HAUFE.
Ausgabe 05/2025

ACCESS

IM UNTERNEHMEN

WARUM ACCESS UNSICHER IST

Erfahren Sie, wie jeder die Daten Ihrer Datenbank auslesen kann (Seite 23).

In diesem Heft:

VERPACKUNGSPROZESSE VERWALTEN

Behalten Sie den Überblick, welche Lieferung in welchem Karton landet.

SEITE 56

SUCHEN UND ERSETZEN-FUNKTIONEN

Fügen Sie dem Suchformular den Code zum Suchen und Ersetzen in Datenblättern hinzu.

SEITE 11

LISTENFELDER GEZIELT STEUERN

Einlesen, auslesen, selektieren – wir zeigen alle Techniken in diesem Beitrag.

SEITE 2

Sicher ist nicht sicher

Wir haben unseren Kunden zuletzt ein Access-Audit angeboten, das sehr gern angenommen wurde. Dabei haben wir uns die Anwendungen der Kunden angesehen und verschiedene Aspekte berücksichtigt. Ein wichtiger Aspekt war für uns das Thema Sicherheit. Und hier stellte sich heraus, dass es in den meisten Access-Anwendungen keine oder nur sehr wenige Maßnahmen gibt, um zu verhindern, dass beispielsweise ein verärgerter Mitarbeiter Kundendaten entwendet und damit zur Konkurrenz geht. Daher gehen wir in dieser und auch in den kommenden Ausgaben von Access im Unternehmen einmal genauer auf das Thema Sicherheit ein.

Den ersten Schritt gehen wir dabei mit dem Beitrag **Einfacher Datenklau durch mangelnde Sicherheit** (ab Seite 23). Hier zeigen wir, wie jeder, der an einem Rechner mit der geöffneten Datenbank sitze, die in dieser Datenbank enthaltenen Tabellen, Tabellenverknüpfungen und Abfragen vollständig auslesen kann. Selbst wenn es sich um eine in Frontend und Backend aufgeteilte Datenbank handelt, bei der das Backend kennwortgeschützt ist, können die Daten in den meisten Fällen einfach ausgelesen werden.

Im zweiten Beitrag zu diesem Thema namens **Datenbank durch Runtime-Modus sicher machen?** zeigen wir ab Seite 28, wie man es dem Benutzer zumindest erschweren kann, die Bereiche der Datenbank sichtbar zu machen, die er eigentlich nicht sehen soll. Beinahe jede Sicherheitsmaßnahme von Access kann allerdings umgangen werden – daher sind die dortigen Ideen nur als Lösung zum Schutz der Daten vor unbedarften Anwendern gedacht. Wer ambitioniert ist, die Daten auszulesen, wird sich dadurch nicht aufhalten lassen.

Produktiv wird es im Beitrag **Verpackungsprozesse mit Access: Formulare** ab Seite 56. Hier zeigen wir, wie man einen Packvorgang von Produkten in verschiedene Behälter mit einer Access-Anwendung erfassen kann – damit man zum Beispiel jederzeit sieht, welches Produkt in welchem Behälter gelandet ist und wie viele Produkte für die jeweilige Bestellung noch zu packen sind.

Listenfelder sind nach wie vor ein beliebtes Steuerelement zur Anzeige einfacher Listen. Im Beitrag **Listenfeldwerte auswählen, abwählen und auslesen** zeigen wir ab Seite 2, wie sich Einträge auswählen lassen, wie man diese wieder abwählt und wie man die aktuelle Selektion auslesen kann.

Manchmal braucht man viele gleichartige Steuerelemente. Diese kann man von Hand anlegen, benennen und mit den gewünschten Eigenschaften versehen. Diese Aufgabe lässt sich aber auch automatisieren. Wie das gelingt, zeigen wir ab Seite 9 im Beitrag **Schaltflächen-Matrix per VBA erzeugen**.

Wie wir die dort generierte Schaltflächenmatrix einsetzen können, sehen wir in der Lösung aus dem Beitrag **Icons in der Datenbank verwalten** (ab Seite 34). Hier stellen wir ein Formular vor, mit dem wir die Icons, die in der Tabelle **MSysResources** einer Anwendung gespeichert sind, komfortabel verwalten kann.

Und schließlich reichen wir im Beitrag **Optimierter »Suchen und Ersetzen«-Dialog: Funktionen** ab Seite 11 noch die Suchen- und Ersetzen-Funktionen für die Lösung nach, die wir in der vorherigen Ausgabe gestartet haben.

Herzlichst, Ihr André Minhorst

Listenfeldwerte auswählen, abwählen und auslesen

Das Listenfeld-Steuerelement ist ein sehr praktisches Steuerelement, wenn es um die Darstellung von Listen ohne größere optische Ansprüche geht. Sie zeigen einfach nur Text in Zeilen und Spalten an und die Inhalte sind nicht direkt bearbeitbar. Gegenüber der Datenblattansicht bieten sie aber auch Vorteile, zum Beispiel die Mehrfachauswahl. Dennoch gibt es einige Dinge, die man beachten muss, wenn man dem Benutzer die richtige Auswahlmöglichkeit bereitstellen möchte – und auch das Auslesen kann gegebenenfalls unvorhergesehene Ergebnisse liefern, vor allem nach der Aktualisierung der Inhalte. In diesem Beitrag schauen wir uns die verschiedenen Selektionsmöglichkeiten an und zeigen, wie wir diese auslesen können.

Beispieldaten

Zu Beispielzwecken haben wir eine einfache Tabelle namens **tblProdukte** mit den Feldern **ProduktID**, **Produktname** und **Kategorie** angelegt (siehe Bild 1).

Listenfeld anlegen und Daten anzeigen

Ein Listenfeld ist schnell angelegt (siehe Bild 2). Damit es die Daten aus der Beispieltabelle anzeigt, stellen wir die Datensatzherkunft auf die gewünschte Tabelle oder Abfrage ein.

Wir wollen die Daten der Tabelle **tblProdukte** anzeigen, also legen wir für die Eigenschaft **Datensatzherkunft** den Wert **tblProdukte** fest.

Damit zeigt sie im Ausgangszustand jedoch nur das Feld **ProduktID** an. Der Grund sind die beiden Eigenschaften **Spaltenanzahl** und **Spaltenbreiten**.

Die erste steht auf **1**, die zweite ist noch leer – was dazu führt, dass die erste Spalte die gesamte Breite einnimmt. Die anzuzeigende Tabelle enthält aber drei Felder.

Damit wir beim Auslesen des oder der ausgewählten Einträge der Tabelle über die Standardeigenschaft **Value** auf den eindeutigen Wert des gewählten Datensatzes zugreifen können, sollte das entsprechende Feld in der Datensatzherkunft den 1-basierten Index aufweisen, der in der Eigenschaft **Gebundene Spalte** festgelegt ist. Gleichzeitig

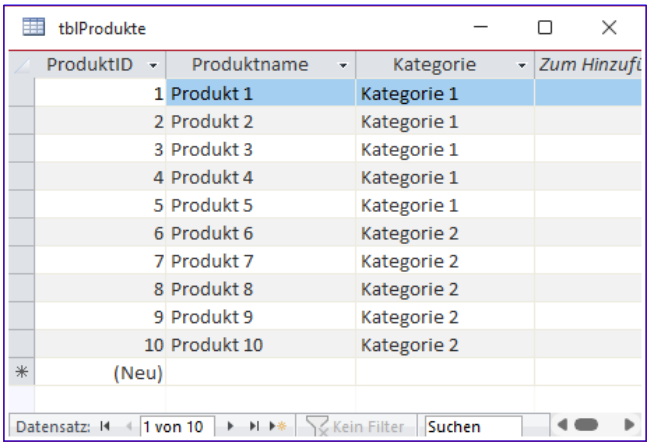

Bild 1: Beispieldaten der Tabelle **tblProdukte**

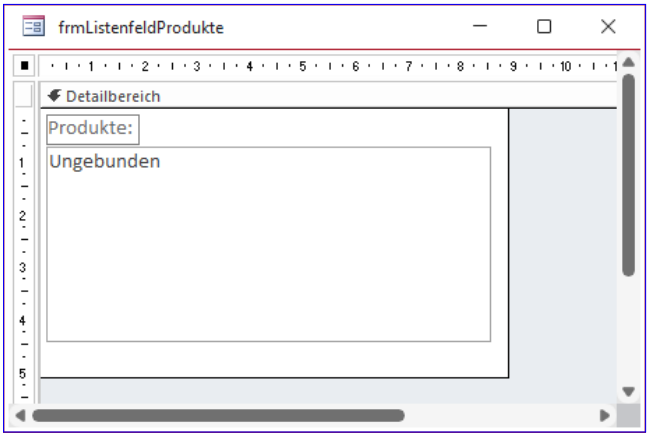

Bild 2: Einfaches Listenfeld

wollen wir den Primärschlüsselwert der zugrunde liegenden Datensatzherkunft nicht unbedingt anzeigen.

Dies führt zu den folgenden Einstellungen:

- **Spaltenanzahl**: **3** (alle drei Felder der Datensatzherkunft **tblProdukte** werden angezeigt)
- **Spaltenbreiten**: **0cm;3cm** (das erste Feld mit dem Index wird ausgeblendet, das zweite mit 3cm Breite angezeigt und das dritte nimmt den Rest der Listenfeldbreite ein)
- **Gebundene Spalte**: Bleibt auf dem Wert **1**, da die erste Spalte das Primärschlüsselfeld enthält.

Damit erhalten wir das Ergebnis aus Bild 3.

