HAUFE.
Ausgabe 01/2020

ACCESS
IM UNTERNEHMEN

OUTLOOK-TERMINE

Termine importieren, Termin-Formulare erweitern und benutzerdefinierte Felder hinzufügen (ab S. 25)

In diesem Heft:

TABELLEN UND CO. IM RIBBON ANZEIGEN

Machen Sie Access-Objekte einfach über das Ribbon zugänglich.

SEITE 55

DATENSÄTZE NACH ZAHL AUSGEBEN

Ermitteln Sie Datensätze wie im Drucken-Dialog mit Angaben wie 1, 2, 5 oder 4-7.

SEITE 16

OPTIONEN UPDATE-SICHER SPEICHERN

Speichern Sie Anwendungsoptionen so, dass diese bei Updates nicht überschrieben werden.

SEITE 3

Outlook-Termine im Griff

Die Kooperation zwischen Access und Outlook ist immer wieder ein gefragtes Thema in den Leseranfragen. In dieser Ausgabe schauen wir uns daher die Outlook-Termine genauer an, und zwar in gleich drei Beiträgen. Da wir dieses Thema schon in früheren Ausgaben bearbeitet haben, können wir diesmal etwas tiefer in die Materie einsteigen – allerdings ohne die Grundlagen zu vernachlässigen.

Der erste Beitrag mit dem Titel **Benutzerdefinierte Felder in Outlook** (ab Seite 25) bezieht sich nicht explizit auf Outlook-Termine, sondern auch auf die übrigen Elementtypen wie E-Mails, Kontakte oder Aufgaben. Hier beschreiben wir, wie Sie den vielen ohnehin schon vorhandenen Eigenschaften der Outlook-Elemente noch weitere hinzufügen können. Damit können Sie dann beispielsweise Termine um ein Feld ergänzen, dass den Namen des Mitarbeiters speichert, der diesen Termin durchgeführt hat. Im gleichen Beitrag beschreiben wir, wie Sie der Benutzeroberfläche von Outlook, in diesem Fall einem Outlook-Termin, Steuerelemente hinzufügen, mit denen Sie die Inhalte der benutzerdefinierten Felder einsehen oder bearbeiten können.

Die so erfassten Daten machen natürlich nur Sinn, wenn wir später auch auswerten können, wann welcher Mitarbeiter Zeit mit welchem Kunden verbracht hat – zum Beispiel, um dem Kunden die geleistete Zeit in Rechnung zu stellen. Dazu müssen wir nicht nur auf die üblichen Eigenschaften der Outlook-Termine wie die Startzeit und die Endzeit zugreifen können, sondern auch auf die zuvor angelegten Werte für die benutzerdefinierten Felder, in diesem Fall den Namen oder die Nummer des Mitarbeiters. Wie das gelingt, zeigt der Beitrag **Outlook-Termine nach Access einlesen** ab Seite 34.

Die im eingangs genannten Beitrag erweiterten Outlook-Formulare samt Steuerelementen können Sie natürlich auch programmieren. Wie das gelingt, zeigen wir im Beitrag **Outlook-Termine programmieren** ab Seite 45. Hier erfahren Sie, wie Sie auf die Outlook-Fenster zugreifen, die enthaltenen Steuerelemente referenzieren und Ereignisprozeduren implementieren. Diese können Sie beispielsweise nutzen, um beim Anlegen eines neuen Termins direkt den Mitarbeiter in das dafür vorgesehene benutzerdefinierte Feld einzutragen.

Ein weiterer Beitrag namens **Optionen und andere Daten updatesicher speichern** (ab Seite 3) beschreibt, wie Sie die Daten einer Optionentabelle so speichern, dass diese nicht von Updates der Frontend-Datenbank betroffen sind.

Unter **Datensätze nach Zahl ausgeben** erfahren Sie ab Seite 16, wie Sie die im Datenblatt angezeigten Datensätze so referenzieren können, wie Sie im **Drucken**-Dialog von Windows die zu druckenden Seiten angeben – also mit Ausdrücken wie 1, 2, 5 oder 4-6. Und es sind auch Kombinationen dieser Möglichkeiten erlaubt.

Mit dem Beitrag **Objekte im Ribbon verfügbar machen** erhalten Sie ab Seite 55 eine praktische Lösung, mit der Sie die Objekte einer Datenbankanwendung wie Tabellen, Abfragen, Formulare und Berichte im Ribbon anzeigen und über die entsprechenden Schaltflächen öffnen können.

Und schließlich liefern wir unter **HTML-Code optimieren per VBA** noch Informationen, wie Sie HTML-Code per VBA optimieren und für die Ausgabe auf verschiedenen Plattformen optimieren können (ab Seite 67).

Und nun: Viel Spaß beim Lesen!

Ihr André Minhorst

Microsoft To Do

Wer Access- oder andere Softwareprojekte durchführt, hat in der Regel immer eine prall gefüllte To-Do-Liste. Bei dem einen findet sich die Liste in Papierform, andere verwenden computergestützte Lösungen. Wer keine eigene Datenbank dafür programmieren möchte, kann ein Produkt aus dem Hause Microsoft für die Verwaltung seiner Aufgaben nutzen: Microsoft To Do.

Mit **Microsoft To Do**, das auf dem weitverbreiteten und von To Do abgelösten Tool **Wunderlist** basiert, können Sie Listen, Aufgaben und Unterpunkte verwalten. Sie können Aufgaben in die Aufgabenliste für den aktuellen Tag übertragen und Termine für die Erledigung von Aufgaben festlegen. Außerdem bietet To Do die Möglichkeit, Aufgaben zu teilen, wodurch mehrere Menschen gleichzeitig auf Listen zugreifen können und Aufgaben hinzufügen oder als erledigt markieren können.

Eines der besten Features, das dieses Tool bietet: Es gibt nicht nur eine Desktop-Version (siehe Bild 1), sondern auch eine für die verschiedenen mobilen Endgeräte, also für iOS oder Android, und es gibt auch noch eine Webversion der Anwendung. Sie brauchen also nicht den Desktop, sondern können sich die Aufgabenliste bequem über Smartphone oder Tablet anzeigen lassen.

Interessant ist die Frage, wie weit Microsoft dieses Tool in die Infrastruktur der übrigen Microsoft-Anwendungen wie etwa Microsoft Outlook einbettet und ob wir per VBA auf die gespeicherten Aufgabenlisten zugreifen können.

Voraussetzung ist ein kostenloses Microsoft-Konto.

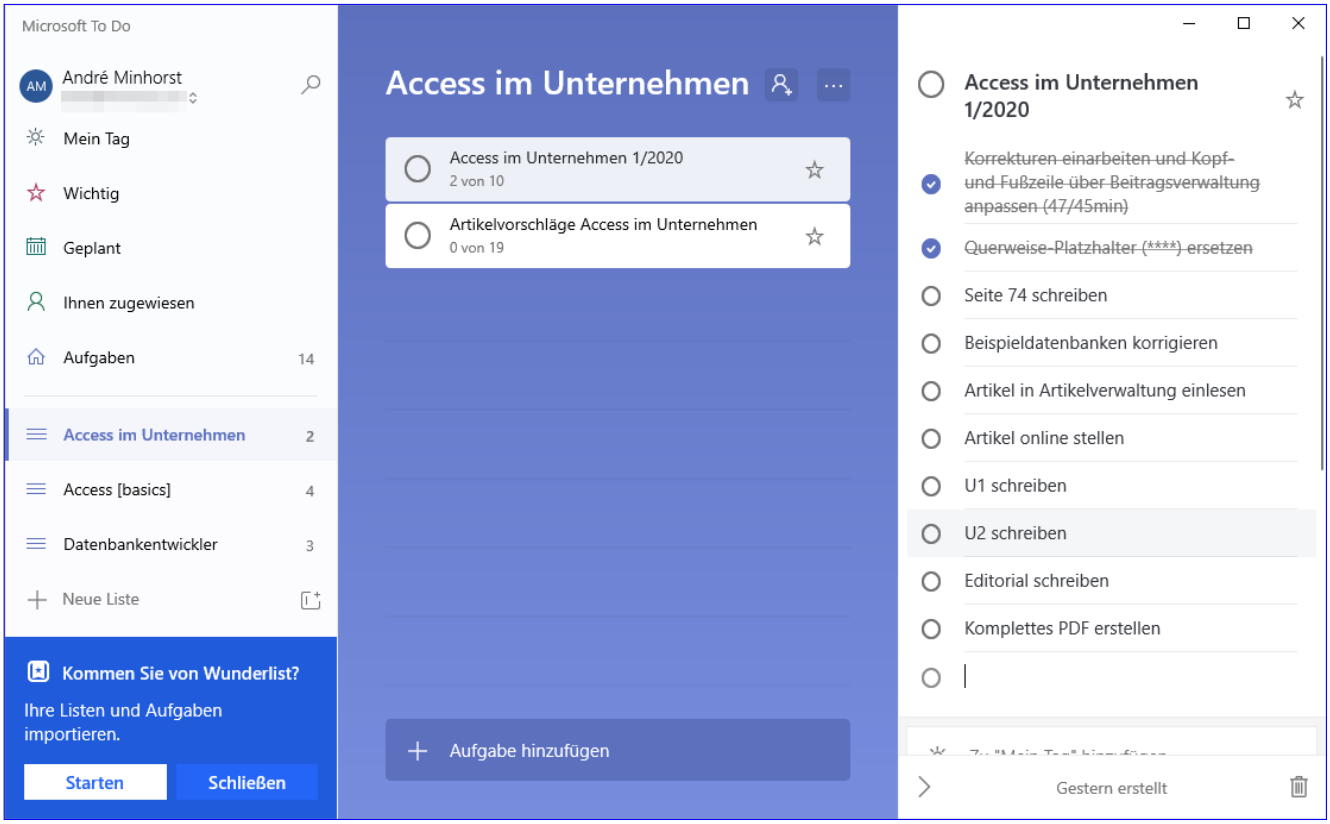

Bild 1: Microsoft To Do in der Desktopversion

Optionen und andere Daten update-sicher speichern

Wenn mehrere Benutzer über Frontend-Anwendungen an verschiedenen Arbeitsplätzen auf die Daten einer Backend-Datenbank zugreifen, ist das kein Problem. In vielen Anwendungen ist es dabei sinnvoll, die Möglichkeit zum Speichern von Optionen je Benutzer vorzusehen. Oft legt man dabei eine Tabelle namens »tblOptionen« im Frontend an. Das ist aber nur sinnvoll, wenn auch immer der gleiche Benutzer am gleichen Frontend arbeitet – anderenfalls würde er ja die Optionen eines anderen Benutzers vorgesetzt bekommen. Oder Sie führen ein Update des Frontends durch – auch dann wären die Optionen des Benutzers nicht mehr vorhanden. Welche Alternativen dazu gibt es? Sie könnten zum Beispiel die Optionen in einer Tabelle im Backend speichern und diese beim Anmelden des Benutzers auslesen und in eine lokale Optionentabelle übertragen. Oder Sie fügen dem Frontend ein lokales Backend hinzu, das nur die Daten enthält, die beim Update des Frontends nicht überschrieben werden sollen. Wie das gelingt, zeigen wir im vorliegenden Artikel.

Am einfachsten ist das Speichern der Optionen eines Benutzers im Frontend. Und auch wenn wir das schon so in früheren Beiträgen propagiert haben: Die Lösung ist nur unter ganz bestimmten Rahmenbedingungen pragmatisch. Die folgenden Bedingungen würden es erlauben, die Optionen einfach in einer Tabelle namens **tblOptionen** im Frontend zu speichern:

- Am Rechner mit dem Frontend arbeitet immer nur der Mitarbeiter, dessen Optionen in der Tabelle **tblOptionen** dieses Frontends gespeichert sind.
- Es gibt keine Updates für das Frontend. Sonst würden mit einem Update, das durch einfaches Ersetzen über die vorhandene Version kopiert wird, immer die bestehenden Optionswerte gelöscht werden.

Es kann sein, dass beide Bedingungen erfüllt werden. Gerade wenn ein Frontend auf einem Notebook läuft, das einem Mitarbeiter fest zugeteilt ist, wird sich dort kaum ein anderer Mitarbeiter anmelden und die Access-Anwendung nutzen, der seinen eigenen Satz von Optionen erwartet. Und es gibt sicher auch Anwendungen, die bereits fertig programmiert sind und keinerlei Erweiterungen oder Änderungen mehr benötigen, sodass es schlicht und einfach keine Updates mehr zu diesen Anwendungen gibt.

Tatsächlich ist aber nicht sichergestellt, dass Microsoft nicht einmal einen Fehler in ein Update integriert, der die übliche Funktion von Access verhindert – und dann kann es gegebenenfalls doch einmal nötig sein, dass ein Update über die aktuelle Version mit den gespeicherten Optionen des Benutzers kopiert werden muss (tatsächlich ist genau das vor kurzem geschehen – plötzlich funktionierten nämlich Abfragen nicht mehr fehlerfrei, was sich entweder kurzfristig durch eine Änderung der Programmierung beheben ließ oder durch das Warten auf das nächste Update oder den Bugfix von Microsoft).

Wir gehen in diesem Beitrag zunächst davon aus, dass jeder Benutzer seinen eigenen Rechner hat und wir die Optionentabelle einfach in einem lokalen Backend anlegen. Später schauen wir uns an, wie die Lösung für das Speichern der Optionen aller Benutzer im Backend auf dem Server aussehen kann.

Die Optionentabelle

Die Optionentabelle kann unterschiedlich aufgebaut sein. Die erste Möglichkeit ist, die Optionentabelle so zu gestalten, dass jede Option fest als Feld vorgesehen ist. Das sieht dann etwa wie in Bild 1 aus. Diese Variante hat den Vorteil, dass man für jede Option den Felddatentyp individuell festlegen kann – so, wie wir es hier auch gemacht haben.

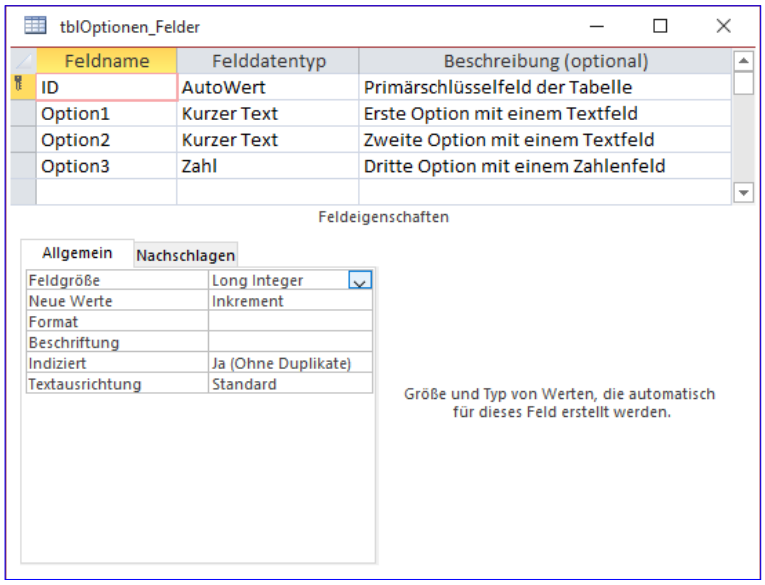

Bild 1: Optionstabelle mit je einem Feld pro Option

In dieser Variante braucht man eigentlich kein Primärschlüsselfeld, denn es gibt ja zumindest in einer Anwendung für nur einen Arbeitsplatz auch nur einen Datensatz mit Optionen. Da wir aber auch davon ausgehen müssen, dass einmal die Optionen für mehr als einen Anwender gespeichert werden müssen, haben wir vorsichtshalber schon einmal ein Primärschlüsselfeld hinzugefügt.

Die Alternative ist, für jede Option einen eigenen Datensatz anzulegen, der dann in einem Feld den Namen der Option enthält und in einem weiteren Feld den Optionswert. Das sieht im Entwurf wie in Bild 2 aus.

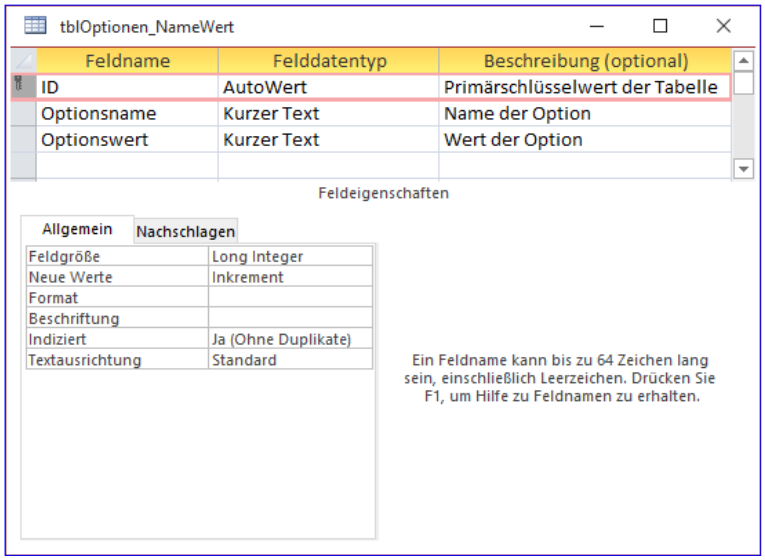

Bild 2: Optionstabelle mit einer Option je Datensatz

Der Vorteil dieser Variante ist, dass Sie für das Anlegen neuer Optionen nicht in den Entwurf des Datenmodells eingreifen müssen, sondern nur einen neuen Datensatz hinzuzufügen brauchen. Der Nachteil ist, dass wir ausschließlich Werte eines zuvor festzulegenden Datentyps als Optionswerte festlegen können. Dieser Nachteil ist aber nicht allzu gewichtig, denn wir können ja alle Datentypen auch im Textformat speichern und gegebenenfalls in den benötigten Datentyp umwandeln.

Optionentabelle im lokalen Backend

Wenn wir die Optionen updatesicher speichern wollen, müssen wir diese außerhalb des Frontends unterbringen. Updatesicher speichern bedeutet dabei, dass wir sicherstellen wollen, dass wir das Frontend durch eine neue Version ersetzen können, ohne dass die für den Benutzer der Anwendung gespeicherten Optionen verloren gehen. Das erledigen wir, indem wir die Optionentabelle in eine eigene Backenddatenbank auslagern, die wir dann im gleichen Verzeichnis wie die Frontendanwendung speichern. Wir fügen dem Frontend dann eine Verknüpfung zu dieser Tabelle hinzu, die automatisch beim Öffnen der Frontendanwendung geprüft und erneuert wird.

Verknüpfung beim Start aktualisieren

Die Aktualisierung der Verknüpfung zur Optionentabelle erledigen wir direkt beim Start der Anwendung. Dazu gibt es verschiedene Möglichkeiten, wo wir den Aufruf des dazu notwendigen VBA-Code platzieren:

- Im **Load**- oder **Open**-Ereignis eines Formulars, das automatisch beim Starten der Frontendanwendung angezeigt wird, oder
- in einer VBA-Prozedur, die durch die Makroaktion **AusführenCode** im automatisch beim Start ausgeführten **AutoExec**-Makro aufgerufen wird.

Wir gehen an dieser Stelle davon aus, dass Sie bereits eine Frontendanwendung vorliegen haben und eine Backenddatenbank, welche die Optionentabelle enthält. Was kann bei der Verknüpfung alles geschehen und den Vorgang blockieren?

