notiz

Es gibt noch weitere Elementtypen, die wir aber in diesem Buch nicht behandeln.

Für jedes *Folder*-Element ruft die Prozedur *Outlookfolder* die rekursive Prozedur *Outlookfolder_Rek* auf, die wie folgt aussieht:

```
Public Sub Outlookfolder_Rek(objParent As Folder, intEbene As Integer)
    Dim objFolder As Outlook.Folder
    For Each objFolder In objParent.Folders
        With objFolder
            Debug.Print Space(intEbene * 4) & .Name, .DefaultItemType
            Outlookfolder_Rek objFolder, intEbene + 1
        End With
    Next objFolder
End Sub
```

Diese Prozedur erwartet mit *objParent* den aktuellen Folder der aufrufenden Prozedur, damit sie die darin enthaltenen *Folder*-Objekte durchlaufen kann. Außerdem verwendet sie einen *Integer*-Parameter, um die aktuelle Ebene in der Hierarchie zu übergeben. Diese benötigen wir nur, um die Ordner bei der Ausgabe ein wenig einzurücken.

Das Ergebnis sieht auf einer relativ frischen Installation etwa so aus:

```
andre@minhorst.com          0
    Gelöschte Elemente      0
    Posteingang             0
    Postausgang             0
```

```
Gesendete Elemente       0
Junk-E-Mail              0
Einstellungen für Unterhaltungsaktionen              0
Einstellungen für QuickSteps           0
Entwürfe   0
Kalender   1
Kontakte   2
    Kontakte_Unterordner               2
Journal    4
Notizen    5
Aufgaben   3
RSS-Feeds  0
Vorgeschlagene Kontakte  2
Newsfeed   0
```

Manchmal heißt das Hauptelement auch Outlook, was dem Namen der Datendatei ohne Dateiendung entspricht (*Outlook.pst* ist Ihnen sicher ein Begriff). Im vorliegenden Fall wurde eine neue Datendatei namens *andre@minhorst.com.pst* angelegt.

13.5.1 Ordner auswählen

Wenn Sie von Access aus auf die Daten eines der Outlook-Ordner zugreifen wollen, müssen Sie den Namen des Ordners kennen – oder noch besser direkt das entsprechende Folder-Objekt verwenden. Sie müssen gar nicht erst die Outlook-Ordner in Access einlesen und diese etwa in einem Listenfeld eines Formulars zur Auswahl bereitstellen. Nein: Outlook bietet zu diesem Zweck einen eigenen Dialog an.

Diesen starten Sie, Sie werden es ahnen, über eine Methode des *Namespace*-Objekts. Die folgende Prozedur deklariert ein *Folder*-Objekt und ruft dann die Methode *PickFolder* des mit *GetMAPI* gelieferten *Namespace*-Objekts auf. Dies zeigt den Dialog aus Abbildung 13.9 an. Die Prozedur wird solange angehalten, bis der Benutzer den Dialog entweder mit der *OK*- oder der *Abbrechen*-Schaltfläche schließt.

Danach prüft die Prozedur, ob *objFolder* nicht leer ist (was der Fall wäre, wenn der Benutzer auf *Abbrechen* klickt) und gibt anderenfalls den Pfad des gewählten Ordners aus:

```
Public Sub OrdnerAuswaehlen()
    Dim objFolder As Outlook.Folder
    Set objFolder = GetMAPI.PickFolder
    If Not objFolder Is Nothing Then
        Debug.Print objFolder.FolderPath
    End If
End Sub
```

14 Mails mit Outlook senden, empfangen und verarbeiten

Es gibt eine Reihe von Dingen, die Sie von Access aus mit der Outlook-Mail-Funktion erledigen können. Sie können von Access aus eine neue Outlook-Mail öffnen und diese dann vom Benutzer ausfüllen und verschicken lassen. Oder Sie füllen die neue Outlook-Mail direkt mit Informationen aus der E-Mail.

Sie können auch die Daten für die E-Mail wie Empfänger, Betreff und Inhalt in einem Formular der Datenbankanwendung eingeben und darauf basierend unsichtbar eine neue E-Mail erstellen und verschicken. Vielleicht möchten Sie auch eine Serien-E-Mail an mehrere Empfänger versenden.

Andersherum gibt es auch einige sinnvolle Nutzungsmöglichkeiten: Sie können beispielsweise alle Outlook-Mails oder auch nur die E-Mails aus einem Verzeichnis in eine Access-Tabelle einlesen. Dabei könnten Sie die Attachments auf der Festplatte speichern und von der Datenbank aus referenzieren – oder Sie speichern die Anlagen gleich im passenden Anlage-Feld. Outlook können Sie auch soweit automatisieren, dass eingehende Mails direkt in irgendeiner Weise in der Datenbank erfasst werden. Oder Sie sehen einen speziellen Ordner vor, in den der Benutzer E-Mails ziehen kann, die dann in der Datenbank erfasst werden sollen.

Darauf aufbauend gibt es natürlich noch umfangreichere Lösungen – so könnten Sie, wenn Sie alle E-Mails, die Sie an einen bestimmten Kontakt geschickt und von diesem erhalten haben, in der Datenbank speichern und im Verlauf darstellen.

Dieses Kapitel des Buchs wirft einen Blick auf die notwendigen Grundlagen.

Hinweis

Dieses Kapitel baut auf den Techniken auf, die Sie im Kapitel »Outlook programmieren« ab Seite 339 finden. Das Testen der Beispiele dieses Kapitels setzt voraus, dass das VBA-Projekt Ihrer Datenbank einen Verweis auf die Bibliothek *Microsoft Outlook x.0 Object Library* enthält. Außerdem sollten Sie dieser das Modul *mdlOutlook* hinzufügen, da dieses einige Funktionen enthält, die wir im Folgenden nutzen.

14.1 Mails verschicken

Die folgenden Abschnitte beschreiben verschiedene Möglichkeiten, um eine Mail von Access aus mit Outlook zu verschicken.

14.1.1 Neue Mail in Outlook öffnen

Die einfachste Variante, von Access aus eine E-Mail zu versenden, nutzt einfach den entsprechenden Dialog von Outlook. Das Ergebnis soll etwa so wie in Abbildung 14.1 aussehen. Da hierzu gar nicht erst das vollständige Outlook-Hauptfenster eingeblendet werden muss, macht diese Lösung einen stimmigen Eindruck – das Mail-Fenster sieht fast so aus, als würde es zur aktuellen Anwendung gehören.

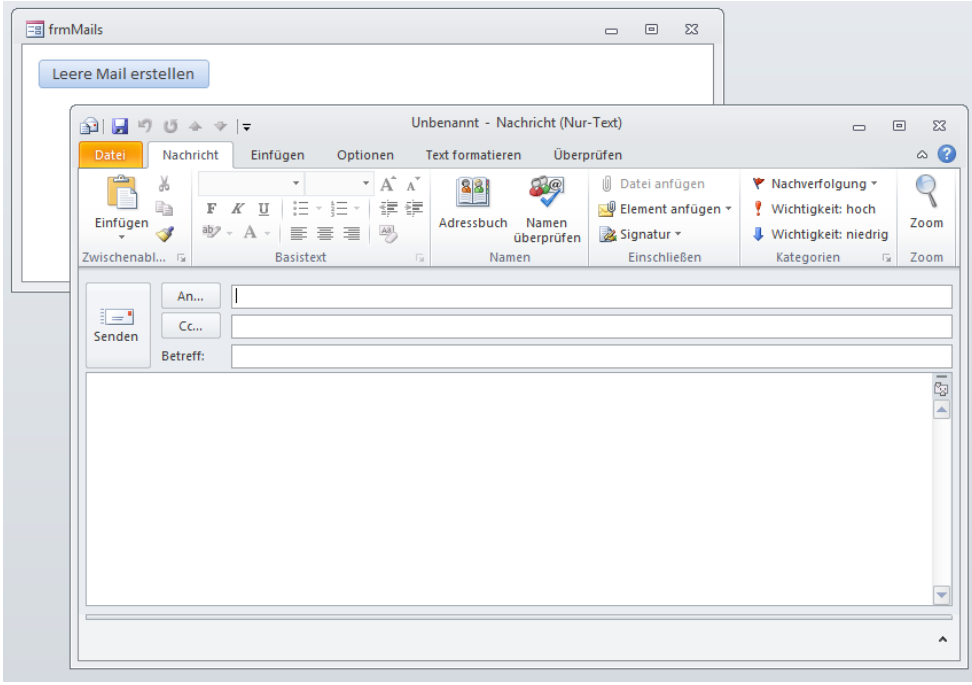


Abbildung 14.1: Erstellen einer neuen, leeren E-Mail per Schaltfläche

Für diesen Ansatz ist nur ein Formular notwendig, von dem aus wir die Prozedur starten. Die dazu verwendete Schaltfläche heißt *cmdMailErstellen* und löst für das Ereignis *Beim Klicken* die folgende Prozedur aus. Diese erstellt ein *Outlook.Application*-Objekt und ruft dann dessen Funktion *CreateItem* mit dem Wert *olMailItem* für den Parameter *ItemType* auf.

Das Ergebnis speichert die Prozedur in der Variablen *objMailItem*. Dies ist nötig, um das Elemente mit der *Display*-Methode sichtbar zu machen:

```
Private Sub cmdMailErstellen_Click()
    Dim objOutlook As Outlook.Application
    Dim objMailItem As Outlook.MailItem
    Set objOutlook = New Outlook.Application
```

```
    Set objMailItem = objOutlook.CreateItem(olMailItem)
    With objMailItem
        .Display
    End With
End Sub
```

Das ist allerdings auch die Lösung für die geringsten Ansprüche – quasi ein Shortcut, um direkt von Access aus eine Mail zu erstellen, ohne dafür extra zu Outlook wechseln zu müssen. Andererseits wird die so erstellte und versendete E-Mail von Outlook im *Gesendete Elemente*-Ordner gespeichert – Sie haben also allen Komfort, den auch der direkt Versand aus Outlook heraus bietet.

14.1.2 Einfache Mail erstellen und anzeigen

Nun gehen einen Schritt weiter und wollen die wichtigsten Inhalte der E-Mail bereits vorher in einem Formular der Access-Anwendung aufnehmen. Erste dann soll die Mail angezeigt werden, damit der Benutzer gegebenenfalls noch Änderungen durchführen und die Mail dann verschicken kann (siehe Abbildung 14.2). Das Formular enthält die drei Textfelder *txtAn*, *txtBetreff* und *txtInhalt* zur Aufnahme der Mail-Daten.

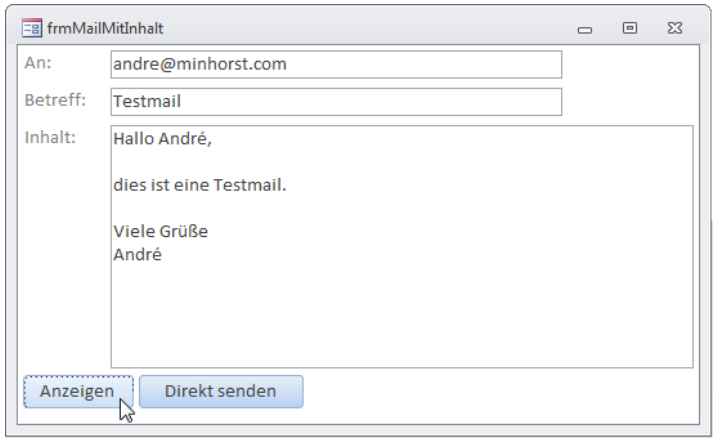


Abbildung 14.2: Vorbereiten und versenden einer E-Mail

Die Schaltfläche *cmdAnzeigen* soll dann eine neue Mail erstellen, die im Formular eingegebenen Daten in die Mail übertragen und diese anzeigen. Die Ereignisprozedur erstellt wieder ein Outlook-Objekt und eine neue E-Mail. Diesmal füllt Sie vor dem Anzeigen die Eigenschaften *To*, *Subject* und *Body* der E-Mail und zeigt diese erst danach mit der *Display*-Methode an:

```
Private Sub cmdAnzeigen_Click()
    Dim objOutlook As Outlook.Application
```

```
    Dim objMailItem As Outlook.MailItem
    Set objOutlook = New Outlook.Application
    Set objMailItem = objOutlook.CreateItem(olMailItem)
    With objMailItem
        .To = Me!txtAn
        .Subject = Me!txtBetreff
        .Body = Me!txtInhalt
        .Display
    End With
End Sub
```

Dies funktioniert prima – das Ergebnis finden Sie in Abbildung 14.3.

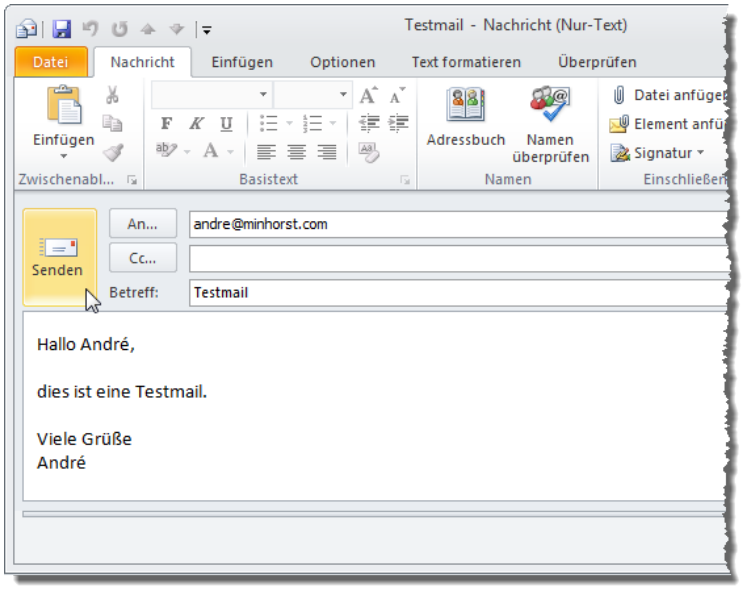


Abbildung 14.3: Eine per VBA erstellt und gefüllte E-Mail

Parameter der Display-Methode

Die *Display*-Methode bietet einen Parameter, den wir bisher ignoriert haben. Er heißt *Modal* und legt fest, ob das *MailItem*-Fenster als modaler Dialog geöffnet werden soll. Wenn Sie hier den Wert *True* übergeben, kann der Benutzer erst mit der Access-Anwendung weiterarbeiten, wenn das *MailItem*-Fenster wieder geschlossen ist – also beispielsweise, wenn die Mail versendet wurde. Der Standardwert lautet *False*. Die Einstellung auf den Wert *True* sieht etwa so aus:

```
objMailItem.Display True
```

14.1.3 Einfache Mail erstellen und direkt senden

Auf ähnliche Weise wie im vorherigen Beispiel können Sie die per VBA mit den Daten des Formulars gefüllte E-Mail auch direkt verwenden. Dies erledigen Sie mit einem Mausklick auf die Schaltfläche *cmdDirektSenden*, die so aussieht:

```
Private Sub cmdDirektSenden_Click()
    Dim objOutlook As Outlook.Application
    Dim objMailItem As Outlook.MailItem
    Set objOutlook = New Outlook.Application
    Set objMailItem = objOutlook.CreateItem(olMailItem)
    With objMailItem
        .To = Me!txtAn
        .Subject = Me!txtBetreff
        .Body = Me!txtInhalt
        .Send
    End With
End Sub
```

Grundsätzlich erledigt die Prozedur die gleichen Schritte wie die vorherige, aber sie verschickt die Mail gleich mit der *Send*-Methode. Wichtig ist: Die Mail wird in diesem Fall nicht mehr angezeigt!

Jetzt kommt der interessante Teil dieser Methode: Wenn Sie diese Schaltfläche betätigen, nachdem Sie zuvor eine Mail mit der anderen Schaltfläche (also *cmdAnzeigen*) versendet haben, klappt auch *cmdDirektSenden* reibungslos. Aber beenden Sie Outlook zuvor einmal komplett (notfalls per Task-Manager) und versuchen dann erneut, *cmdDirektSenden* zu betätigen. Das Ergebnis sieht wie in Abbildung 14.4 aus.

Wie ist dieser Fehler zu erklären? Aus der Fehlermeldung allein leider gar nicht. In so einem Fall hilft nur eine Google-Suche, welche die Erklärung lieferte, dass dieser Fehler aus Sicherheitsgründen ausgelöst wird – um zu verhindern, dass bösartige Skripte direkt auf Outlook zugreifen und beispielsweise Mails versenden.

Der erste Workaround, der einem einfällt, ist das vorherige kurze Einblenden der E-Mail – also das Voranstellen von *objMail.Display* vor der Anweisung *objMail.Send*. Wenn dies geschieht, kann ein Angreifer jedenfalls nicht mehr unbemerkt E-Mails versenden. Allerdings sieht es auch nicht besonders schön aus, wenn die Mail kurz ins Bild flackert.

Wir können das Problem jedoch dennoch beheben. Dazu müssen wir nur das *MailItem*-Objekt auf andere Weise anlegen als mit der *CreateItem*-Methode. Die folgende Prozedur verwendet zusätzlich ein Objekt des Typs *Namespace* und eines des Typs *Folder*. Sie erstellt für die Variable *objMAPI* einen Verweis auf den MAPI-Namespace und für *objFolder* einen Verweis auf den Ordner *Posteingang* von Outlook (*GetDefaultFolder(olFolderInbox)*).