Einfachauswahl setzen und auslesen

Standardmäßig ist die Eigenschaft **Mehrfachauswahl** auf **Keine** eingestellt, was bedeutet, dass wir maximal nur einen Eintrag gleichzeitig auswählen können.

Wenn wir einen Eintrag anklicken, wird dieser durch einen schwarzen Hintergrund markiert (siehe Bild 4). Auf diese Weise können wir nacheinander verschiedene Einträge selektieren. Allerdings können wir die Auswahl, wenn wir einmal einen Eintrag ausgewählt haben, standardmäßig nicht mehr aufheben. Es ist dann immer ein Eintrag markiert.

Wie wir diese Markierung wieder aufheben können, zeigen wir weiter unten.

Um den aktuell markierten Eintrag auszulesen, können wir einfach auf die Eigenschaft **Value** des Listenfeldes zugreifen. Von außen, zum Beispiel vom Direktbereich aus, sieht das so aus:

```
? Forms!frmListenfeldProdukte!lstProdukte.Value
2
```

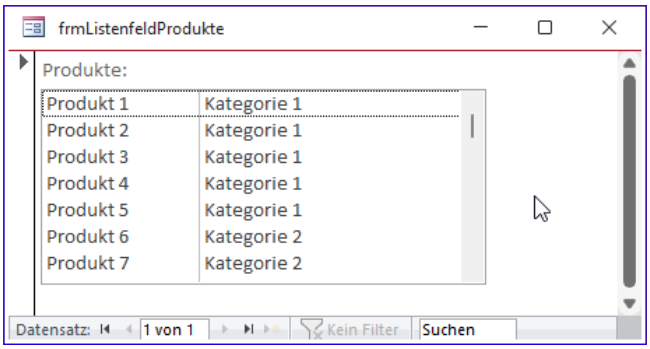

Bild 3: Listenfeld mit den Daten der Tabelle **tblProdukte**

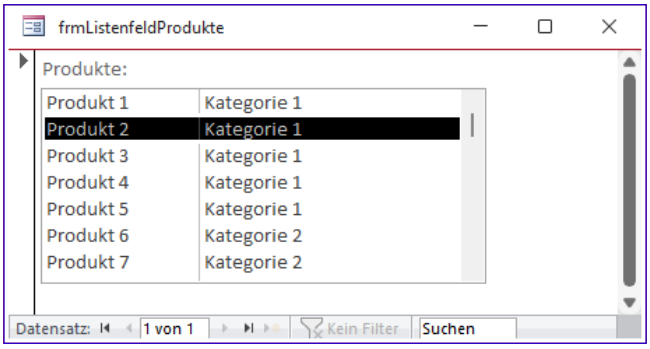

Bild 4: Einfachauswahl eines Eintrags

Da **Value** die Standardeigenschaft ist, können wir diese auch weglassen:

```
? Forms!frmListenfeldProdukte!lstProdukte
2
```

Innerhalb des Klassenmoduls des Formulars greifen wir einfach über folgenden Ausdruck auf den markierten Eintrag im Listenfeld zu:

```
Debug.Print Me.lstProdukte
```

Weitere Spaltenwerte des aktuellen Eintrags auslesen

Value liefert also den Wert der gebundenen Spalte. Wenn wir auch die übrigen Spalten auslesen wollen, verwenden wir die **Column**-Eigenschaft mit dem 0-basierten Index der jeweiligen Spalte. In der folgenden Prozedur geben wir nach der Aktualisierung des Listenfeldes den Wert der

gebundenen Spalte aus sowie den der Spalten mit den Indizes **1** und **2**:

```
Private Sub lstProdukte_AfterUpdate()
    Debug.Print "Gebundene Spalte: " & Me.lstProdukte.Value
    Debug.Print "Spalte 1: " & Me.lstProdukte.Column(1)
    Debug.Print "Spalte 2: " & Me.lstProdukte.Column(2)
End Sub
```

Listenfeldeintrag per VBA einstellen

Wollen wir per VBA einen der Einträge selektieren, weisen wir einfach dem Listenfeld den gewünschten Wert der gebundenen Spalte, hier mit dem Primärschlüsselwert gefüllt, zu:

```
Forms!frmListenfeldProdukte!lstProdukte = 3
```

Markierung per VBA aufheben

Wenn wir die aktuelle Markierung wieder aufheben wollen, stellen wir das Listenfeld auf den Wert **Null** ein:

```
Forms!frmListenfeldProdukte!lstProdukte = Null
```

Spaltenüberschriften einblenden

Mit der Eigenschaft **Spaltenüberschriften** können wir die Spaltenüberschriften des Listenfeldes einblenden (siehe Bild 5).

Diese orientieren sich an den Feldnamen beziehungsweise Beschriftungen der Datensatzherkunft. Wenn wir also nicht die eigentlichen Feldnamen, sondern eine andere Beschriftung anzeigen wollen, müssen wir dazu die Eigenschaft **Beschriftung** des jeweiligen Feldes anpassen.

Diese wird als Spaltenüberschrift übernommen.

Die Spaltenüberschriften können wir per VBA ein- und ausschalten (siehe Formular **frmListenfeldProdukte_Spaltenueberschriften**:

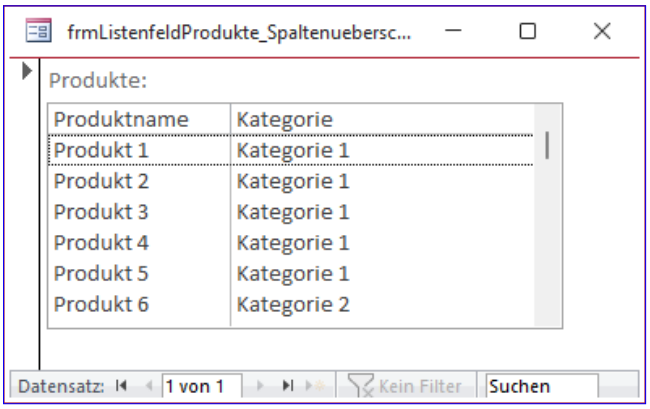

Bild 5: Anzeige von Spaltenüberschriften

```
Private Sub cmdueberschriftenAus_Click()
    Me.lstProdukte.ColumnHeads = False
End Sub

Private Sub cmdueberschriftenEin_Click()
    Me.lstProdukte.ColumnHeads = True
End Sub
```

Im Formular aus Bild 6 haben wir noch einige weitere Schaltflächen zum Experimentieren mit den Spaltenüberschriften eingebaut.

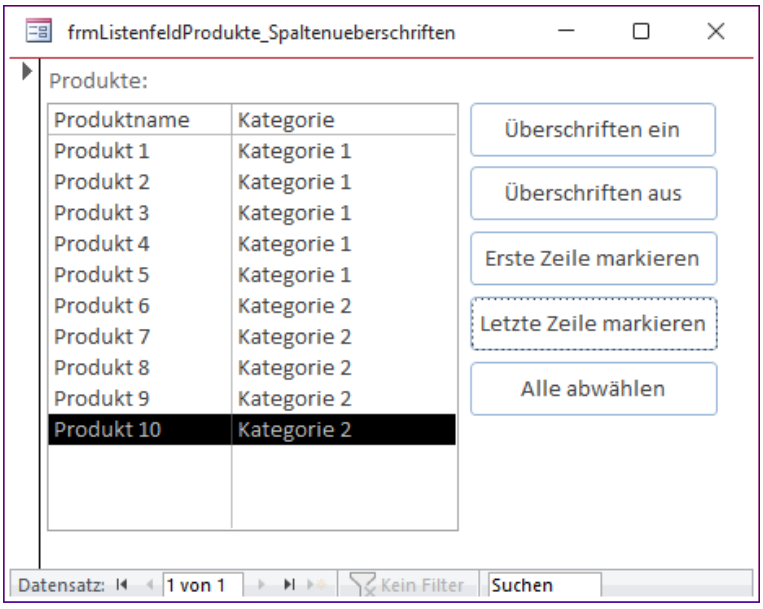

Bild 6: Experimente mit den Spaltenüberschriften

Schaltflächen-Matrix per VBA erzeugen

Manchmal benötigt man viele gleichartige Steuerelemente. Ein bekanntes Beispiel ist eine Kalenderansicht, wo man mehrere Wochen mit jeweils sieben Tagen über Schaltflächen anzeigen möchte. Ein aktuelles Beispiel aus dieser Ausgabe ist die Iconverwaltung aus dem Artikel »Icons in der Datenbank verwalten«. Hier wollen wir für die zu erzeugenden Schaltflächen die verfügbaren Icons in drei Reihen mit je 15 Icons anzeigen, damit der Benutzer schnell das gewünschte Icon auswählen kann. In diesem Beitrag zeigen wir, wie man solche Schaltflächen schnell zu einem Formular in der Entwurfsansicht hinzufügen kann.

Die gewünschte Matrix aus Schaltflächen sehen wir in Bild 1.

Bild 1: Diese Schaltflächenmatrix wollen wir per VBA erzeugen.

Wenn wir solche Schaltflächen anlegen wollen, haben wir zwei Möglichkeiten:

- Wir erledigen die Aufgabe von Hand. Dazu legen wir die Schaltflächen einzeln an, positionieren diese, weisen ihnen Steuerelementnamen zu und stellen gegebenenfalls noch den Wert für die Eigenschaft **Beim Klicken** ein.
- Oder wir nutzen die Zeit mit einer anspruchsvolleren Aufgabe, bei der wir die Schaltflächen durch eine VBA-Prozedur erstellen lassen.