- Die Verknüpfung in der Frontenddatenbank ist vorhanden, aber verweist auf die falsche oder eine nicht vorhandene Tabelle.
- Die Verknüpfung in der Frontenddatenbank ist nicht vorhanden.
- Die Backenddatenbank ist nicht vorhanden.
- Die Backenddatenbank ist vorhanden, aber die Optionentabelle ist nicht darin enthalten.

Die folgenden Funktionen sollen all dies addressieren, um die in der Konstanten **cStrOptionentabelle** angegebene Tabelle der in der Konstanten **cStrOptionenbackend** angegebenen Datenbank aus dem Verzeichnis des Frontends zu verknüpfen.

Name von Backend und Tabelle speichern

Wir müssen an irgendeiner Stelle die Informationen darüber speichern, wie die Datenbankdatei mit der Optionentabelle und wie die darin enthaltene Optionentabelle heißt – was auch dem Namen der Verknüpfung für die Optionentabelle in der Frontend-Datenbank entspricht. Dazu legen wir im Modul **mdlBackendoptionen**, dem wir nachfolgend die Funktionen zum Prüfen und Wiederherstellen der Verknüpfung hinzufügen, die folgenden beiden Konstanten mit dem Datentyp **String** an:

```
Const cStrOptionenbackend As String = "Backend_Optionen.
accdb"
Const cStrOptionentabelle As String = "tblOptionen_Felder"
```

Die Backenddatenbank soll also **Backend_Optionen.accdb** heißen, als Backendtabelle mit den Optionen verwenden wir die Tabelle namens **tblOptionen_Felder**.

Wo aber speichern wir den Pfad, in dem sich die Backenddatenbank befindet? Diesen speichern wir gar nicht, denn die Backenddatenbank mit der Optionentabelle soll sich schlicht im gleichen Verzeichnis wie die Frontenddatenbank befinden. Und diesen ermitteln wir per VBA zur Laufzeit mit der Eigenschaft **CurrentProject.Path**.

Prüfen, ob die Backenddatenbank vorhanden ist

Die erste Möglichkeit für eine Fehlfunktion ist, dass die Backenddatenbank gar nicht an Ort und Stelle ist. Das prüfen wir mit einer Funktion namens **IstBackendVorhanden**. Diese stellt in der Variablen **strPfad** den Pfad zum Backend zusammen, der aus dem Verzeichnis der aktuellen Datenbank und dem Datenbanknamen aus der Konstanten **cStrOptionenbackend** besteht.

Mit der **Dir**-Funktion ermitteln wir die erste Datei, die unter dieser Pfadangabe gefunden werden kann. Dies sollte die in **cStrOptionenBackend** angegebene Datei liefern oder, wenn diese Datei nicht gefunden werden kann, eine leere Zeichenkette. Beides landet in der Variablen **strBackenddatei**. Um zu prüfen, ob diese Variable nicht leer ist, ermitteln wir die Anzahl der Zeichen des Wertes dieser Variablen. Ist diese gleich **0**, konnte die Datei nicht im angegebenen Verzeichnis gefunden werden und die

Funktion liefert den Wert **False** zurück. Wenn die Zeichenkette aus **strBackenddatei** jedoch eine Länge größer als **0** aufweist, dann ist die Datei vorhanden und die Funktion liefert den Wert **True** zurück:

```
Public Function IstBackendVorhanden() As Boolean
    Dim strPfad As String
    Dim strBackenddatei As String
    strPfad = CurrentProject.Path & "\" _
                                & cStrOptionenbackend
    strBackenddatei = Dir(strPfad)
    If Not Len(strBackenddatei) = 0 Then
        IstBackendVorhanden = True
    End If
End Function
```

Prüfen, ob die Optionentabelle im Backend vorliegt

Damit gehen wir einen Schritt weiter, nämlich zur Prüfung des Vorhandenseins der Optionentabelle in der Backenddatenbank. Dazu referenzieren wir die Backenddatenbank und prüfen, ob dort ein **TableDef**-Objekt mit dem angegebenen Namen vorhanden ist.

Dazu stellen wir wieder in **strPfad** den Pfad zur Backenddatenbank zusammen. Diesmal prüfen wir allerdings nicht, ob diese vorhanden ist, sondern öffnen mit der **OpenDatabase**-Methode ein **Database**-Objekt auf Basis dieser Datenbank und speichern dieses in der Variablen **db**. Dann versuchen wir wieder, bei deaktivierter Fehlerbehandlung auf das **TableDef**-Objekt aus **cStrOptionenbackend** zuzugreifen. Ist die damit gefüllte Variable **tdf** danach nicht leer, ist die Tabelle in der Backenddatenbank enthalten:

```
Public Function IstOptionentabelleVorhanden() As Boolean
    Dim db As DAO.Database
    Dim tdf As DAO.TableDef
    Dim strPfad As String
    strPfad = CurrentProject.Path & "\" _
                                & cStrOptionenbackend
    Set db = OpenDatabase(strPfad)
    On Error Resume Next
    Set tdf = db.TableDefs(cStrOptionentabelle)
    On Error GoTo 0
    If Not tdf Is Nothing Then
        IstOptionentabelleVorhanden = True
    End If
End Function
```

Prüfen, ob Verknüpfung vorhanden ist

Die erste Hilfsfunktion prüft, ob die Verknüpfung überhaupt im Frontend vorhanden ist. Sie heißt **IstVerknuepfungVorhanden** und liefert einen Rückgabewert des Typs **Boolean**. Die Funktion erstellt ein Objekt mit einem Verweis auf das **Database**-Objekt der Frontenddatenbank. Dann deaktiviert sie die eingebaute Fehlerbehandlung und versucht, über die **TableDefs**-Auflistung des **Database**-Objekts auf die Tabelle mit dem in **cStrOptionenTabelle** gespeicherten Namen zuzugreifen und einen Verweis darauf mit der Variablen **tdf** zu referenzieren.

Wir prüfen danach gar nicht erst, ob dabei ein Fehler aufgetreten ist oder nicht (ein Fehler würde beim Zugriff auf ein nicht vorhandenes **TableDef**-Element ausgelöst werden), sondern aktivieren danach einfach die Fehlerbehandlung wieder und prüfen dann, ob **tdf** mit einem Objektverweis gefüllt ist. Falls ja, wurde die Verknüpfung offensichtlich gefunden, falls nicht, fehlt sie in der aktuellen Datenbank:

```
Public Function IstVerknuepfungVorhanden() As Boolean
    Dim db As DAO.Database
    Dim tdf As DAO.TableDef
    Set db = CurrentDb
    On Error Resume Next
    Set tdf = db.TableDefs(cStrOptionentabelle)
    On Error GoTo 0
    If Not tdf Is Nothing Then
        IstVerknuepfungVorhanden = True
    End If
End Function
```

Wenn die Verknüpfung gar nicht vorhanden ist, müssen wir diese anlegen. Das erledigen wir gleich in der Hauptfunktion, die auch die Funktion **IstVerknuepfungVorhanden** aufruft.

Prüfen, ob die Verknüpfung auf die richtige Tabelle in der richtigen Datenbank verweist

Es kann auch sein, dass die Verknüpfung zwar vorhanden ist, aber nicht auf die richtige Tabelle verweist. Oder dass sie zwar auf die Tabelle mit dem richtigen Namen verweist, diese sich aber nicht in der richtigen Datenbank im richtigen Verzeichnis befindet.

Dies prüfen wir beides in der Funktion **IstVerknuepfungKorrekt** (siehe Listing 1). Diese referenziert das **TableDefs**-Objekt aus **cStrOptionentabelle** und schreibt daraus die Inhalte der **Connect**-Eigenschaft in die Variable **strConnect** und der **SourceTableName**-Eigenschaft in die Variable **strTabelle**. Die **Connect**-Eigenschaft liefert einen Ausdruck, der mit **;DATABASE=** beginnt und dann den Pfad zu der Datenbank liefert, in welcher sich die verknüpfte Tabelle befindet. Die **SourceTableName**-Eigenschaft enthält den Namen der Tabelle in dieser Backenddatenbank.

```
Public Function IstVerknuepfungKorrekt() As Boolean
    Dim db As DAO.Database
    Dim tdf As DAO.TableDef
    Dim strConnect As String
    Dim strTabelle As String
    Set db = CurrentDb
    Set tdf = db.TableDefs(cStrOptionentabelle)
    strConnect = tdf.Connect
    strTabelle = tdf.SourceTableName
    If strTabelle = cStrOptionentabelle Then
        If strConnect = ";DATABASE=" & CurrentProject.Path & "\" _
                & cStrOptionenbackend Then
            IstVerknuepfungKorrekt = True
        End If
    End If
End Function
```

Listing 1: Prüfen, ob die Verknüpfung die richtige Datenbank und Tabelle addressiert

Wir prüfen zunächst, ob der Wert in **strTabelle** mit dem Namen aus **cStrOptionentabelle** übereinstimmt. Ist das der Fall, untersuchen wir, ob der Wert aus **strConnect** mit der Zeichenkette bestehend aus **;DATABASE=**, dem Verzeichnis der aktuellen Datenbank und dem Datenbanknamen aus **cStrOptionenbackend** besteht. Sind beide Bedingungen erfüllt, stellen wir den Rückgabewert der Funktion auf **True** ein.

Prozedur zum Prüfen und Korrigieren der Verknüpfung

Mit diesen Hilfsfunktionen können wir nun entspannt in die Hauptprozedur einsteigen, die prüft, ob alle Bedingungen erfüllt sind und gegebenenfalls Korrekturen vornimmt, damit dies der Fall ist. Aber in welcher Reihenfolge prüfen wir die Bedingungen und wann und wie greifen wir korrigierend ein?

Wir können beispielsweise damit einsteigen, dass wir mit der Funktion **IstVerknuepfungVorhanden** prüfen, ob die Verknüpfung zur Optionentabelle überhaupt im Frontend vorliegt. Aber was tun, wenn das schon nicht der Fall ist? Sollen wir diese dann einfach anlegen und dann die weiteren Bedingungen prüfen? Das ist wenig sinnvoll, denn wenn wir nicht wissen, ob die Backenddatenbank überhaupt vorhanden ist, brauchen wir auch nicht die Verknüpfung zu reparieren. Also starten wir in der ersten Version der Funktion **OptionentabelleVerknuepfen** bei den wesentlichen Voraussetzungen (siehe Listing 2). Die Erste lautet: Ist die Backenddatenbank überhaupt an Ort und Stelle? Das prüfen wir mit der Funktion **IstBackendVorhanden**. Falls ja, gehen wir mit der Prüfung der

```
Public Function OptionentabelleVerknuepfen() As Boolean
    Dim bol As Boolean
    If IstBackendVorhanden Then
        If IstOptionentabelleVorhanden Then
            If IstVerknuepfungVorhanden Then
                If IstVerknuepfungKorrekt Then
                    bol = True
                Else
                    MsgBox "Die Verknüpfung ist nicht korrekt."
                End If
            Else
                MsgBox "Die Verknüpfungstabelle fehlt."
            End If
        Else
            MsgBox "Die Tabelle '" & cStrOptionentabelle & "' konnte nicht in der folgenden Datenbank gefunden " _
                & "werden:" & vbCrLf & vbCrLf & CurrentProject.Path & "\" & cStrOptionenbackend
        End If
    Else
        MsgBox "Die Datenbank '" & cStrOptionenbackend & "' mit der Optionentabelle konnte nicht gefunden werden."
    End If
    OptionentabelleVerknuepfen = bol
End Function
```

Listing 2: Prüfen aller Bedingungen in einer Funktion

nächsten Bedingung weiter. Falls nicht, müssen wir dem Benutzer mitteilen, was nicht in Ordnung ist oder dies selbst beheben.

Die nächste Bedingung prüfen wir mit der Funktion **IstOptionentabelleVorhanden**. Auch hier geben wir eine Meldung aus, wenn die Bedingung nicht erfüllt wurde.

Liefert die Funktion jedoch den Wert **True**, dann gehen wir mit der Prüfung der nächsten Bedingung weiter – also **IstVerknuepfungVorhanden** – und dann mit **IstVerknuepfungKorrekt**. In beiden Fällen wird für den Wert **False** eine entsprechende Meldung ausgegeben.

Optionentabelle verknüpfen mit Korrekturmöglichkeit

Nun wollen wir dem Benutzer noch die Möglichkeit geben, selbst die nicht erfüllten Bedingungen zu erfüllen und die Bedingungen dann erneut zu prüfen. Dazu müssen wir die ganzen **If...Then**-Bedingungen in eine Schleife einfassen, die prüft, ob **bol** den Wert **True** liefert, was der Fall ist, wenn alle Bedingungen erfüllt sind.

Wir müssen aber auch in Erwägung ziehen, dass der Benutzer oder die Anwendung selbst die Bedingungen nicht erfüllen können – beispielsweise wenn der Benutzer die Backend-Datenbank per Dateiauswahl-Dialog auswählen soll, weil diese gegebenenfalls an einem anderen Ort liegt, an dem sich auch die Frontend-Datenbank vorher befunden hat, aber diese Datenbank nicht findet. Dann soll der Benutzer die Möglichkeit erhalten, die Schleife abzubrechen.

Dazu haben wir die Funktion **OptionentabelleVerknuepfen** wie in Listing 3 ordentlich aufgebohrt. Als Erstes finden wir hier die schon angesprochene **Do While**-Schleife, die erst abgebrochen wird, wenn die Variable **bol** den Wert **True** aufweist. Diese Variable wird erst in der inneren der vier verschachtelten **If...Then**-Bedingungen auf **True** eingestellt. Dadurch ist die Funktion noch nicht beendet,

Datensätze nach Zahl ausgeben

Sie kennen das sicher vom Drucken-Dialog, wo Sie einzelne Zahlen oder Zahlenbereiche angeben können, um die betroffenen Seiten zu drucken – also beispielsweise 1-2, 3-5 und so weiter. Eine solche Möglichkeit wollen wir auch für das Filtern von Datensätzen in der Datenblattansicht schaffen. Der Benutzer soll also eine oder mehrere Seitenzahlen oder Bereiche von Seiten in ein Textfeld eingeben können, nach denen dann gefiltert wird.

Vorbereitung

Wir benötigen ein Hauptformular, welches das Textfeld für die Eingabe der gewünschten Filterausdrücke enthält, sowie ein Unterformular, das die gefilterten Datensätze anzeigt. Das Unterformular soll die Tabelle **tblArtikel** als Datensatzquelle verwenden, und zwar als Abfrage formuliert, welche die Artikel aufsteigend nach dem Wert des Feldes **ArtikelID** sortiert. Das Unterformular namens **sfmArtikelNachBereich** enthält alle Felder der zugrundeliegenden Abfrage. Außerdem stellen wir seine Eigenschaft **Standardansicht** auf **Datenblatt** ein.

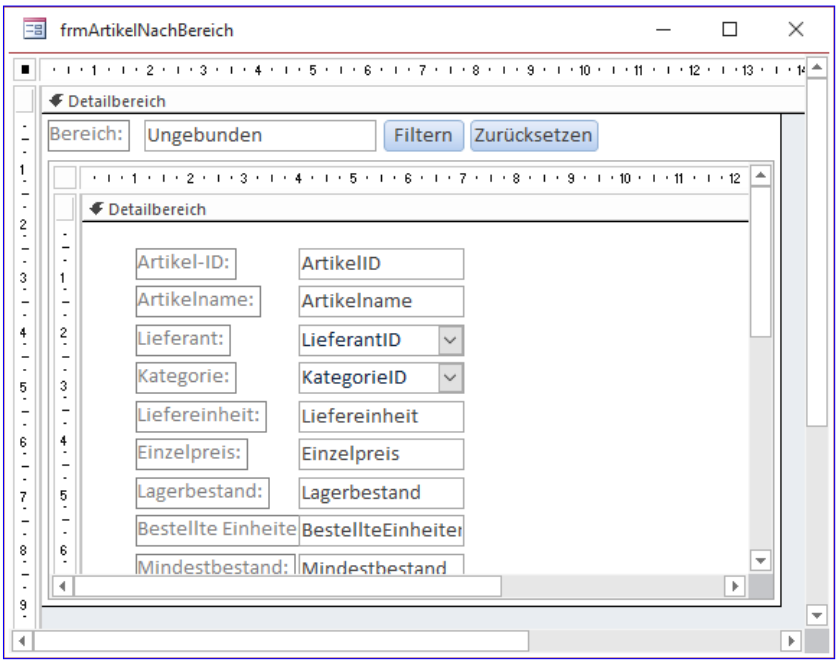

Bild 1: Entwurf des Formulars zum Filtern nach Zahlenbereichen

Das Formular **sfmArtikelNachBereich** fügen wir dann dem Hauptformular als Unterformular hinzu. Außerdem legen wir im Hauptformular ein Textfeld namens **txtBereiche** an. Neben diesem Textfeld legen wir zwei Schaltflächen an – eine zum Anwenden des Filters und eine zum Zurücksetzen. Für das Hauptformular stellen wir außerdem die Eigenschaften **Datensatzmarkierer**, **Navigationsschaltflächen** und **Bildlaufleisten** auf **Nein** und **Automatisch zentrieren** auf **Ja** ein.

Die beiden Eigenschaften **Horizontaler Anker** und **Vertikaler Anker** des Unterformular-Steuerelements erhalten den Wert **Beide**, das Bezeichnungsfeld des Unterformular-Steuerelements löschen wir. Das Hauptformular sieht dann inklusive Steuerelementen und Unterformular wie in Bild 1 aus.

Anforderungen

Welche Filtermöglichkeiten wollen wir schaffen? Hier sind die einzelnen Möglichkeiten:

- Durch Komma getrennte Zahlen, also zum Beispiel **1, 3, 5**, sollen die erste, dritte und fünfte Seite liefern.
- Zwei durch einen Bindestrich verbundene Zahlen geben einen Bereich an, zum Beispiel soll **1-4** alle Datensätze von **1** bis **4** liefern.

- Ein Bindestrich vor einer Zahl soll alle Datensätze bis zu dieser Zahl liefern. **-5** soll also alle Datensätze von **1** bis **5** ausgeben.

- Ein Bindestrich nach einer Zahl soll alle Datensätze ab dieser Zahl liefern (einschließlich dieser Zahl). **3-** soll also alle Datensätze ab **3** ausgeben.

Diese Möglichkeiten sollen auch noch kombiniert werden können, und zwar jeweils getrennt durch das Komma-Zeichen.

Und was auch noch nicht geklärt ist: Worauf genau sollen sich die Zahlen beziehen? Dazu gibt es zwei Möglichkeiten:

- Den Index der angezeigten Datensätze

- Den Primärschlüsselwert der angezeigten Datensätze

Da wir vorher nie wissen, wie die Wünsche des Benutzers aussehen, fügen wir dazu noch eine Optionsgruppe mit den beiden Möglichkeiten hinzu.

So kann der Benutzer auch nach der Sortierung etwa nach dem Artikelnamen die Datensätze nach dem Index filtern lassen. Dazu fügen wir noch die Optionsgruppe **ogrFilter** wie in Bild 2 zum Hauptformular hinzu.

Einzelne Filterausdrücke extrahieren

Wir beginnen einfach und wollen zunächst alle Filterausdrücke aus dem Textfeld **txtBereiche** extrahieren. Das sind also alle Teile der im Textfeld enthaltenen Zeichenkette, die durch ein Komma voneinander getrennt werden.

Um diese schnell zu erhalten, nutzen wir die **Split**-Funktion, um aus der Zeichenkette ein **String**-Array mit den einzelnen Elementen zu bilden.