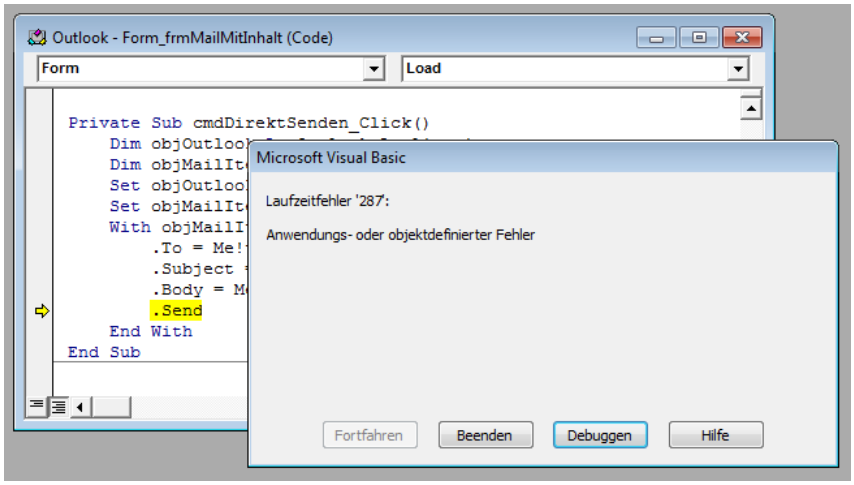


Abbildung 14.4: Fehler beim Versuch, eine Mail direkt zu versenden

Dann nutzt sie die *Add*-Methode der *Items*-Auflistung des *Folder*-Objekts für den Posteingang, um die neue E-Mail zu erstellen. Damit scheinen die Sicherheitsbedürfnisse von Outlook gestillt zu sein – der Versand funktioniert so:

```
Private Sub cmdDirektSenden_Click()
    Dim objOutlook As Outlook.Application
    Dim objMailItem As Outlook.MailItem
    Dim objMAPI As Outlook.NameSpace
    Dim objFolder As Outlook.Folder
    Set objOutlook = New Outlook.Application
    Set objMAPI = objOutlook.GetNamespace("MAPI")
    Set objFolder = objMAPI.GetDefaultFolder(olFolderInbox)
    Set objMailItem = objFolder.Items.Add(olMailItem)
    With objMailItem
        .To = Me!txtAn
        .Subject = Me!txtBetreff
        .Body = Me!txtInhalt
        .Send
    End With
End Sub
```

14.1.4 Account zum Versenden von Mails abrufen und einstellen

Bisher haben wir nur die drei Informationen Empfänger, Betreff und Inhalt angegeben. Was aber ist mit dem Versender? Immerhin kann man ja mit Outlook durchaus mehrere E-Mail-Konten

verwalten, mit denen man auch man auch Mails verschicken und empfangen kann. Mit welcher Eigenschaft also legen wir den Versender einer E-Mail fest? Wenn Sie nur ein einziges E-Mail-Konto verwenden, brauchen Sie hier nichts weiter zu tun – Outlook verwendet dann automatisch das vorliegende Konto.

Für die folgenden Beispiele habe ich drei Konten in meiner Outlook-Instanz angelegt (siehe Abbildung 14.5).

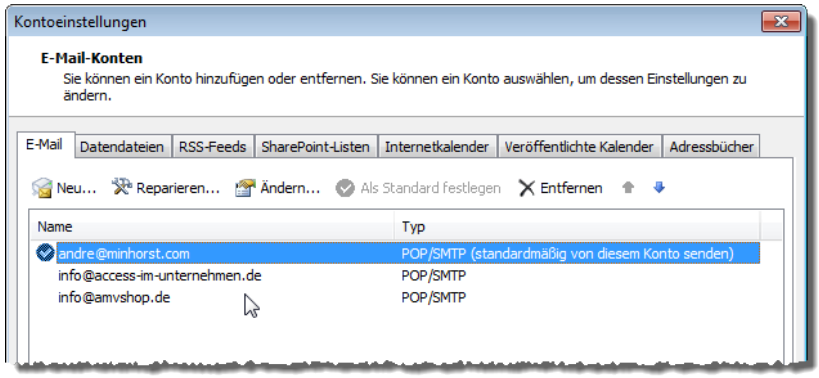


Abbildung 14.5: Die Outlook-Kontoeinstellungen mit den drei E-Mail-Konten

Wenn Sie nun in Outlook eine neue E-Mail erstellen, können Sie mit der Schaltfläche *Von* eines der vorhandenen E-Mail-Konten auswählen (siehe Abbildung 14.6).

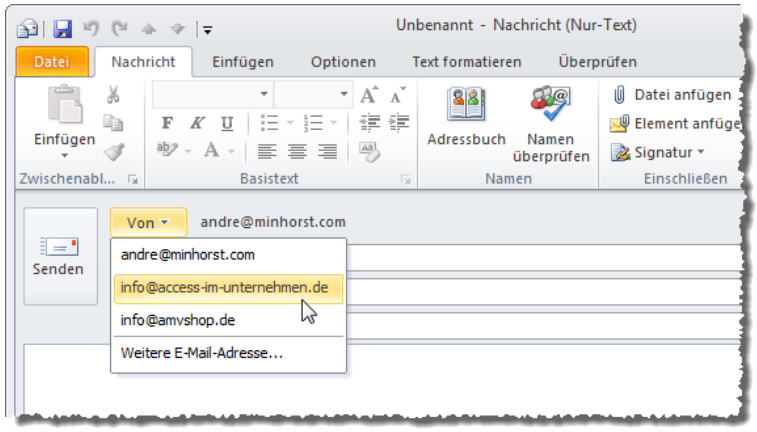


Abbildung 14.6: Auswahl des zu verwendenden Kontos

Wie können wir nun von Access aus festlegen, welches Konto als Versender einer neuen E-Mail eingestellt wird? Dazu fügen wir einem neuen Formular namens *frmMailAbsenderEinstellen*

zunächst ein Textfeld namens *txtVon* hinzu (siehe Abbildung 14.7). Dort fügen Sie einfach die E-Mail-Adresse für den Absender hinzu.

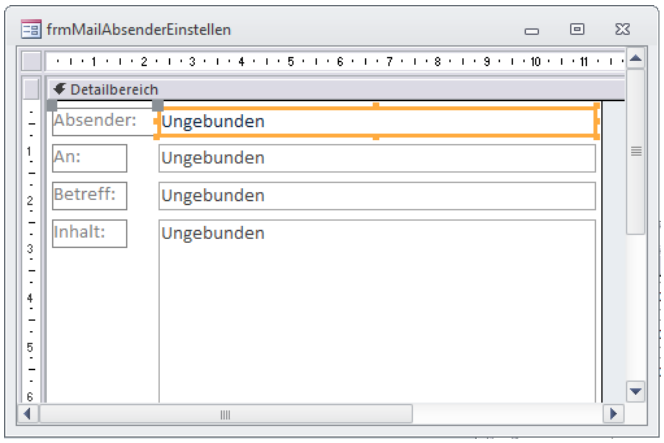


Abbildung 14.7: Mail-Formular mit Absender

Wie aber übergeben wir diesen Wert nun an die neu erstellte Mail? Zumindest liefert das *MailItem*-Objekt schon mal keine Eigenschaft namens *From*. Im Objektkatalog werden wir jedoch unter dem Begriff *Sender* fündig (siehe Abbildung 14.8).

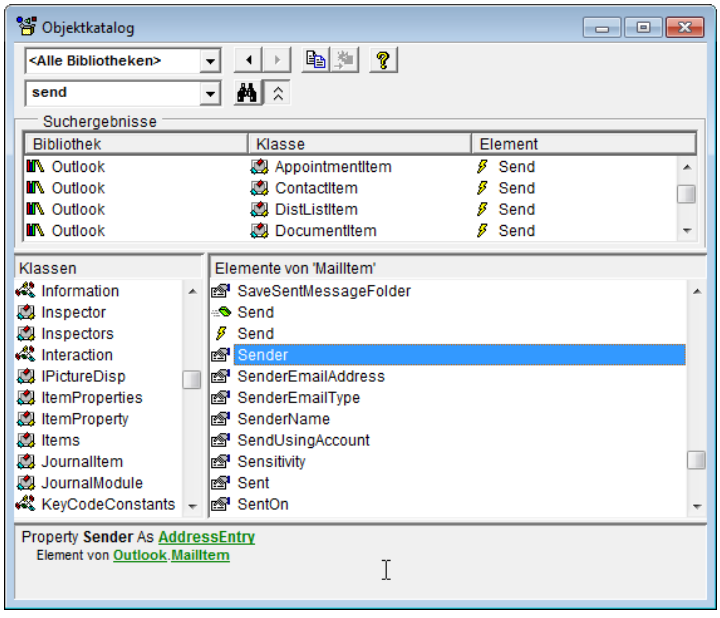


Abbildung 14.8: Eigenschaften für den Sender einer E-Mail

Dabei hat Sender den Datentyp *AddressEntry*. *SenderEMailAddress*, *SenderEMailType* und *SenderName* sind schreibgeschützte Eigenschaften. *SendUsingAccount* hört sich hingegen interessant für uns an – über die Outlook-Accounts haben Sie ja bereits unter »Outlook-Accounts« ab Seite 342 einiges erfahren.

Leider bietet die *Accounts*-Auflistung des *Namespace*-Objekts keine Möglichkeit, direkt etwa über die E-Mail-Adresse auf das entsprechende Objekt zuzugreifen. Also schreiben wir eine kleine Funktion, welche alle *Account*-Objekte des MAPI-Namespaces durchläuft und die Eigenschaft *SMTPAddress* des aktuellen Accounts mit der übergebenen E-Mail-Adresse vergleicht.

Sobald diese eine Übereinstimmung gefunden hat, stellt die Funktion den Rückgabewert auf das gefundene *Account*-Objekt ein und beendet die Funktion mit *Exit Function*:

```
Public Function AccountByEMail(strEMail As String) As Outlook.Account
    Dim objAccount As Outlook.Account
    For Each objAccount In GetMAPI.Accounts
        If objAccount.SmtpAddress = strEMail Then
            Set AccountByEMail = objAccount
            Exit Function
        End If
    Next objAccount
End Function
```

Sie können diese Funktion beispielsweise mit dem folgenden Aufruf im Direktfenster von Access ausprobieren:

```
? AccountByEMail("andre@minhorst.com").DisplayName
André Minhorst
```

Die Funktion *AccountByEMail* finden Sie in der Beispieldatenbank übrigens im Modul *mdlOutlook*, wo wir auch die übrigen allgemeinen Funktionen zum Thema Outlook abgelegt haben.

Nun wollen wir die Prozedur zum Erstellen und Anzeigen der E-Mail mit dem neuen Absender programmieren. Diese ist grundsätzlich wieder wie die vorherigen Prozeduren aufgebaut. Der entscheidende Unterschied ist die Zeile, die der Eigenschaft *SendUsingAccount* das Ergebnis der Funktion *AccountByEMail(Me!txtVon)* zuweist:

```
Private Sub cmdAnzeigen_Click()
    Dim objMailItem As Outlook.MailItem
    Dim objAccount As Outlook.Account
    Set objMailItem = GetOutlook.CreateItem(olMailItem)
    With objMailItem
        .SendUsingAccount = AccountByEMail(Me!txtVon)
        .To = Me!txtAn
        .Subject = Me!txtBetreff
```

```
            .Body = Me!txtInhalt
            .Display
        End With
    End Sub
```

Hinweis: Fehlergefahr! In dieser Prozedur weisen wir der Eigenschaft *SendUsingAccount* ein Objekt zu. Dennoch verwenden wir nicht die *Set*-Anweisung! Dies führt nämlich zu einem Fehler, der mich fast in den Wahnsinn getrieben hat. Warum hier Set weggelassen werden muss, weiß wohl nur der zuständige Microsoft-Entwickler ...

Und es funktioniert, wie Abbildung 14.9 zeigt.

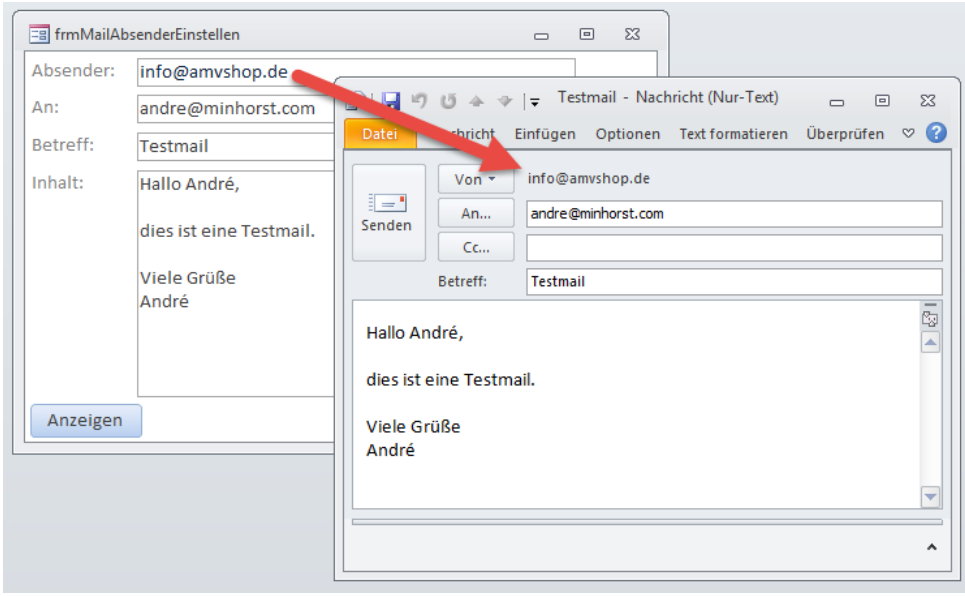


Abbildung 14.9: Einstellen des Absenders einer E-Mail

Nun können wir das Formular noch etwas benutzerfreundlicher gestalten, indem wir das Textfeld *txtVon* durch ein Kombinationsfeld *cboVon* ersetzen, das gleich beim Öffnen mit den E-Mail-Adressen aus den verfügbaren Accounts gefüllt wird. Das Ergebnis soll dann etwa wie in Abbildung 14.10 aussehen.

Damit das Kombinationsfeld beim Start mit den verfügbaren E-Mail-Adressen gefüllt wird, passen wir die Ereignisprozedur, die durch das Ereignis *Beim Laden* des Formulars ausgelöst wird, wie folgt an. Die Prozedur stellt zunächst den *Herkunftstyp* des Kombinationsfeldes auf *Wertliste* ein. Dann durchläuft sie die Elemente der *Accounts*-Auflistung und fügt für jedes Element einen neuen Eintrag zum Kombinationsfeld hinzu. Schließlich stellt sie den ersten Wert als Standardwert ein.

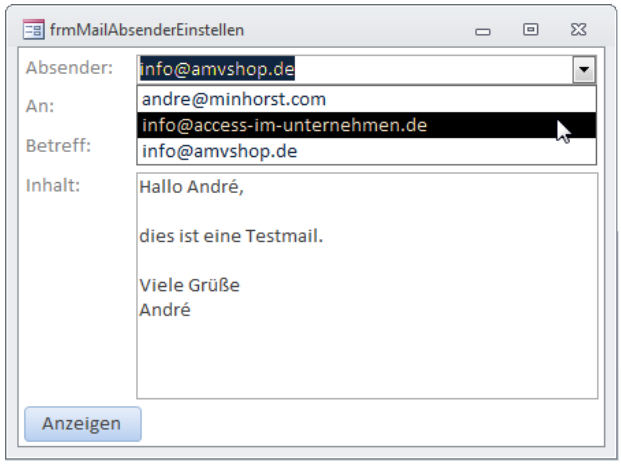


Abbildung 14.10: Kombinationsfeld zum Auswählen der verfügbaren Accounts

Die übrigen Anweisungen dienen dem Füllen der verbleibenden Steuerelemente mit Testdaten (diese müssen Sie an Ihre Gegebenheiten anpassen!):

```
Private Sub Form_Load()
    Dim objAccount As Outlook.Account
    Me!cboVon.RowSourceType = "Value List"
    For Each objAccount In GetMAPI.Accounts
        Me!cboVon.AddItem objAccount.SmtpAddress
    Next objAccount
    Me!cboVon = Me!cboVon.ItemData(0)
    'Me!txtVon = "info@amvshop.de"
    Me!txtAn = "andre@minhorst.com"
    Me!txtBetreff = "Testmail"
    Me!txtInhalt = "Hallo André," & vbCrLf & vbCrLf & "dies ist eine Testmail." _
        & vbCrLf & vbCrLf & "Viele Grüße" & vbCrLf & "André"
End Sub
```

Dies klappt ganz ausgezeichnet, dauert aber ein Weilchen, wenn Outlook zuvor noch nicht von der Anwendung geöffnet wurde. Da muss noch eine bessere Lösung her. Diese sieht so aus, dass wir die Account-Daten in einer Tabelle speichern und diese nur bei Bedarf aktualisieren.

Die Tabelle soll im Entwurf so wie in Abbildung 14.11 aussehen. Wir nehmen hier nur die wichtigsten Informationen auf und fügen noch ein Primärschlüsselfeld namens *MailAccountID* hinzu.

Damit wir die Liste auch aktualisieren können, fügen wir dem Formular *frmMailAbsenderEinstellen* noch eine entsprechende Schaltfläche namens *cmdAktualisieren* hinzu.

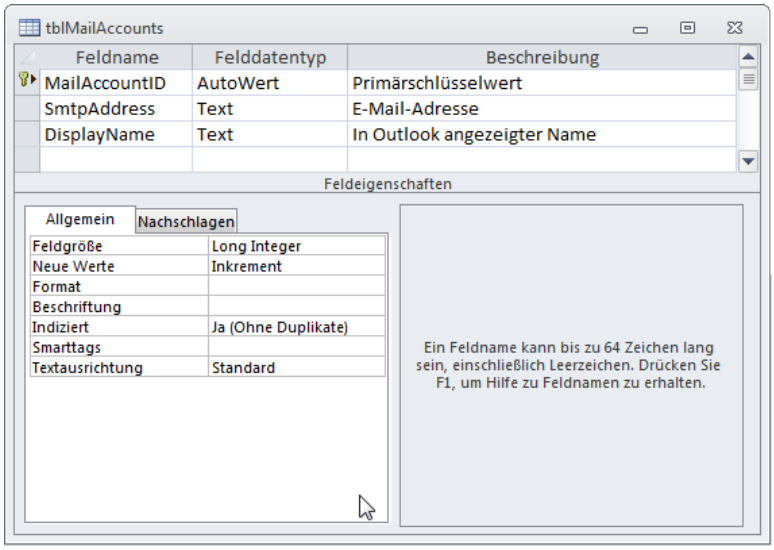


Abbildung 14.11: Tabelle zum Speichern der Account-Informationen

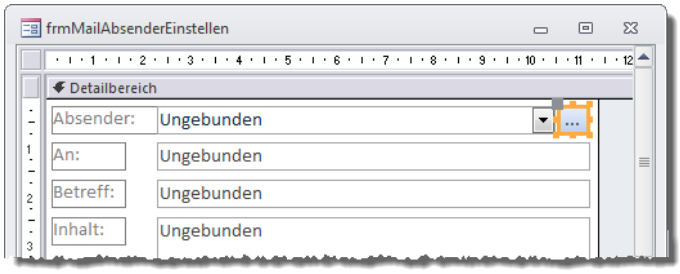


Abbildung 14.12: Schaltfläche zum Aktualisieren der Account-Liste

Diese soll die folgende Ereignisprozedur auslösen:

```
Private Sub cmdAktualisieren_Click()
    Dim objAccount As Outlook.Account
    Dim db As DAO.Database
    Set db = CurrentDb
    db.Execute "DELETE FROM tblMailAccounts", dbFailOnError
    For Each objAccount In GetMAPI.Accounts
        With objAccount
            db.Execute "INSERT INTO tblMailAccounts(SmtpAddress, DisplayName) VALUES('" _
                & .SmtpAddress & "', '" & .DisplayName & "')", dbFailOnError
        End With
```

```
    Next objAccount
    Me!cboVon.Requery
    Me!cboVon = Me!cboVon.ItemData(0)
    Set db = Nothing
End Sub
```

Die Prozedur leert zunächst die Tabelle *tblMailAccounts*, da ja nun ohnehin alle Einträge neu hinzugefügt werden. Dann durchläuft sie wiederum alle Elemente der *Accounts*-Auflistung des MAPI-Namespaces und trägt dabei für jeden Account einen neuen Datensatz in die Tabelle *tblMailAccounts* ein.