Die erste Variante hat den Vorteil, dass man hier nicht viel nachzudenken braucht. Es gibt sicherlich Gelegenheiten, an denen man Lust auf eine solche Aufgabe hat. Das Problem: Es dauert lange und man muss jederzeit mit Flüchtigkeitsfehlern rechnen, weil man beispielsweise die Schaltflächen nicht konsistent benennt. Außerdem ist das genaue Positionieren recht aufwändig.

Die zweite Variante erfordert eine gewisse Denkleistung und Programmierskills. Deshalb zeigen wir in diesem Beitrag, wie es genau funktioniert. Der Vorteil dieser Variante ist, dass Fehler kein Problem sind. Wenn sich herausstellt, dass die Größe der Schaltflächen nicht stimmt oder die Abstände nicht passen, können wir einfach die vorhandenen Steuerelemente löschen und die Prozedur erneut ausführen. Wichtig ist, dass wir wirklich alle Eigenschaften, die wir benötigen, direkt mit dem Code einstellen und diese nicht nachträglich ändern, weil wir sonst mit einer Neuerstellung der Schaltflächen die manuell hinzugefügten Änderungen wieder verwerfen würden.

Wie wir diese Matrix nutzen, lesen Sie im Beitrag **Icons in der Datenbank verwalten** (**www.access-im-unternehmen.de/1564**).

Optimierter »Suchen und Ersetzen«-Dialog: Funktionen

Im Beitrag »Besserer »Suchen und Ersetzen«-Dialog: Grundgerüst« haben wir das Grundgerüst für einen Ersatz der eingebauten Suchen- und Ersetzen-Funktion von Access erstellt. Im vorliegenden Artikel wollen wir nun die Such- und Ersetzungsfunktionen mit Leben füllen. Dabei werden Sie nicht nur die begonnene Lösung vervollständigen, sondern auch noch Einiges über das VBA-gesteuerte Suchen und Ersetzen in Datenblättern lernen.

Die Lösung aus dem oben genannten Beitrag bildet bisher die vollständige Benutzeroberfläche des eingebauten **Suchen und Ersetzen**-Dialogs ab.

Allerdings haben wir die eigentlichen Suchen- und Ersetzen-Funktionen noch nicht eingebaut. Das holen wir in diesem Beitrag nach.

Dabei unterscheiden wir verschiedene Fälle, die wir in den folgenden Abschnitten im Detail beschreiben.

Einfaches Weitersuchen in einem Feld

Der erste und einfachste Fall ist die Suche im aktuell markierten Feld (siehe Bild 1). Dazu enthält die Eigenschaft **Suchen** den Wert **Aktuelles Feld**. Vorher müssen wir uns allerdings noch um einige weitere Informationen kümmern.

Referenzieren des aktuellen Datenblatts

Wichtig ist für den **Suchen und Ersetzen**-Dialog, dass wir einen Verweis auf das Datenblatt erhalten, das wir untersuchen wollen. Dazu führen wir in der Ereignisprozedur, die durch das Öffnen des Formulars **frmSuchenUndErsetzen** ausgelöst wird, einige Aktionen aus (siehe Listing 1).

Die Prozedur ruft zuerst die Funktion **AktuelleDatenblattansichtObjekt** auf. Diese sehen wir in Listing 2. Sie versucht, bei deaktivierter Fehlerbehandlung über **Screen.ActiveDatasheet** auf das aktive **Datasheet**-Objekt zuzugreifen. Das gelingt in vielen Fällen, aber zum Beispiel dann nicht, wenn sich das Datenblatt in einem Unterformular befindet. Dann liefert der Zugriff auf **Screen.ActiveDatasheet** einen Fehler zurück.

Deshalb deaktivieren wir hier die Fehlerbehandlung und prüfen anschließend, ob **objAktuellesObjekt** einen Wert enthält. Falls nicht, versuchen wir es auf einem anderen Weg. Wir versuchen, ebenfalls bei deaktivierter Fehlerbehandlung, mit **Screen.ActiveForm.ActiveControl** einen Verweis auf das Unterformular zu erhalten und in

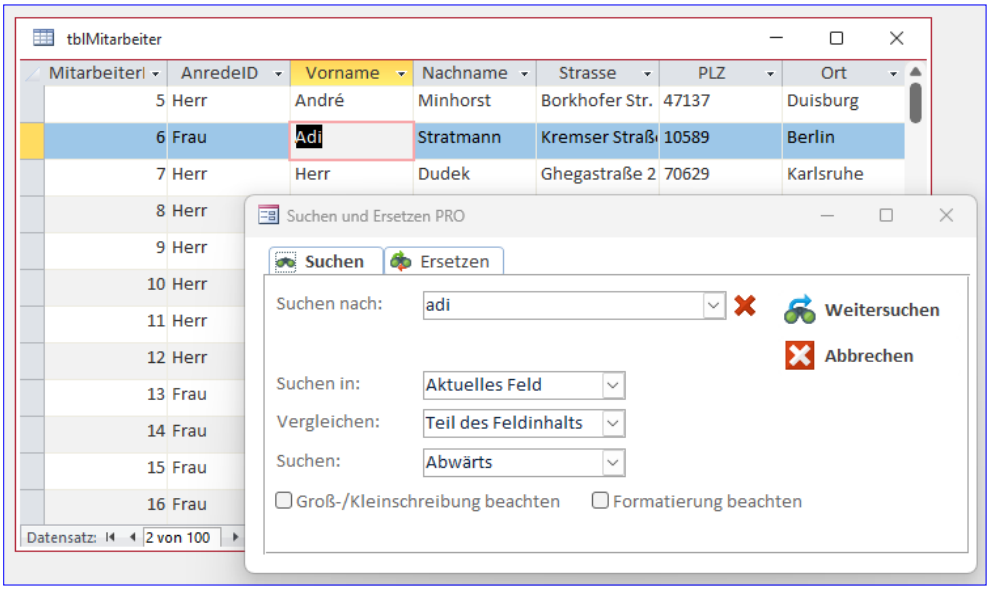

Bild 1: Erster Fall: Suchen in einem einzelnen Feld

```
Private Sub Form_Open(Cancel As Integer)
    Dim strObjektname As String
    Set objAktuellesDatenblatt = AktuelleDatenblattansichtObjekt
    If objAktuellesDatenblatt Is Nothing Then
        MsgBox "Suchen und Ersetzen steht nur für die Datenblattansicht zur Verfügung.", vbOKOnly + vbExclamation, _
            "Keine Datenblattansicht aktiviert"
        Cancel = True
        Exit Sub
    Else
        Set rstDatenblatt = objAktuellesDatenblatt.Recordset
        Me.TimerInterval = 100
    End If
    If Not IsNull(Me.OpenArgs) Then
        If Me.OpenArgs = "Ersetzen" Then
            bolErsetzen = True
        End If
    End If
End Sub
```

Listing 1: Aktionen beim Öffnen des Formulars

objAktuellesObjekt einzutragen. Wie gesagt: Datenblätter in Hauptformularen, Tabellen oder Abfragen liefert uns bereits **Screen.ActiveDatasheet** das aktuelle Datenblatt.

Die Funktion prüft dann, ob das aktive Steuerelement ein Datenblatt ist. Falls ja, liefert sie einen Verweis auf die **Form**-Eigenschaft von **objAktuellesObjekt** zurück.

Die Prozedur **Form_Open** schreibt das Ergebnis in die Variable **objAktuelles-Datenblatt**, das wie folgt im Kopf des Klassenmoduls des Formulars deklariert wird:

```
Private objAktuellesDatenblatt As Object
```

Dort deklarieren wir direkt auch noch:

```
Private rstDatenblatt As DAO.Recordset
```

Wenn **objAktuellesDatenblatt** nun leer ist, gibt **Form_Open** eine Meldung aus, dass die **Suchen und Ersetzen**-Funktion nur für die Datenblattansicht zur Verfügung steht.

Danach wird die Prozedur mit **Cancel = True** verlassen, sodass das Formular gar nicht erst angezeigt wird.

```
Public Function AktuelleDatenblattansichtObjekt() As Object
    Dim objAktuellesObjekt As Object
    On Error Resume Next
    Set objAktuellesObjekt = Screen.ActiveDatasheet
    On Error GoTo 0
    If objAktuellesObjekt Is Nothing Then
        On Error Resume Next
        Set objAktuellesObjekt = Screen.ActiveForm.ActiveControl
        If objAktuellesObjekt.CurrentView = acCurViewDatasheet Then
            Set AktuelleDatenblattansichtObjekt = objAktuellesObjekt.Form
            Exit Function
        End If
        On Error GoTo 0
    Else
        Set AktuelleDatenblattansichtObjekt = objAktuellesObjekt
    End If
End Function
```

Listing 2: Aktuelle Datenblattansicht holen

Anderenfalls schreibt sie einen Verweis auf das Recordset in **objAktuellesDatenblatt** in die Variable **rstDatenblatt** und aktiviert einen Timer, der alle 100 Millisekunden feuern soll (es ginge vermutlich auch ein größeres Intervall). Diesen Timer schauen wir uns gleich an.

Schließlich prüft die Prozedur noch das Öffnungsargument. Enthält dieses den Wert **Ersetzen**, wird **bolErsetzen** auf **True** eingestellt. Dies sorgt später dafür, dass neben dem **Suchen**- auch das **Ersetzen**-Tab im Formular angezeigt wird.