Dieses durchlaufen wir dann in einer **For...Next**-Schleife über alle Elemente. Die Indexwerte dieser Elemente

Bild 2: Optionsgruppe zum Einstellen des Kriteriums zum Filtern

ermitteln wir dabei mit den Funktionen **LBound** (unterer Indexwert) und **UBound** (oberer Indexwert):

```
Private Sub cmdFiltern_Click()
    Dim strFilter As String
    Dim strBereiche As String
    Dim strVergleichswerte() As String
    Dim i As Integer
    strBereiche = Nz(Me!txtBereiche, "")
    strVergleichswerte = Split(strBereiche, ",")
    For i = LBound(strVergleichswerte) To
                                UBound(strVergleichswerte)
        Debug.Print strVergleichswerte(i)
    Next i
End Sub
```

Wenn der Benutzer nun **1,3,5** eingibt, erhalten wir diese Ausgabe:

```
1
3
5
```

Damit lässt sich schon einmal arbeiten. Wenn wir Bereiche eingeben, erhalten wir auch die gewünschten Informationen, zum Beispiel bei **-2, 4, 7-9, 75-**. Das liefert:

```
-2
4
7-9
75-
```

Unzulässige Zeichen erkennen

Bevor wir überhaupt in die Analyse der einzelnen, durch Kommata getrennten Elemente einsteigen, wollen wir direkt prüfen, ob der Ausdruck unzulässige Zeichen enthält. Dazu nutzen wir die folgende Funktion, der wir zwei Parameter übergeben:

- die zu untersuchende Zeichenfolge und
- eine Zeichenkette mit den zulässigen Zeichen.

Der Aufruf dieser Funktion sieht beispielsweise wie folgt aus:

```
? NurZulaessigeZeichen("x-2, 4, 7-9, 75-", "[0-9,-]")
```

Als Ergebnis soll eine Zeichenkette ohne nicht zulässige Zeichen zurückgeliefert werden.

In diesem Fall ist das führende **x** ein nicht zulässiges Zeichen, also sollte der Aufruf dieses Ergebnis liefern:

```
-2, 4, 7-9, 75-
```

Die Funktion sieht wie folgt aus:

```
Public Function NurZulaessigeZeichen(strText As String, _
                                     strZeichen As String)
    Dim i As Integer
    Dim strTemp As String
    For i = 1 To Len(strText)
        If Mid(strText, i, 1) Like strZeichen Then
            strTemp = strTemp & Mid(strText, i, 1)
        End If
    Next i
    NurZulaessigeZeichen = strTemp
End Function
```

Die Funktion erwartet die nicht zulässigen Zeichen in einer Form, wie Sie mit einem **Like**-Vergleich genutzt werden kann, also in eckigen Klammern. Außerdem können Bereiche von Zeichen wie etwa die Zahlen von **0** bis **9** vereinfacht durch **0-9** angegeben werden.

Die Funktion durchläuft alle Zeichen der zu untersuchenden Zeichenkette in einer **For...Next**-Schleife, wobei jeweils ein Zeichen mit dem **Like**-Operator mit den Vergleichszeichen vergleichen wird. Ist das Zeichen in den mit dem zweiten Parameter übergebenen Zeichen enthalten, wird dieses an die Zeichenkette der Variablen **strTemp** angehängt. Diese wird nach dem Durchlaufen aller Zeichen an die aufrufende Instanz zurückgegeben.

Der Sinn dieser Funktion ist in unserem Fall, unerwünschte und gegebenenfalls versehentlich eingegebene Zeichen zu extrahieren. Diese Funktion bauen wir daher direkt in die Ereignisprozedur **cmdFiltern_Click** ein:

```
Private Sub cmdFiltern_Click()
    ...
    strBereiche = Nz(Me!txtBereiche, "")
    strBereiche = NurZulaessigeZeichen(strBereiche, _
                                       "[,-0-9]")
    ...
End Sub
```

Bereiche analysieren

Diese Bereiche analysieren wir nun. Dabei gibt es die folgenden Fälle:

- eine Zahl, zum Beispiel **4**,
- eine Zahl mit folgendem Bindestrich, zum Beispiel **75-**,
- eine Zahl mir führendem Bindestrich, zum Beispiel **-2** und
- zwei durch einen Bindestrich getrennte Zahlen, zum Beispiel **7-9**.

Diese vier Fälle müssen wir zunächst erkennen. Dazu verwenden wir eine neue Funktion, die wir in der **For...**

Next-Schleife für jedes durch Komma getrennte Element einmal aufrufen:

```
For i = LBound(strVergleichswerte) To UBound(strVergleichswerte)
    Debug.Print strVergleichswerte(i)
    FilterkriteriumErmitteln strVergleichswerte(i)
Next i
```

Die Funktion sieht zunächst wie folgt aus:

```
Private Function FilterkriteriumErmitteln(ByVal strVergleichswert As String)
    Dim intPos As Integer
    strVergleichswert = Replace(strVergleichswert, " ", "")
    intPos = InStr(1, strVergleichswert, "-")
    Select Case intPos
        Case 0
            Debug.Print "kein Minus"
        Case 1
            Debug.Print "Minus an erster Stelle"
        Case Else
            If intPos = Len(strVergleichswert) Then
                Debug.Print "Minus an letzter Stelle"
            Else
                Debug.Print "Minus dazwischen"
            End If
    End Select
End Function
```

Die Funktion ersetzt zunächst alle Leerzeichen durch leere Zeichenketten. Dann ermittelt sie in der Variablen **intPos** die Position des Minuszeichens (**-**). Im Falle von **0** ist kein Minuszeichen vorhanden, was wir aktuell einfach im Direktbereich vermerken. Lautet der Wert von **intPos** hingegen **1**, befindet sich das Minuszeichen an der ersten Stelle, was auf einen Ausdruck wie **-3** hindeutet. In allen anderen Fällen ist eine weitere Untersuchung nötig, die wir im **Else**-Zweig der **Select Case**-Bedingung prüfen. Hier prüfen wir, ob **intPos** gleich der Anzahl der Zeichen der übergebenen Zeichenkette entspricht, was bedeuten würde, dass sich das Minus-Zeichen an der letzten Stelle befindet. In allen anderen Fällen befindet sich das Minuszeichen nicht an erster und nicht an letzter Stelle, der Ausdruck sollte also wie 1-3 lauten.

Vergleichsausdrücke ermitteln

Nun reichern wir die obige Prozedur um die resultierenden Vergleichsausdrücke an. Dabei betrachten wir zunächst nur den Fall, dass wir das **ID**-Feld als Vergleichsfeld nutzen wollen und nicht den Index der angezeigten Elemente – der ist nämlich noch etwas komplizierter zu handhaben.

Die neue Version der Prozedur finden Sie in Listing 1. Die Prozedur erwartet nun mit dem ersten Parameter auch noch den Namen des Feldes, das als Vergleichsfeld verwendet werden soll. Im Vergleichswert entfernen wir die Leerzeichen und ermitteln dann die Position eines gegebenenfalls enthaltenen Minuszeichens.

Im ersten **Case**-Zweig der **Select Case**-Bedingung tragen wir für die Variable **strVergleichsausdruck** den Namen des Vergleichsfeldes, ein Gleichheitszeichen und den Vergleichswert ein, und zwar in der Form **<Vergleichsfeld> = <Vergleichswert>**.

Der **Case**-Zweig, der davon ausgeht, dass das Minuszeichen gleich das erste Zeichen ist, fügt den Namen des Vergleichsfelds mit dem Vergleichsoperator **<=** und dem Teil aus **strVergleichswert** zusammen, der nach dem Minuszeichen folgt, also in der Form **<Vergleichsfeld> <= <Vergleichswert>**.

Im **If**-Teil der **If...Then**-Bedingung des **Else**-Zweigs der **Select Case**-Bedingung, der davon ausgeht, dass sich das Minuszeichen am Ende von **strVergleichswert** befindet, fügen wir in **strVergleichsausdruck** den Wert aus **strVergleichsfeld**, den Vergleichsoperator **>=** und den Inhalt aus **strVergleichswert** bis zum Minuszeichen zusammen. Der Ausdruck lautet dann **<Vergleichsfeld> >= <Vergleichswert>**.

Benutzerdefinierte Felder in Outlook

Outlook bietet noch viel mehr Möglichkeiten, als es die Standardeinstellungen vermuten lassen. So können Sie beispielsweise zu einem Termin noch weitere benutzerdefinierte Felder hinzufügen, mit denen Sie wichtige Informationen zum Termin hinzufügen – zum Beispiel die Nummer des Kunden, dem Sie den Termin in Rechnung stellen, die Leistungsart oder auch die Nummer des Mitarbeiters, der den Termin durchgeführt hat. Das alles hat noch nicht direkt etwas mit Access zu tun, aber Outlook ist ja nichts anderes als eine Benutzeroberfläche für die Eingabe von Daten, die später mit Access weiterverarbeitet werden – zum Beispiel, um Rechnungen auf Basis der angefallenen Arbeitszeiten zu erstellen. Also schauen wir uns in diesem Beitrag zunächst einmal an, wie wir die benötigten Daten direkt über die Benutzeroberfläche von Outlook erfassen können.

Natürlich ist es für den geübten Access-Entwickler am einfachsten, Access zu nutzen, um Eingabeformulare zur Erfassung der gewünschten Daten zu programmieren. Allerdings funktioniert die tatsächliche Eingabe der Daten am besten, wenn dies für den Benutzer den geringsten Zusatzaufwand bedeutet. Und wenn dieser ohnehin mit Outlook arbeitet und seine Termine mit Kunden im Terminkalender speichert, kann man diese Daten auch gleich nutzen und nach der Übertragung nach Access dort auswerten und Rechnungen und andere Dokumente auf ihrer Basis erstellen. Wenn dann allerdings noch weitere Informationen abgefragt werden sollen, für deren Eingabe Outlook standardmäßig keine Möglichkeit vorsieht, kommt man schnell ins Straucheln und denkt: Vielleicht machen wir das doch lieber komplett mit Access. Ist ja schließlich gewohntes Terrain für

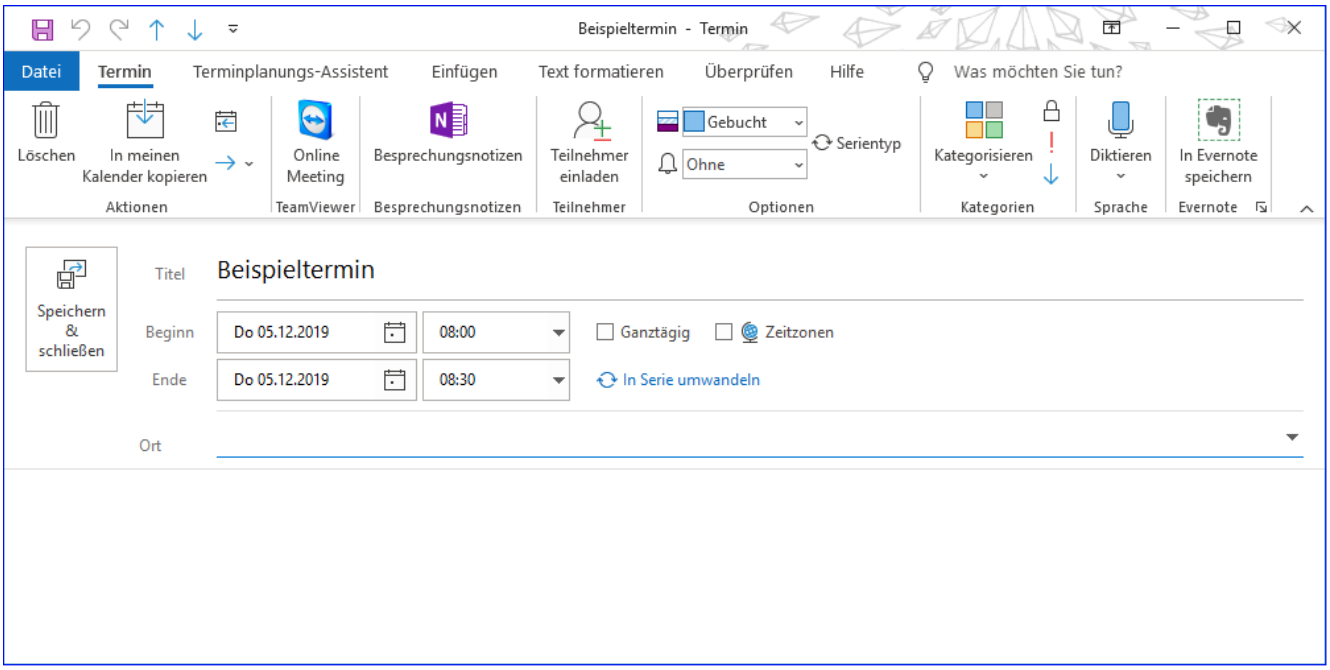

Bild 1: Ein Outlook-Termin mit den Standard-Steuerelementen

uns Entwickler. Besinnt man sich jedoch wieder zurück in Richtung optimaler Ergonomie, findet man in Outlook durchaus die Möglichkeit, auch zusätzliche Informationen zu erfassen. Wie das gelingt, wollen wir uns in diesem Beitrag ansehen.

Outlook-Termine

Outlook-Termine fügen Sie am einfachsten durch einen Doppelklick in die Kalenderansicht von Outlook hinzu. Dann finden Sie einen Termin vor, der wie in Bild 1 aussieht und bereits einige Felder zur Eingabe der wichtigsten Informationen vorsieht.

Termine können Sie aber auch noch auf andere Weise erstellen, und zwar, indem Sie Aufgaben aus der Aufgabenleiste in den Kalenderbereich ziehen. Dann wird direkt der Titel der Aufgabe als Titel des Termins übernommen und es erscheinen auch noch einige weitere Informationen der Aufgabe in dem für die Beschreibung vorgesehenen Textfeld des Termins (siehe Bild 2).

Wenn die Aufgabe weitere Informationen enthält wie eine Kategorie oder Abrechnungsinformationen, werden diese auch in diesen Beschreibungstext übernommen.

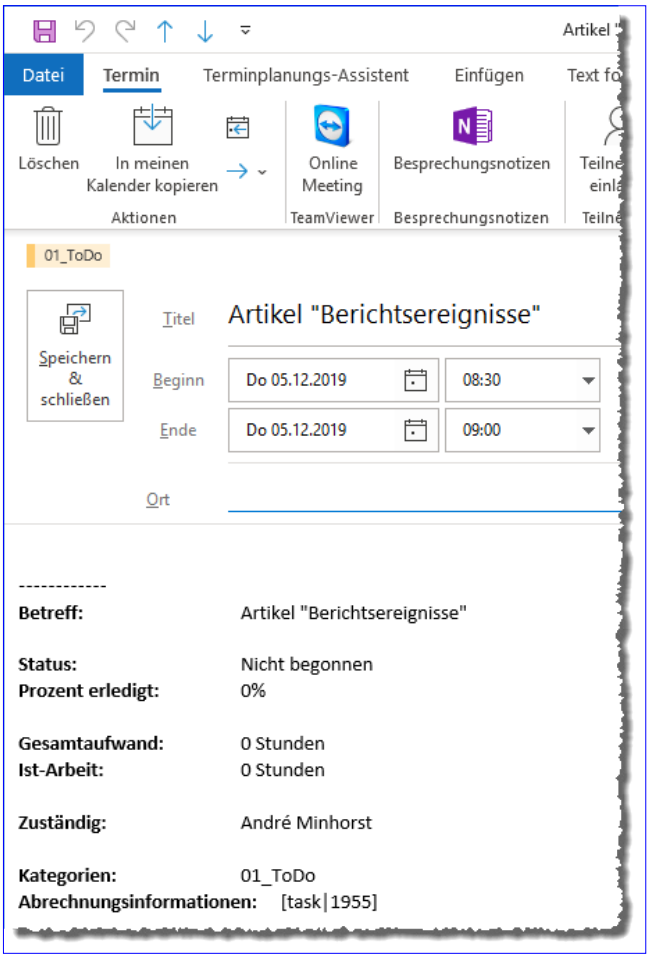

Bild 2: Aus einer Aufgabe erstellter Termin

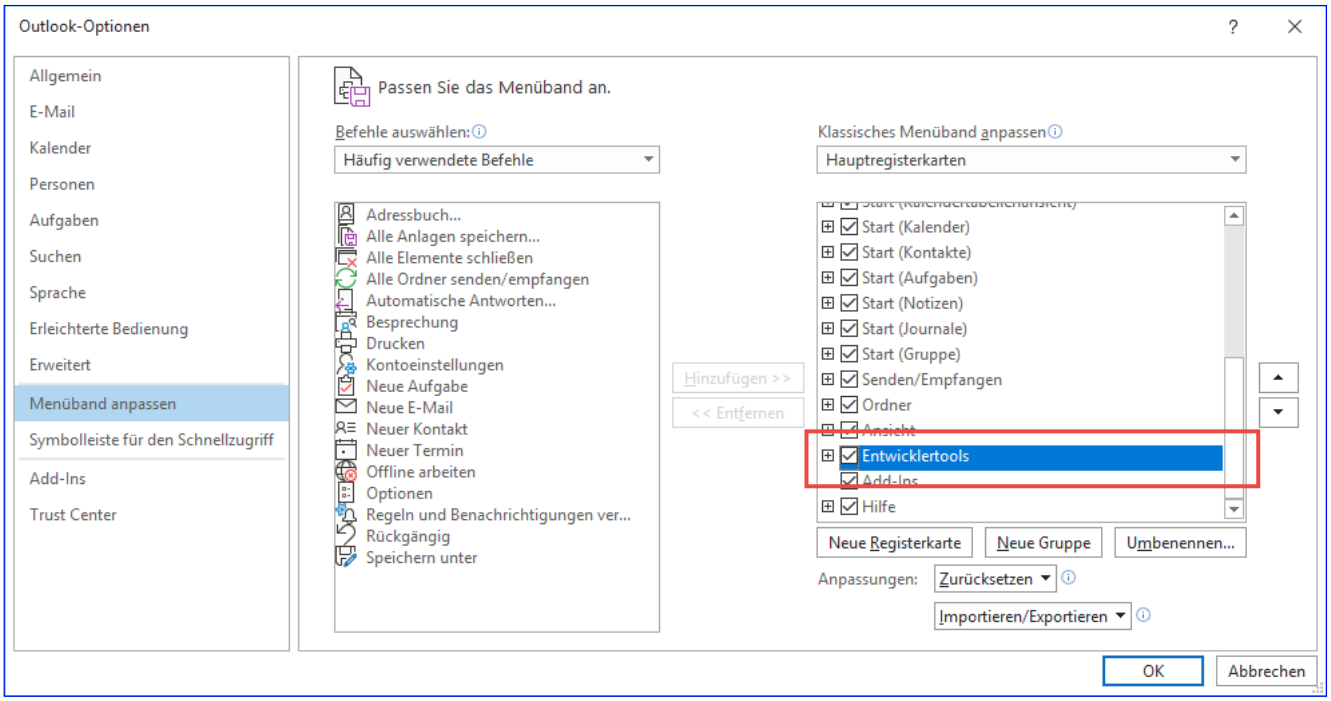

Bild 3: Aktivieren der Entwicklertools für Outlook

Entwicklerwerkzeuge verfügbar machen

Wie können wir nun einen Outlook-Termin anpassen und ihm so benutzerdefinierte Felder hinzufügen, die auch noch durch passende Steuerelemente in der Benutzeroberfläche des Fensters mit dem Termin erscheinen?