Schließlich aktualisiert sie die Datensatzherkunft des Kombinationsfeldes *cboVon* und stellt dieses auf den ersten Eintrag der Datensatzherkunft ein.

Apropos *cboVon*: Hier müssen wir natürlich noch die Datensatzherkunft auf eine Abfrage einstellen, welche die entsprechenden Daten aus der Tabelle *tblMailAccounts* liefert. Diese soll einen zusammengesetzten Wert aus den Feldern *DisplayName* und *SmtpAddress* liefern sowie das Feld *SmtpAddress* – das Ganze nach *DisplayName* sortiert (siehe Abbildung 14.13).

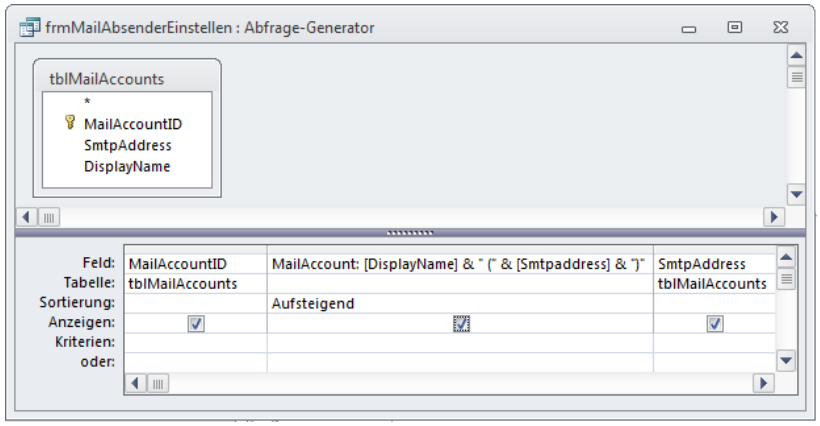


Abbildung 14.13: Datensatzherkunft für das Kombinationsfeld *cboVon*

Damit nur das zweite Feld angezeigt wird, stellen Sie die Eigenschaft *Spaltenanzahl* auf *3* und die Eigenschaft *Spaltenbreiten* auf *0cm;;0cm* ein – dies blendet die erste und dritte Spalte aus.

Die Prozedur *Form_Load* braucht sich nun nicht mehr um das Laden der Account-Liste zu kümmern, sondern soll nur noch den jeweils ersten Eintrag der Liste einstellen:

```
Private Sub Form_Load()
    Me!cboVon = Me!cboVon.ItemData(0)
    ...
End Sub
```

Schließlich müssen wir Ereignisprozedur, die durch die Schaltfläche *cmdAnzeigen* ausgelöst wird, noch etwas anpassen – und zwar so, dass Sie sich die E-Mail-Adresse für den zum Versenden benötigten Account aus der dritten Spalte des Kombinationsfeldes holt (also aus dem Feld *SmtpAddress*):

```
Private Sub cmdAnzeigen_Click()
    ...
    With objMailItem
        .SendUsingAccount = AccountByEMail(Me!cboVon.Column(2))
        ...
    End With
End Sub
```

14.2 To, Cc und Bcc

Neben den Eigenschaften *Von*, *An*, *Betreff* und *Inhalt* gibt es noch eine Reihe weiterer Eigenschaften, die für den Versand von E-Mails wichtig sein können: Beispielsweise die Eingabe weiterer Empfänger, welche die Mail als *Cc:* oder *Bcc:* erhalten sollen oder auch die Erweiterung der Haupt-Empfängerliste.

Die Beispiele dieses Abschnitts finden Sie im Formular *frmMailToCcBcc* der Beispieldatenbank.

14.2.1 Adressaten hinzufügen

Die erste dieser Eigenschaften haben Sie ja bereits kennen gelernt. Allerdings haben wir dieser Eigenschaft nur einen einzigen Adressaten hinzugefügt. Was aber, wenn Sie die E-Mail gleich mehreren Empfängern schicken möchten? Dann haben Sie zwei Möglichkeiten:

» Eingabe einer durch Semikola getrennten Liste der Empfänger oder

» Verwendung der Auflistung *Recipients* des *MailItem*-Objekts.

Gehen wir einmal vereinfacht davon aus, dass der Benutzer die Empfänger als Liste in das Textfeld *txtAn* einträgt (siehe Abbildung 14.14).

Dann können Sie den Inhalt des Textfeldes direkt in dieser Form an die Eigenschaft *To* des neuen *MailItem*-Objekts übergeben.

Alternativ wählen wir einmal die *Recipients*-Auflistung. Dazu lesen wir zunächst die im Textfeld *txtAn* befindliche, durch Semikola getrennte Liste der Empfänger mit der *Split*-Funktion in ein *String*-Array ein – auf diese Weise können wir die Einträge einfacher abarbeiten.

Die *Split*-Funktion teilt die als ersten Parameter übergebene Zeichenfolge an den Stellen auf, die das als zweiten Parameter enthaltene Trennzeichen enthalten.

Abbildung 14.14: Liste der Empfänger

Danach durchlaufen wir eine *For...Next*-Schleife über die Grenzen des neu erstellten Arrays *strAn()*.

In der Schleife fügt die Prozedur für jedes Element des Arrays mit der *Add*-Methode einen neuen Eintrag zur *Recipients*-Auflistung des *MailItem*-Objekts hinzu:

```
Private Sub cmdAnzeigen_Click()
    Dim objMailItem As Outlook.MailItem
    Dim objAccount As Outlook.Account
    Dim strAn() As String
    Dim i As Integer
    Set objMailItem = GetOutlook.CreateItem(olMailItem)
    With objMailItem
        .SendUsingAccount = AccountByEMail(Me!cboVon.Column(2))
        strAn() = Split(Me!txtAn, ";")
        For i = LBound(strAn) To UBound(strAn)
            .Recipients.Add strAn(i)
        Next i
        .Subject = Me!txtBetreff
        .Body = Me!txtInhalt
        .Display
    End With
End Sub
```

Wenn die Empfängerdaten für eine E-Mail ohnehin aus einer Tabelle kommen, ist die zweite Methode natürlich sinnvoller als erst eine Semikola-getrennte Liste zusammenzustellen.

Die Tabelle *tblKunden* enthält unter anderem die E-Mail-Adressen der Kunden (siehe Abbildung 14.15).

Wenn Sie nun eine nicht personalisierte E-Mail an alle Kunden senden möchten, müssen Sie einfach die E-Mail-Adressen aller Datensätze der Tabelle *tblKunden* an die Auflistung *Recipients* anhängen. Dies können Sie per Code so abbilden wie im Beispielformular für die Schaltfläche *cmdAnAlleKunden* geschehen:

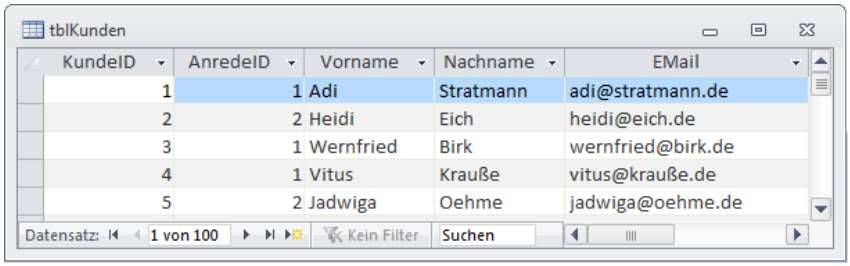


Abbildung 14.15: Kundendaten mit E-Mail-Adressen

```
Private Sub cmdAnAlleKunden_Click()
    Dim objMailItem As Outlook.MailItem
    Dim objAccount As Outlook.Account
    Dim db As DAO.Database
    Dim rst As DAO.Recordset
    Set objMailItem = GetOutlook.CreateItem(olMailItem)
    With objMailItem
        .SendUsingAccount = AccountByEMail(Me!cboVon.Column(2))
        Set db = CurrentDb
        Set rst = db.OpenRecordset("SELECT EMail FROM tblKunden", dbOpenDynaset)
        Do While Not rst.EOF
            .Recipients.Add rst!EMail
            rst.MoveNext
        Loop
        .Subject = Me!txtBetreff
        .Body = Me!txtInhalt
        .Display
    End With
End Sub
```

Die Prozedur öffnet ein Recordset auf Basis der Datensätze der Tabelle *tblKunden* und durchläuft alle Datensätze in einer *Do While*-Schleife – allerdings erst nach Erstellen des neuen *MailItem*-Objekts.

Innerhalb der Schleife fügt die Prozedur der *Recipients*-Auflistung jeweils den Wert des Feldes *EMail* der Tabelle *tblKunden* hinzu. Das Ergebnis sieht dann wie in Abbildung 14.16 aus.

Dies ist allerdings keine besonders gute Idee, wenn Sie es sich nicht mit Ihren Kunden verscherzen möchten: Es soll schließlich nicht jeder Kunde erfahren, wer die anderen Kunden sind. Aber zum Glück gibt es ja noch andere Möglichkeiten, E-Mails an den Mann/die Frau zu bringen.

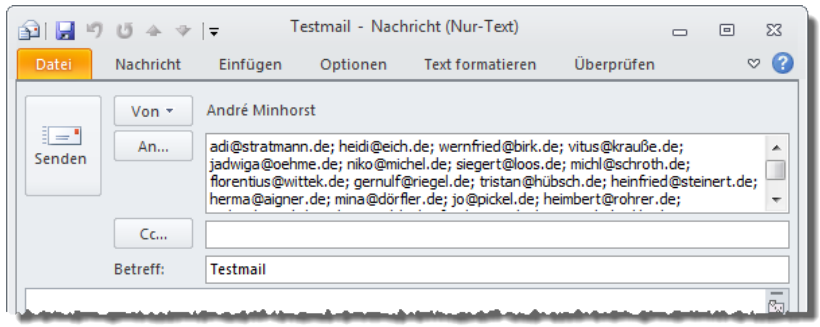


Abbildung 14.16: E-Mail an alle Einträge der Tabelle *tblKunden*

14.2.2 Mails per Cc:

Die erste Variante ist es, weitere Empfänger der Eigenschaft *Cc:* hinzuzufügen (Abkürzung für *Carbon Copy*). Diese sind jedoch auch für alle Empfänger sichtbar. *Cc:* ist auch eigentlich dazu gedacht, neben dem Hauptempfänger noch weitere Adressaten hinzuzufügen, die zwar nicht am Mailverkehr teilnehmen sollen beziehungsweise von denen keine Antwort erwartet wird, die aber dennoch Kenntnis vom Mailverkehr erhalten sollen.

Als Beispiel schreibe ich eine Mail bezüglich der Stornierung einer Rechnung an einen Kunden und setze die Buchhaltung auf *Cc:*, damit diese über den aktuellen Stand informiert ist.

Die Eigenschaft *Cc:* ist in der Benutzeroberfläche von Outlook standardmäßig sichtbar und wird genauso gefüllt wie das Feld *An*. Sie können die entsprechenden Empfänger also per Angabe einer Semikola-separierten Liste übergeben oder ... ja, welche Alternative gibt es noch?

Es gibt zwar jeweils eine Eigenschaft namens *To*, *Cc* oder *Bcc*, aber nur eine Auflistung namens *Recipients*. Sollten wir die Auflistung etwa nur für die direkten Empfänger verwenden können?

Nein: Es gibt eine einfache Einstellung, mit der Sie einen Eintrag zur Auflistung *Recipients* hinzufügen und diesen dann in einen *Cc:*-Empfänger umwandeln. Die Einträge, die wir im vorherigen Beispiel hinzugefügt haben, sind nämlich nur deshalb als *To:*-Empfänger klassifiziert worden, weil dies die Standardeinstellung beim Hinzufügen von Elementen zur *Recipients*-Auflistung ist.

Wir fügen für das folgende Beispiel eine weitere Schaltfläche namens *cmdAnAlleKundenCC* zum Formular *frmMailWeitereEigenschaften* hinzu. Die durch diese ausgelöste Ereignisprozedur deklariert zunächst ein Objekt namens *objRecipientCC* mit dem Objekttyp *Recipient*.

Danach fügen wir wie zuvor alle Kunden-E-Mails zur Auflistung *Recipients* hinzu, referenzieren den neuen Eintrag aber jeweils mit der Variablen *objRecipientCC*. Diese bietet eine Eigenschaft namens *Type* an, mit der Sie den Typ des Empfängers festlegen können. Es gibt die folgenden Werte:

15 Outlook-Kontakte im- und exportieren

Outlook-Kontakte sind sehr praktisch: Mittlerweile kann man diese automatisch mit allen möglichen anderen Programmen synchronisieren – beispielsweise über die Cloud mit seinem Smartphone. Nur eine automatische Synchronisierung zwischen Outlook-Kontakten und etwa den Kundendaten einer Access-Datenbank gibt es nicht. Grund genug, dass wir uns in diesem Kapitel anschauen, wie wir die Kontakte aus Outlook am besten mit unserer Datenbank abgleichen – und umgekehrt.

Hinweis

Dieses Kapitel baut auf den Techniken auf, die Sie im Kapitel »Outlook programmieren« ab Seite 339 finden. Die Beispiele dieses Kapitels setzen einen Verweis auf die Bibliothek *Microsoft Outlook x.0 Object Library* voraus. Außerdem sollten Sie dieser das Modul *mdlOutlook* hinzufügen, da dieses einige Funktionen enthält, die wir im Folgenden nutzen.

15.1 Die EntryID

Beim Synchronisieren von Adressen zwischen zwei verschiedenen Datenquellen ist es wichtig, in beiden Datenquellen eine gemeinsame Eigenschaft zu pflegen, die möglichst eindeutig ist. Auf Datenbankseite sind wir ja relativ flexibel, deshalb schauen wir uns an, was Outlook zu diesem Thema zu bieten hat. Wie in der Überschrift dieses Abschnitts bereits angedeutet, handelt es sich dabei im die Eigenschaft *EntryID*. Dies ist eine hexadezimale Zeichenkette mit 48 Zeichen:

```
00000000C7FFC59C3722AA4B9618A092CD6C9E8DE4BF2000
```

Früher war es einmal so, dass sich die *EntryID* geändert hat, wenn man ein Element von einem Ordner in einen anderen verschoben hat. Dieses Verhalten konnten wir mit den aktuellen Outlook-Versionen nicht mehr reproduzieren. Es macht ja auch keinen Sinn: Selbst der Ort eines Elements ist ja über die *EntryID* des *Folder*-Objekts eindeutig definiert. Wie können wir die *EntryID* für die Synchronisierung der Kontakte in Outlook mit den in unserer Datenbank gespeicherten Daten nutzen? Fest steht, dass wir in der entsprechenden Tabelle unserer Datenbank ein Feld namens *EntryID* einbauen müssen, um eine Referenz zum jeweiligen Outlook-Kontakt herzustellen. Danach kommt es darauf an, in welcher Richtung und wie die Synchronisierung erfolgt:

» Wenn Sie mit einer Reihe Outlook-Kontakte und einer leeren Datenbank beginnen, dann lesen Sie einfach die Kontante samt *EntryID* in die Zieltabelle ein.

» Wenn Sie mit einer vollen Tabelle und einem leeren Kontakte-Ordner starten, übertragen Sie die Kontakte nach Outlook und ermitteln nach dem Anlegen die *EntryID*, die Sie wiederum in die Tabelle zurückschreiben.

» Sind sowohl Tabelle als auch die Outlook-Kontakte gefüllt, müssen Sie tatsächlich mehr oder weniger automatisch die Daten der beiden Quellen abgleichen. Finden Sie einen Kontakt, der in einer Quelle vorhanden ist und in der anderen nicht, gehen Sie so vor wie in den beiden vorherigen Schritten beschrieben. Stoßen Sie auf einen Kontakt, der in beiden Datenquellen vorliegt, übernehmen Sie die *EntryID* des Outlook-Kontakts in die Tabelle der Datenbank und gleichen gegebenenfalls unterschiedliche Informationen noch ab.

15.2 Struktur eines Kontakts

Ein Kontakt-Element kann in Outlook eine ganze Reihe Eigenschaften aufnehmen. Eindeutig sind der Name, die Firma und die Position zu erkennen (siehe Abbildung 15.1). Die E-Mail lässt sich mit den Eigenschaften *E-Mail*, *E-Mail 2* und *E-Mail 3* abbilden. Bei den Telefonnummern gibt es dann vier Möglichkeiten, die Sie mit den zusätzlichen Informationen *Assistent*, *Geschäftlich*, *Geschäftlich 2*, *Fax geschäftl.*, *Rückmeldung*, *Auto*, *Firma*, *Privat* und so weiter versehen können.

Auch lassen sich drei verschiedene Adressen einrichten – dies unter den Bezeichnungen *Geschäftlich*, *Privat* und *Weitere*. Auf den ersten Blick trägt man hier einfach die Adressdaten untereinander in ein Textfeld ein. Es gibt jedoch noch einen Dialog namens *Adresse überprüfen*, mit dem Sie die Interpretation von Outlook prüfen können (siehe Abbildung 15.2).