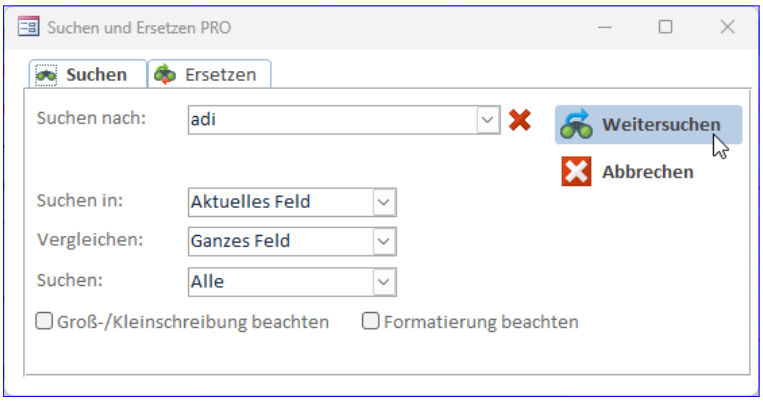

Bild 2: Einfache Suche

Die Prozedur, die durch das Ereignis **Bei Zeitgeber** ausgelöst wird, prüft bei jedem Aufruf, ob das Datenblatt aus **rstDatenblatt** noch vorhanden ist. Dazu greifen wir einfach auf die **RecordCount**-Eigenschaft zu. Löst dies einen Fehler aus, sind das Recordset und somit wohl auch die Datenblattansicht, für die der **Suchen und Ersetzen**-Dialog geöffnet wurde, wohl nicht mehr vorhanden:

```
Private Sub Form_Timer()
    Dim lngRecordcount As Long
    On Error Resume Next
    lngRecordcount = rstDatenblatt.RecordCount
    If Not Err.Number = 0 Then
        On Error GoTo 0
        Me.TimerInterval = 0
        DoCmd.Close acForm, Me.Name
    End If
End Sub
```

Einfachen Suchvorgang ausführen

Damit kommen wir zum eigentlichen Suchvorgang. Wir schauen uns zuerst einen einfachen Suchvorgang mit den Parametern aus Bild 2 an.

Ein Klick auf die Schaltfläche **Weitersuchen** löst die Prozedur aus Listing 3 aus. Diese holt sich den Suchbegriff aus dem Kombinationsfeld **cboSuchenNach** in die Variable **txtSuchbegriff**. Wenn dieser eine Länge von **0** hat, bricht sie die Prozedur ab.

Anderenfalls erfolgt eine Unterscheidung nach dem Umfang der Suche, der mit dem Kombinationsfeld **cboSuchenIn** festgelegt wird. Im Fall von **Aktuelles Feld** beginnt der Teil, den wir hier beschreiben wollen.

Es folgt ein Aufruf der Funktion **KriteriumAktuellesFeld**, der als Ergebnis das Suchkriterium für die Suche liefern soll. Das ist recht aufwendig, da wir prüfen müssen, welchen Datentyp das Feld enthält, ob dieser zum eingegebenen Suchbegriff passt und so weiter – mehr dazu weiter unten. Wir gehen an dieser Stelle davon aus, dass wir zum Beispiel den Suchbegriff **adi** eingegeben haben und als Kriterium mit dem Parameter **strKriterium** den folgenden Ausdruck zurückerhalten:

```
Vorname LIKE '*adi*'
```

Dann wird die Variable **bolFound** zunächst auf **False** eingestellt. Nun prüfen wir den Inhalt von **cboSuchen** und steuern abhängig davon einen der **Case**-Zweige für **Alle**, **Abwärts** oder **Aufwärts** an.

Suchen nach allen Vorkommen im aktuellen Feld

Wenn der Benutzer **Alle** gewählt hat, haben wir die umfangreichste Ausgabe vor uns. Während wir bei Abwärts oder Aufwärts einfach vom aktuellen Datensatz aus nach

```
Private Sub cmdWeitersuchen_Click()
    Dim strFeld As String
    Dim strSuchbegriff As String
    Dim strKriterium As String
    Dim bolFound As Boolean
    Dim strMeldung As String

    strSuchbegriff = Nz(Me.cboSuchenNach.Column(1), "")
    If Len(strSuchbegriff) = 0 Then Exit Sub
    Select Case Me.cboSuchenIn
        Case "Aktuelles Feld"
            If KriteriumAktuellesFeld(strSuchbegriff, strFeld) = True Then
                bolFound = False
                Select Case Me.cboSuchen
                    Case "Alle"
                        Call SuchenDatensatz(bolFound, strKriterium, strFeld, strSuchbegriff)
                    Case "Abwärts"
                        rstDatenblatt.FindNext strKriterium
                        If rstDatenblatt.NoMatch Then
                            bolFound = False
                        Else
                            bolFound = True
                        End If
                    Case "Aufwärts"
                        rstDatenblatt.FindPrevious strKriterium
                        If rstDatenblatt.NoMatch Then
                            bolFound = False
                        Else
                            bolFound = True
                        End If
                End Select
                If Not bolFound Then
                    lngAbsolutePositionStart = -1
                    MsgBox "Kein weiterer Eintrag gefunden."
                Else
                    Select Case Me.cboVergleichen
                        Case "Teil des Feldinhalts", "Anfang des Feldinhalts"
                            Call SuchbegriffMarkieren(objAktuellesDatenblatt.ActiveControl, Me.cboSuchenNach.Column(1))
                    End Select
                End If
            Else
                MsgBox strMeldung, vbExclamation + vbOKOnly, "Suche nicht erfolgreich"
            End If
        Case "Aktuelles Dokument"
            MsgBox "Nicht implementiert."
    End Select
End Sub
```

Listing 3: Code hinter der Schaltfläche **Weitersuchen**

oben oder unten suchen können, müssen wir in diesem Fall vom aktuellen Datensatz aus nach unten suchen, dann oben neu beginnen und vor dem aktuellen Datensatz stoppen. Da dies ein wenig aufwendiger ist, haben wir diesen Teil in eine eigene Prozedur namens **Suchen-Datensatz** ausgelagert (siehe Listing 4).

Dieser übergeben wir verschiedene Parameter, die zum größten Teil auch als Rückgabeparameter dienen:

- **bolGefunden** gibt an, ob ein Datensatz gefunden wurde.
- **strKriterium** nimmt das Kriterium entgegen.

Die beiden folgenden Variablen müssen wir noch im Modulkopf deklarieren, da diese von mehreren Prozeduren genutzt werden:

```
Private lngAbsolutePositionStart As Long
Private varLetzteFundposition As Variant
```

Beim Aufrufen des Formulars **frmSuchenUndErsetzen** wird der Wert von **lngAbsolutePositionStart** auf **-1** eingestellt. Deshalb ist die erste Bedingung beim ersten Suchlauf nach dem Öffnen immer wahr und wir stellen nun **lngAbsolutePositionStart** auf die aktuelle Position des Datensatzzeigers aus **rstDatenblatt.AbsolutePosition** ein.

Außerdem legen wir hier den Wert der Variablen **bolVonVornGestartet** auf **False** fest. Diese Variable gibt an, ob wir den Suchlauf bereits von vorn begonnen haben. Das ist wichtig, wenn wir mittendrin mit der Suche beginnen, damit auch die vor dem Startpunkt liegenden Datensätze noch durchsucht werden können.

Wenn der Benutzer nach dem ersten Suchlauf noch weitere Suchvorgänge startet, hat **lngAbsolutePositionStart** bereits einen anderen Wert als **-1**. Also wird der zweite Teil der **If...Then**-Bedingung angesteuert.

Hier prüfen wir, ob sich der Datensatzzeiger aktuell auf der letzten Fundstelle befindet. Das sollte immer der Fall sein, wenn der Benutzer mehrere Suchvorgänge hintereinander aufruft. Wenn sich der Datensatzzeiger woanders befindet, gehen wir davon aus, dass der Benutzer diesen manuell an eine andere Stelle verschoben hat. In diesem Fall stellen wir **lngAbsolutePositionStart** wieder auf die aktuelle Position ein und **bolVonVornGestartet** auf **False**, da selbst ein Suchvorgang, der bis zum Ende lief und von vorn beginnt, nun zurückgesetzt werden soll.

Damit können wir endlich suchen und nutzen dafür entweder die **FindFirst**- oder die **FindNext**-Methode mit dem Kriterium aus **strKriterium**. **FindFirst** nutzen wir nur, wenn zuvor **bolStart** auf **True** eingestellt wurde, was der Fall ist, wenn die Funktion zum ersten Mal aufgerufen wird und nicht wiederholt für den gleichen Suchbegriff. Dann wird **bolStart** direkt auf **False** eingestellt. Bei den folgenden Aufrufen suchen wir dann mit **FindNext**.

Die folgende **If...Then**-Bedingung prüft durch den Vergleich der Eigenschaft **NoMatch** mit dem Wert **False**, ob es einen Suchtreffer gab. In diesem Fall prüfen wir, ob **bolVonVornGestartet** den Wert **True** hat, also ob wir die Suche bereits wieder ganz oben fortgesetzt haben. Ist das der Fall, kann es sein, dass wir mit der Suche ein zweites Mal auf dem gleichen Datensatz landen. Um das herauszufinden, ermitteln wir, ob die aktuelle Position des Datensatzzeigers aus **rstDatenblatt.AbsolutePosition** größer ist als die Startposition aus **lngAbsolutePosition-Start**. Ist diese Bedingung wahr, sind wir mehr als einmal durch die Datensätze navigiert. Dann verschieben wir den Datensatzzeiger wieder auf die Position der vorherigen Fundstelle, geben für **bolGefunden** den Wert **False** zurück und verlassen die Prozedur.