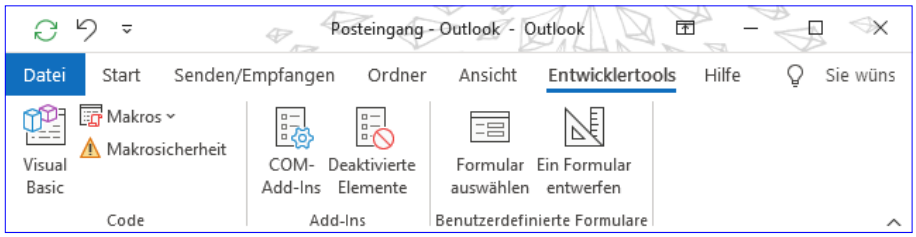

Bild 4: Entwicklertools-Tab im Hauptfenster von Outlook

Dazu müssen wir zunächst die Entwicklertools von Outlook verfügbar machen. Dazu gehen Sie wie folgt vor:

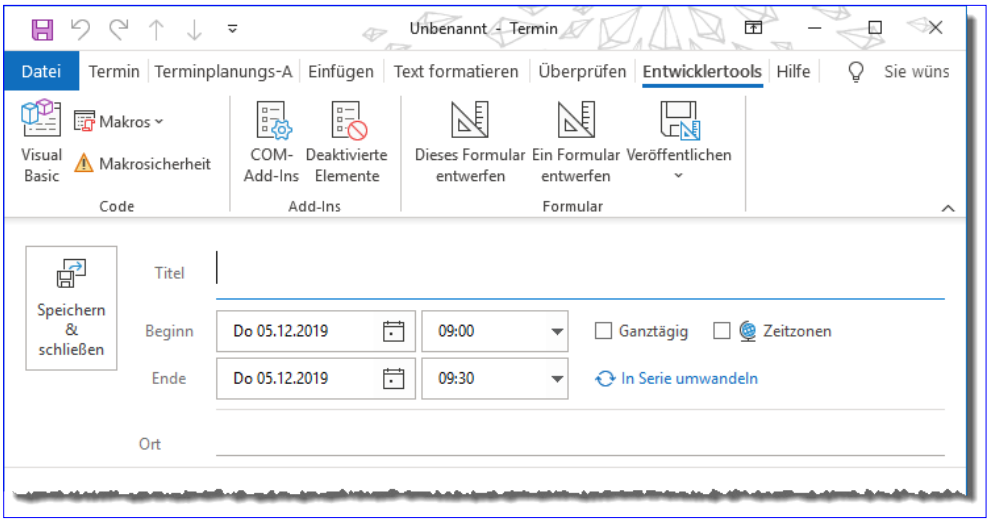

Bild 5: Entwicklertools-Tab im Termin-Fenster

Öffnen Sie die Outlook-Optionen mit dem Ribbon-Eintrag **Datei|Optionen**. Wechseln Sie im nun erscheinenden Dialog Outlook-Optionen in den Bereich **Menüband anpassen**. Aktivieren Sie in der rechten Liste den Eintrag **Entwicklertools** (siehe Bild 3).

Danach finden Sie bereits im Hauptfenster von Outlook das Tab **Entwicklertools** im Ribbon vor (siehe Bild 4). Hier benötigen wir dieses allerdings nicht, denn wir wollen ja

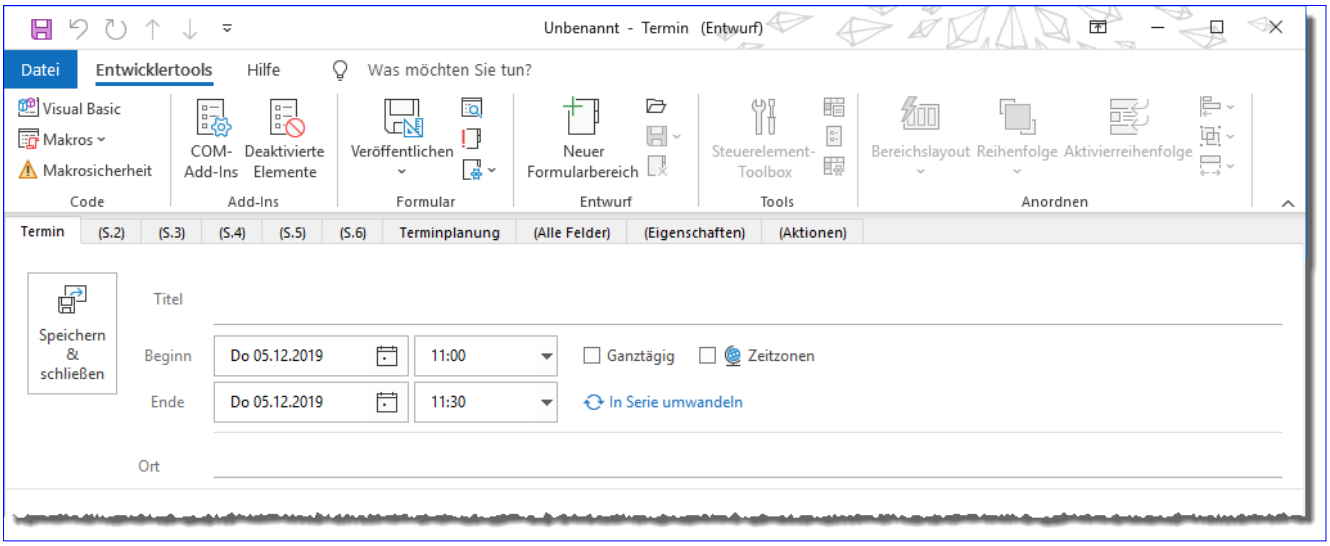

Bild 6: Neue Möglichkeiten in der Entwurfsansicht des Outlook-Termins

nicht das Hauptfenster, sondern das Termin-Fenster anpassen.

Termin-Fenster anpassen

Nun erstellen wir einen neuen Termin, auf dessen Basis wir unsere Anpassungen vornehmen wollen. Auch hier finden Sie nun das Tab **Entwicklertools** vor. Allerdings gibt es hier nicht wie im Hauptfenster von Outlook den Bereich **Benutzerdefinierte Formulare**, sondern den Bereich **Formular** mit den drei Befehlen **Dieses Formular entwerfen**, **Ein Formular entwerfen** und **Veröffentlichen** (siehe Bild 5).

Klicken wir hier auf **Dieses Formular entwerfen**, wird es spannend. Es tauchen dann nämlich einige neue Elemente im Ribbon auf, unter anderem zum Anlegen neuer Formularbereiche oder eine Steuerelement-Toolbox (siehe Bild 6). Und im unteren Bereich finden wir neben der Seite **Termin** noch Vorlagen für weitere Seiten wie **(S.2)** vor und Seiten, mit denen die Felder, Eigenschaften und Aktionen verwaltet werden können.

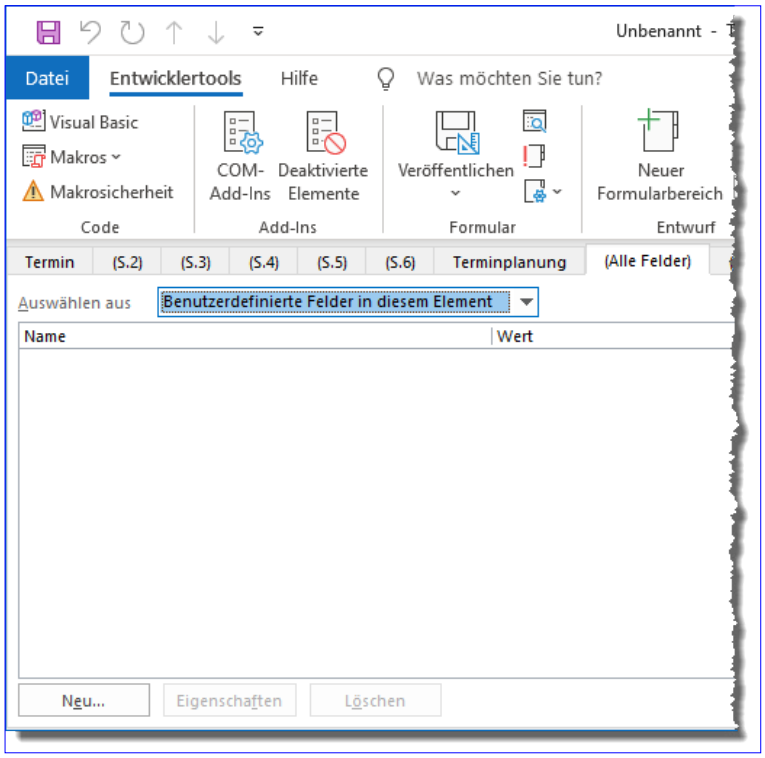

Bild 7: Verwalten der benutzerdefinierten Felder

Die schlechte Nachricht gleich zu Beginn: Die Startseite mit den Basiseigenschaften eines Termins können wir nicht bearbeiten. Das heißt, dass der Benutzer zur Eingabe weiterer Informationen zum Termin auf die zweite Seite wechseln muss.

Benutzerdefinierte Felder verwalten

Bevor wir Steuerelemente anlegen, um Daten in benutzerdefinierte Felder einzugeben, müssen wir diese zunächst definieren. Das erledigen wir, indem wir zur Registerkarte **(Alle Felder)** des Outlook-Termins in der Entwurfsansicht wechseln. Hier können wir mit der Schaltfläche **Neu...** neue Elemente hinzufügen (siehe Bild 7).

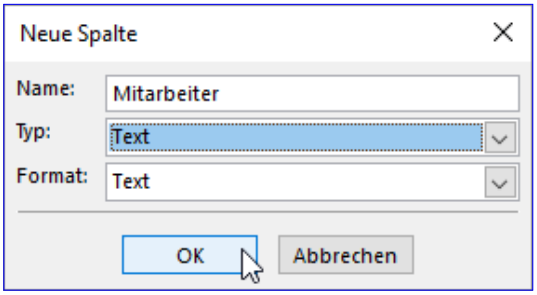

Bild 8: Neues Feld anlegen

Es erscheint dann der Dialog **Neue Spalte**, mit dem Sie die Bezeichnung des Feldes eingeben und den Typ und das Format auswählen (siehe Bild 8). Nachdem Sie hier etwa **Mitarbeiter** eingeben und die Schaltfläche **OK** unter Beibehaltung der übrigen Eigenschaften betätigen, wird anschließend das Feld **Mitarbeiter** in der Liste der Felder angezeigt.

Steuerelement für benutzerdefiniertes Feld hinzufügen

Um ein Textfeld etwa zu der mit **(S.2)** markierten Seite hinzuzufügen, wechseln wir zunächst zu dieser Seite. Dann aktivieren wir mit dem Ribbon-Befehl **Entwicklertools|Tools|Steuerelement-Toolbox** die **Toolsammlung** und fügen daraus ein Textfeld zum Entwurf des Bereichs **(S.2)** hinzu und auch noch ein passendes

Outlook-Termine nach Access einlesen

Outlook-Termine, die man über verschiedene Clients pflegt, möchte man vielleicht später einmal nach Access exportieren, um diese dort beispielsweise für die Erfassung der Projektzeiten beim Kunden heranzuziehen. Dafür gibt es verschiedene Methoden. Outlook selbst bietet einen Export an, mit dessen Ergebnis Sie arbeiten können. Oder Sie greifen direkt von Access aus per VBA auf das Objektmodell von Outlook zu, um die Termine Stück für Stück einzulesen. Dieser Beitrag zeigt, was es dabei zu beachten gibt und wie Sie auch benutzerdefinierte Felder in Terminen einlesen können.

Export per Bordmittel

Als Erstes schauen wir uns die Bordmittel zum Exportieren von Outlook-Terminen an, denn wir wollen ja keine aufwendige Access-Programmierung starten, wenn es auch einfach geht.

Den notwendigen Befehl finden Sie in Outlook 2016 im Backstage-Bereich unter **Öffnen und Exportieren**. Die benötigte Schaltfläche heißt **Importieren/Exportieren** (siehe Bild 1).

Nachdem wir diese Taste betätigt haben, finden wir den Dialog **Import/Export-Assistent** vor.

Dieser bietet nur einen für uns interessanten Eintrag in der Liste der Aktionen an, nämlich **In Datei exportieren** (siehe Bild 2).

Der nach der Auswahl von **In Datei exportieren** und dem Betätigen der **Weiter**-Schaltfläche angezeigte Dialog bietet wiederum zwei Möglichkeiten an:

- **Durch Trennzeichen getrennte Werte** und
- **Outlook-Datendatei (.pst)**.

Hier wählen wir den Eintrag **Durch Trennzeichen getrennte Werte** aus und klicken wiederum auf **Weiter** (siehe Bild 3).

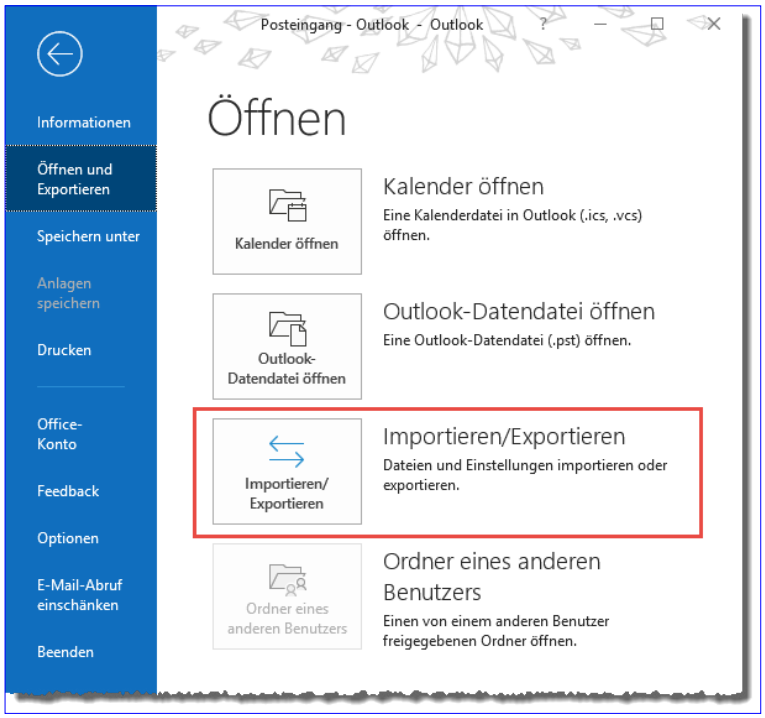

Bild 1: Befehl zum Exportieren von Outlook-Daten

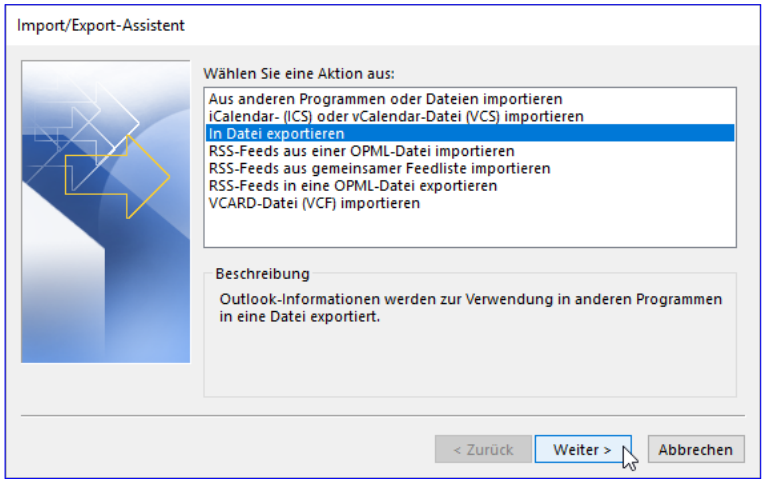

Bild 2: Import- und Export-Möglichkeiten

Das öffnet die nächste Seite des Dialogs und wir finden die Ordnerstruktur von Outlook vor (siehe Bild 4). Hier suchen wir nach dem Ordner **Kalender** und markieren diesen, bevor wir zum nächsten Schritt wechseln.

Hier legen wir fest, in welche Datei die Termindaten exportiert werden sollen. Dazu geben Sie entweder Pfad und Datei ein oder wählen diese mit dem Dialog aus, den Sie über die **Durchsuchen...**-Schaltfläche öffnen können (siehe Bild 5).

Danach wird es interessanter: Im nun erscheinenden Übersichtsdialog öffnen Sie über die Schaltlfäche **Benutzerdefinierte Felder** zuordnen einen weiteren Dialog namens **Benutzerdefinierte Felder zuordnen** öffnen.

Hier können Sie festlegen, welche der Felder überhaupt exportiert werden sollen und mit welcher Spaltenüberschrift diese versehen werden sollen (siehe Bild 6).

Haben Sie diese Liste nach Ihren Wünschen bearbeitet, schließen Sie den Dialog wieder und führen den Export mit einem Klick auf die Schaltfläche **Fertigstellen** aus.

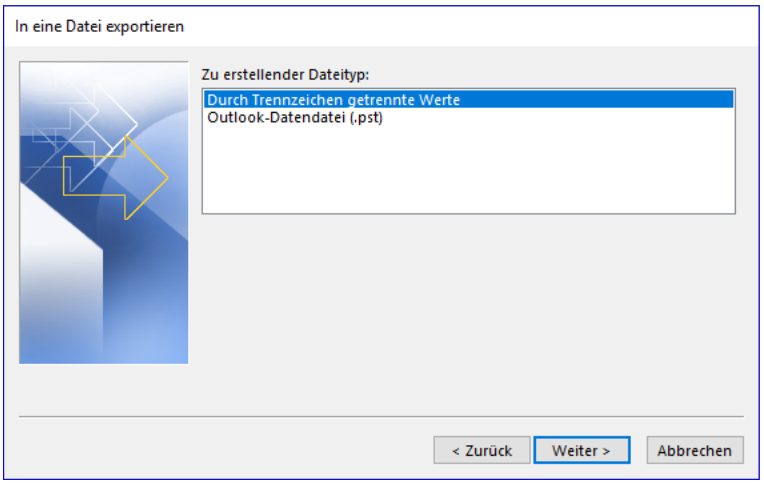

Bild 3: Export in eine Liste mit Trennzeichen

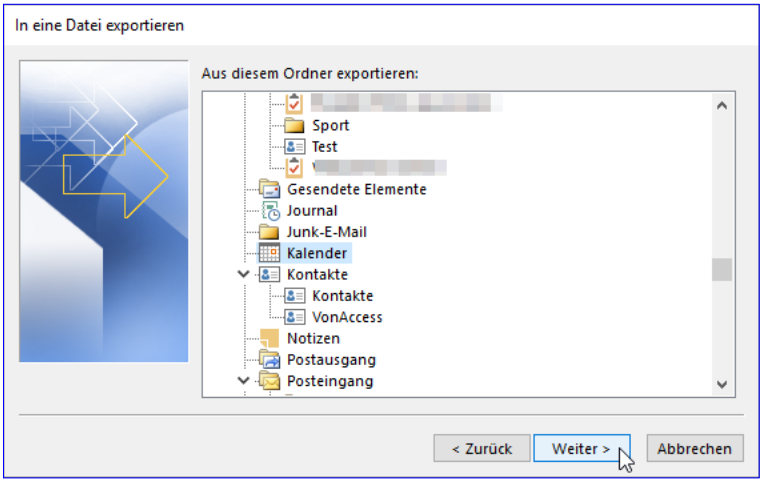

Bild 4: Export der Kalenderdaten

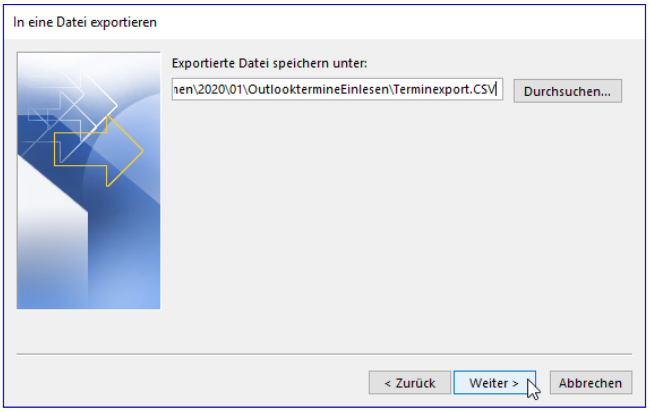

Bild 5: Angabe der Exportdatei

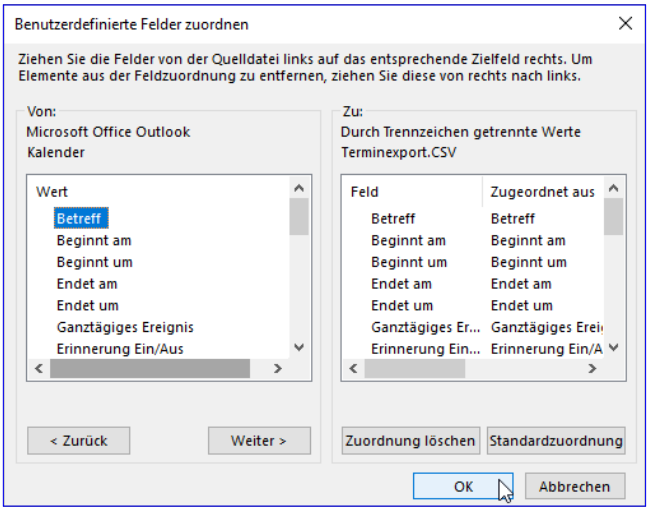

Bild 6: Zuordnen der Felder an die Spalten der Exportdatei

Danach erscheint noch ein abschließender Dialog namens **Zeitraum festlegen** (siehe Bild 7). Hier können Sie den vom Export vorgeschlagenen Zeitraum anpassen. Außerdem erhalten Sie hier den Hinweis, dass von Termin- und Aufgabenserien, die in diesen Zeitraum fallen, nur die Termine dieses Zeitraums berücksichtigt werden.