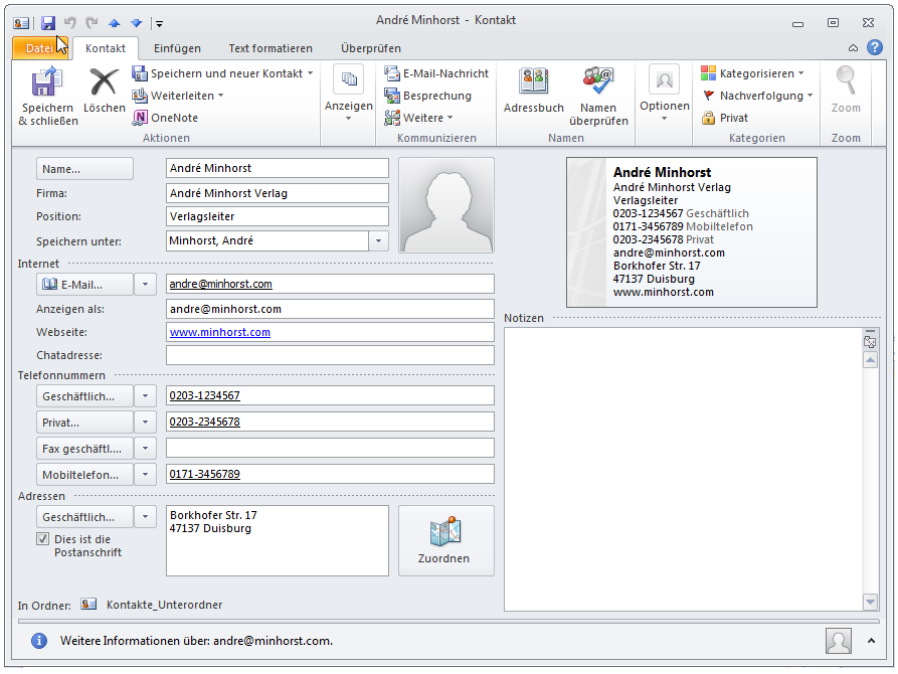


Abbildung 15.1: Beispiel für einen Kontakt in Outlook

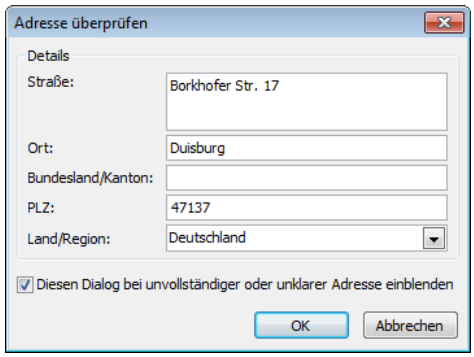


Abbildung 15.2: Kontrollieren einer Adresse

Es sieht also so aus, als ob Outlook die Adresse nicht nur einfach in einem Textfeld ablegt, sondern diese nach der Eingabe erkennt und in einzelnen Feldern speichert, wie sie beispielsweise im Dialog *Adresse überprüfen* erscheinen.

Wenn Sie im Ribbon den Befehl *Kontakt|Anzeigen|Details* anklicken, zeigt das Kontakt-Fenster weitere Informationen wie etwa zum Job oder zu privaten Bereichen an (siehe Abbildung 15.3).

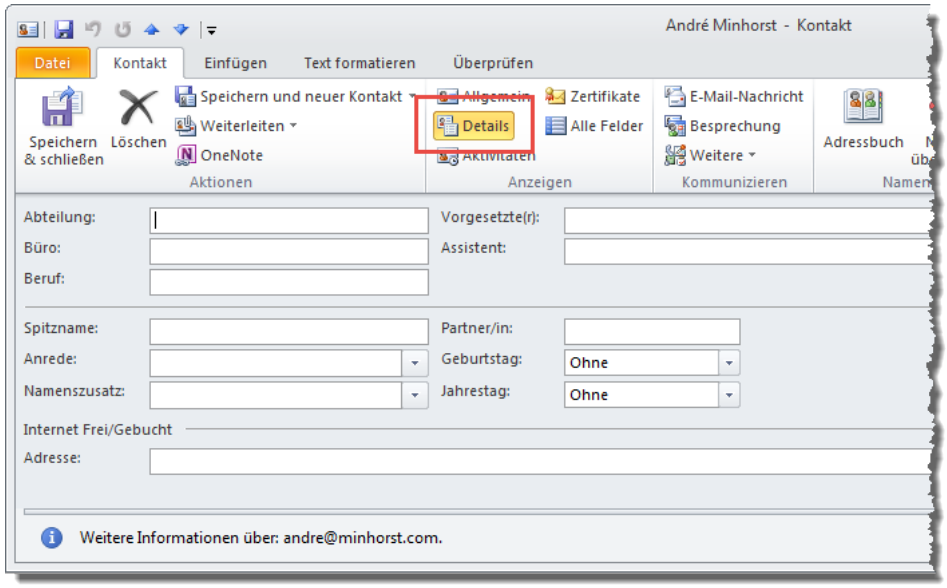


Abbildung 15.3: Weitere Informationen

Damit wissen wir nun schon einmal, welche Vielfalt an Eigenschaften uns erwartet, wenn wir uns an das Objektmodell des Contact-Objekts begeben.

15.3 Datenmodell für die Kontakte

Es wird fast nie der Fall auftreten, dass Sie die Daten aus Outlook genauso in die Zieldatenbank übernehmen können – zumindest dann nicht, wenn schon eine Zieldatenbank vorliegt. Und selbst wenn nicht, lautet die Frage, ob Sie ein Datenmodell erstellen möchten, das genau an die Gegebenheiten eines Outlook-Kontakts angepasst ist. In der Regel sehen Tabellen mit den Daten von Personen, Mitarbeitern, Kunden et cetera doch immer ähnlich aus. Also verwenden wir auch einfach Tabellen, welche die Kontakte von Outlook ähnlich wie Outlook verwalten.

15.4 Outlook-Kontakte einlesen

Beim Einlesen von Daten aus Outlook ist die schwierigste Aufgabe, dafür zu sorgen, dass die Daten gut gepflegt sind. Das betrifft weniger das Problem, dass einige Felder nicht ausgefüllt sein können, sondern vielmehr das Problem falsch ausgefüllter Eigenschaften eines Kontakts.

Dazu bietet Outlook natürlich eine Menge Möglichkeiten, denn allein für die Telefonnummern haben wir mehr als zehn verschiedene Bezeichnungen, die wir den jeweiligen Telefonnummern zuordnen können. Wenn Sie dies in eine Datenbank übertragen wollen, können Sie eigentlich nur die Telefonnummern plus Bezeichnungen in je eine eigene Tabelle übertragen. Wir schauen uns aber zunächst an, wie wir überhaupt auf Outlook-Kontakte zugreifen, und kümmern uns dann um das Übertragen in eine Access-Datenbank.

15.4.1 Kontakt einlesen

Starten wir doch gleich mit dem Einlesen eines Kontakts, wobei wir ein kleines Formular zum Experimentieren erstellen. Dieses heißt *frmKontakteEinlesen* und sieht wie in Abbildung 15.4 aus. Die Steuerelemente oben kennen Sie bereits aus dem Kapitel »Mails mit Outlook senden, empfangen und verarbeiten« ab Seite 371, die Schaltfläche darunter soll einfach alle Kontakte des oben ausgewählten Ordners durchlaufen und ein paar Eigenschaften der Kontakte ausgeben.

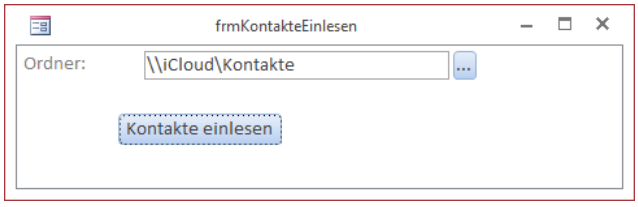


Abbildung 15.4: Formular zum Einlesen von Kontakten

Die Prozedur referenziert mit der benutzerdefinierten Funktion *GetFolderByPath* das *Folder*-Objekt zum angegebenen Pfad. Danach durchläuft sie in einer *For Each*-Schleife alle Elemente

dieses Ordners und gibt verschiedene Informationen zu den eingelesenen *ContactItem*-Objekten im Direktfenster des VBA-Editors aus – darunter auch die *EntryID*:

```
Private Sub cmdKontakteEinlesen_Click()
    Dim objContactItem As Outlook.ContactItem
    Dim objFolder As Outlook.Folder
    Set objFolder = GetFolderByPath(Me!txtOrdner)
    For Each objContactItem In objFolder.Items
        With objContactItem
            Debug.Print "Ordner: " & Me!txtOrdner
            Debug.Print "EntryID: " & .EntryID
            Debug.Print "Vorname: " & .FirstName
            Debug.Print "Nachname: " & .LastName
            Debug.Print "Strasse: " & .MailingAddressStreet
            Debug.Print "PLZ: " & .MailingAddressPostalCode
            Debug.Print "Ort: " & .MailingAddressCity
        End With
    Next objContactItem
End Sub
```

Die meisten Eigenschaften eines *ContactItem*-Elements haben einfache Datentypen wir *String*, *Date* oder *Boolean*. Es gibt ein paar Ausnahmen:

» *Attachments*: Mit dieser Auflistung können Sie Dokumente an ein *ContactItem*-Element anfügen und diese auch wieder auslesen.

» *Gender* (Geschlecht): Erwartet ein Element einer Auflistung als Wert (*olFemale – 1*, *olMale – 2* oder *olUnspecified – 0*).

» *Importance* (Wichtigkeit): Erwartet ebenfalls eine Konstante.

» *SelectedMailingAdress*: Gibt die für Lieferungen ausgewählte Adresse an (*olBusiness – 2*, *olHome – 1*, *olNone – 0*, *olOther – 3*)

15.4.2 Datenmodell für die Kontakte

Nachfolgend wollen wir die Daten der Kontakte aus Outlook in einen Satz von Access-Tabellen einlesen. Wir verwenden insgesamt vier Tabellen:

» *tblKunden*: Erfasst die grundlegenden Kundendaten.

» *tblAnreden*: Lookup-Tabelle für das Feld *Anrede* der Tabelle *tblKunden*.

» *tblKommunikationsdetails*: Enthält Telefonnummern und E-Mail-Adressen.

» *tblKommunikationsarten*: Liefert die Typen für die Tabelle *tblKommunikationsdetails*.

Die erste Tabelle namens *tblKontakte* enthält *AnredeID*, *Vorname* und *Nachname*, die *EntryID*, zwei Sätze von Adressdaten (Privatadresse und Firmenadresse) sowie das Geburtsdatum und ein Anlage-Feld zum Speichern eines Bildes (siehe Abbildung 15.5).

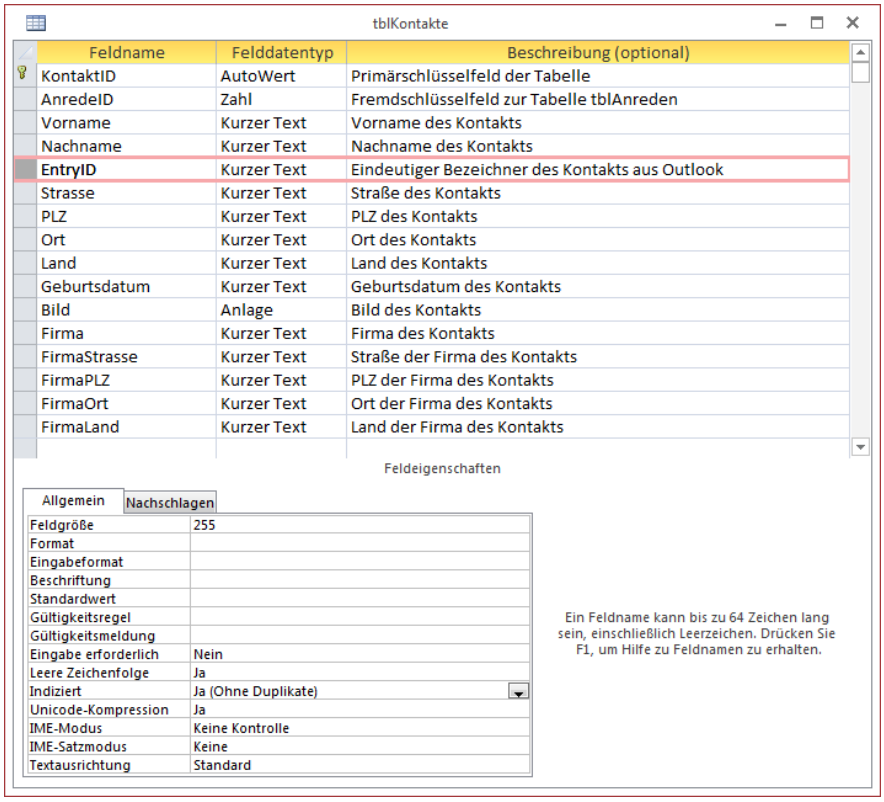


Abbildung 15.5: Entwurf der Tabelle *tblKontakte*

Nun werden Sie das Fehlen jeglicher E-Mail- und Telefonnummern-Felder feststellen – und das, obwohl Outlook sehr viele verschiedene Eigenschaften für diesen Zweck anbietet: nämlich *Business2TelephoneNumber*, *BusinessFaxNumber*, *BusinessTelephoneNumber*, *CallbackTelephoneNumber*, *CarTelephoneNumber*, *CompanyMainTelephoneNumber*, *Email1Address*, *Email2Address*, *Email3Address*, *Home2TelephoneNumber*, *HomeFaxNumber*, *HomeTelephoneNumber*, *IMAddress*, *MobileTelephoneNumber*, *OtherFaxNumber*, *OtherTelephoneNumber*, *RadioTelephoneNumber*, *TTYTDDTelephoneNumber* und *PrimaryTelephoneNumber*.

Da wohl kaum für jeden Kontakt alle Felder auch Werte enthalten, verwenden wir zum Speichern diese Informationen die oben bereits erwähnten Tabellen *tblKommunikationsdetails* und *tblKommunikationsarten*. Die Tabelle *tblKommunikationsarten* erfasst die Bezeichnung der Kommunikationsart in Outlook, also etwa *Business2TelephoneNumber*.

Damit können wir Informationen auch wieder so nach Outlook zurückschreiben, wie wir diese eingelesen haben – oder auch einmal neue Werte übermitteln. Das Feld *Kommunikationsart* kann der Benutzer selbst mit dem gewünschten Wert füllen. Dieser soll im Formular als Bezeichner der jeweiligen Kommunikationsart dienen (siehe Abbildung 15.6).

Das Feld *KommunikationsartOutlook* haben wir mit einem eindeutigen Index versehen, damit es garantiert jeden Wert nur einmal enthält.

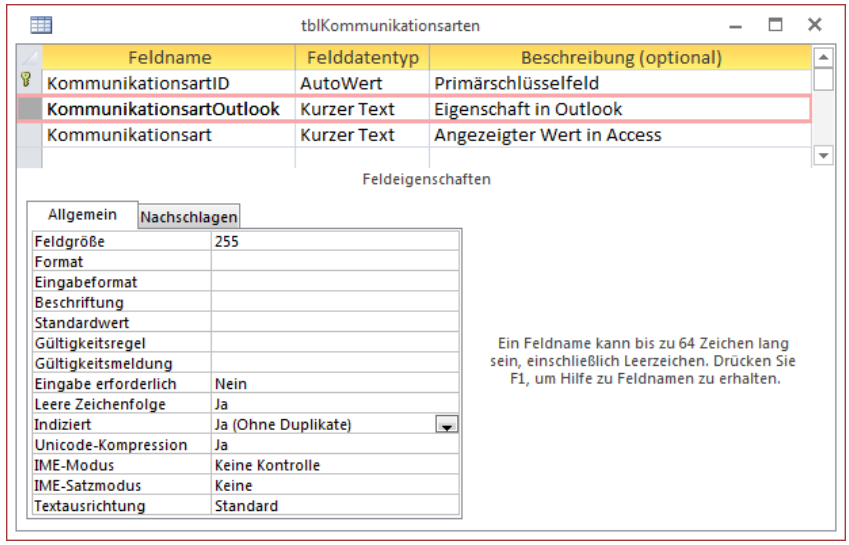


Abbildung 15.6: Die Tabelle *tblKommunikationsarten* im Entwurf

Wie bringen wir aber nun die Kontakte und die Kommunikationsarten zusammen? Da wir theoretisch für jede Kombination aus Kontakt und Kommunikationsart einen Wert anlegen müssen, verwenden wir eine m:n-Beziehung und benötigen somit noch eine entsprechende Verknüpfungstabelle.

Diese Tabelle heißt *tblKommunikationsdetails* und sieht wie in Abbildung 15.7 aus. Das Feld Kommunikationsdetail nimmt die jeweilige Telefonnummer, E-Mail-Adresse et cetera auf.

Die beiden Felder *KontaktID* und *KommunikationsartID* sind Fremdschlüsselfelder, welche für die eigentliche Verknüpfung zwischen den beiden Tabellen *tblKontakte* und *tblKommunikationsarten* sorgen. Für diese beiden Felder haben wir einen zusammengesetzten, eindeutigen Index festgelegt – immerhin soll jeder Kontakt ja jede Kommunikationsart nur einmal enthalten.

Fehlt noch die Tabelle *tblAnreden*, die wir aber nur in der Übersicht des Datenmodells darstellen möchten (siehe Abbildung 15.8). Diese Abbildung verdeutliche die Zusammenhänge. Nun wollen wir eine Prozedur erstellen, welche die Kontakte aus Outlook einliest und die Informationen in die entsprechenden Tabellen einliest.