Sind die letzten beiden Bedingungen nicht erfüllt, haben wir einen Suchtreffer. Wir speichern die Position des Datensatzzeigers für diesen Datensatz in der Variablen **lngLetzteFundstelle**, geben den Wert **True** für **bolGefunden** zurück und verlassen die Prozedur.

Einfacher Datenklau durch mangelnde Sicherheit

Wir haben in den letzten Wochen einige Datenbanken von Kunden untersucht und erhebliche Mängel bei der Sicherheit der Daten festgestellt. Primär haben wir uns angeschaut, ob ein Mitarbeiter ohne Weiteres auf die in der Datenbank gespeicherten Daten zugreifen kann. In den meisten Fällen konnte er sich einfach die Backend-Datenbank kopieren und so in den Besitz wichtiger Daten des Unternehmens gelangen. Oft war zumindest das Backend durch ein Kennwort geschützt, sodass man nur über das Frontend auf die Daten zugreifen kann. Ob das sicher ist, schauen wir uns im vorliegenden Beitrag an. Um es vorwegzunehmen: Es ist nicht sicher.

Trügerische Sicherheit? Noch nicht einmal das.

In einer Aktion, bei der wir uns die Datenbanken von Kunden angesehen haben, sind gravierende Sicherheitsmängel aufgetaucht:

- Oft waren die Daten im Backend völlig ungesichert. Da die Benutzer schreibenden Zugriff auf die Backend-Datenbank benötigen, können sie theoretisch leicht die Backend-Datenbank kopieren und auf einen USB-Stick ziehen.

- Teilweise haben wir zumindest per Kennwort geschützte Backenddatenbanken vorgefunden. Die Verknüpfung zu den Tabellen im Backend enthielt dann jedoch das Kennwort in den Verknüpfungseigenschaften, was sich ebenfalls leicht auslesen lässt.

- In den meisten Fällen konnte der Benutzer über den Navigationsbereich direkt auf die verknüpften Tabellen zugreifen. Der Manipulation von Daten wie etwa geleistete Arbeitsstunden oder dem Gehalt sind so leicht möglich, wenn die entsprechenden Tabellen vorliegen.

Aber selbst wenn der Entwickler die Anwendung so weit geschützt hat, dass der Benutzer nicht auf die eingebundenen Tabellen zugreifen kann, lassen sich alle Daten von außen leicht auslesen. In diesem Beitrag zeigen wir, wie das funktioniert. Voraussetzung dafür ist, dass der Benutzer die Frontend-Datenbank geöffnet hat.

Daten auslesen per VBA

Wir legen dazu in einer neuen Datenbank die Prozedur aus Listing 1 an. Die Vorbedingung für ihre Funktion ist:

- Die auszulesende Datenbank ist bereits geöffnet.

- Die auslesende Datenbank mit unserer VBA-Prozedur wird erst danach geöffnet.

In der Prozedur erstellen wir ein Objekt des Typs **Access.Application** auf Basis des Aufrufs **GetObject(, "Access.Application")**. Unter den oben angeführten Voraussetzungen sollten wir damit einen Verweis auf die zu untersuchende Datenbank erhalten.

Danach weisen wir der Variablen **db** einen Verweis auf das **Database**-Objekt dieser Datenbank zu und durchlaufen die enthaltenen Tabellen. Für alle Tabellen, deren Eigenschaft **Connect** einen Wert enthält und die wir so als Tabellenverknüpfung identifizieren können, geben wir zunächst den Namen aus.

Danach erstellen wir ein Recordset auf Basis der aktuellen Tabelle und durchlaufen die enthaltenen Datensätze. Dabei geben wir jeweils den Namen und den Inhalt der

```
Public Sub DatenbankAuslesen()
    Dim db As DAO.Database
    Dim tdf As DAO.TableDef
    Dim rst As DAO.Recordset
    Dim objAccess As Access.Application
    Dim fld As DAO.Field
    Set objAccess = GetObject(, "Access.Application")
    Set db = objAccess.CurrentDb
    For Each tdf In db.TableDefs
        If Not Len(tdf.Connect) = 0 Then
            Debug.Print tdf.Name
            Set rst = db.OpenRecordset("SELECT * FROM " & tdf.Name, dbOpenDynaset)
            Do While Not rst.EOF
                For Each fld In rst.Fields
                    Debug.Print fld.Name; ": ",
                    On Error Resume Next
                    Debug.Print fld.Value,
                    On Error GoTo 0
                Next fld
                Debug.Print
                rst.MoveNext
            Loop
        End If
    Next tdf
End Sub
```

Listing 1: Auslesen der Inhalte der Tabellen einer anderen geöffneten Access-Datenbank

Felder aus. Wir verwenden noch eine Fehlerbehandlung, wenn sich der Inhalt nicht direkt ausgeben lässt – zum Beispiel für Felder des Typs **OLE-Objekt**.

Wir können so sehr einfach alle Daten der verknüpften Tabellen im zu untersuchenden Frontend ausgeben.

Das dient nur Beispielzwecken – wenn wir die Daten sichern und beispielsweise in einer eigenen Datei speichern und diese mitnehmen wollten, wäre auch das einfach möglich.

Wir könnten einfach das Datenmodell nachbauen, in dem wir dieses auslesen und in einer neuen Datenbank reproduzieren. Dann können wir Tabelle für Tabelle und Datensatz für Datensatz die gewünschten Informationen in die Zieltabelle übertragen.

Übertragen der Tabellen der geöffneten Datenbank in die Zieldatenbank

Zum Übertragen des kompletten Inhalts einer Datenbank bedarf es gar nicht viel. Wir haben diese Funktion auf drei Routinen aufgeteilt:

- Die erste heißt **TabellenUebertragen** und ermittelt einen Verweis auf das Database-Objekt der Datenbank mit den zu übertragenden Tabellen und durchläuft die **TableDef**-Objekte.
- Dabei ruft sie für jede Tabelle die zweite Routine namens **TabelleNachbauen** auf. Diese kopiert die Tabellen und die enthaltenen Felder Stück für Stück.
- Die dritte Prozedur **DatenKopieren** überträgt schließlich die Daten der Felder der Quelltabelle in die Zieltabelle.

Datenbank durch Runtime-Modus sicher machen?

Access-Datenbanken, die man als .accdb bereitstellt, können dem Benutzer immer noch Elemente eröffnen, mit denen dieser auf Bereiche zugreift, die er eigentlich nicht sehen soll. Wir sprechen dabei vor allem vom Navigationsbereich oder von den eingebauten Ribbon-Befehlen, mit denen Elemente in der Entwurfsansicht angezeigt oder sogar bearbeitet werden können. Durch die Umwandlung in eine .accde-Datenbank lässt sich zumindest die Bearbeitung von Formularen, Berichten und VBA-Code verhindern. An die übrigen Elemente kommt der Benutzer aber dennoch heran. Dieser Beitrag zeigt, wie wir für den Benutzer noch einige Möglichkeiten dieser Art sperren können – und wie wir die größtmögliche Sicherheit erreichen.

Eine Access-Datenbank, die wir als **.accdb**-Datenbank an den Benutzer weitergeben, erlaubt den vollständigen Zugriff auf alle Elemente. Selbst wenn wir den Navigationsbereich zunächst ausblenden und die Access-Spezialtasten blockieren, kann der Benutzer dies in den meisten Fällen und mit ein wenig Geschick umgehen. Das eingebaute Ribbon können wir unsichtbar machen, indem wir eine eigene Ribbondefinition hinzufügen, die

```
<?xml version="1.0"?>
<customUI xmlns="http://schemas.microsoft.com/office/2009/07/customui" loadImage="loadImage">
  <ribbon startFromScratch="true">
    <tabs>
      <tab id="grpMain" label="Main-Gruppe">
        <group id="grpBeispiel" label="Beispielgruppe">
          <button image="banana" label="Formular öffnen" id="btnFormOeffnen" onAction="onAction" size="large"/>
        </group>
      </tab>
    </tabs>
  </ribbon>
  <backstage>
    <tab idMso="PlaceTabHome" visible="false"/>
    <tab idMso="TabOfficeStart" visible="false"/>
    <tab idMso="TabRecent" visible="false"/>
    <tab idMso="TabInfo" visible="false"/>
    <button idMso="FileSave" visible="false"/>
    <tab idMso="TabSave" visible="false"/>
    <tab idMso="TabPrint" visible="false"/>
    <button idMso="FileCloseDatabase" visible="false"/>
    <tab idMso="TabHelp" visible="false"/>
    <button idMso="ApplicationOptionsDialog" visible="false"/>
 </backstage>
</customUI>
```

Listing 1: Ribbon-Anpassung, um alle eingebauten Elemente auszublenden

wir als Anwendungsribbon festlegen. Aber auch diese Einstellung lässt sich leicht durch Verwenden der Umschalttaste umgehen.