Danach geht der Export endlich los und Sie können den Fortschritt im Dialog aus Bild 8 verfolgen.

Danach schauen wir uns die Export-Datei direkt in Excel an. Bild 9 zeigt nur zwei Termine

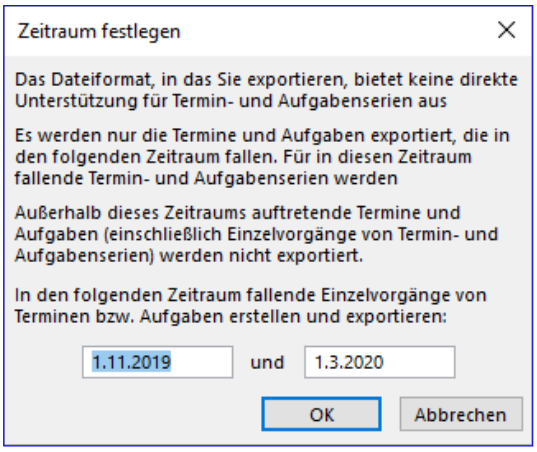

Bild 7: Festlegen des Zeitraums

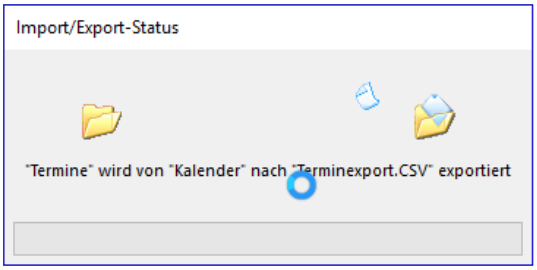

Bild 8: Exportfortschritt

an, die so aus Outlook in die **.csv**-Datei exportiert wurden.

Die erste Zeile enthält die Spaltenüberschriften, die zweite den ersten Termin, die folgenden Zeilen den zweiten Termin. Dieser erstreckt sich über mehrere Zeilen, weil der Inhalt des Feldes **Beschreibung** mit Text gefüllt ist. Dies ist der Fall, wenn ein Termin erstellt wurde, indem der Benutzer eine Aufgabe in den Kalender gezogen hat.

Analyse des Ergebnisses

Wenn wir uns das Ergebnis anschauen, finden wir ein paar Nachteile vor. Der erste ist, dass wir unter anderem Termine

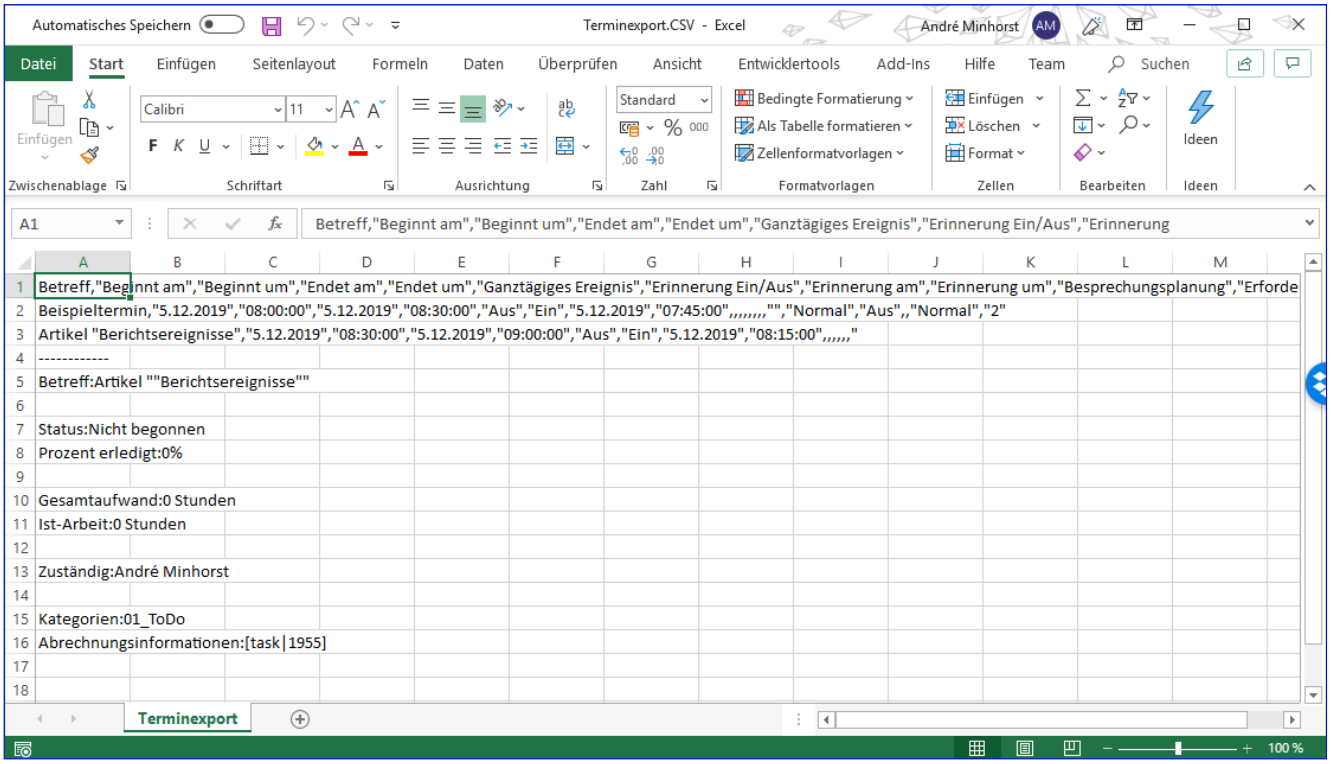

Bild 9: Export von zwei Terminen

haben, die mehrere Zeilen in Beschlag nehmen. Der zweite ist, dass wir hier sicher nicht alle Eigenschaften vorfinden, die ein Termin hat – zumindest die benutzerdefinierten Eigenschaften wurden nicht mitexportiert.

Zusammenfassend scheint es also praktischer zu sein, direkt per VBA auf die Termine und ihre Eigenschaften zuzugreifen.

Zugriff per VBA

Um von einer Access-Datenbank auf Outlook zuzugreifen, benötigen wir einen Verweis auf die entsprechende Objektbibliothek. Diese binden wir im VBA-Editor über den Menüeintrag **Extras|Verweise** ein, der den Dialog **Verweise** öffnet. Hier wählen wir den Eintrag **Microsoft Outlook 16.0 Object Library** aus (für Office 2016 – sonst wählen Sie die verfügbare Version für die aktuelle Office-Installation). Das Ergebnis sieht dann im **Verweise**-Dialog wie in Bild 10 aus.

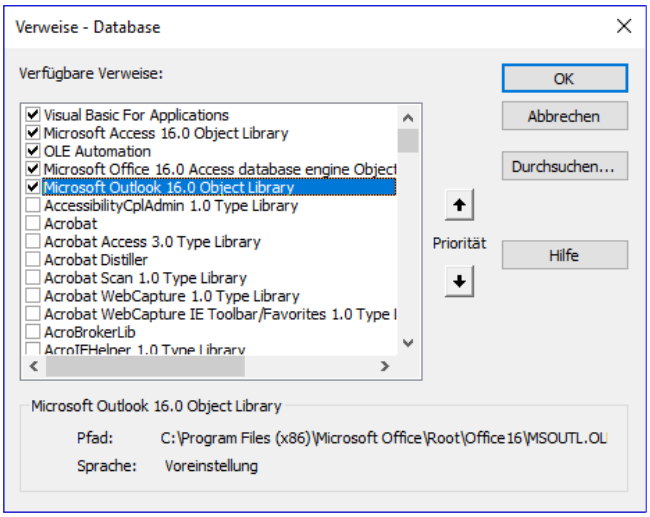

Bild 10: Verweis auf die Outlook-Bibliothek

Outlook referenzieren

Als Erstes benötigen wir eine Objektvariable, mit der wir die geöffnete Outlook-Instanz referenzieren oder eine neue erstellen. Das realisieren wir, indem wir zunächst eine privat deklarierte Objektvariable in einem neuen Standardformular anlegen:

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

Diese füllen wir mit der Funktion **GetOutlook**. Sie prüft, ob **m_Outlook** den Wert **Nothing** enthält. Falls ja, wird **m_Outlook** mit einer mit dem **New**-Schlüsselwort erzeugten neuen Instanz gefüllt. Danach liefert **GetOutlook** den Wert von **m_Outlook** als Funktionswert zurück. Damit braucht **m_Outlook** nur beim ersten Aufruf gefüllt zu werden und die Funktion kann bei folgenden Aufrufen auf die bereits existierende Instanz zugreifen:

```
Public Function GetOutlook() As Outlook.Application
    If m_Outlook Is Nothing Then
        Set m_Outlook = New Outlook.Application
    End If
    Set GetOutlook = m_Outlook
End Function
```

Das **New**-Schlüsselwort holt übrigens nur eine neue Outlook-Instanz, wenn keine Instanz läuft.

Dies kann zum Beispiel in dem Fall schiefgehen, wenn eine Outlook-Instanz durch den Benutzer geöffnet wurde, die obige Prozedur **m_Outlook** mit einem Verweis auf diese Instanz füllt, der Benutzer Outlook über die Benutzeroberfläche schließt und **GetOutlook** dann nochmals auf **m_Outlook** zugreift.

In diesem Fall löst der Zugriff auf eine der Eigenschaften des **Outlook-Application**-Objekts den Fehler aus Bild 11 aus, weil die Objektvariable ein ungültiges Objekt enthält.

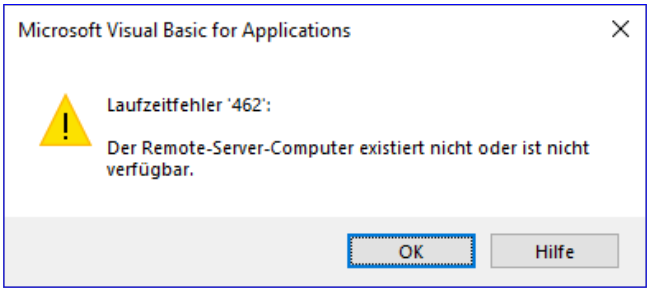

Bild 11: Fehler beim Zugriff auf eine leere Objektvariable

Also erweitern wir die Funktion ein wenig:

```
Public Function GetOutlook() As Outlook.Application
    Dim strDummy As String
    If m_Outlook Is Nothing Then
        Set m_Outlook = New Outlook.Application
    End If
    On Error Resume Next
    strDummy = m_Outlook.Name
    If Not Err.Number = 0 Then
        Set m_Outlook = New Outlook.Application
    End If
    On Error GoTo 0
    Set GetOutlook = m_Outlook
End Function
```

Hier versuchen wir nun, bei deaktivierter Fehlerbehandlung auf die Eigenschaft **Name** von **m_Outlook** zuzugreifen. Wenn **m_Outlook** zu diesem Zeitpunkt nicht korrekt gefüllt, löst dies einen Fehler aus, den wir mir der folgenden **If...Then**-Bedingung abfragen und **m_Outlook** gegebenenfalls mit einer neuen Outlook-Instanz füllen.

Ordner mit Terminen referenzieren

Danach benötigen wir Zugriff auf den Ordner mit den Terminen. Wie bekommen wir diesen? Dazu gibt es mehrere Möglichkeiten. Die gängigste ist, den Standardordner für die Termine aufzurufen. Das gelingt mit der folgenden Funktion:

```
Public Function GetDefaultCalendar() As Outlook.Folder
    Dim objMAPI As Outlook.NameSpace
    Dim objFolder As Outlook.Folder
    Set objMAPI = GetOutlook.GetNamespace("MAPI")
    Set objFolder = _
                objMAPI.GetDefaultFolder(olFolderCalendar)
    Set GetDefaultCalendar = objFolder
End Function
```

Hier holen wir erst den **MAPI**-Namespace über die **GetNamespace**-Methode des mit **GetOutlook** referenzierten **Outlook.Application**-Objekts und referenzieren diesen mit der Variablen **objMAPI**. Dann verwenden wir dessen Funktion **GetDefaultFolder** mit dem Parameter **olFolderCalendar**. Die Funktion gibt das erhaltene Objekt dann als Verweis des Typs **Outlook.Folder** an die aufrufende Anweisung zurück.

Damit können Sie nun beispielsweise den Pfad zu diesem Ordner innerhalb von Outlook mit folgender Anweisung ausgeben:

```
? GetDefaultCalendar.FolderPath
\\Outlook\Kalender
```

In einer aufrufenden Prozedur können wir den Verweis auf dieses **Folder**-Objekt nun auch in einer Objektvariablen speichern und dann weiter damit arbeiten:

```
Public Function AppointmentsLesen()
    Dim objAppointments As Outlook.Folder
    Set objAppointments = GetDefaultCalendar
    Debug.Print objAppointments.FolderPath
End Function
```

Ordner selbst auswählen

Sie können dem Benutzer auch die Möglichkeit geben, einen anderen Ordner als Kalenderordner auszuwählen. Das ist beispielsweise sinnvoll, wenn die zu untersuchenden Termine sich nicht im Standardordner befinden, sondern in einem anderen Termin-Ordner.

Die folgende Funktion öffnet einen entsprechenden Dialog:

```
Public Function GetCalenderFolder() As Outlook.Folder
    Dim objMAPI As Outlook.NameSpace
    Dim objFolder As Outlook.Folder
    Set objMAPI = GetOutlook.GetNamespace("MAPI")
    Set objFolder = objMAPI.PickFolder
    Set GetCalenderFolder = objFolder
End Function
```

Outlook-Termine programmieren

Wie wir im Beitrag »Benutzerdefinierte Felder in Outlook« gezeigt haben, können Sie beispielsweise einem Outlook-Termin weitere Seiten mit benutzerdefinierten Steuerelementen hinzufügen. Wie nicht anders zu erwarten, lassen sich diese Elemente auch programmieren. Ein Einsatzzweck ist, die benutzerdefinierten Steuerelemente beziehungsweise die dahinter stehenden Felder vorab mit Standardwerten zu füllen – beispielsweise das Feld »Mitarbeiter« mit dem Namen des aktuellen Mitarbeiters.

Ereignisse von Terminen und Steuerelementen

Uns als Access-Entwickler interessiert neben den Objekten und Eigenschaften auch, welche Ereignisse die Objekte auslösen, mit denen wir arbeiten. Und natürlich lösen auch Outlook-Termine Ereignisse aus und auch die durch den Entwickler hinzugefügten Steuerelemente.

Schauen wir in den Objektkatalog, finden wir allein für das **AppointmentItem**-Objekt, um das es hier im Wesentlichen geht, einige Ereignisse (siehe Bild 1).

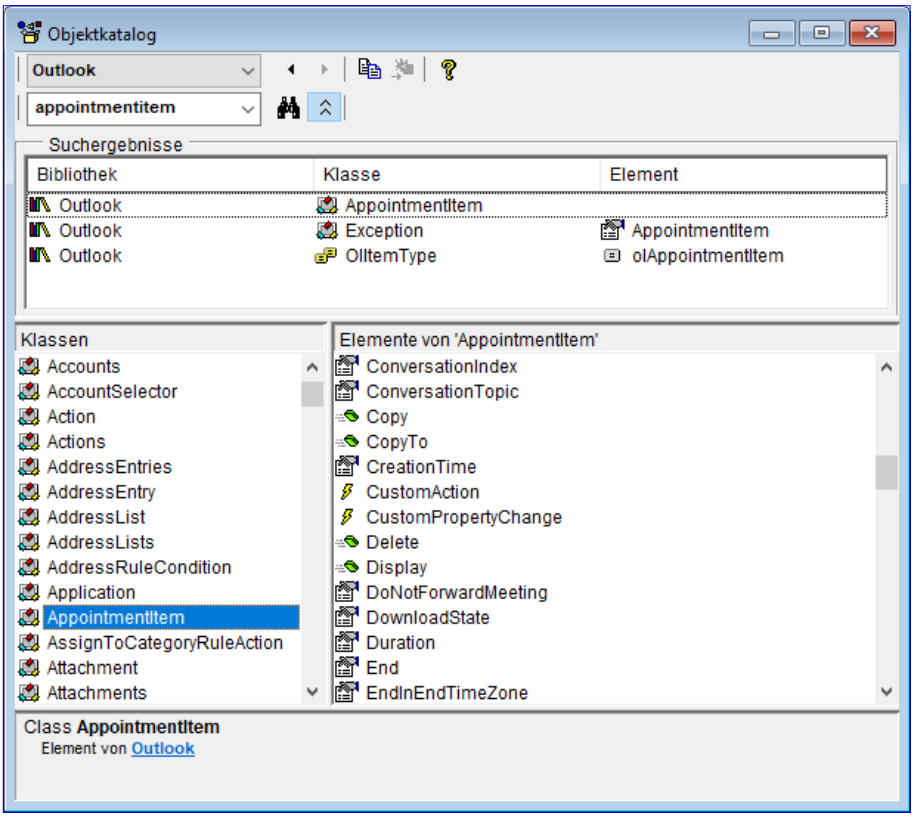

Bild 1: Ereignisse des **AppointmentItem**-Elements im Objektkatalog

Um ein **AppointmentItem**, in diesem Beitrag auch Termin genannt, so zu programmieren, dass wir die hier aufgeführten Ereignisse implementieren, benötigen wir eine andere Herangehensweise als beispielsweise bei der Programmierung eines Access-Formulars oder seiner Steuerelemente.