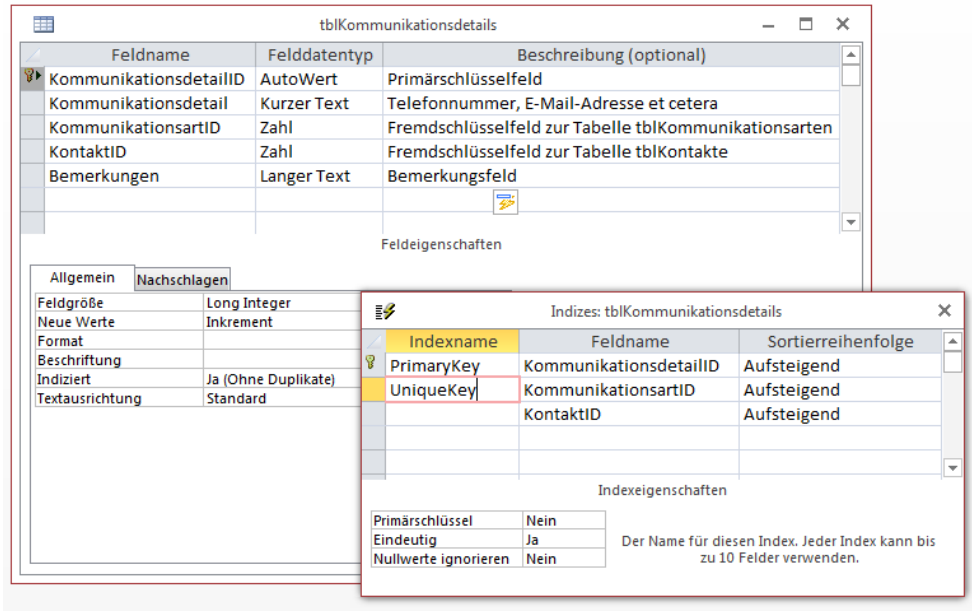


Abbildung 15.7: Die Tabelle *tblKommunikationsdetails* in der Entwurfsansicht

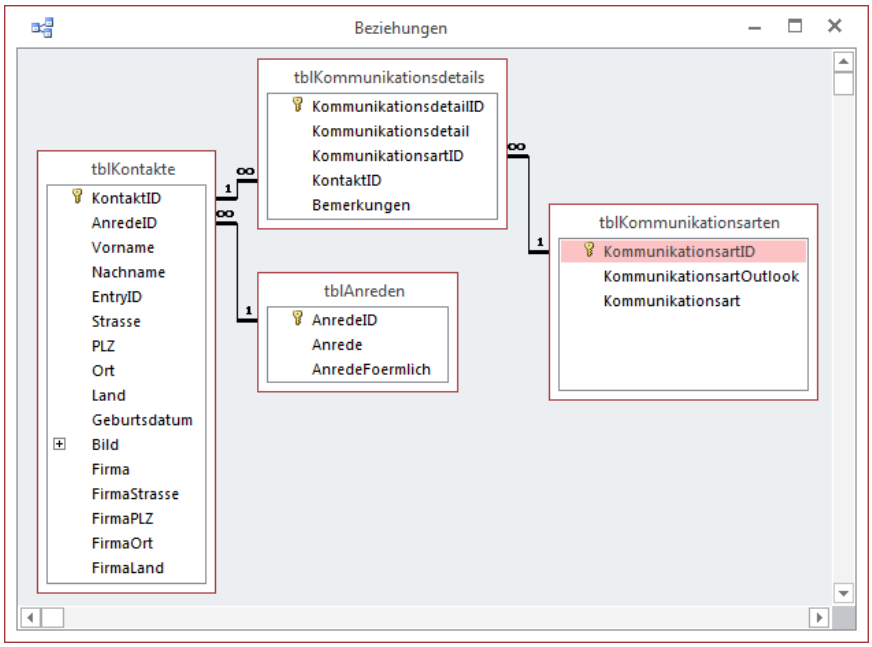


Abbildung 15.8: Datenmodell der Kontaktverwaltung

15.4.3 Basisdaten schreiben

Die folgende Prozedur wird durch eine neue Schaltfläche im Formular *frmKontakteEinlesen* ausgelöst. Sie erstellt zunächst ein Recordset auf Basis der Tabelle *tblKontake*. Danach liest sie das *Folder*-Objekt zu dem im Textfeld *txtOrdner* angegebenen Outlook-Ordner ein und speichert den Verweis in der Variablen *objFolder*. Dieses Objekt stellt die Kontakte über die *Items*-Auflistung bereits, welche die Prozedur nachfolgend in einer *For Each*-Schleife durchläuft.

Innerhalb dieser Schleife erstellt die Prozedur zunächst einen neuen Datensatz in der Tabelle *tblKontakte*. Gleich das erste Feld dieser Tabelle heißt *AnredeID*, weshalb wir die entsprechende Information aus dem Outlook-Kontakt herausholen wollen. Dies misslingt jedoch: Ein Kontakt enthält keine Informationen über die Anrede. Allerdings gibt es ein Feld namens *Gender* (Geschlecht). Das sollte für unsere Zwecke ausreichen. Hat *Gender* also den Wert der Konstanten *olFemale*, handelt es sich um einen weiblichen Kontakt und wir können den Primärschlüsselwert der Tabelle *tblAnreden* für den Datensatz mit der Anrede *Frau* in das Feld *AnredeID* eintragen. Ähnlich geschieht dies, wenn *Gender* den Wert *olMale* enthält. Der dritte mögliche Wert heißt *olUnspecified*. In diesem Fall wollen wir die Anrede *Firma* verwenden beziehungsweise den entsprechenden Primärschlüsselwert der Tabelle *tblAnreden* (siehe Abbildung 15.9) in das Feld *AnredeID* eintragen.

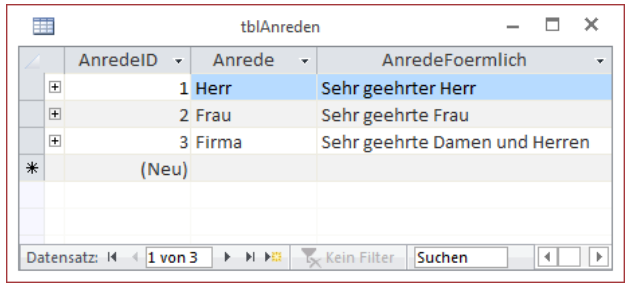


Abbildung 15.9: Die Tabelle *tblAnreden* in der Datenblattansicht

Anschließend füllt die Prozedur die übrigen Felder der Tabelle *tblKontakte* mit den Informationen aus den entsprechenden Eigenschaften des aktuellen *ContactItem*-Objekts. Bevor die Prozedur den Datensatz mit der *Update*-Methode speichert, liest sie noch den Wert des Feldes *KontaktID* ein, also den automatisch erstellten Primärschlüsselwert des neuen Datensatzes, und speichert diesen in der Variablen *lngKontaktID*. Diese benötigen wir für die folgenden Aufrufe einer weiteren Prozedur namens *KommunikationsdetailAnlegen*, die sich um Telefonnummern, E-Mail-Adressen et cetera kümmert.

```
Private Sub cmdKontakteImportieren_Click()
    Dim objContactItem As Outlook.ContactItem
    Dim objFolder As Outlook.Folder
    Dim db As dao.Database
```

16 Mit Outlook-Terminen arbeiten

Outlook-Termine sind ein weiteres Outlook-Objekt, das der eine oder andere gern mit seiner Datenbank synchronisieren möchte. Anwendungsfälle gibt es genug: Beispielsweise möchte Sie vielleicht einfach die Termine von Outlook in eine Bericht in einer Form ausgeben, die Outlook nicht beherrscht. Oder Sie legen mit Ihrer Bestellverwaltung Liefertermine fest, die Sie gern in Outlook übernehmen möchten. Natürlich gibt es auch „Einmal-Anwendungen", mit denen Sie beispielsweise Ihre in Access vorliegende Geburtstagsliste in Outlook-Termine umwandeln wollen (natürlich in Serientermine!).

Hinweis

Dieses Kapitel baut auf den Techniken auf, die Sie weiter vorn in *****Kapitel Outlook programmieren* finden. Das Testen der Beispiele dieses Kapitels setzt voraus, dass das VBA-Projekt Ihrer Datenbank einen Verweis auf die Bibliothek *Microsoft Outlook x.0 Object Library* enthält. Außerdem sollten Sie dieser das Modul *mdlOutlook* hinzufügen, da dieses einige Funktionen enthält, die wir im Folgenden nutzen.

16.1 Termine auslesen

Wir beginnen wieder mit dem Weg von Outlook nach Access und schauen uns an, wie wir Outlook-Termine auslesen können. Den Standard-Ordner für Termine können Sie mit der *GetDefaultFolder*-Funktion und dem Parameterwert *olFolderCalendar* auslesen. Allerdings ist heutzutage ja längst nicht mehr sichergestellt, dass dieser Termine-Ordner auch tatsächlich die relevanten Termine enthält. Vielleicht greifen Sie ja auch längst auf einen Termin-Kalender in der Cloud zu oder nutzen eine andere Möglichkeit.

Also verwenden wir auch hier wieder die individuelle Auswahl des Ordners, aus dem wir die Termine beziehen wollen. Einem neuen Formular namens *frmTermineEinlesen* fügen wir deshalb das Textfeld *txtOrdner* sowie die Schaltfläche *cmdOrdnerAuswaehlen* hinzu, die wir schon in den vorherigen Kapiteln erläutert haben. Außerdem erhält das Formular eine Schaltfläche namens *cmdTermineEinlesen*, mit der wir die Termine des gewählten Ordners einlesen möchten.

Die Termine wollen wir für den ersten Überblick in einem Listenfeld namens *lstTermine* einfügen. Dabei sollen nur die *EntryID* als nicht sichtbarer Wert in der ersten Spalte und der Betreff des Termins (*Subject*) in der einzigen sichtbaren Spalte landen. Das Listenfeld müssen wir daher noch ein wenig anpassen: Stellen Sie die Eigenschaft *Spaltenanzahl* auf den Wert *2* und *Spaltenbreiten* auf *0cm* ein. Außerdem verwenden wir keine Tabelle oder Abfrage als Datensatzherkunft, als müssen wir die Eigenschaft *Herkunftstyp* auf den Eintrag *Wertliste* einstellen. Das Formular sieht dann im Entwurf wie in Abbildung 16.1 aus.

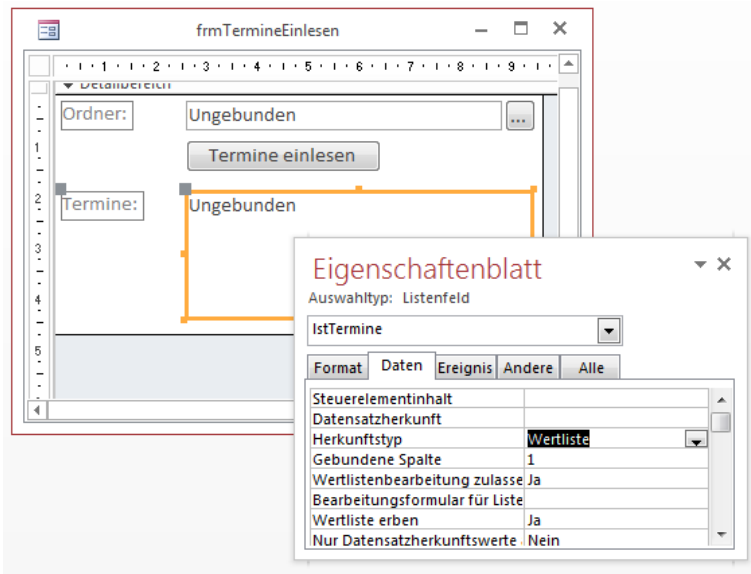


Abbildung 16.1: Listenfeld zur Anzeige von Terminen

Die Prozedur zum Einlesen der Termine referenziert das passende *Folder*-Objekt und durchläuft dann alle darin enthaltenen Elemente. Diese landen in der Variablen *objAppointmentItem*, deren Eigenschaften *EntryID* und *Subject* dann mit einem trennenden Semikolon als ein Element in das Listenfeld eingefügt werden:

```
Private Sub cmdTermineEinlesen_Click()
    Dim objAppointmentItem As Outlook.AppointmentItem
    Dim objFolder As Outlook.Folder
    Set objFolder = GetFolderByPath(Me!txtOrdner)
    For Each objAppointmentItem In objFolder.Items
        With objAppointmentItem
            Me!lstTermine.AddItem .EntryID & ";" & .Subject
        End With
    Next objAppointmentItem
End Sub
```

Das Ergebnis sieht wie in Abbildung 16.2 aus. Dort finden sich aber weder oben in der Liste noch ganz unten die aktuellen Termine meines Outlook-Kalenders. Woran kann das liegen? Die Ursache ist eine Limitierung der Zeichenanzahl der Eigenschaft *RowSource* des Listenfeldes auf ca. 32.000 Zeichen.

Der betroffene Outlook-Kalender enthält eine durchaus beträchtliche Menge Einträge, die dann wohl nicht komplett in der Wertliste eines Listenfeldes Platz finden.

Wir könnten nun direkt die Termine in eine Tabelle schreiben und diese dann problemlos im Listenfeld anzeigen, aber warum sollen wir diese natürliche Beschränkung des Platzes nicht nutzen, um etwa nur die neuesten Daten anzuzeigen?

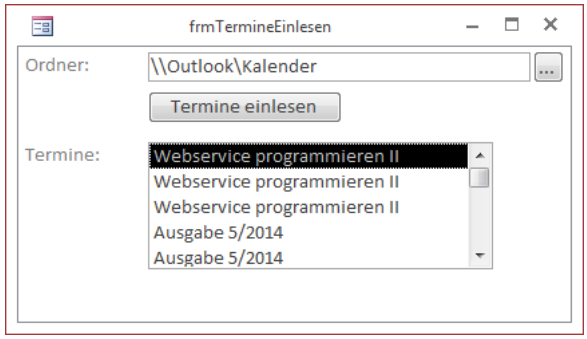


Abbildung 16.2: Liste mit einigen Terminen

16.1.1 Aktuellste Termine einlesen

Es gibt nämlich durchaus Möglichkeiten, nur eine bestimmte Menge der vorliegenden Termine zu durchlaufen. Die erste ist eine Sortierung nach einer bestimmten Eigenschaft, in diesem Fall dem Startdatum des Termins (*Start*). Wir fügen eine neue Schaltfläche zum Formular hinzu und hinterlegen dafür die folgende Prozedur:

```
Private Sub cmdErsteZehnEinlesen_Click()
    Dim objAppointmentItem As Outlook.AppointmentItem
    Dim objFolder As Outlook.Folder
    Dim objItems As Outlook.Items
    Dim i As Integer
    Set objFolder = GetFolderByPath(Me!txtOrdner)
    Set objItems = objFolder.Items
    objItems.Sort "[Start]", True
    Me!lstTermine.RowSource = ""
    For i = 1 To 10
        Set objAppointmentItem = objItems.Item(i)
        With objAppointmentItem
            Me!lstTermine.AddItem .EntryID & ";" & .Start & ": " & .Subject
        End With
    Next i
End Sub
```

Diese führt ein neues Objekt ein, nämlich *objItems* mit dem Datentyp *Items*. Dieses bietet genauso Zugriff auf die Elemente wie die Eigenschaft *Items* des *Folder*-Objekts. Allerdings erhalten

Sie hier noch Zugriff auf Methoden wie die hier verwendete *Sort*-Methode zum Sortieren der Einträge. Der erste Parameter gibt das zu sortierende Feld an, der zweite, ob die Einträge absteigend sortiert werden sollen (*True*) oder aufsteigend (*False*).

Da wir in diesem Fall nur die ersten zehn Einträge zum Listenfeld hinzufügen möchten, durchlaufen wir eine *For...Next*-Schleife mit den Werten von *1* bis *10* und geben jeweils die Daten des mit *Item(i)* ermittelten Elements im Listenfeld aus. Dabei erhält die angezeigte Spalte neben dem Betreff auch noch Datum und Zeit, damit wir sichergehen können, dass es sich um die aktuellsten Termine handelt (siehe Abbildung 16.3).

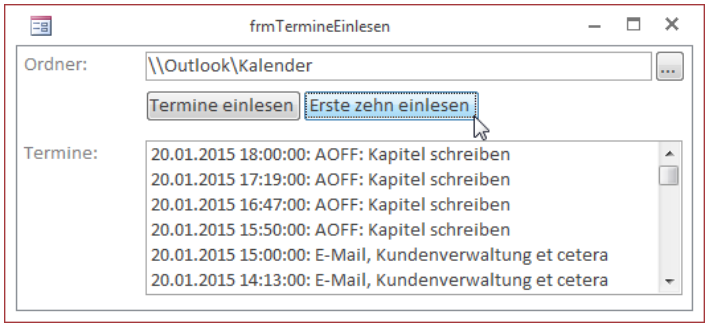


Abbildung 16.3: Anzeige der neuesten zehn Termine in absteigender Reihenfolge

16.1.2 Termine nach Datum filtern

Nun wollen wir beispielsweise nur die Termine eines bestimmten Zeitraums in Tagen anzeigen. Dazu fügen wir dem Formular zwei Textfelder namens *txtDatumVon* und *txtDatumBis* hinzu sowie eine Schaltfläche namens *cmdTermineNachDatum*. Diese löst die folgende Prozedur aus:

```
Private Sub cmdTermineNachDatum_Click()
    Dim objAppointmentItem As Outlook.AppointmentItem
    Dim objFolder As Outlook.Folder
    Dim objItems As Outlook.Items
    Dim i As Integer
    Dim strFind As String
    If IsNull(Me!txtDatumVon) Then
        MsgBox "Geben Sie ein Startdatum ein."
        Me!txtDatumVon.SetFocus
        Exit Sub
    End If
    Set objFolder = GetFolderByPath(Me!txtOrdner)
    Set objItems = objFolder.Items
    Me!lstTermine.RowSource = ""
```

```
    strFind = "[Start] > '" & Me!txtDatumVon & " 00:00'"
    If Not IsNull(Me!txtDatumBis) Then
        strFind = strFind & " AND [Start] < '" & Me!txtDatumBis & " 00:00'"
    End If
    Set objAppointmentItem = objItems.Find(strFind)
    Do While Not objAppointmentItem Is Nothing And i <= 10
        With objAppointmentItem
            i = i + 1
            Me!lstTermine.AddItem .EntryID & ";" & .Start & ": " & .Subject
        End With
        Set objAppointmentItem = objItems.FindNext
    Loop
End Sub
```

Die Prozedur prüft, ob ein Startdatum angegeben wurde und weist den Benutzer gegebenenfalls auf die fehlende Eingabe hin. Danach stellt die Prozedur den ersten Teil des Suchausdrucks zusammen, der etwa so aussehen könnte:

```
[Start] > '20.01.2015 00:00'
```

Sollte das Textfeld *txtDatumBis* auch einen Wert enthalten, wird dieser durch den *AND*-Operator getrennt an das erste Kriterium angefügt. Danach sucht die Prozedur mit der *Find*-Funktion nach dem ersten Element, das dem Kriterium genügt, und übergibt es an das Objekt *objAppointmentItem*.

Damit steigen wir in eine *Do While*-Schleife ein, die erst beendet wird, wenn das Objekt *objAppointmentItem* leer ist oder die Schleife mehr als zehn Mal durchlaufen wurde (*i <= 10*). Enthält *objAppointmentItem* ein Objekt, trägt die Prozedur dessen Daten wie gewohnt in das Listenfeld ein. Danach ruft die Prozedur immer am Ende der *Do While*-Schleife die Funktion *FindNext* auf, welche das nächste Objekt mit dem zuvor durch die *Find*-Funktion festgelegten Kriterium sucht. Dies geschieht so lange, bis das Abbruchkriterium erfüllt ist.