Wir schauen uns erst einmal an, wie wir diese Einstellungen für den unbedarften Benutzer vornehmen können und der Anwendung so einen provisorischen, aber auch leicht zu umgehenden Schutz mitgeben:

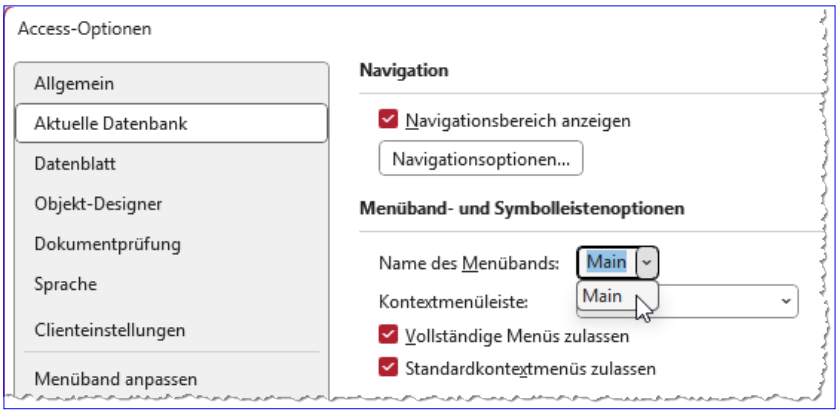

Bild 1: Ribbondefinition festlegen, die beim Start angewendet wird

- Ribbon ausblenden
- Navigationsbereich ausblenden
- Access-Standardtasten sperren

Ausblenden des eingebauten Ribbons

Das eingebaute Ribbon blenden wir aus, indem wir der Anwendung eine einfache Ribbondefinition wie die aus Listing 1 hinzufügen. Diese bewirkt folgende Änderungen:

- Mit **startFromScratch="true"** im Element **ribbon** sorgen wir dafür, dass die eingebauten Elemente im Ribbon ausgeblendet werden.
- Durch das Einstellen von **visible** auf **false** in den Elementen unterhalb von **backstage** blenden wir die **Datei**-Befehle aus.
- Wir blenden lediglich die benutzerdefinierten Befehle ein, die wir tatsächlich bereitstellen wollen.

Damit die Ribbondefinition direkt beim Start angewendet wird, müssen wir den Namen nach dem Anlegen in den Access-Optionen für die Eigenschaft **Name des Menübands** hinterlegen (siehe Bild 1).

Daneben benötigen wir noch ein wenig Code für die Funktionalität der Ribbonanpassung, den Sie in den beiden Modulen **mdlRibbon** und **mdlRibbonImages** finden. Außerdem müssen wir das anzuzeigende Bild **banana.png** noch zur Tabelle **MSysResources** hinzufügen.

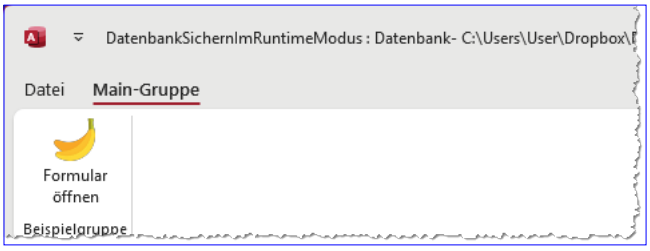

Bild 2: Das Ribbon mit ausgeblendeten Standardelementen

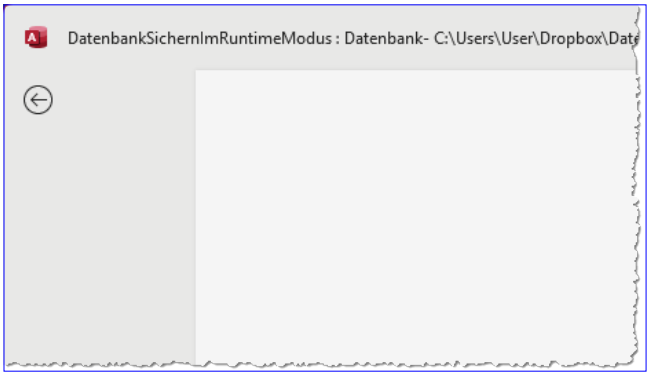

Bild 3: Auch das Datei-Menü ist nun völlig leer.

Wenn wir die Anwendung nun schließen und erneut öffnen, sieht das Ribbon der Anwendung bereits wie in Bild 2 aus.

Hier hat der Benutzer nun keine Gelegenheit mehr, die Objekte zu manipulieren.

Wenn wir auf **Datei** klicken, finden wir dort ebenfalls keine Einträge mehr vor (siehe Bild 3). Dafür haben wir mit dem Bereich **backstage** in der Ribbondefinition gesorgt.

Auch hier kann der Benutzer nun keinerlei Änderungen mehr vornehmen.

Navigationsbereich ausblenden und Spezialtasten deaktivieren

Nun nehmen wir zwei weitere Änderungen vor, die dem Benutzer den Zugriff auf den Navigationsbereich verwehren sollen.

Um den Navigationsbereich beim Starten gar nicht erst einzublenden, deaktivieren wir, wieder in den Access-Optionen, die Eigenschaft **Navigationsbereich anzeigen**.

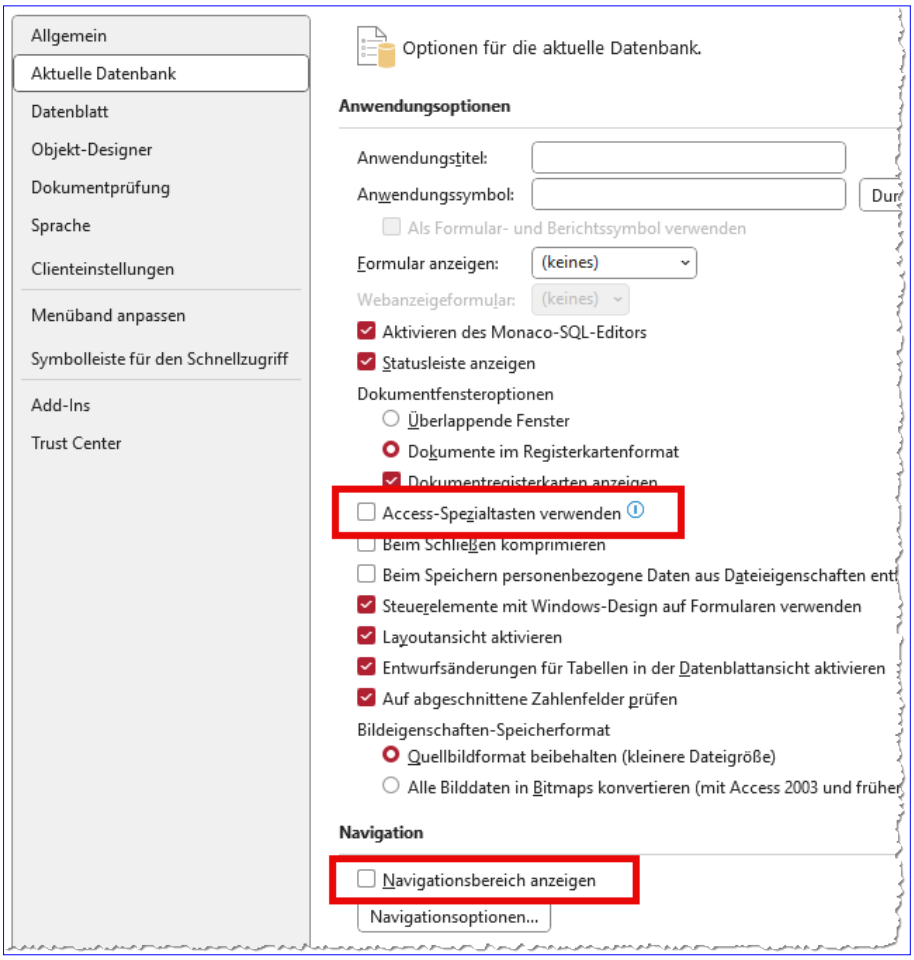

Bild 4: Deaktivieren weiterer Eigenschaften

Da der Benutzer diesen Bereich nun problemlos mit der Taste **F11** wieder einblenden könnte, verhindern wir auch dies. Dazu deaktivieren wir die Option **Access-Spezialtasten verwenden** (siehe Bild 4).

Nach einem erneuten Start bleibt der Navigationsbereich nun ausgeblendet und der Benutzer kann diesen auch gar nicht erst wieder einblenden.

Bisherige Änderungen mit der Umschalt-Taste umgehen

Wenn der Benutzer sich etwas auskennt, wird er jedoch direkt probieren, die Datenbank erneut bei gedrückter **Umschalt**-Taste zu öffnen. Und er wird damit Erfolg haben: Alle Elemente sind wieder sichtbar, und auch die Einstellung zum Sperren der Access-Spezialtasten kann er so umgehen.

Verwendung der Umschalttaste beim Start verhindern

Um dies zu verhindern, reichen die Access-Optionen nicht mehr aus. Hier müssen wir per VBA eine bestimmte Eigenschaft setzen. Diese heißt **AllowBypassKey** und muss zunächst zur Anwendung hinzugefügt werden. Wir implementieren die folgenden Funktionen direkt in einer eigenen Datenbankdatei, hier namens **EnableDisableBypassKey.accdb**.

Wenn **AllowBypassKey** den Wert **True** aufweist, kann man die Anwendung bei gedrückter Umschalttaste öffnen

Icons in der Datenbank verwalten

Mit den neueren Versionen von Access ab der Version 2010 kann man Icons, die an verschiedenen Stellen der Benutzeroberfläche wie in Bildsteuerelementen oder Schaltflächen angezeigt werden sollen, relativ einfach verwalten. Sie landen nach dem Einfügen in einer Systemtabelle namens MSysResources, die ein Anlagefeld zum Speichern der Bilddateien und weitere Felder für die Metadaten wie Bildname oder Dateiendung enthält. Allerdings ist die Verwaltung nicht perfekt gelöst. Daher stellen wir in diesem Beitrag ein Formular vor, mit dem sich die Bilder wesentlich besser hinzufügen lassen. Außerdem enthält das Formular eine Suchfunktion, mit der sich die Bilder nach dem Namen filtern lassen, sowie eine Übersicht der Bilder, mit der sich die jeweiligen Bilddateien anschauen lassen. Schließlich können wir auch noch den Bildnamen herausfinden, um das gewünschte Icon schnell einer Schaltfläche oder einem Bildsteuerelement zuweisen zu können.