Dort brauchen wir nur explizit die gewünschten Ereignisse in das jeweilige Klassenmodul zu schreiben und für die Ereigniseigenschaft den Wert **[Ereignisprozedur]** zu hinterlegen. Unter Outlook sieht das etwas anders aus – wie genau, schauen wir uns nun im Detail an.

Ereignisse beim Bearbeiten

Als Erstes schauen wir uns an, wie wir Ereignisse abfangen können, die beim Bearbeiten eines Termins ausgelöst werden. Dazu brauchen wir eine Instanz des Termins. Wie kommen wir an diese heran? Dazu gehen wir den Weg über das sogenannte **Inspector**-Element. **Inspector** ist das Fenster, in dem ein Outlook-Objekt – wie in diesem Fall ein Termin – angezeigt wird. Über das **Inspector**-Objekt können wir schließlich auf den enthaltenen Termin

zugreifen, wenn der Benutzer diesen geöffnet hat.

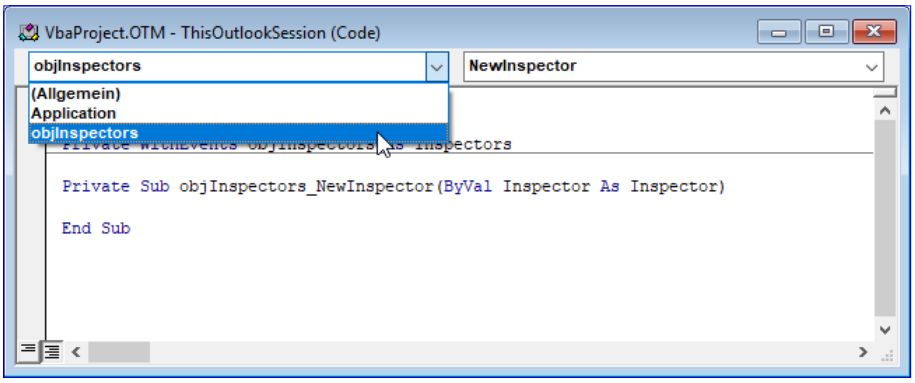

Bild 2: Anlegen des Ereignisses für die **Inspectors**-Auflistung

Neu geöffneten Inspector referenzieren

Um auf ein frisch geöffnetes Termin-Objekt zugreifen zu können, benötigen wir das **Inspector**-Objekt. Dieses erhalten wir über die **Inspectors**-Auflistung. Dazu deklarieren wir eine Objektvariable wie die folgende im Klassenmodul **ThisOutlookSession** des VBA-Projekts von Outlook:

```
Private WithEvents objInspectors As Inspectors
```

Wir verwenden das Schlüsselwort **WithEvents**, um die Ereignisse dieses Objekts implementieren zu können. Die **Inspectors**-Auflistung hat nur eine Ereigniseigenschaft, die wir implementieren können, indem wir im linken Kombinationsfeld im VBA-Fenster des Moduls **ThisOutlookSession** den Eintrag **objInspectors** auswählen. Im rechten Kombinationsfeld wird dann automatisch das einzige Ereignis ausgewählt und im VBA-Fenster die Ereignisprozedur **objInspectors_NewInspector** angelegt (siehe Bild 2). Diese Prozedur sieht zunächst wie folgt aus:

```
Private Sub objInspectors_NewInspector(ByVal Inspector
                                       As Inspector)

End Sub
```

Diese Ereignisprozedur wird im aktuellen Zustand des Moduls niemals ausgelöst, denn die Objektvariable **objInspectors** wurde noch nicht gefüllt. Das holen wir nun nach, indem wir ihr in der Ereignisprozedur **Application_Startup** die Auflistung zuweisen:

```
Private Sub Application_Startup()
    Set objInspectors = Outlook.Application.Inspectors
End Sub
```

Die Prozedur **Application_Startup** wird bei jedem Start von Outlook aufgerufen. Sie können nun Outlook neu starten, um diese Prozedur auszulösen, oder Sie rufen diese einfach durch Platzieren der Einfügemarke in der Prozedur und Betätigen der Taste **F5** auf. Setzen Sie außerdem einen Haltepunkt in die Prozedur **objInspectors_NewInspector**. Wenn Sie nun zum Anwendungsfenster von Outlook wechseln und durch einen Doppelklick in den Kalender einen neuen Termin öffnen, wird die Prozedur **objInspectors_NewInspector** ausgelöst und stoppt am Haltepunkt.

Neues Inspector-Fenster referenzieren

Damit kommen wir zum nächsten Schritt: Der Parameter **Inspector** der Prozedur **objInspectors_NewInspector** liefert einen Verweis auf das neue **Inspector**-Fenster. Dieses können wir nun wiederum mit einer Objektvariablen referenzieren, um seine Ereignisse implementieren zu können. Dazu fügen wir dem Modul **ThisOutlookSession** die folgende Objektvariable hinzu, wiederum mit dem Schlüsselwort **WithEvents** ausgestattet:

```
Private WithEvents objInspector As Inspector
```

Wenn wir das neue Objekt im linken Kombinationsfeld des VBA-Fensters auswählen, finden wir im rechten Kombinationsfeld alle dafür zur Verfügung stehenden Ereignisse vor – zum Beispiel eines, das beim Wechseln der Seite ausgelöst wird (siehe Bild 3). Um dieses auszuprobieren, stellen wir in der Ereignisprozedur **objInspectors_NewIn-**

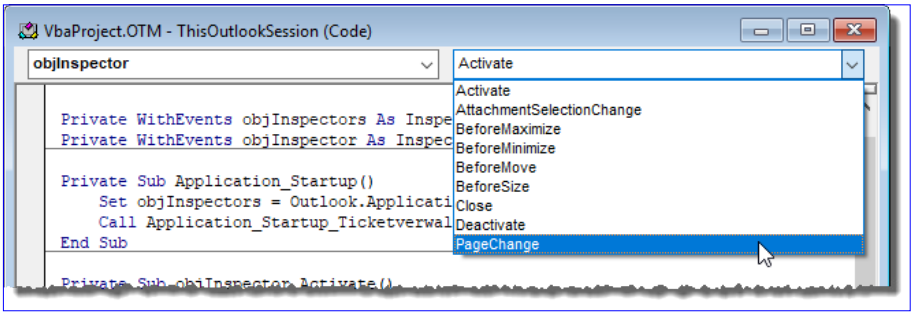

Bild 3: Ereignisse eines Outlook-Inspectors

spector die Objektvariable **objInspector** auf das mit dem Parameter **Inspector** gelieferte Objekt ein:

```
Private Sub objInspectors_NewInspector(_
                            ByVal Inspector As Inspector)
    Set objInspector = Inspector
End Sub
```

Dann implementieren wir das Ereignis **PageChange** der neuen Objektvariablen und geben darin den Wert des Parameters **ActivePageName** aus:

```
Private Sub objInspector_PageChange(_
                                ActivePageName As String)
    Debug.Print "Aktuelle Seite: " & ActivePageName
End Sub
```

Damit können wir nun zwischen den Seiten **Termin** und **Terminplanung** hin- und herwechseln und erhalten jeweils den Titel der aktiven Seite im Direktbereich. Wenn Sie, wie im Beitrag **Benutzerdefinierte Felder in Outlook** (**www.access-im-unternehmen.de/1221**) beschrieben, bereits eine weitere Seite namens **Termindetails** angelegt haben, können Sie auch diese aufrufen und den jeweiligen Namen ausgeben.

Unterscheidung zwischen verschiedenen Inspector-Objekten

Wir haben nun bewusst immer Termin-Einträge im **Inspector**-Objekt geöffnet. Aber auch beim Öffnen von E-Mails oder Kontakten wird das Ereignis **objInspectors_NewInspector** ausgelöst. Wenn wir nun nicht mehr nur mit **objInspector** das geöffnete **Inspector**-Objekt referenzieren wollen, sondern auch mit einer weiteren Objektvariablen namens **objAppointmentItem** den im Inspector enthaltenen Termin, kann es zu Problemen kommen – beispielsweise, wenn wir gar keinen Termin-Inspector geöffnet haben.

In diesem Fall müssen wir also zuvor prüfen, ob überhaupt ein Inspector mit einem Termin geöffnet wurde.

Dazu deklarieren wir wieder eine Objektvariable mit dem Schlüsselwort **WithEvents** für das **AppointmentItem**-Objekt im neu geöffneten Inspector:

```
Private WithEvents objAppointmentItem As AppointmentItem
```

Dann erweitern wir die Prozedur **objInspectors_NewInspector** so, dass wir den Typ des enthaltenen Elements prüfen und nur im Falle eines Appointments einen Verweis auf das Element in die Variable **objAppointmentItem** schreiben:

```
Private Sub objInspectors_NewInspector(_
                            ByVal Inspector As Inspector)
    Set objInspector = Inspector
    Select Case Inspector.CurrentItem.Class
        Case olAppointment
            Set objAppointmentItem = Inspector.CurrentItem
    End Select
End Sub
```

Interessant sind zum Beispiel die beiden Ereignisse **PropertyChange** und **CustomPropertyChange**, die jeweils beim Ändern von eingebauten und benutzerdefinierten Ereignissen ausgelöst werden. Das Ereignis **PropertyChange** können Sie so implementieren:

```
Private Sub objAppointmentItem_PropertyChange(_
                                ByVal Name As String)
    Debug.Print "Eigenschaft geändert: " & Name
End Sub
```

Hier können wir leider nicht über die **Properties**-Auflistung auf die Eigenschaft mit dem geänderten Wert zugreifen, denn diese Auflistung gibt es schlicht nicht. Sie könnten aber per **Select Case** auf die Änderung bestimmter Eigenschaften reagieren, die für den Anwendungsfall interessant sind. Im Beispiel aus Listing 1 geben wir die Werte für die Eigenschaften **Subject** und **Start** aus, wenn diese geändert wurden.

Objektvariablen erneut initialisieren

Wenn Sie mit den hier vorgestellten Ereignisprozeduren experimentieren, werden Sie gelegentlich feststellen, dass diese nicht mehr wie gewünscht funktionieren. Es kann sein, dass die Objektvariablen bei Codeänderungen zur Laufzeit oder durch Laufzeitfehler geleert werden. Damit werden dann natürlich auch nicht mehr die für diese Objektvariablen implementierten Ereignisse ausgelöst. Sie sollten also gelegentlich die Prozedur **Application_Startup** erneut ausführen, um die Objektvariablen zu initialisieren.

Änderungen von benutzerdefinierten Eigenschaften

Im oben genannten Beitrag **Benutzerdefinierte Felder in Outlook** haben wir einem Termin eine neue Seite mit einem Feld namens **Mitarbeiter** hinzugefügt. Eine Änderungen an einem solchen Feld können wir nicht über die Ereignisprozedur **PropertyChange** erfassen, sondern wir benötigen dazu die Ereignisprozedur **CustomPropertyChange**. Diese liefert ebenfalls mit dem **Name**-Parameter den Namen der geänderten Eigenschaft. Im Gegensatz zu den eingebauten Eigenschaften können wir hier allerdings über die Auflistung **UserProperties** des **AppointmentItem**-Elements auf die Eigenschaft und ihren Wert zugreifen. Das erledigen wir wie folgt:

```
Private Sub objAppointmentItem_CustomPropertyChange(_
                                ByVal Name As String)
    Debug.Print "Benutzerdefinierte Eigenschaft _
                                geändert: " & Name
    Debug.Print "Neuer Wert: " _
                & objAppointmentItem.UserProperties(Name)
End Sub
```

Schließen von Termin-Objekten

Mit den oben vorgestellten Ereignissen können Sie bereits auf Änderungen an den eingebauten und benutzerdefinierten Eigenschaften von Outlook-Terminen zugreifen. Damit könnten Sie die geänderten Eigenschaften beispielsweise direkt mit einer Tabelle in einer Access-Datenbank synchronisieren. Allerdings wäre es ressourcenschonender, wenn wir dies erst beim Schließen des Termin-Elements erledigen.

```
Private Sub objAppointmentItem_PropertyChange(ByVal Name As String)
    Select Case Name
        Case "Subject"
            Debug.Print "Die Eigenschaft '" & Name & "' wurde geändert."
            Debug.Print "Neuer Wert: " & objAppointmentItem.Subject
        Case "Start"
            Debug.Print "Die Eigenschaft '" & Name & "' wurde geändert."
            Debug.Print "Neuer Wert: " & objAppointmentItem.Start
    End Select
    Debug.Print "Eigenschaft geändert: " & Name
End Sub
```

Listing 1: Ereignisprozedur beim Ändern einer eingebauten Eigenschaft

Dazu können wir beispielsweise eines der Ereignisse **Close** oder **Unload** nutzen. Wir legen beide Ereignisprozeduren an und versehen diese mit Haltepunkten, um zu prüfen, zu welchem Zeitpunkt diese beiden Ereignisse ausgelöst werden und in welcher Reihenfolge. In unseren Tests haben wir festgestellt, dass nur **Close**

ausgelöst wird, **Unload** jedoch nicht.

Für unsere Zwecke würde es jedoch reichen, wenn die **Close**-Ereignisprozedur ausgelöst wird – in dieser können wir noch auf alle Eigenschaften des Termin-Elements zugreifen und wir können das Schließen gegebenenfalls auch noch abbrechen, wenn Informationen fehlen. Das gestalten wir dann beispielsweise so:

```
Private Sub objAppointmentItem_Close(Cancel As Boolean)
    If objAppointmentItem.UserProperties("Mitarbeiter") _
                                        = "" Then
        MsgBox "Tragen Sie den Mitarbeiter ein."
        Cancel = True
    End If
End Sub
```

Das setzt natürlich voraus, dass Sie wie im Beitrag **Benutzerdefinierte Felder in Outlook** beschrieben, ein benutzerdefiniertes Feld namens **Mitarbeiter** angelegt und auch über die Benutzeroberfläche verfügbar gemacht haben. Praktisch wäre es, wenn wir den Bereich, in dem sich das Steuerelement zur Eingabe des benutzerdefinierten Feldes befindet, noch einblenden könnten. Dazu erweitern wir die Prozedur wie in Listing 2. Hier fragen wir den Benutzer noch, ob er den Mitarbeiter eintragen möchte. Falls ja, wechseln wir mit der Methode **SetCurrentFormPage** des **Inspector**-Objekts zum Formularbereich **Termindetails**.

```
Private Sub objAppointmentItem_Close(Cancel As Boolean)
    Dim intResult As VbMsgBoxResult
    If objAppointmentItem.UserProperties("Mitarbeiter") = "" Then
        intResult = MsgBox("Es ist kein Mitarbeiter zugeordnet. Noch eintragen?", vbYesNo)
        If intResult = vbYes Then
            objInspector.SetCurrentFormPage "Termindetails"
            Cancel = True
        End If
    End If
End Sub
```

Listing 2: Abbrechen des Schließens, wenn noch kein Mitarbeiter eingetragen wurde

Damit landen wir dann auf der Seite, die in Bild 4 abgebildet ist. Nach der Eingabe eines Wertes in das Textfeld können Sie das Termin-Element allerdings noch nicht schließen – es erscheint die gleiche Meldung wie zuvor. Der Grund ist, dass der eingegebene Wert nun zwar im Textfeld vorliegt, aber noch nicht in der zugrundeliegenden benutzerdefinierten Eigenschaft. Aber wann landet der Wert im benutzerdefinierten Feld und wie können wir nicht nur auf das benutzerdefinierte Feld, sondern auch auf das Steuerelement und seinen Inhalt zugreifen? Das schauen wir uns im nächsten Abschnitt an.

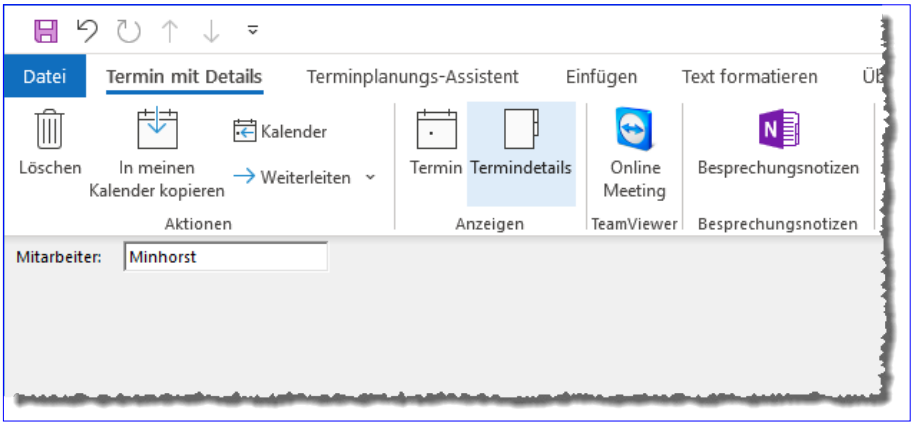

Bild 4: Wechseln zum benutzerdefinierten Formularbereich

Steuerelemente im Formularbereich referenzieren

Um Steuerelemente in einem Formularbereich zu referenzieren, benötigen wir zunächst einen Verweis auf den Formularbereich. Aber welchen Objekttyp hat der Formularbereich eigentlich? Das finden wir im Zweifel mit der **Typename**-Funktion heraus. Und wie kommen wir überhaupt an

Objekte im Ribbon verfügbar machen

Manchmal wird die Liste der Tabellen, Abfragen, Formulare und Co. im Navigationsbereich etwas unübersichtlich. Da wünscht sich der eine oder andere, dass die wichtigsten Elemente einer Datenbank schnell zur Verfügung stehen. Eine Möglichkeit, das zu erledigen, ist das Ribbon. Wie Sie die dort anzuzeigenden Elemente auswählen und diese dort hinzufügen, zeigt der vorliegende Beitrag.

Zwar verstummen allmählich die Stimmen, die eine Rückkehr des Datenbankfensters fordern, aber dass der Navigationsbereich nicht gerade die übersichtlichste Variante ist, um Datenbankobjekte anzuzeigen, ist offensichtlich (siehe Bild 1). Auf vielfachen Leserwunsch schauen wir uns daher einmal an, welche Möglichkeiten es gibt, Elemente wie Tabellen, Abfragen, Formulare, Berichte, Makros und Module über das Ribbon zugänglich zu machen.

Als Erstes benötigen wir eine Tabelle, in die wir die Elemente des Navigationsbereichs einlesen und der wir ein Feld hinzufügen, mit dem wir angeben, ob das jeweilige Element zusätzlich zur Anzeige im Navigationsbereich noch im Ribbon erscheinen soll. Dann stellen wir ein Formular zusammen, mit dem wir die Elemente anzeigen und die Möglichkeit bieten, diese für die Anzeige im Ribbon auszuwählen. Schließlich gibt es verschiedene Varianten, um Daten wie die beschriebenen Objekte im Ribbon anzuzeigen.