Wie Abbildung 16.4 zeigt, liefert das aber ganz und gar nicht die erwarteten Ergebnisse, sondern eine Reihe Geburtstage – die immerhin alle zumindest im Januar liegen ...

Die Frage ist hier grundsätzlich: Wieso tauchen hier nun plötzlich die Geburtstagseinträge auf, die zuvor gar nicht angezeigt wurden?

Anscheinend läuft hier etwas mit den Datumskriterien falsch. Dies war das Ergebnis für die Suche nach den Terminen des aktuellen Tages, also etwa mit folgendem Kriterium:

```
[Start] > '20/01/2015 00:00' AND [Start] < '21/01/2015 00:00'
```

Der folgende Ausdruck lieferte alle Termine des 19.1.2015, aber auch nicht die des 20.1.2015:

```
[Start] >= '19.01.2015 00:00'
```

Allerdings folgten auch hier eine Menge Geburtstagstermine.

Dieser hier liefert immerhin alle Termine des 18. und 19. Januar 2015 – wieder gefolgt von Geburtstagen:

```
[Start] >= '18.01.2015 00:00' AND [Start] < '20.01.2015 00:00'
```

Das Problem an dieser Stelle sind auch gar nicht ausschließlich die Such-Kriterien, sondern die Einstellungen des *Items*-Objekts: Dieses sollte nämlich sortiert werden.

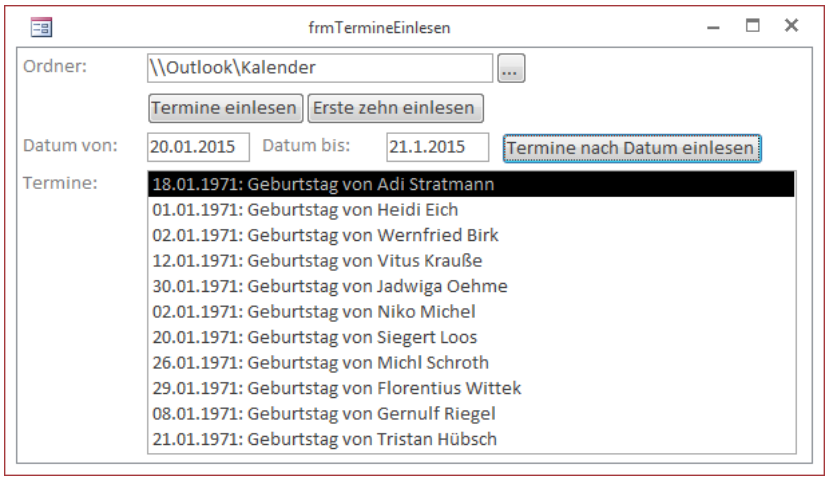


Abbildung 16.4: Falsches Ergebnis für dieses Suchkriterium

16.1.3 Filtern per Restrict

Es gibt noch eine zweite Möglichkeit, auf Termine mit bestimmten Eigenschaften zuzugreifen: *Restrict*. Dabei handelt es sich um eine Eigenschaft des *Items*-Objekts. Wir haben dem Formular noch eine weitere Schaltfläche hinzugefügt, die ebenfalls die in den beiden Textfeldern *txtDatumVon* und *txtDatumBis* nutzt.

Diese Schaltfläche löst die folgende Prozedur aus, hier zunächst der Prozedurkopf und die Deklarationszeilen:

```
Private Sub cmdTermineNachDatumRestrict_Click()
    Dim objAppointmentItem As Outlook.AppointmentItem
    Dim objFolder As Outlook.Folder
    Dim objItems As Outlook.Items
    Dim strRestriction As String
    Dim strDatumVon As String
    Dim strDatumBis As String
```

Die Prozedur prüft wiederum, ob *txtDatumVon* überhaupt einen Datumswert enthält und fordert den Benutzer gegebenenfalls auf, einen solchen einzutragen:

```
If IsNull(Me!txtDatumVon) Then
    MsgBox "Geben Sie ein Startdatum ein."
    Me!txtDatumVon.SetFocus
    Exit Sub
End If
```

Die Prozedur liest dann den Zielordner in die Variable *objFolder* ein und referenziert die enthaltenen Elemente mit der Objektvariablen *objItems*:

```
Set objFolder = GetFolderByPath(Me!txtOrdner)
Set objItems = objFolder.Items
```

Nun folgen zwei wichtige Schritte: Die Prozedur stellt die Sortierung nach der Eigenschaft *Start* ein, also nach Datum und Uhrzeit des Beginns des Termins:

```
objItems.Sort "[Start]"
```

Danach legen wir fest, dass Serientermine berücksichtigt werden sollen. Wenn Sie *IncludeRecurrences* nicht auf *True* einstellen, aber vorher die Sortierung aktivieren und dann mit *Restrict* filtern, liefert dies alle Termine – inklusive der Serientermine:

```
objItems.IncludeRecurrences = True
```

Nun leert die Prozedur das Listenfeld:

```
Me!lstTermine.RowSource = ""
```

Anschließend folgt der erste Teil der Restriktion. Die Prozedur schreibt das Startdatum in die Variable *strDatumVon* und formatiert diese im Datums-Kurzformat. Außerdem hängt sie die Uhrzeit *12:00 AM* an:

```
strDatumVon = Format(Me!txtDatumVon, "ddddd") & " 12:00 AM"
```

Dieser Wert landet dann in der ersten Bedingung – die Endzeit muss größer als die Startzeit sein:

```
strRestriction = "[End] > '" & strDatumVon & "'"
```

Danach prüft die Routine das Enddatum. Ist dieses leer, verwendet sie das Startdatum als Enddatum, was dazu führt, dass nur die Termine eines Tages ermittelt werden.

Anderenfalls nutzt die Routine das entsprechend formatierte und erweiterte Enddatum für das Enddatum:

```
If IsNull(Me!txtDatumBis) Then
    strDatumBis = Format(Me!txtDatumVon, "ddddd") & " 11:59 PM"
Else
```

```
            strDatumBis = Format(Me!txtDatumBis, "ddddd") & " 11:59 PM"
        End If
```

Der Ausdruck aus *strDatumBis* landet dann im zweiten Teil der Bedingung:

```
        strRestriction = strRestriction & " AND [Start] <= '" & strDatumBis & "'"
```

Dann wendet die Prozedur die Bedingung an und durchläuft alle gefundenen Termine in einer *For Each*-Schleife. In dieser legt sie wie in den vorherigen Beispielen die Einträge im Listenfeld an:

```
        Set objItems = objItems.Restrict(strRestriction)
        For Each objAppointmentItem In objItems
            With objAppointmentItem
                Me!lstTermine.AddItem .EntryID & ";" & .Start & ": " & .Subject
            End With
        Next objAppointmentItem
    End Sub
```

Der Filterausdruck für nur einen Tag sieht dann etwa so aus:

```
[End] >= '19.01.2015 12:00 AM' AND [Start] < '19.01.2015 11:59 PM'
```

Wenn Sie einen Zeitraum von etwa zehn Tagen berücksichtigen wollen, verwendet die Prozedur den folgenden Filterausdruck:

```
[End] > '10.01.2015 12:00 AM' AND [Start] <= '20.01.2015 11:59 PM'
```

16.2 Termine in eine Tabelle importieren

Bevor wie Termine aus Outlook in eine Tabelle importieren, müssen wir uns für die gewünschten Felder entscheiden, die wir speichern wollen. Dafür ist ausschlaggebend, ob wir die Termine einfach nur in die Datenbank holen und dort etwa in einem Bericht ausgeben möchten oder ob wir diese gegebenenfalls anpassen und wieder zurück nach Outlook schreiben möchten. Im ersten Fall reicht es sicher aus, die Zeitangaben zum Termin sowie den Betreff und gegebenenfalls noch weitere Informationen einzulesen. Im zweiten Fall benötigen wir natürlich alle Informationen, die für ein Wiederherstellen des Termins in Outlook nötig sind.

Außerdem stellt sich die Frage, ob Sie nur einfache Termine einlesen möchten oder auch Terminserien. Letzteres erhöht den Grad der Komplexität nochmals.

Wir wollen uns eine Termin-Tabelle erstellen, die zunächst alle einfachen Werte eines Outlook-Termins aufnimmt. Das können Sie in Handarbeit erledigen, indem Sie sich den Objektkatalog des Objekts *AppointmentItem* neben eine Tabelle in der Entwurfsansicht legen. Oder Sie verwenden die folgende Prozedur dazu:

```
Public Sub TabelleErstellen(strTabellenname As String, strPKFeld As String)
    Dim db As DAO.Database
    Dim tdf As DAO.TableDef
    Dim fld As DAO.Field
    Dim objItemProperty As Outlook.ItemProperty
    Dim objAppointmentItem As Outlook.AppointmentItem
    Set db = CurrentDb
    On Error Resume Next
    db.TableDefs.Delete strTabellenname
    On Error GoTo 0
    Set tdf = db.CreateTableDef(strTabellenname)
    Set fld = tdf.CreateField(strPKFeld, dbLong)
    fld.Attributes = dbAutoIncrField
    tdf.Fields.Append fld
    Set objAppointmentItem = GetOutlook.CreateItem(olAppointmentItem)
    For Each objItemProperty In objAppointmentItem.ItemProperties
        Set fld = Nothing
        Select Case TypeName(objItemProperty.Value)
            Case "String"
                Set fld = tdf.CreateField(objItemProperty.Name, dbText, 255)
                fld.AllowZeroLength = True
            Case "Long"
                Set fld = tdf.CreateField(objItemProperty.Name, dbLong)
            Case "Date"
                Set fld = tdf.CreateField(objItemProperty.Name, dbDate)
            Case "Boolean"
                Set fld = tdf.CreateField(objItemProperty.Name, dbBoolean)
            Case Else
                Debug.Print objItemProperty.Name, TypeName(objItemProperty.Value)
        End Select
        If Not fld Is Nothing Then
            If fld.Name = "Body" Then
                fld.Type = dbMemo
            End If
           Debug.Print "    rst!" & objItemProperty.Name & " = ." & objItemProperty.Name
            tdf.Fields.Append fld
        End If
    Next objItemProperty
    db.TableDefs.Append tdf
    Application.RefreshDatabaseWindow
End Sub
```

Die Prozedur erwartet die folgenden Parameter:

» *strTabellenname*: Name der zu erstellenden Tabelle

» *strPKFeld*: Name des Autowert-Feldes für diese Tabelle

Ein Aufruf sieht etwa so aus:

```
TabelleErstellen "tblTermineVonOutlook", "TerminID", olAppointmentItem
```

Damit erstellt die Prozedur zunächst eine neue Tabelle mit dem angegebenen Namen aus dem ersten Parameter sowie ein Autowert-Feld mit dem Namen des zweiten Parameters. Danach erstellt die Prozedur mit der Prozedur *CreateItem* des mit *GetOutlook* referenzierten *Outlook.Application*-Objekts ein neues Objekt des Typs *olMailItem*. Danach durchläuft die Prozedur alle Elemente der *ItemProperties*-Auflistung des *AppointmentItem*-Objekts.

Per *Select Case* prüft sie den Datentyp der in der Eigenschaft *objItemProperty* enthaltenen Daten und legt je nach Datentyp ein neues Feld mit dem entsprechenden Datentyp und dem Namen der Eigenschaft in der Tabelle an.

Nach der *Select Case*-Anweisung fügt die Prozedur das neue Feld aus der Variablen *fld* noch zur *Fields*-Auflistung des *TableDef*-Objekts der Tabelle hinzu. Sollten Datentypen auftauchen, die in der *Select Case*-Anweisung noch nicht berücksichtigt sind, werden diese per *Debug.Print* im Direktfenster ausgegeben.

Nach dem ersten Test haben wir festgestellt, dass das Feld *Body* mit dem Datentyp *dbMemo* statt *dbText* ausgestattet werden muss – dies haben wir durch eine individuelle Zuweisung erledigt.

Außerdem gibt die Prozedur, um später beim Schreiben der Routine zum Importieren der Termine Tipparbeit zu sparen, zu jeder Eigenschaft eine Zeile aus, welche den Inhalt der Eigenschaft einem Feld des Recordsets *rst* zuweist – dies sieht etwa so aus und wird weiter unten klarer werden:

```
rst!EntryID = .EntryID
```

Schließlich fügt die Prozedur die neue Tabelle zur *TableDefs*-Auflistung hinzu und aktualisiert den Navigationsbereich. Diese sieht nun in der Entwurfsansicht wie in Abbildung 16.5 aus.

Es fehlen nur noch die Eigenschaften, die Objekte oder Auflistungen enthalten – diese wollen wir jedoch vorerst außen vor lassen.

16.2.1 Termine in die Tabelle schreiben

Nun erstellen wir eine Prozedur, die durch einen Klick auf die Schaltfläche *cmdTermineImportieren* ausgelöst wird. Sie erstellt ein Recordset auf Basis der Zieltabelle *tblTermineVonOutlook* und holt dann mit der Funktion *GetAppointmentItems* eine Liste der Termine, die in dem durch die beiden Textfelder *txtDatumVon* und *txtDatumBis* eingegrenzten Zeitraum gelten.

```
Dim rst As dao.Recordset
Dim lngKontaktID As Long
Set db = CurrentDb
Set rst = db.OpenRecordset("SELECT * FROM tblKontakte", dbOpenDynaset)
Set objFolder = GetFolderByPath(Me!txtOrdner)
For Each objContactItem In objFolder.Items
    With objContactItem
        rst.AddNew
        Select Case .Gender
            Case olFemale
                rst!AnredeID = DLookup("AnredeID", "tblAnreden", "Anrede='Frau'")
            Case olMale
                rst!AnredeID = DLookup("AnredeID", "tblAnreden", "Anrede='Herr'")
            Case Else
                rst!AnredeID = DLookup("AnredeID", "tblAnreden", "Anrede='Firma'")
        End Select
        rst!Vorname = .FirstName
        rst!Nachname = .LastName
        rst!EntryID = .EntryID
        rst!Strasse = .HomeAddressStreet
        rst!PLZ = .HomeAddressPostalCode
        rst!Ort = .HomeAddressCity
        rst!Land = .HomeAddressCountry
        rst!Geburtsdatum = .Birthday
        rst!Firma = .CompanyName
        rst!FirmaStrasse = .BusinessAddressStreet
        rst!FirmaPLZ = .BusinessAddressPostalCode
        rst!FirmaOrt = .BusinessAddressCity
        rst!FirmaLand = .BusinessAddressCountry
        lngKontaktID = rst!KontaktID
        rst.Update
        KommunikationsdetailAnlegen db, lngKontaktID, .Business2TelephoneNumber, _
            "BusinessTelephoneNumber"
        KommunikationsdetailAnlegen db, lngKontaktID, .BusinessFaxNumber, _
            "BusinessFaxNumber"
        KommunikationsdetailAnlegen db, lngKontaktID, .BusinessTelephoneNumber, _
            "BusinessTelephoneNumber"
        ... weitere Aufrufe der Prozedur KommunikationsdetailAnlegen
        KommunikationsdetailAnlegen db, lngKontaktID, .PrimaryTelephoneNumber, _
            "PrimaryTelephoneNumber"
    End With
```

17 PowerPoint programmieren

Neben Word und Excel ist PowerPoint die dritte Office-Anwendung, mit der Sie Dokumente anlegen und bearbeiten können. Diesmal liegt der Fokus jedoch darin, die erstellten Dokumente als Präsentation zu nutzen. Da die Programmierung von PowerPoint mit dem Hintergrund des Datenaustauschs mit Access wohl eher seltener vorkommen wird, also es mit Excel oder Word der Fall ist, beschränken wir uns in diesem Buch auf ein Kapitel, das den Umgang mit PowerPoint-Dokumenten erläutert und ein Beispiel liefert, wie Sie etwa einen kleinen Produktkatalog auf Basis einiger Beispieldaten erstellen.

17.1 Vorbereitungen

Voraussetzung für eine einfache Programmierung von PowerPoint unter der Verwendung von IntelliSense und der Konstanten von PowerPoint ist das Einbinden der entsprechenden Bibliothek in das VBA-Projekt der aktuellen Datenbank (siehe Abbildung 17.1).

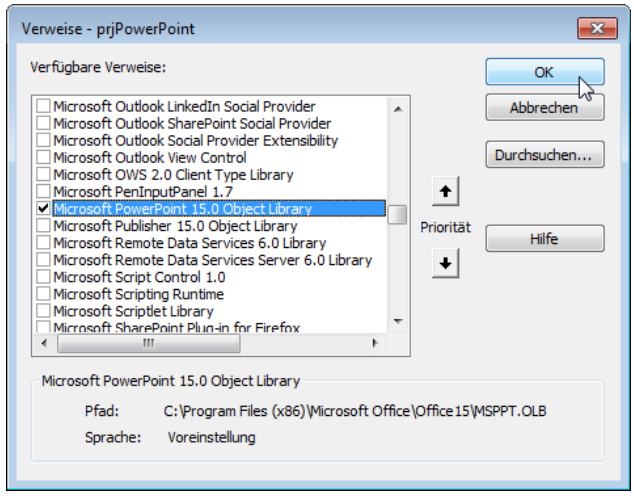


Abbildung 17.1: Verweis auf PowerPoint einbinden

17.2 Aufbau von PowerPoint-Dokumenten

Ein PowerPoint-Dokument hat den Objekttyp *Presentation*. Ein *Presentation*-Objekt kann ein oder mehrere Folien enthalten, die den Objekttyp *Slide* aufweisen. Daneben gibt es einige Folien, die als Vorlage für die übrigen Folien dienen. Um diese einzusehen, müssen Sie in

die Folienmaster-Ansicht wechseln. Dazu aktivieren Sie zunächst mit dem Ribbon-Eintrag *Ansicht|Präsentationsansichten|Gliederungsansicht* die Ansicht aus Abbildung 17.2. Dort finden Sie unten eine Schaltfläche, die beim Überfahren mit der Maus den Text *Normal* anzeigt. Klicken Sie bei gedrückter *Umschalt*-Taste auf diese Schaltfläche, aktivieren Sie die Folienmaster-Ansicht.