Nachteile der eingebauten Verwaltung von Icons

Die eingebauten Funktionen zum Hinzufügen von Icons zu einer Access-Datenbank sind bereits recht hilfreich, wenn man dies mit früheren Versionen vergleicht, wo man Bilder bestenfalls im OLE-Format nutzen konnte.

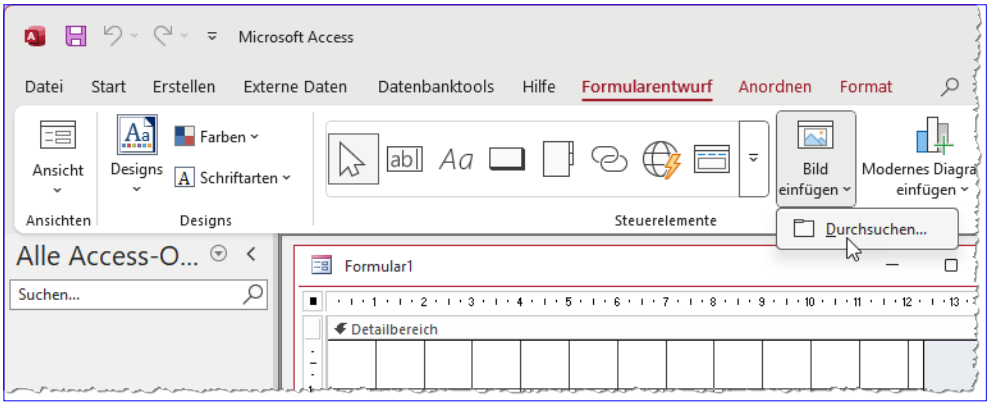

Bild 1: Funktion zum Hinzufügen von Bilddateien

Wenn wir ein Formular in der Entwurfsansicht geöffnet haben, können wir im Ribbon auf den Befehl **FormularentwurflSteuerelementelBild einfügen** einen Dateidialog öffnen, mit dem wir das einzufügende Bild auswählen (siehe Bild 1).

Unabhängig davon, ob wir danach in das Formular klicken und ein Bildsteuerelement gefüllt mit diesem Bild anlegen, wird das Bild in der Tabelle **MSysResources** gespeichert und steht nachfolgend in Steuerelementen

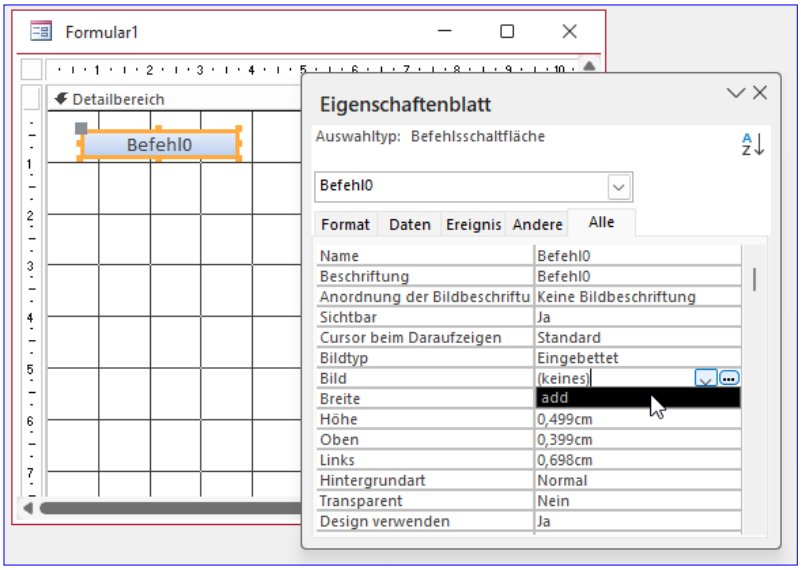

Bild 2: Das hinzugefügte Bild

wie der Schaltfläche oder dem Bildsteuerelement zur Auswahl bereit (siehe Bild 2).

Anschließend können wir uns einen Überblick über die enthaltenen Bilder verschaffen, indem wir erneut auf **Bild einfügen** klicken (siehe Bild 3).

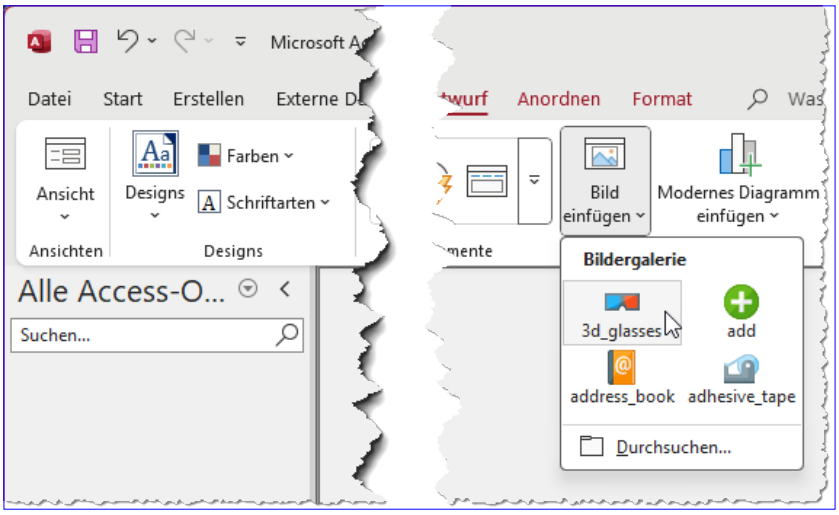

Bild 3: Auswahl der verfügbaren Bilder der Tabelle **MSysResources**

Was hier fehlt, sind folgende Funktionen:

- Wir können immer nur ein Bild gleichzeitig zur Tabelle **MSysResources** hinzufügen.
- Die Bildergalerie zeigt nur eine begrenzte Anzahl von Bildern an. Unsere Sammlung von ca. 1.700 Icons konnte nicht vollständig angezeigt werden.
- Es fehlt eine Suchmöglichkeit nach dem Bildnamen.

All diese Funktionen rüsten wir mit unserem Formular nach, dass sich leicht in eigene Anwendungen einbauen lässt.

Formular zum Verwalten von Icons

Das Formular dieser Lösung heißt **frmIconsVerwalten** und sieht wie in Bild 4 aus.

Es enthält 45 Schaltflächen in drei Reihen, die zur Anzeige der ersten verfügbaren Bilder dienen.

Wenn sich mehr als 45 Bilder in der Tabelle **MSysResources** befinden, können wir mit den beiden **Nach oben**- und **Nach unten**-Schaltflächen reihenweise vor- und zurücknavigieren.

Oben sehen wir ein Textfeld namens **Filter**. Hier können wir einen Filterausdruck eingeben, mit dem wir die Bilder in der Tabelle **MSysResources** nach dem Namen filtern können. Die Anzeige wird nach der Eingabe eines jeden Zeichens aktualisiert und sucht stets an beliebiger Stelle im Dateinamen der Bilder nach dem Filterbegriff.

Rechts daneben befindet sich eine Schaltfläche, mit der wir das aktuell markierte Bild aus der Tabelle **MSysResources** löschen können.

Die Schaltfläche **Hinzufügen** öffnet einen Dateiauswahl-Dialog, mit dem wir weitere Bilder zur Tabelle **MSysResources** hinzufügen können.

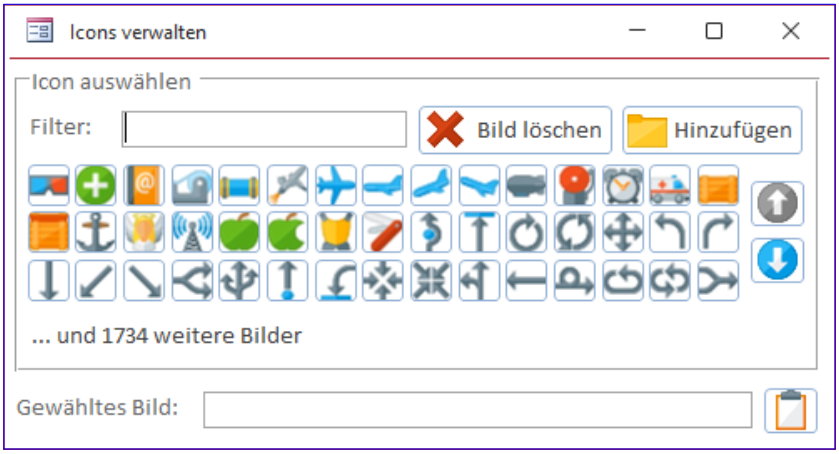

Bild 4: Formular zum Verwalten von Icons

Unter der Anzeige der Bilder sehen wir die Anzahl der noch nachfolgend vorhandenen Bilder. Diese Anzahl ändert sich, wenn wir mit der **Nach unten**-Schaltfläche zu weiteren Zeilen navigieren.

Schließlich können wir ein Bild anklicken und erhalten im Textfeld **Gewähltes Bild** den Namen des Bildes, das wir mit einem Klick auf die Schaltfläche rechts davon in die Zwischenablage kopieren und beispielsweise für die Eigenschaft **Bild** einer Schaltfläche einfügen können.