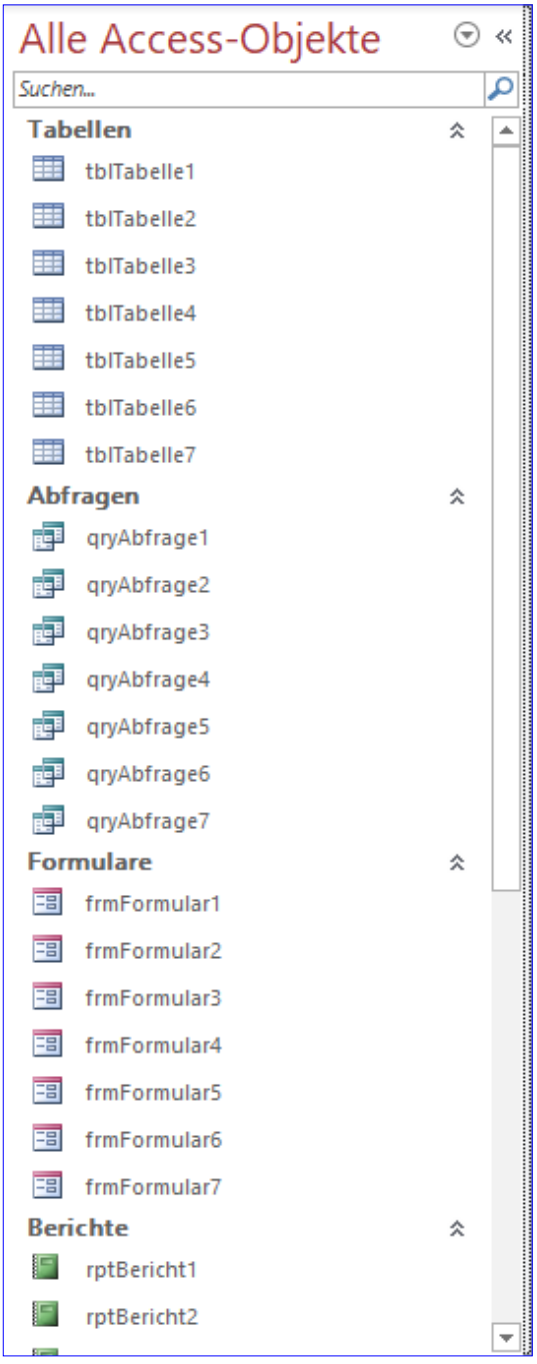

Bild 1: Navigationsbereich

Wir schauen uns die verschiedenen Varianten an und prüfen die Machbarkeit und Vor- und Nachteile.

Objekte in Tabelle

Zum Speichern der Objekte wollen wir die Tabelle **tblObjekteInRibbon** verwenden, die im Entwurf wie in Bild 2 aussieht.

Die Tabelle verwendet neben dem Primärschlüsselfeld **ID** das Feld **Objektname**, in das wir den Namen der Tabelle, Abfrage et cetera eintragen. Für das Feld **Objektname** legen wir über die Eigenschaft **Indiziert** mit dem Wert **Ja (Ohne Duplikate)** einen eindeutigen Index fest. Damit stellen wir sicher, dass jedes Objekt nur einmal zur Tabelle hinzugefügt wird.

Das Feld **ObjekttypID** ist verknüpft mit der Tabelle **tblObjekttypen**, in der wir die sechs Objekttypen aufführen (siehe Bild 3). Das letzte Feld heißt **InRibbon** und hat den Datentyp **Ja/Nein**. Mit diesem Feld legen wir fest, ob das Objekt im Ribbon angezeigt werden soll oder nicht.

Achtung: Die Tabelle muss die Daten genau mit den Primärschlüsselwerten enthalten, wie sie hier abgebildet sind. Später verwenden wir im Code die Werte von **1** bis **6**, um Elemente zu referenzieren, die zu einer der Objektkategorien gehören.

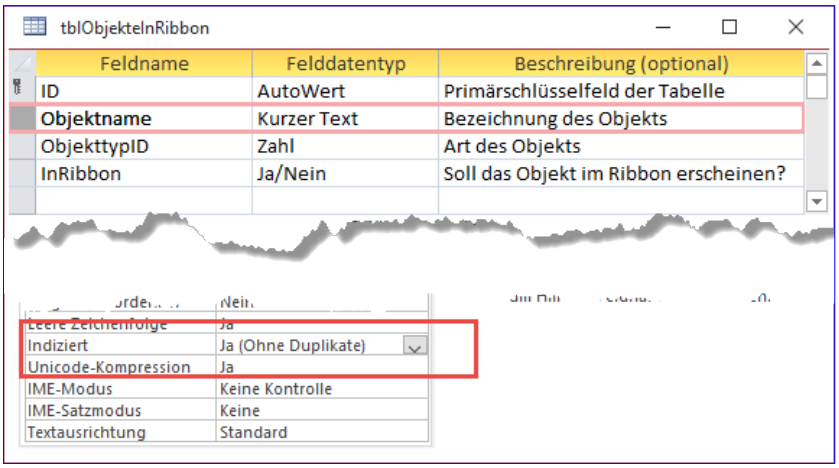

Bild 2: Tabelle zum Speichern der Objekte

Prozedur zum Erfassen aller Objekte in der Tabelle

Schließlich benötigen wir eine Prozedur, mit der wir die Objekte der Datenbank durchlaufen und diese zur Tabelle **tblObjekteInRibbon** hinzufügen.

Diese sieht wie in Listing 1 aus. Die Prozedur deklariert die Variable **db**, die Sie mit einem Verweis auf das **Database**-Objekt der aktuellen Datenbank füllt. **tdf** und **qdf** haben die Typen **TableDef** und **QueryDef** und dienen zum Durchlauen der Tabellen und Abfragen.

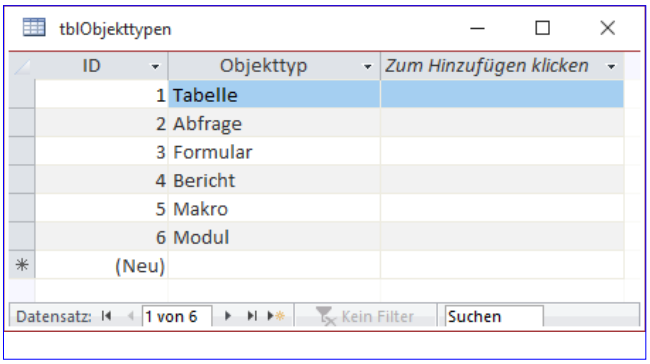

Bild 3: Die verschiedenen Objekttypen in der Tabelle **tblObjekttypen**

objAccess hat den Typ **AccessObject** und erlaubt das Durchlaufen der Auflistungen **AllForms**, **AllReports**, **AllMacros** und **AllModules** der **CurrentProject**-Klasse.

Die Fehlerbehandlung deaktivieren wir mit **On Error Resume Next**. Der Hintergrund ist, dass beim erneuten Einlesen der Objekte gegebenenfalls vorhandene Elemente gleichen Namens zu Fehlern führen können, da der eindeutige Index für das Feld **Objektname** verletzt wird. Dies wollen wir einfach ignorieren und Objekte, deren Namen bereits vorhanden sind, einfach nicht nochmals hinzufügen.

In der ersten **For Each**-Schleife durchläuft die Prozedur alle Elemente der **TableDefs**-Auflistung des **Database**-Objekts und trägt für jedes Element einen neuen Datensatz in die Tabelle **tblObjekteInRibbon** ein.

Dabei wird als **Objektname** der Wert der Eigenschaft **Name** des **TableDef**-Objekts und als **ObjekttypID** der Wert **1** eingetragen.

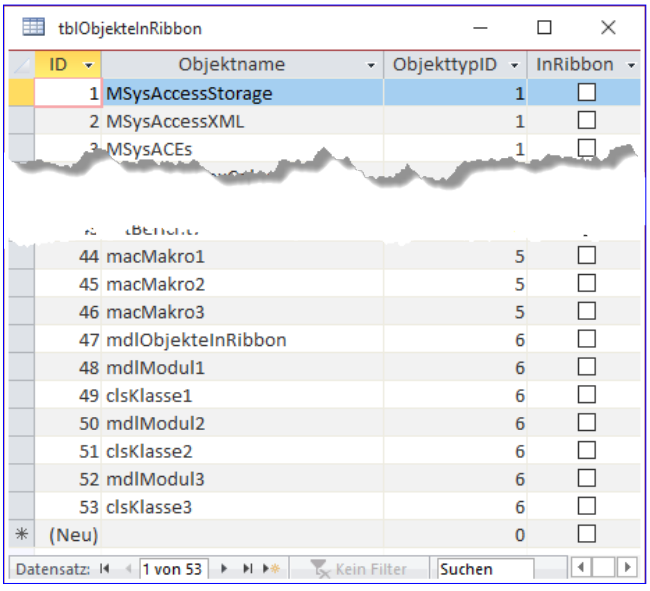

Bild 4: Die Tabelle **tblObjekteInRibbon** mit den Objekten der Datenbank

```
Public Sub ObjekteInTabelleSchreiben()
    Dim db As DAO.Database
    Dim tdf As DAO.TableDef
    Dim qdf As DAO.QueryDef
    Dim objAccess As AccessObject
    Set db = CurrentDb
    On Error Resume Next
    For Each tdf In db.TableDefs
        db.Execute "INSERT INTO tblObjekteInRibbon(Objektname, ObjekttypID) VALUES('" & tdf.Name & "', 1)", dbFailOnError
    Next tdf
    For Each qdf In db.QueryDefs
        db.Execute "INSERT INTO tblObjekteInRibbon(Objektname, ObjekttypID) VALUES('" & qdf.Name & "', 2)", dbFailOnError
    Next qdf
    For Each objAccess In CurrentProject.AllForms
        db.Execute "INSERT INTO tblObjekteInRibbon(Objektname, ObjekttypID) VALUES('" & objAccess.Name _
            & "', 3)", dbFailOnError
    Next objAccess
    For Each objAccess In CurrentProject.AllReports
        db.Execute "INSERT INTO tblObjekteInRibbon(Objektname, ObjekttypID) VALUES('" & objAccess.Name _
            & "', 4)", dbFailOnError
    Next objAccess
    For Each objAccess In CurrentProject.AllMacros
        db.Execute "INSERT INTO tblObjekteInRibbon(Objektname, ObjekttypID) VALUES('" & objAccess.Name _
            & "', 5)", dbFailOnError
    Next objAccess
    For Each objAccess In CurrentProject.AllModules
        db.Execute "INSERT INTO tblObjekteInRibbon(Objektname, ObjekttypID) VALUES('" & objAccess.Name _
            & "', 6)", dbFailOnError
    Next objAccess
End Sub
```

Listing 1: Prozedur zum Einlesen aller Access-Objekte in die Tabelle **tblObjekteInRibbon**

Die zweite **For Each**-Schleife erledigt das Gleiche für die Elemente der Auflistung **QueryDefs**. Hier fügt sie der Tabelle jedoch für das Feld **ObjekttypID** den Wert **2** hinzu.

Die dritte bis sechste **For Each**-Schleife der Prozedur verwendet die Auflistungen **AllForms**, **AllReports**, **AllMacros** und **AllModules** des **CurrentProject**-Objekts, um die Formulare, Berichte, Makros und Module der Datenbank zu ermitteln. Dabei nutzen wir jeweils die **Name**-Eigenschaft des **AccessObject**-Elements als Wert für das Feld **Objektname**. Für das Feld **ObjekttypID** tragen wir je nach Objekttyp die Werte **3**, **4**, **5** oder **6** ein. Das Ergebnis sieht wie in Bild 4 aus.

Formular zur Auswahl der anzuzeigenden Elemente

Nun benötigen wir ein Formular, dass die Elemente der Datenbank in alphabetischer Reihenfolge und nach Objekttyp aufgeteilt anzeigt und das Markieren der im Ribbon anzuzeigenden Objekte erlaubt. Dazu erstellen wir ein neues Formular namens **frmObjekteInRibbon**. Das Formular soll seine Daten in einem Register-Steuerelement mit sechs Registerseiten anzeigen, die jeweils ein Unterformular enthalten. Das Unterformular soll die Elemente des jeweiligen Objekttyps anzeigen. Da das Unterformular jeder Registerseite die Daten aus der gleichen Tabelle anzeigen soll, die nur unterschiedlich gefiltert werden,

benötigen wir nur ein Formular, das dann in den sechs Unterformular-Steuerelementen angezeigt wird.

Wir legen als Erstes das Formular **frmObjekteInRibbon** an und fügen das Register-Steuerelement hinzu. Dann wählen wir aus dem Kontextmenü dieses Steuerelements so oft den Eintrag **Seite einfügen** aus, bis das Register-Steuerelement sechs Registerseiten enthält (siehe Bild 5). Die sechs Registerseiten beschriften wir über die Eigenschaft **Beschriftung** mit **Tabellen**, **Abfragen**, **Formulare**, **Berichte**, **Makros** und **Module**. Die Seiten erhalten die Namen **pgeTabellen**, **pgeAbfragen**, **pgeFormulare**, **pgeBerichte**, **pgeMakros** und **pgeModule**.

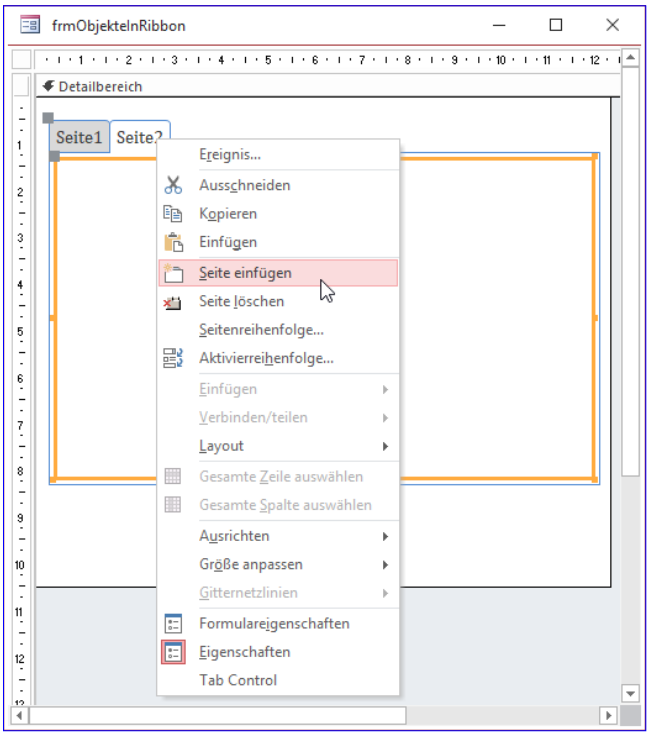

Bild 5: Hinzufügen von Registerseiten zum Register-Steuerelement

Das Unterformular **sfmObjekteInRibbon** statten wir mit einer Abfrage namens **qryObjekteInRibbonNachObjektname** als **Datensatzquelle** aus, welche alle Felder der Tabelle **tblObjekteInRibbon** enthält und diese nach dem Feld **Objektname** sortiert (siehe Bild 6).

Wir fügen dem Formular die beiden Felder **InRibbon** und **Objektname** der Datensatzquelle hinzu. Außerdem stellen wir die Eigenschaft **Standardansicht** auf **Datenblatt** ein.

Danach speichern und schließen Sie das Unterformular **sfmObjekteInRibbon**. Für das Formular **frmObjekteInRibbon** stellen wir noch die Eigenschaften **Datensatzmarkierer**, **Navigationsschaltflächen** und **Bildlaufleisten** auf **Nein** ein und die Eigenschaft **Automatisch zentrieren** auf **Ja**.

Dann folgt ein wenig Kleinarbeit: das Einfügen des Unterformulars in das Register-Steuerelement. Hier gibt es zwei Möglichkeiten:

- Entweder wir fügen auf jeder Registerseite ein neues Unterformular-Steuerelement ein, das wir dann jeweils nach dem Wert für das Feld **ObjekttypID** filtern, der die Objekte passend zur Beschriftung der Registerseite liefert.
- Oder wir platzieren ein Unterformular-Steuerelement in z-Reihenfolge vor dem Register-Steuerelement und ändern den Filter jeweils bei Änderung der ausgewählten Registerseite.

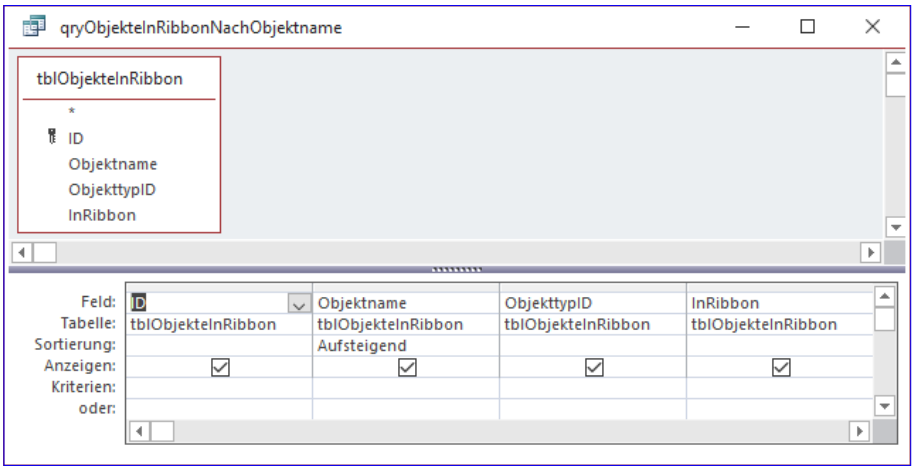

Bild 6: Datensatzquelle des Formulars **sfmObjekteInTabelle**

Wer schon einmal auf mehreren Registerseiten ein Steuerelement so eingefügt hat, dass es sich beim Blättern im Register immer an der gleichen Position befindet, weiß, dass das eine langweilige Arbeit ist. Also entscheiden wir uns für die zweite Variante mit nur einem Unterformular, das wir einfach vor dem Register-Steuerelement platzieren. Dazu fügen wir dieses zuerst unter oder neben dem Register-Steuerelement ein, entfernen das Beschriftungsfeld und schieben es dann über das Register-Steuerelement (siehe Bild 7).

Damit sich beide beim Ändern der Formulargröße der Größe anpassen, stellen wir die Eigenschaften **Horizontaler Anker** und **Vertikaler Anker** jeweils auf **Beide** ein.

Nun müssen wir nur noch dafür sorgen, dass das Unterformular nur noch die Elemente anzeigt, die dem im Register-Steuerelement ausgewählten Objekttyp entsprechen. Dazu hinterlegen wir für die Ereigniseigenschaft **Bei Änderung** des Register-Steuerelements, das wir zuvor noch mit **regObjekte** benennen, die folgende Ereignisprozedur:

```
Private Sub regObjekte_Change()
    With Me!sfmObjekteInRibbon.Form
        .Filter = "ObjekttypID = " & Me!regObjekte.Value + 1
        .FilterOn = True
    End With
End Sub
```

Diese stellt die Eigenschaft **Filter** des Unterformulars auf einen Ausdruck ein, der den Wert der jeweils ausgewählten Registerseite ermittelt, eins hinzuaddiert und diesen mit dem Wert des Feldes **ObjekttypID** gleichsetzt. Dabei kommt dann etwa für die erste Registerseite das folgende Kriterium heraus:

```
ObjekttypID = 1
```

Wenn wir das Formular nun öffnen, erscheinen allerdings zunächst alle Elemente in der Liste (siehe Bild 8).