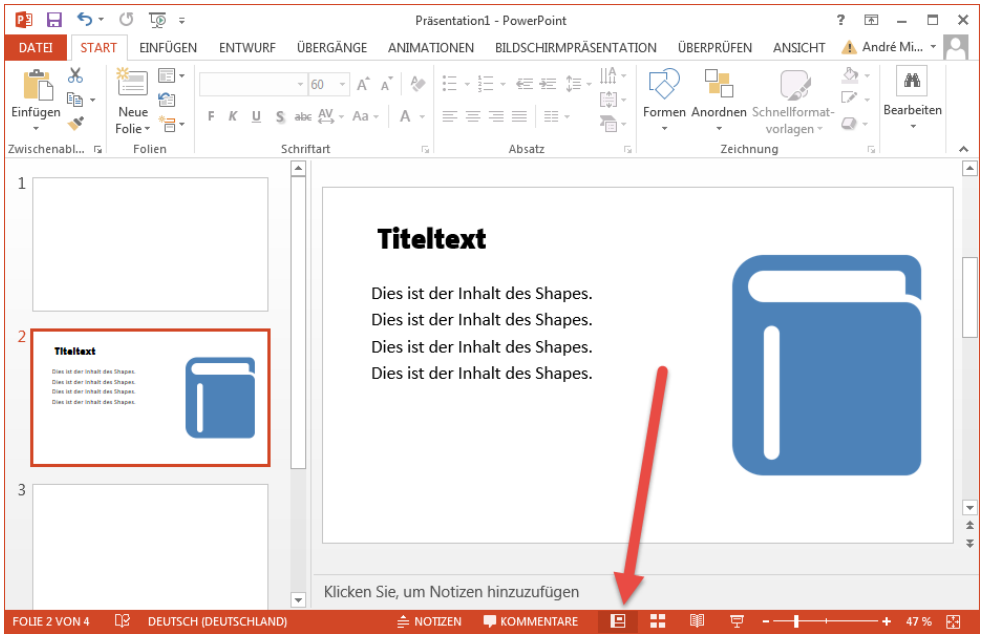


Abbildung 17.2: Umstellen auf die Folienmaster-Ansicht

Die Titelmaster-Ansicht sieht unter PowerPoint 2013 wie in Abbildung 17.3 aus. Wichtig ist zunächst der linke Bereich, der den Titelmaster sowie die Layouts des Titelmasters enthält. Der Titelmaster ist etwas breiter, die Layouts sind etwas kleiner und nach rechts eingerückt.

Wenn Sie nun beispielsweise möchten, dass ein bestimmtes Element auf allen Seiten der Präsentation erscheint, dann fügen Sie es auf der Titelmaster-Folie ein – zum Beispiel einen Hintergrund für die Präsentation. Es wird dann zunächst auf allen Layouts sichtbar und somit auch auf allen Folien, die Sie auf Basis der Layouts entwickeln.

Wenn Sie hier noch weiter nach unten scrollen, finden Sie noch weitere Titelmaster-Folien mit den entsprechenen untergeordneten Layouts vor. Wenn Sie beispielsweise die Schriftart der Überschriften anpassen möchten und dies einheitlich geschehen soll, dann stellen Sie die Schriftart für das obere Element mit der Überschrift im Titelmaster ein. Diese wird dann zunächst auch für alle Layouts geändert.

Das ist ein nettes Konzept, wenn man Folien von Hand erstellen muss. Wir schauen uns PowerPoint in diesem Kapitel allerdings nur an, um automatisiert von Access beispielsweise ei-

nen Artikelkatalog zu erstellen. Theoretisch könnten wir dann auch direkt alle Elemente per VBA auf die Folien bringen. Allerdings werden wir gleich feststellen, dass man beim Erstellen einer Folie immer ein Layout angeben muss – auch wenn es sich dabei um eine leere Seite handelt. Es kann also nicht schaden, sich grundsätzlich anzusehen, ob man die Grundlage für den Katalog nicht doch von Hand legt.

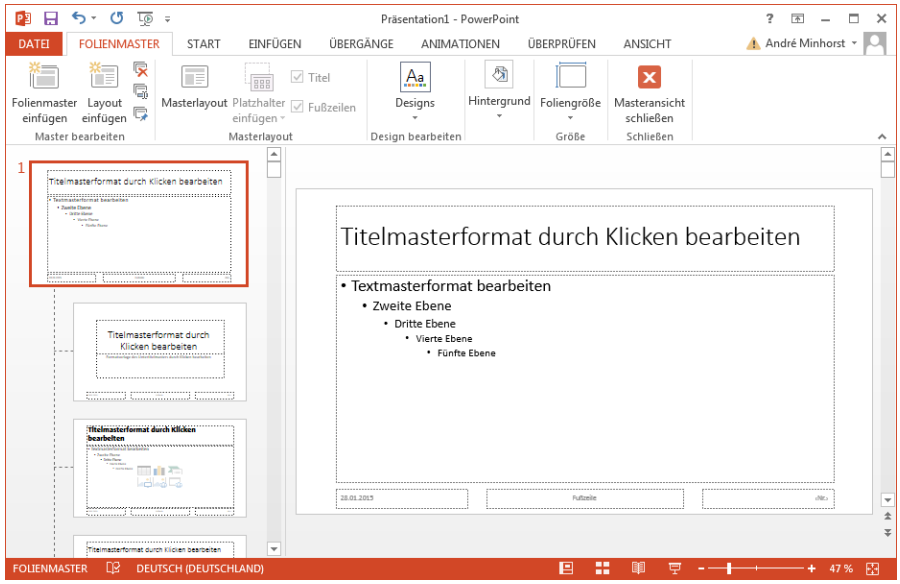


Abbildung 17.3: Die Titelmaster-Ansicht

Neben den Titelmaster-Folien und den Layout-Folien gibt es nun noch die Folien, welche die tatsächlichen Inhalte aufnehmen sollen. Diese legen Sie, wie oben erwähnt, immer im Kontext einer der Layoutfolien an – mehr dazu weiter unten.

17.3 PowerPoint programmieren

Die folgenden Abschnitte zeigen, wie Sie über das Objektmodell von PowerPoint neue Präsentationen anlegen, Folien hinzufügen und diese mit Inhalten füllen.

17.3.1 PowerPoint starten

Eine Instanz von PowerPoint starten Sie genau wie Outlook. Ist PowerPoint schon gestartet, greifen Sie auf die laufende Instanz zu. Dies ist also anders als bei Word oder Excel, wo Sie auch eine eigene Instanz erzeugen können. Die folgende Prozedur startet PowerPoint und blendet das Fenster ein:

```
Public Sub PowerPointStarten()
    Dim objPowerPoint As PowerPoint.Application
    Set objPowerPoint = New PowerPoint.Application
    objPowerPoint.Visible = True
End Sub
```

17.3.2 Präsentation erstellen

Die folgende Prozedur ist eine Erweiterung der vorherigen. Sie erstellt nach dem Instanzieren von PowerPoint ein neues *Presentation*-Objekt und referenziert dieses mit der Variablen *objPresentation*. Danach blendet sie das PowerPoint-Fenster ein und passt mit der Eigenschaft *ViewType* des *ActiveWindow*-Objekts die Ansicht an.

```
Public Sub PowerPointStartenUndPraesentationAnlegen()
    Dim objPowerPoint As PowerPoint.Application
    Dim objPresentation As PowerPoint.Presentation
    Set objPowerPoint = New PowerPoint.Application
    Set objPresentation = objPowerPoint.Presentations.Add
    objPowerPoint.Visible = True
    objPowerPoint.ActiveWindow.ViewType = ppViewSlide
End Sub
```

17.3.3 Zugriff auf geöffnete Präsentation

Damit wir in den folgenden Beispielen nicht jedesmal von vorn beginnen müssen, sondern auf das aktuell geöffnete und angezeigte Dokument zugreifen können, verwenden wir die folgende Funktion. Diese referenziert zunächst die geöffnete PowerPoint-Instanz und dann das darin angezeigte *Presentation*-Objekt. Wird eines von beiden nicht angetroffen, erscheint eine entsprechende Meldung. Die Funktion liefert ein Objekt des Typs *Presentation* zurück:

```
Public Function CurrentPresentation() As PowerPoint.Presentation
    Dim objPowerPoint As PowerPoint.Application
    Dim objPresentation As PowerPoint.Presentation
    On Error Resume Next
    Set objPowerPoint = GetObject(, "Powerpoint.Application")
    If Err.Number = 429 Then
        MsgBox "Es ist keine PowerPoint-Instanz geöffnet, " _
            & "die mit CurrentPresentation referenziert werden kann."
        Exit Function
    Else
        Set objPresentation = objPowerPoint.ActivePresentation
        If Not Err.Number = 0 Then
```

```
            MsgBox "Es ist keine PowerPoint-Präsentation in der aktuellen " _
                & "PowerPoint-Instanz geöffnet."
            Exit Function
        Else
            Set CurrentPresentation = objPresentation
        End If
    End If
End Function
```

17.4 Neue Folie einfügen

Über die Benutzeroberfläche fügen Sie eine neue Folie wie in Abbildung 17.4 ein. Den Typ der Folie wählen Sie über das entsprechende Menü aus, die Position legen Sie fest, indem Sie die bestehende Folie markieren, hinter der die neue Folie eingefügt werden soll.

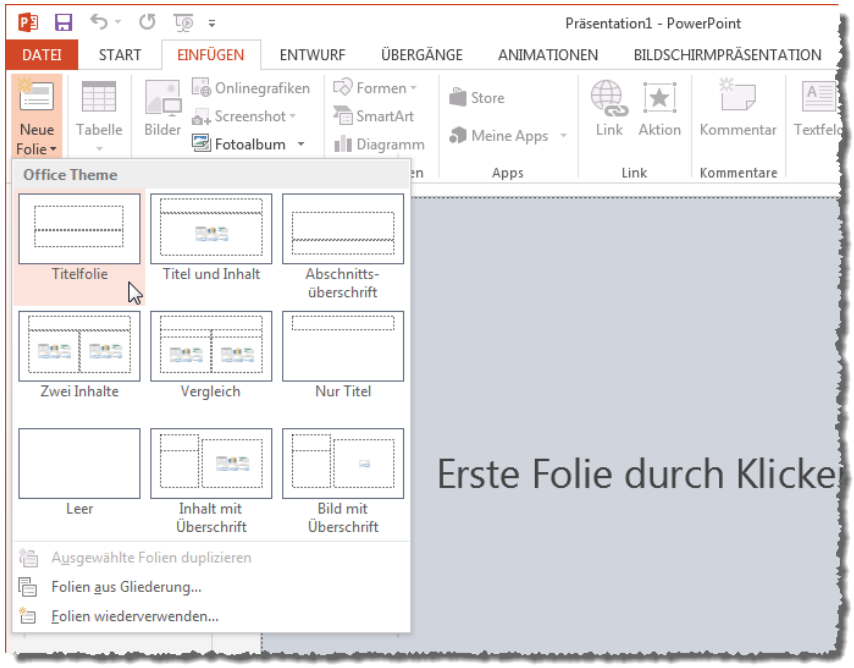


Abbildung 17.4: Einfügen einer neuen Folie über die Benutzeroberfläche

Die unter *OfficeTheme* abgebildeten Layouts können Sie per VBA in einer *For Each*-Schleife über die *CustomLayouts*-Auflistung des *SlideMaster*-Objekts der aktuellen Präsentation durchlaufen. Hier nutzen wir dies und geben die Namen der Layouts im Direktfenster des VBA-Editors aus:

```
Public Sub CustomLayoutsAusgeben()
    Dim objCustomLayout As CustomLayout
    For Each objCustomLayout In CurrentPresentation.SlideMaster.CustomLayouts
        Debug.Print objCustomLayout.Index, Replace(objCustomLayout.Name, vbLf, "")
    Next objCustomLayout
End Sub
```

Das Ergebnis sieht so aus und entspricht den Elementen aus dem Ribbon-Menü:

```
1           Titelfolie
2           Titel und Inhalt
3           Abschnitts-überschrift
4           Zwei Inhalte
5           Vergleich
6           Nur Titel
7           Leer
8           Inhalt mit Überschrift
9           Bild mit Überschrift
10          Titel und vertikaler Text
11          Vertikaler Titel und Text
```

Warum schauen wir so genau auf die Layout-Arten? Weil wir beim Erstellen einer Folie mit der *AddSlide*-Methode als zweiten Parameter ein *CustomLayout*-Objekt übergeben müssen. Der erste Parameter der *AddSlide*-Methode ist der Index, welcher die Position angibt. Der Index beginnt bei *1* und kann für die erste Folie auch keinen größeren Wert annehmen:

```
Public Sub TitelSlideEinfuegen()
    Dim objCustomLayout As PowerPoint.CustomLayout
    Set objCustomLayout = CurrentPresentation.SlideMaster.CustomLayouts(1)
    CurrentPresentation.Slides.AddSlide 1, objCustomLayout
End Sub
```

Wenn Sie eine weitere Folie hinten anfügen möchten, rufen Sie die folgende Prozedur auf. Diese zählt mit der *Count*-Eigenschaft der *Slides*-Auflistung die Anzahl der bisher vorhandenen Folien und verwendet diesen Wert plus eins als Index. Außerdem wählen wir hier als Design den Index *2*, was dem Design *Titel und Inhalt* entspricht:

```
Public Sub SlideHintenAnfuegen()
    Dim objCustomLayout As PowerPoint.CustomLayout
    Dim intCount As Integer
    intCount = CurrentPresentation.Slides.Count
    Set objCustomLayout = CurrentPresentation.SlideMaster.CustomLayouts(2)
    CurrentPresentation.Slides.AddSlide intCount + 1, objCustomLayout
End Sub
```

Sollten Sie einen Wert als Index angeben, der größer als die Anzahl der bisherigen Folien plus eins ist, löst dies einen Fehler aus. Wenn Sie eine Folie vor einer anderen Folie einfügen möchten, müssen Sie als Index einfach nur den Index der Folie angeben, vor der Sie die neue Folie einfügen wollen. Die neue Folie sieht nun wie in Abbildung 17.5 aus.

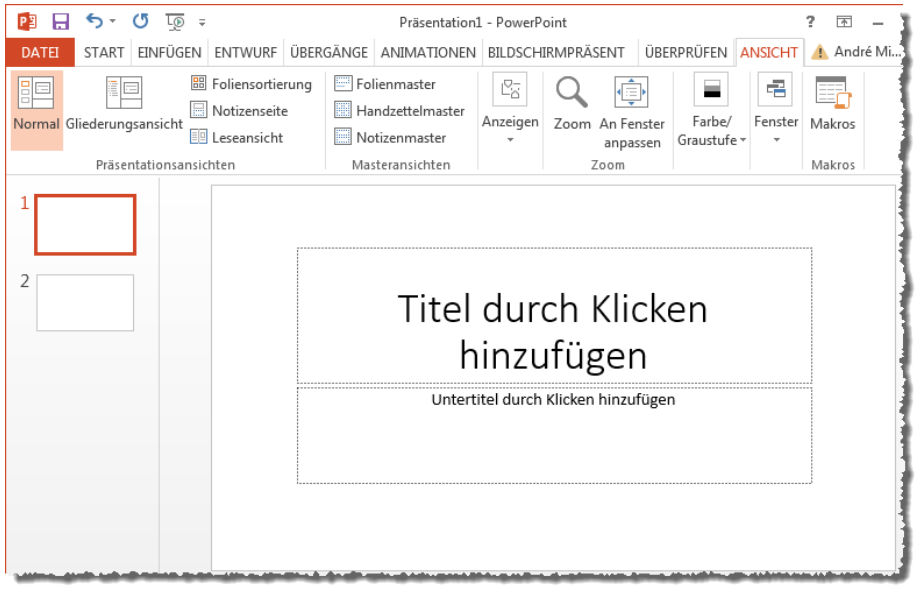


Abbildung 17.5: Zwei Bereiche einer neuen Folie

17.5 Design-Elemente ansprechen

Nun kümmern wir uns um die einzelnen Elemente einer neu hinzufügten Folie – vorausgesetzt, Sie haben nicht den Typ *7* (*Leer*) für die Ermittlung des *CustomLayout*-Objekts verwendet. Zunächst wollen wir herausfinden, wie wir diese Elemente referenzieren und ihnen Inhalte zuweisen können. Dazu durchlaufen wir zwei Schleifen, mit denen wir eine PowerPoint-Präsentation analysieren wollen, die zwei Folien mit je zwei *Shape*-Elementen aufweist:

```
Public Sub ShapesDurchlaufen()
    Dim objSlide As PowerPoint.Slide
    Dim objShape As PowerPoint.Shape
    Dim i As Integer
    Dim j As Integer
    For i = 1 To CurrentPresentation.Slides.Count
        Debug.Print "Slide " & i
        Set objSlide = CurrentPresentation.Slides(i)
```

```
        For j = 1 To objSlide.Shapes.Count
            Debug.Print "  Shape " & j
            Set objShape = objSlide.Shapes(j)
            Debug.Print "    " & objShape.Name
        Next j
    Next i
End Sub
```

Das Ergebnis sieht wie folgt aus:

```
Slide 1
  Shape 1
    Title 1
  Shape 2
    Subtitle 2
Slide 2
  Shape 1
    Title 1
  Shape 2
    Content Placeholder 2
```

17.6 Shape-Elemente füllen

Wie können wir die Shape-Elemente nun mit Inhalten füllen – also beispielsweise mit Texten oder Bildern?

17.6.1 Shape-Element mit Text füllen

Wenn Sie dem oberen *Shape*-Objekt zur Anzeige des Titels den Text *Titeltext* zuweisen möchten, können Sie dies wie in der folgenden Prozedur erledigen.

Außerdem fügen wir auch dem zweiten *Shape*-Objekt noch einige Zeilen Text hinzu:

```
Public Sub ShapeMitTextFuellen()
    Dim objSlide As PowerPoint.Slide
    Dim objShape As PowerPoint.Shape
    Set objSlide = CurrentPresentation.Slides(2)
    Set objShape = objSlide.Shapes(1)
    With objShape
        .TextFrame.TextRange.Text = "Titeltext"
    End With
   Set objShape = objSlide.Shapes(2)
```

18 Office-Dokumente im Access-Formular

Wenn man über die Zusammenarbeit von Access mit den übrigen Office-Komponenten schreibt, dann dürfen die Möglichkeiten zur Anzeige von Office-Dokumenten in Access nicht fehlen. Dies ist vor allem dann interessant, wenn Sie Word oder Excel als Ausgabe-Tool für Access-Daten nutzen:

Sie sehen dann direkt in einem Formular im aktuellen Access-Fenster, wie die Ausgabe aussieht und müssen nicht noch zwischen Access und einem zusätzlichen Anwendungsfenster wie etwa für Word oder Excel hin- und herwechseln – geschweige denn, dass immer wieder mal ein Fenster hinter dem anderen verschwindet.

Standardmäßig liefern Access-Tabellen mit dem OLE-Feld und seit Access 2007 mit dem Anlage-Feld die Möglichkeit, Dateien in Access-Tabellen zu speichern. Wenn Sie ein Office-Dokument in einem OLE-Feld speichern, können Sie dieses sogar in einem entsprechenden Steuerelement im Formular anzeigen und bearbeiten.

Leider springen beim Wechsel von den Access-eigenen Elementen zu denen des jeweiligen Office-Programms einige Elemente hin- und her und der Bildschirm flackert, sodass hier keine rechte Freude aufkommen will.

Das ist aber kein Problem, denn es gibt eine tolle Alternative – das *DSOFramer*-Steuerelement, das Microsoft bereitstellt und in der in der Beispieldatenbank zu diesem Kapitel verwendeten Version von Sascha Trowitzsch noch angepasst wurde. Dazu gehört auch das Anpassen der enthaltenen Texte an die deutsche Sprache.

18.1 DSOFramer-Steuerelement installieren

Wenn Sie das *DSOFramer*-Steuerelement zur Anzeige von Office-Dokumenten in einem Access-Formular nutzen wollen, müssen Sie zunächst eine DLL installieren, die Sie im Download zum Buch finden.

Die Datei heißt *DSOFramer_moss.ocx* und Sie speichern diese in eine Verzeichnis Ihrer Wahl (wir haben es unter *c:\Windows\SysWow64* gespeichert).

Danach müssen Sie es zunächst registrieren. Dazu starten Sie die Eingabeaufforderung im Administrator-Modus, wozu Sie zunächst im Startmenü von Windows den Wert *cmd* in das *Suchen*-Feld eingeben (im Beispiel unter Windows 7).

Es erscheint der Eintrag *cmd.exe*, auf den Sie mit der rechten Maustaste klicken, um sein Kontextmenü anzuzeigen. Dort finden Sie den Befehl *Als Administrator ausführen*, mit dem Sie die Eingabeaufforderung als Administrator öffnen (siehe Abbildung 18.1).

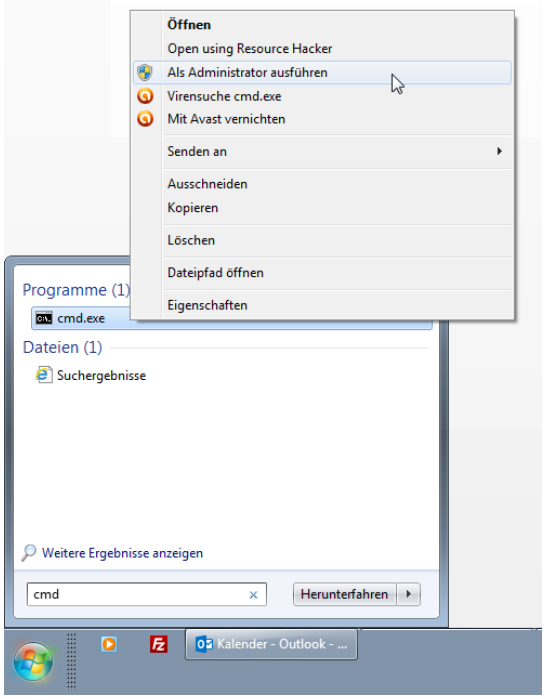


Abbildung 18.1: Öffnen der Eingabeaufforderung als Administrator

Dort navigieren Sie nun zu dem Verzeichnis, in dem Sie die Datei *DSOFramer_moss.dll* gespeichert haben – in unserem Fall nach *c:\Windows\SysWow64*. Dort geben Sie den Befehl *RegSvr32 dsoFramer_moss.ocx* ein. War die Registrierung erfolgreich, quittiert Windows dies mit einer entsprechenden Meldung (siehe Abbildung 18.2).

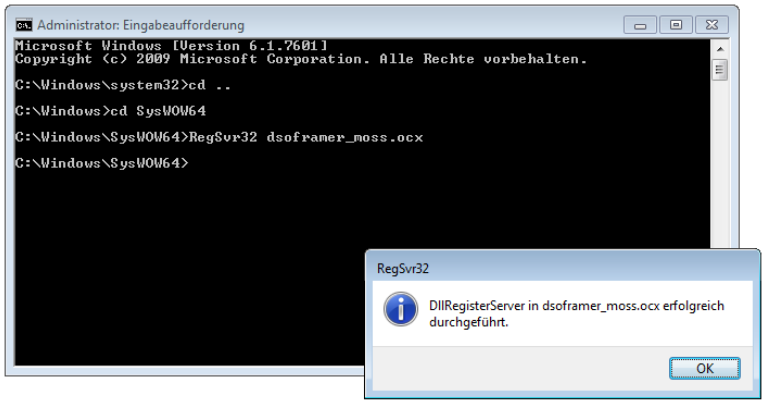


Abbildung 18.2: Registrieren der DSOFramer-DLL

18.2 DSOFramer zum Formular hinzufügen

Nun erstellen Sie ein neues Formular und speichern es unter der Bezeichnung *frmDSOFramer*. Das erste Steuerelement, das wir dem Formular hinzufügen, ist das *DSOFramer*-Steuerelement. Dazu wählen Sie in der Liste der Steuerelemente den Eintrag *Entwurf|Steuerelemente|ActiveX-Steuerelemente* aus (siehe Abbildung 18.3).

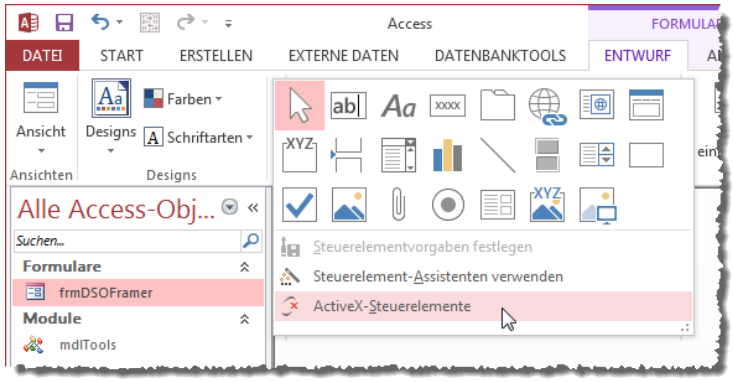


Abbildung 18.3: Einfügen eines ActiveX-Steuerelements

Dies öffnet den Dialog *ActiveX-Steuerelement einfügen* aus Abbildung 18.4, mit dem Sie den Eintrag *DSOFramerControlObject* auswählen und diesen mit einem Klick auf die Schaltfläche *OK* in das gerade im Entwurf geöffnete Formular einfügen.

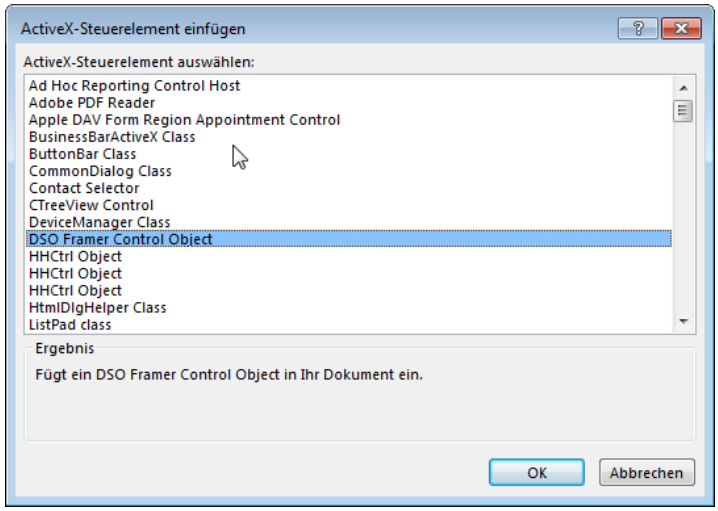


Abbildung 18.4: Auswählen des *DSOFramer*-Steuerelements

Das so hinzugefügte Steuerelement erscheint zunächst wie in Abbildung 18.5 im Formular.

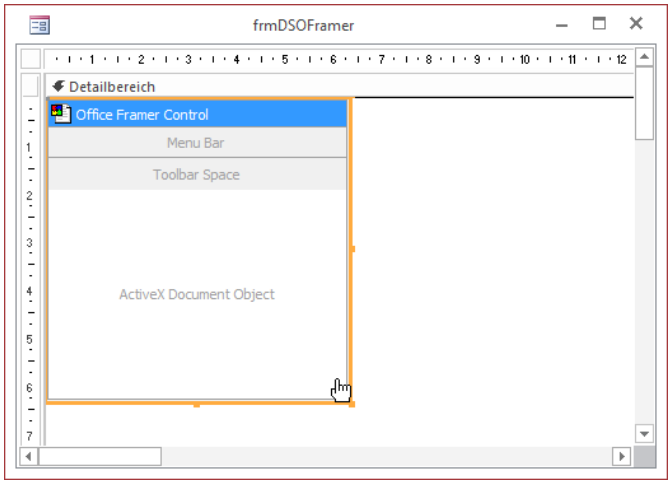


Abbildung 18.5: Das frisch angelegte *DSOFramer*-Steuerelement

Damit wir gleich nicht von störenden Elementen abgelenkt werden, stellen wir nun die Eigenschaften *Bildlaufleisten*, *Navigationsschaltflächen*, *Datensatzmarkierer* und *Trennlinien* auf *Nein* ein.

Außerdem legen wir für die beiden Eigenschaften *Horizontalter Anker* und *Vertikaler Anker* jeweils den Wert *Beide* fest. Das *DSOFramer*-Steuerelement erhält die Bezeichnung *ctlDSOFramer*.

Das Ergebnis sieht nach einem Wechseln in die Formularansicht ja schon einmal nicht schlecht aus (siehe Abbildung 18.6).

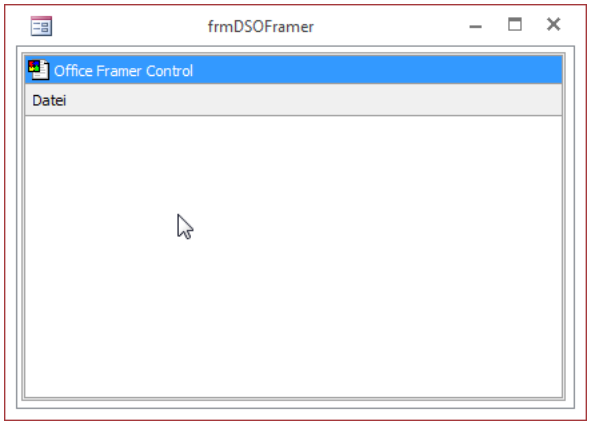


Abbildung 18.6: Der *DSOFramer* in der Formularansicht

Und ein Klick auf das Menü *Datei* macht sogar noch Appetit auf mehr: Dort finden sich die üblichen Befehle für den Umgang mit Dateien (siehe Abbildung 18.7).

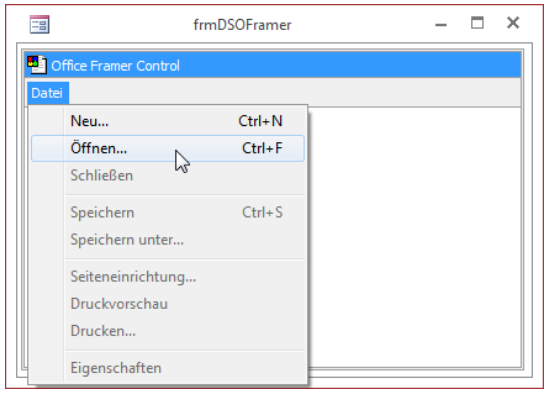


Abbildung 18.7: *Datei*-Menü des *DSOFramer*-Steuerelements

Den Menüpunkt *Datei|Öffnen* wollen wir gleich einmal anhand eines Word-Dokuments ausprobieren. Es erscheint der übliche *Datei öffnen*-Dialog, mit dem wir das gewünschte Dokument auswählen. Nach der Auswahl zeigt der *DSOFramer* nicht nur das Dokument an, sondern praktisch die komplette Word-Benutzeroberfläche im Access-Formular (siehe Abbildung 18.8).

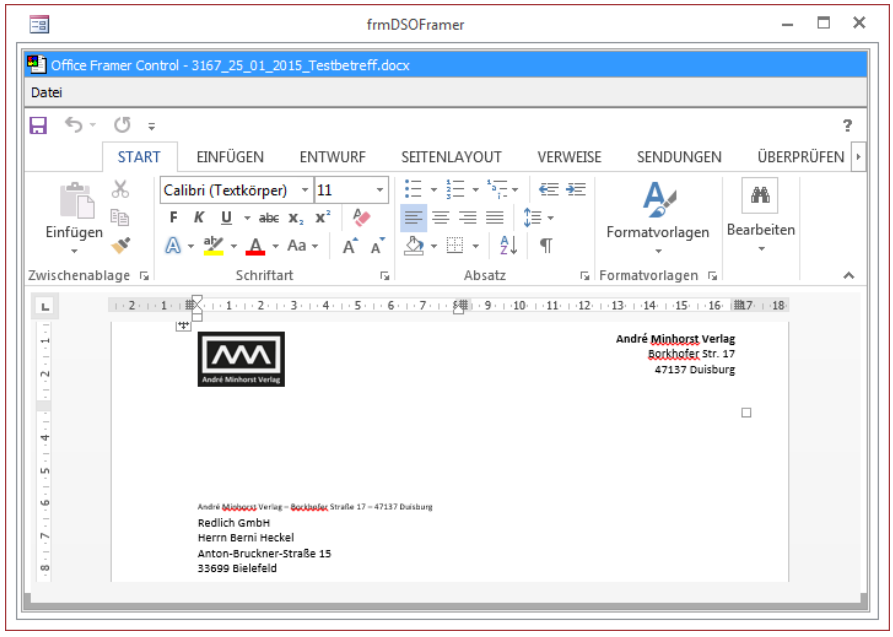


Abbildung 18.8: Ein Word-Dokument im Access-Formular

Bevor jemand auf falsche Gedanken kommt: Natürlich müssen die Office-Anwendungen wie Word oder Excel auf dem aktuellen Rechner installiert sein, bevor Sie die entsprechenden Dokumente im DSOFramer-Steuerelement öffnen können. Dieses Steuerelement liefert nur den Rahmen für die Anzeige der Anwendungen samt Dokumenten im Access-Formular.

Kein Datei-Menü

Interessant ist an dieser Stelle übrigens, dass hier etwa bei Word 2013 das *Datei*-Tab entfallen ist. Die Funktionen rund um das Dokument wie Öffnen, Erstellen, Speichern oder Drucken werden also komplett vom DSOFramer-Steuerelement übernommen.

18.3 DSOFramer programmieren

Damit wir das *DSOFramer*-Steuerelement im Formular programmieren können, um beispielsweise seine Eigenschaften per VBA einzustellen oder Dokumente zu laden, legen wir eine entsprechende Objektvariable fest – und zwar im Kopf des Klassenmoduls des Formulars:

```
Dim WithEvents objDSOFramer As DSOFramer.FramerControl
```

Außerdem hinterlegen Sie für das Ereignis *Beim Laden* des Formulars die folgende Ereignisprozedur:

```
Private Sub Form_Load()
    Set objDSOFramer = Me!ctlDSOFramer.Object
End Sub
```

Die Variable des Typs *FramerControl* haben wir mit dem Schlüsselwort *WithEvents* deklariert, damit wir dessen Ereignisse implementieren können – dazu später mehr. Nun schauen wir uns zunächst die verschiedenen Elemente des *DSOFramer*-Steuerelements an und wie wir diese beeinflussen können.

18.3.1 Eigenschaften des DSOFramer-Objekts

Um uns mit den Eigenschaften des *DSOFramer*-Objekts vertraut zu machen, lassen wir uns diese einmal bei geöffnetem Dokument ausgeben. Dazu legen wir im Formular eine Schaltfläche namens *cmdEigenschaften* an, welche die folgende Ereignisprozedur auslöst:

```
Private Sub cmdEigenschaften_Click()
    With objDSOFramer
        Debug.Print "ActiveDocument.Name: " & .ActiveDocument.Name
        Debug.Print "BackColor:           " & .BackColor
        Debug.Print "BorderColor:         " & .BorderColor
        Debug.Print "BorderStyle:         " & .BorderStyle
```

```
        Debug.Print "Caption:             " & .Caption
        Debug.Print "DocumentFullName:    " & .DocumentFullName
        Debug.Print "ForeColor:           " & .ForeColor
        Debug.Print "HostName:            " & .HostName
        Debug.Print "IsDirty:             " & .IsDirty
        Debug.Print "IsReadOnly:          " & .IsReadOnly
        Debug.Print "MenuBar:             " & .MenuBar
        Debug.Print "ModalState:          " & .ModalState
        Debug.Print "Titlebar:            " & .Titlebar
        Debug.Print "TitlebarColor:       " & .TitlebarColor
        Debug.Print "TitlebarTextColor:   " & .TitlebarTextColor
        Debug.Print "Toolbars:            " & .Toolbars
    End With
End Sub
```

Das Ergebnis sieht wie folgt aus:

```
ActiveDocument.Name: Dokument in DsoFramerControl
BackColor:           -2147483643
BorderColor:         -2147483632
BorderStyle:         1
Caption:             Office Framer Control
DocumentFullName:    Z:\Dokumente\Daten\Buecher\AOFF\Beispieldateien\Word\Test_Format.doc
ForeColor:           -2147483640
HostName:            DsoFramerControl
IsDirty:             Falsch
IsReadOnly:          Falsch
MenuBar:             Wahr
ModalState:          Falsch
Titlebar:            Wahr
TitlebarColor:       -2147483635
TitlebarTextColor:   -2147483634
Toolbars:            Wahr
```

Die meisten dieser Eigenschaften können Sie zum Einstellen des Aussehens des *DSOFramers* nutzen – zum Beispiel:

» *Caption*: Stellt den ersten Teil des Textes in der Titelleiste ein.

» *MenuBar*: Hiermit blenden Sie das Datei-Menü des *DSOFramers* ein und aus.

» *IsDirty*: Gibt an, ob das Dokument seit dem letzten Speichern bearbeitet wurde.

» *IsReadOnly*: Gibt an, ob das Dokument schreibgeschützt geöffnet wurde.

» *Titlebar*: Blendet die Titelleiste ein und aus.