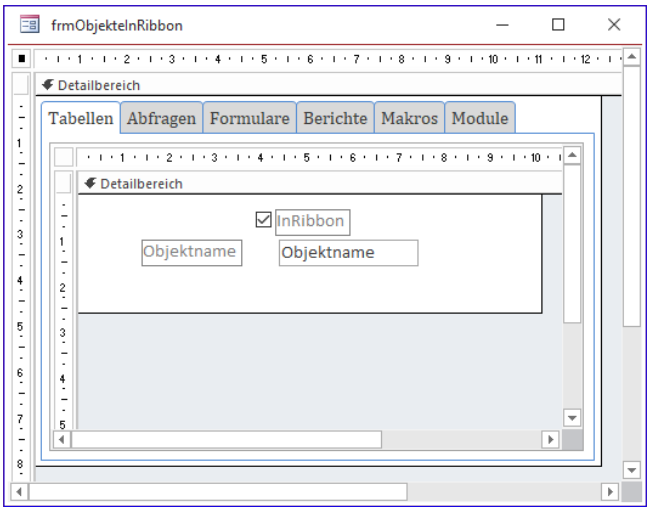

Bild 7: Unterformular-Steuerelement über Register-Steuerelement

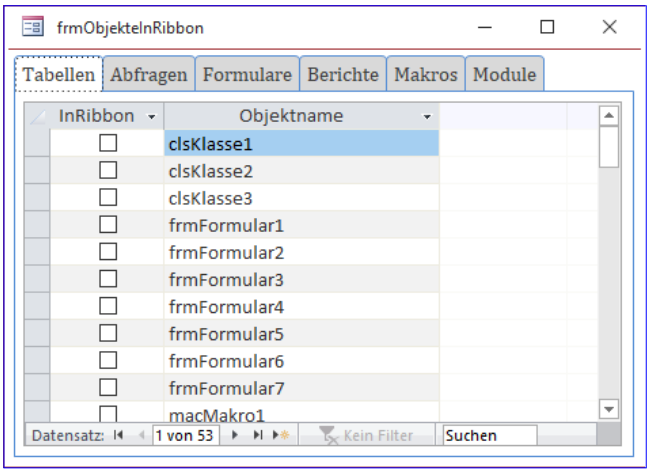

Bild 8: Das Unterformular zeigt noch alle Einträge an.

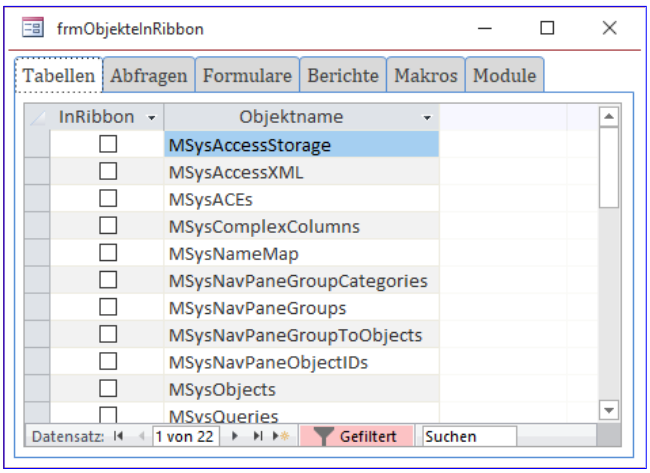

Bild 9: Objekte gefiltert nach Objekttyp, hier alle Tabellen

```
Private Sub cmdAlleLoeschen_Click()
    Dim db As DAO.Database
    Set db = CurrentDb
    If MsgBox("Dies löscht alle Elemente. Diese müssen danach neu eingelesen werden.", vbYesNo) = vbYes Then
        db.Execute "DELETE FROM tblObjekteInRibbon", dbFailOnError
        Me!sfmObjekteInRibbon.Form.Requery
    End If
End Sub
```

Listing 2: **Beim Klicken**-Prozedur der Schaltfläche **cmdAlleLoeschen**

Wenn wir allerdings die Ereignisprozedur regObjekte_Change einmal beim Laden des Formulars aufrufen, erhalten wir auch direkt nach dem Öffnen die korrekt gefilterten Elemente (siehe Bild 9).

Die Ereignisprozedur, die durch das Ereignis **Beim Laden** ausgelöst wird, sieht dann so aus:

```
Private Sub Form_Load()
    regObjekte_Change
End Sub
```

Verwalten der Objekte

Wir wollen noch ein paar weitere Elemente hinzufügen, damit der Benutzer die Objekte im Formular einfacher verwalten kann. Die dazu angelegten Schaltflächen sehen Sie in Bild 10. Damit die Schaltflächen nicht unter den anderen Steuerelementen verschwinden, wenn der Benutzer das Formular nach unten vergrößert, stellen wir die Eigenschaft **Vertikaler Anker** auf **Unten** ein.

Alle Objekte löschen

Die Schaltfläche mit der Beschriftung **Alle löschen** heißt **cmdAlleLoeschen** und soll alle Datensätze aus der Tabelle **tblObjekteInRibbon** löschen. Auf diese Weise kann der Benutzer die Objekte nach umfangreicheren Änderungen anschließend wieder neu einlesen. Die Schaltfläche löst beim Anklicken die Ereignisprozedur aus Listing 2 aus. Diese zeigt zunächst noch eine Meldung an, die den Benutzer darauf hinweist, dass alle Datensätze gelöscht werden. Der Benutzer kann dies durch Betätigen der **Nein**-Schaltfläche abwenden. Ansonsten sorgt eine **DELETE**-Abfrage für das Löschen aller Datensätze der Tabelle **tblObjekteInRibbon**. Die folgende Anweisung aktualisiert die Anzeige der Daten im Unterformular.

Objekte neu einlesen

Die Schaltfläche **cmdNeuEinlesen** löst die folgende Prozedur aus. Diese weist den Benutzer darauf hin, dass nur neue Elemente hinzugefügt werden. Anschließend ruft sie die Prozedur **ObjekteInTabellenSchreiben** auf, die wir bereits weiter oben vorgestellt haben. Auch hier wird der Vorgang durch das Aktualisieren der Daten des Unterformulars abgeschlossen:

```
Private Sub cmdNeuEinlesen_Click()
    MsgBox "Vorhandene Elemente werden beibehalten, 
                nur neue Elemente werden hinzugefügt."
    ObjekteInTabelleSchreiben
```

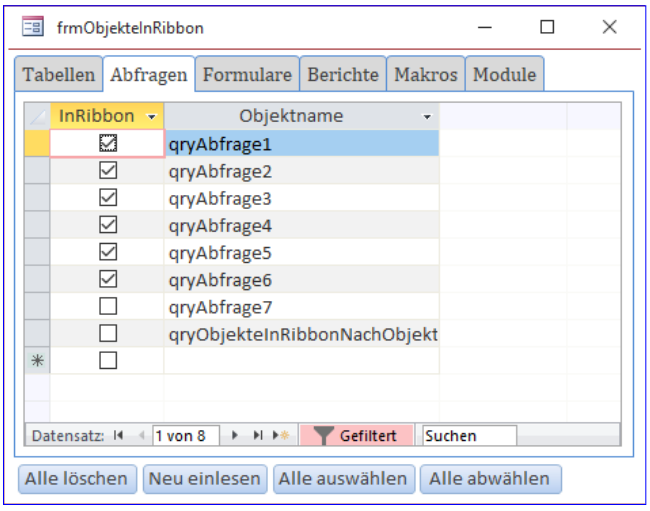

Bild 10: Objekte verwalten mit zusätzlichen Schaltflächen

HTML-Code optimieren per VBA

Im Beitrag »Von Access nach Wordpress« (Heft 6/2019) haben wir Routinen entwickelt, mit denen wir den Inhalt eines Textfeldes in SQL-Anweisungen exportiert und damit eine Wordpress-Webseite gefüllt haben. Der Weg dorthin war nicht so einfach, wie es in diesem Beitrag beschrieben wurde. In der Tag lag der HTML-Code mit den Artikeln so vor, dass er in einem anderen Content Management System, hier Typo3, optimal angezeigt wurde. Wenn wir diesen HTML-Code in das Wordpress-System importiert haben, sah das optisch allerdings nicht so ansprechend aus. Wir mussten also noch einige Änderungen am HTML-Code vornehmen. Für Handarbeit war das bei rund 1.000 Artikel zu viel, also war die Programmierung entsprechender Konvertierungsroutinen angezeigt. Wie das grundlegend funktioniert, zeigen wir im vorliegenden Beitrag.

Beispiele für notwendige Konvertierungen

Typo3 war als Content Management System sehr nachsichtig, was die anzuzeigenden Inhalte anging. So konnten Sie ihm beispielsweise die deutschen Umlaute wie **ä**, **ö** und **ü** sowie das scharfe s (**ß**) unterjubeln und es zeigte diese problemlos an. Unter Wordpress funktioniert das nicht. Hier werden entsprechende HTML-Entitäten erwartet, die wie folgt aussehen:

- **ä**: **ä**
- **ö**: **ö**
- **ü**: **ü**
- **Ä**: **Ä**
- **Ö**: **Ö**
- **Ü**: **Ü**
- **ß**: **ß**

Auch der Vorname des Autors dieser Zeilen führte zu Problemen: Also musste **André** überall in **André** geändert werden. Und davon gab es noch viele weitere Beispiele.

Größer- und Kleiner-Zeichen

Ein weiteres Problem sind im Text abgebildete HTML-Codes wie der folgende:

```
<p>Beispieltext</p>
```

Dies wurde unter Typo3 problemlos angezeigt, unter Wordpress jedoch wurden die HTML-Auszeichnungen mehr oder weniger sinnvoll als solche interpretiert und nicht einfach ausgegeben. Also muss man auch hier die Kleiner- und Größer-Zeichen durch entsprechende Entitäten ersetzen:

- **<**: **<**
- **>**: **>**

Andere Zeichen

Desweiteren haben wir noch einige andere Zeichen gefunden, die wir in den rund 1.000 Artikeln mehr oder weniger konsistent verwendet haben – zum Beispiel verschiedene Anführungszeichen wie '', "", „“ oder »«. Hier haben wir versucht, die Anführungszeichen auf diesem Wege zu vereinheitlichen.

Mit Anführungszeichen gab es vor allem Probleme, weil diese ja irgendwie in die **INSERT INTO**-Anweisungen

integriert werden mussten, die ja ihrerseits die Werte für Textfelder in Anführungszeichen angeben. Unter dem SQL-Dialekt von MySQL kann man Zeichenketten sowohl in Hochkommata (') also auch in Anführungszeichen angeben ("). Je nachdem, welches man verwendet, muss man aber im zu speichernden Text die Hochkommata oder Anführungszeichen »escapen«, also durch ein zusätzliches Zeichen so codieren, dass es nicht als Ende der Zeichenkette interpretiert wird – denn das führt regelmäßig zu Fehlern, da so der Rest der SQL-Anweisung fehlinterpretiert wird. Ein Beispiel:

```
INSERT INTO wp_posts(ID, post_content, ...)
VALUES (55000999, 'Dies ist ein 'lustiger' Beispieltext',
...)
```

Das führt zu einem Fehler, weil der SQL-Interpreter das Hochkomma vor **lustiger** als Ende der Zeichenkette erkennt. Danach wird ein Komma erwartet, aber es folgt noch weiterer Text, was den Fehler auslöst.

Abhilfe schafft es an dieser Stelle, einfach das Hochkomma in dem Text, der in das Feld eingefügt werden soll, durch ein doppeltes Hochkomma zu ersetzen. Alternativ können Sie auch ein Anführungszeichen verwenden – zumindest, wenn zur Markierung der einzufügenden Texte Hochkommata verwendet werden.

HTML-Auszeichnungen

Außerdem hatte ich die interessante Idee, nicht die üblichen HTML-Auszeichnungen etwa für Überschriften zu verwenden, sondern ein Absatz-Element mit einer bestimmten CSS-Klasse (wobei: vermutlich war das gar nicht meine Idee, sondern die des Exports aus Word oder InDesign, das natürlich nicht erkennen konnte, dass die Absatzformatklasse **titel** eine Überschrift beinhaltet):

```
<p class=""titel"">...</p>
```

Dennoch sollten solche Elemente so umformatiert werden, dass Folgendes herauskam:

```
<h1>...</h1>
```

Damit können dann auch die Suchmaschinen viel mehr anfangen – sonst wäre es nicht möglich, zu erkennen, in welchem Element sich die Überschrift eines Textes befindet.

Es gibt noch weitere, ähnliche Beispiele – zum Beispiel für Zwischenüberschriften:

```
<p class="zwischenueberschrift">...</p>
```

Diese Auszeichnung soll beispielsweise zu dieser werden:

```
<h2>...</h2>
```

Für Quellcode gab es die folgende Auszeichnung:

```
<p class="Quellcode">...</p>
```

Diese soll so umformatiert werden:

```
<pre>...</pre>
```

Innerhalb des Fließtextes wurde die folgende Auszeichnung verwendet, um Begriffe hervorzuheben:

```
...<span class="kursiv">...</span>...
```

Das wollen wir durch fette Schrift ersetzen:

```
...<b>...</b>...
```

Nicht sichtbare Zeichen

Die bisher beschriebenen notwendigen Änderungen waren leicht zu erkennen. Entweder führten Sie schon beim Versuch, die Texte in die Textfelder der MySQL-Tabelle zu importieren, zu Fehlern oder sie fielen dann beim manuellen Sichten der Texte auf der Wordpress-Webseite auf. Einige Zeichen sorgten jedoch dafür, dass der Text in Wordpress einfach an einer bestimmten Stelle endete. Diese Zeichen

mussten wir erst aufwendig ermitteln, indem wir die betroffene Stelle eingegrenzt haben und dann Zeichen für Zeichen die ASCII-Codes analysiert haben. Dabei zeigte sich, dass so manche Steuerzeichen aus dem Originaltext aus Word oder InDesign im HTML-Code gelandet sind, die von Typo3 ignoriert wurden – und unter Wordpress dazu führten, dass der Rest des Textes inklusive dieses Zeichens schlicht nicht mehr abgebildet wurde. Die ASCII-Codes dieser Zeichen lauten beispielsweise **7**, **22**, **30**, **63**, **141** oder **157**.

Umwandlungen per VBA

Die meisten Änderungen, die notwendig sind, um die oben genannten, nicht in Wordpress funktionierenden Zeichen und Elemente anzupassen, sind mit dem Aufruf der **Replace**-Funktion zu erledigen. Andere verlangen nach mehr Aufwand, zum Beispiel wenn es um das Ersetzen von Elementen wie ** ... ** durch **<h1>...</h1>** geht.

Einzelne Zeichen ersetzen

Wir haben diese alle in eine einzige Funktion geschrieben, mit der wir einen Text entgegennehmen, diesen auf die zu ersetzenden Zeichen durchsuchen und diese direkt mit der **Replace**-Funktion ersetzen. Das sieht dann beispielsweise wie folgt aus:

```
Public Function Ersetzen(strText As String) As String
    strText = Replace(strText, "'", "'")
    strText = Replace(strText, "'", "''")
    strText = Replace(strText, "\", "\\")
    strText = Replace(strText, "ä", "&auml;")
    strText = Replace(strText, "ö", "&ouml;")
    strText = Replace(strText, "ü", "&uuml;")
    strText = Replace(strText, "Ä", "&Auml;")
    strText = Replace(strText, "Ö", "&Ouml;")
    strText = Replace(strText, "Ü", "&Uuml;")
    strText = Replace(strText, "ß", "&szlig;")
    strText = Replace(strText, "é", "&eacute;")
    strText = Replace(strText, "»", """")
    strText = Replace(strText, "«", """")
    strText = Replace(strText, "§", "&sect;")
    strText = Replace(strText, "-", "-")
    strText = Replace(strText, "€", "&euro;")
    strText = Replace(strText, "à", "&aacute;")
    strText = Replace(strText, "„", """")
    strText = Replace(strText, "“", """")
    ...
    Ersetzen = strText
End Function
```

Für die meisten Sonderzeichen lassen sich leicht die passenden HTML-Entitäten finden. Wenn das nicht gelingt, ist die Chance hoch, dass Sie das Ersetzen über den ASCII-Code des Zeichens wie folgt bewerkstelligen können:

Sie ermitteln den ASCII-Code des Zeichens mit der **Asc**-Funktion:

```
? Asc("š")
 154
```

Dann stellen Sie den Ersetzungscode aus den Zeichen **&#**, dem ASCII-Code und dem Semikolon (**;**) zusammen:

```
&#154;
```

Dementsprechend sieht dann die Ersetzung aus:

```
strText = Replace(strText, "š", "&#154;")
```

oder

```
strText = Replace(strText, Chr(154), "&#154;")
```

Umfangreichere Ersetzungen

Etwas komplizierter wird es, wenn Sie öffnende und schließende HTML-Tags ersetzen wollen.

Soll nur der öffnende Tag angepasst werden, etwa ** ** durch ** **, kann man das genau wie oben mit der **Replace**-Funktion erledigen:

```
strText = Replace(strText, "<p class=""Fliesstext"">", "<p>")
```

Zu beachten ist in diesem Fall, dass Sie das einfache Anführungszeichen in der zu ersetzenden Zeichenkette durch doppelte Anführungszeichen ersetzen, damit das Anführungszeichen hinter **class=** nicht als schließendes Anführungszeichen erkannt wird.

Ein Beispiel für einen komplizierteren Fall ist der folgende:

```
<p class=""titel"">...</p>
```

soll ersetzt werden durch:

```
<h1>...</h1>
```

Das erledigen wir mit der Funktion aus Listing 1. Hier ermitteln wir mit der **InStr**-Funktion zunächst die Position im Text, an der das erste Mal die gesuchte Zeichenkette auftaucht und speichern diese in der Variablen **lngStart**. Danach steigen wir in eine **Do While**-Schleife ein. Warum eine Schleife? Weil wir davon ausgehen, dass es nicht nur ein Vorkommen des zu ersetzenden Textes gibt. Deshalb läuft die **Do While**-Schleife solange, wie **lngStart** größer als **0** ist. Wenn wir beim ersten Aufruf von **InStr** also kein Vorkommen finden, hat **lngStart** den Wert **0** und die Schleife wird gar nicht erst durchlaufen.

Wenn der erste Aufruf der **InStr**-Funktion jedoch einen Treffer liefert, was wir durch einen Wert größer **0** für die Variable **lngStart** erkennen, starten wir in die **Do While**-Schleife. Wir suchen dann nach dem entsprechenden schließenden Element mit dem Text **</p>**. Bei diesem Aufruf der **InStr**-Funktion verwenden wir nicht den Wert **1** als Startposition für die Suche, sondern den Wert der Variablen **lngStart** plus eins, damit wir auf jeden Fall ein Vorkommen von **</p>** finden, das hinter dem öffnenden Element liegt.

Dann folgt eine Anweisung mit Zeichenkettenfunktionen, die folgendes erledigt: Sie ermittelt zunächst den Text, der sich vor der Startposition des zuletzt gefundenen Auftretens von **<p class=""titel"">** befindet, und verwendet diesen als ersten Teil des neuen Inhalts der Variablen **strText**. Als nächstes folgt der ersetzende Ausdruck, nämlich **<h1>**. Dann liefert die **Mid**-Funktion den Text vom Ende des aktuell gefundenen Vorkommens von **<p class=""titel"">** bis zum Anfang des schließenden Tags **</p>**. Danach folgt der neue schließende Tag, also **</h1>**. Und schließlich fügt die **Mid**-Funktion noch den gesamten Text hinzu, der sich hinter dem schließenden Tag **</p>** befindet:

```
Left(strText, lngStart - 1) _
    & "<h1>" _
    & Mid(strText, lngStart + 17, lngEnde - lngStart - 17) _
```

```
Public Function ErsetzenH1(strText As String) As String
    Dim lngStart As Long
    Dim lngEnde As Long
    lngStart = InStr(1, strText, "<p class=""titel"">")
    Do While lngStart > 0
        lngEnde = InStr(lngStart + 1, strText, "</p>")
        strText = Left(strText, lngStart - 1) & "<h1>" & Mid(strText, lngStart + 17, lngEnde - lngStart - 17) _
            & "</h1>" & Mid(strText, lngEnde + 4)
        lngStart = InStr(lngStart + 1, strText, "<p class=""titel"">")
    Loop
    ErsetzenH1 = strText
End Function
```

Listing 1: Ersetzen öffnender und schließender Tags