Bild 5: Das Formular **frmIconsVerwalten** in der Entwurfsansicht

Entwurf des Formulars

Das Formular enthält im Wesentlichen herkömmliche Steuerelemente (siehe Bild 5). Die 45 Schaltflächen, die mit **cmd001** bis **cmd045** benannt sind, haben wir mit der Funktion aus dem Beitrag **Schaltflächen-Matrix per VBA erzeugen** (**www.access-im-unternehmen.de/1562**) erstellt.

Füllen der Schaltflächen mit den verfügbaren Bildern

Die Prozedur **LoadImages** lädt die Bilder aus der Tabelle **MSysResources** als Icons in die 45 Schaltflächen (erster Teil in Listing 1). Sie nimmt optional mit dem Parameter **strFilter** ein Filterkriterium entgegen und stellt einen zweiten Parameter namens **bolMove** zur Verfügung. Dieser gibt an, ob die angezeigten Bilder bereits nach unten gescrollt wurden. Die Variable **strResourcetable** speichert den Namen der Tabelle, aus der die Bilder eingelesen werden, in diesem Fall **MSysResources**.

Die Prozedur referenziert das aktuelle **Database**-Objekt mit der Variablen **db** und deaktiviert das Painting während des Vorgangs, damit das Formular nicht flackert, wenn die Bilder zugewiesen werden.

Die erste **If...Then**-Bedingung prüft, ob der Parameter **strFilter** einen Wert enthält. Ist das nicht der Fall, stellt sie eine SQL-Abfrage zusammen, die alle Datensätze der Tabelle **MSysResources** enthält, deren Feld **Extension** die Dateiendung **png** aufweist. Die Daten werden aufsteigend nach dem Inhalt des Feldes **Name** sortiert.

Ist **strFilter** gefüllt, wird im **Else**-Zweig eine zusätzliche Bedingung in **strSQL** eingebaut, der das Feld **Name** nach dem enthaltenen Wert filtert.

Dann erstellt die Prozedur ein Recordset auf Basis von **strSQL** und prüft, ob das Recordset leer ist.

In diesem Fall ruft sie die Prozedur **SchaltflaechenLeeren** auf, die in einer Schleife über die Werte von **1** bis **45** sowohl die **Picture**- als auch die **ControlTipText**-Eigenschaften der entsprechenden Schaltflächen leert:

```
Private Sub SchaltflaechenLeeren()
    Dim lngIndex As Long
    For lngIndex = 1 To 45
        With Me.Controls("cmd" & Format(lngIndex, "000"))
            .Picture = ""
            .ControlTipText = ""
        End With
    Next lngIndex
End Sub
```

Danach aktiviert die Prozedur das Painting wieder, deaktiviert die Schaltfläche **cmdBildLoeschen**, da nun kein zu löschendes Bild angezeigt wird, und beendet die Prozedur mit **Exit Sub**.

Wenn diese Bedingung jedoch erfüllt ist, trägt die Prozedur den Namen des ersten Bildes in das Textfeld in die Variable **strFirstPicture** ein.

Im nun folgenden Teil prüft eine verschachtelte **If...Then**-Bedingung, ob die Variable **bolMove** den Wert **True** enthält und die Variable **strFilter** nicht leer ist. In diesem Fall stellt sie die Variable **lngStart** auf den Wert **0** ein (siehe Listing 2). **lngStart** wird, damit wir auch beim nächsten Laden der Bilder auf ihren Wert zugreifen können, wie folgt ganz oben im Modul deklariert:

```
Private lngStart As Long
```

Recordset auf Basis vorheriger Scrollvorgänge ermitteln

lngStart könnte anderenfalls in einem zuvor durchgeführten Ladevorgang der Bilder auf einen anderen Wert als **0** eingestellt worden sein, weil der Benutzer bereits gescrollt

```
Private Sub LoadImages(Optional strFilter As String, Optional bolMove As Boolean)
    Dim db As DAO.Database
    Dim rst As DAO.Recordset
    Dim cmd As Access.CommandButton
    Dim strSQL As String
    Dim lngIndex As Long
    Dim lngFirstID As Long
    Dim strFirstPicture As String
    Dim strResourcetable As String

    strResourcetable = "MSysResources"
    Set db = CurrentDb
    Me.Painting = False

    If Len(strFilter) = 0 Then
        strSQL = "SELECT * FROM " & strResourcetable & " WHERE Extension = 'png' ORDER BY Name ASC"
    Else
        strSQL = "SELECT * FROM " & strResourcetable & " WHERE Extension = 'png' AND Name LIKE '*" _
            & strFilter & "*' ORDER BY Name ASC"
    End If

    Set rst = db.OpenRecordset(strSQL, dbOpenDynaset)
    If rst.EOF Then
        Call SchaltflaechenLeeren
        Me.Painting = True
        Me.cmdBildLoeschen.Enabled = False
        Exit Sub
    End If
    strFirstPicture = Nz(rst!Name, "")
    ...
```

Listing 1: Die Prozedur zum Laden von Bildern in die Schaltflächen (Teil 1)

Individueller Standarddatenbankordner je Datenbank

Eines ärgert mich bereits seit Jahrzehnten: Dass man beim Importieren von Objekten aus anderen Datenbanken über den Assistenten immer wieder neu zum jeweiligen Verzeichnis wechseln muss. Dort ist immer das gleiche Verzeichnis voreingestellt. Dieses kann man zwar ändern und so passend für einen Anwendungszweck in einer Datenbank gestalten, aber beim Importieren in die nächste Datenbank muss man das Verzeichnis wieder neu selektieren. Wie schön wäre es, wenn dieser Ordner zumindest immer auf den Ordner eingestellt wird, in dem sich die aktuelle Datenbank befindet. Noch praktischer wäre es, wenn man für jede Anwendung einen individuellen Ordner einstellen könnte, der automatisch beim Öffnen der Anwendung bereitgestellt wird. Die gute Nachricht ist: Wir haben eine Lösung dafür gefunden, die wir in diesem Beitrag vorstellen – damit sind die beiden genannten Varianten möglich! Wie es geht, lesen Sie auf den folgenden Seiten.

Wo wirkt sich der Standarddatenbankordner aus?

Wenn wir Access öffnen, ohne jemals den Standarddatenbankordner geändert zu haben, ist dieser auf den Dokumentenordner des jeweiligen Benutzers eingestellt. Diese Einstellung können wir im Optionen-Dialog von Access auf der Registerseite **Allgemein** im Bereich **Datenbanken erstellen** bearbeiten. Die Option heißt **Standarddatenbankordner** und wir können diese sowohl durch direkte Eingabe oder auch durch Auswahl eines neuen Verzeichnisses über den **Durchsuchen...**-Button ändern (siehe Bild 1).

Hier sehen wir bereits eine Funktion dieses Ordners: Er wird standardmäßig beim Anlegen neuer Datenbankdateien als Zielverzeichnis angegeben (siehe Bild 2).

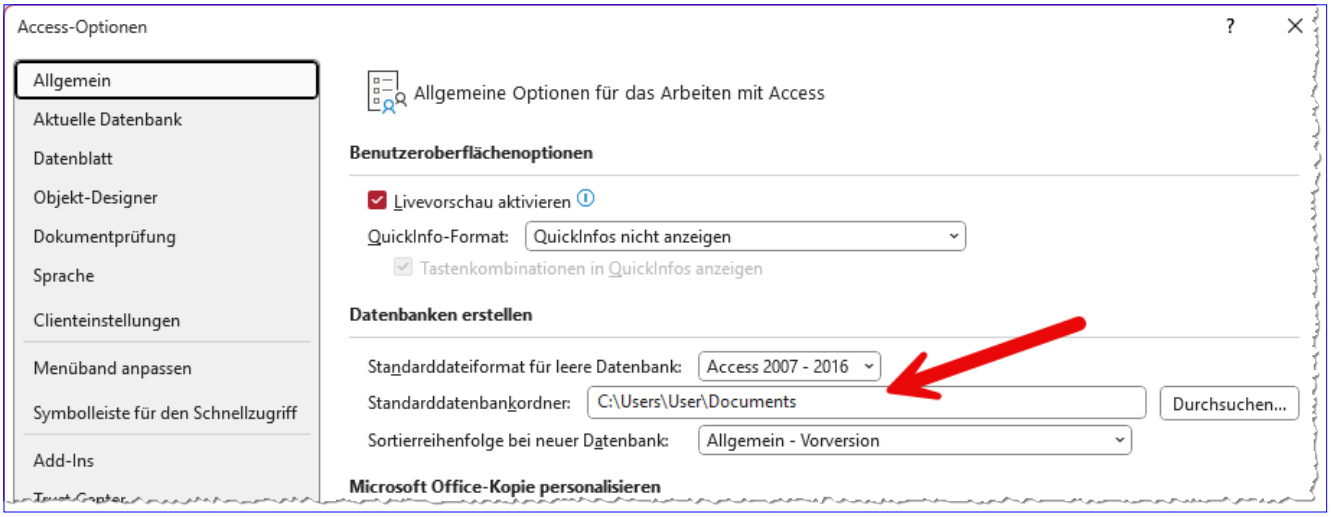

Bild 1: Der Standarddatenbankordner in den Access-Optionen

Und natürlich erscheint dieser Ordner, wenn wir Objekte aus einer anderen Access-Datenbank über den Ribbonbefehl **Externe Daten|Neue Datenquelle|Aus Datenbank|Access** importieren oder verknüpfen wollen (siehe Bild 3).

Es gibt noch andere Dialoge und Orte, an denen der Ordner **c:\Users\<Benutzername>\Documents** erscheint, aber nur in den oben genannten wirkt es sich aus, wenn wir den Standarddatenbankordner anpassen!

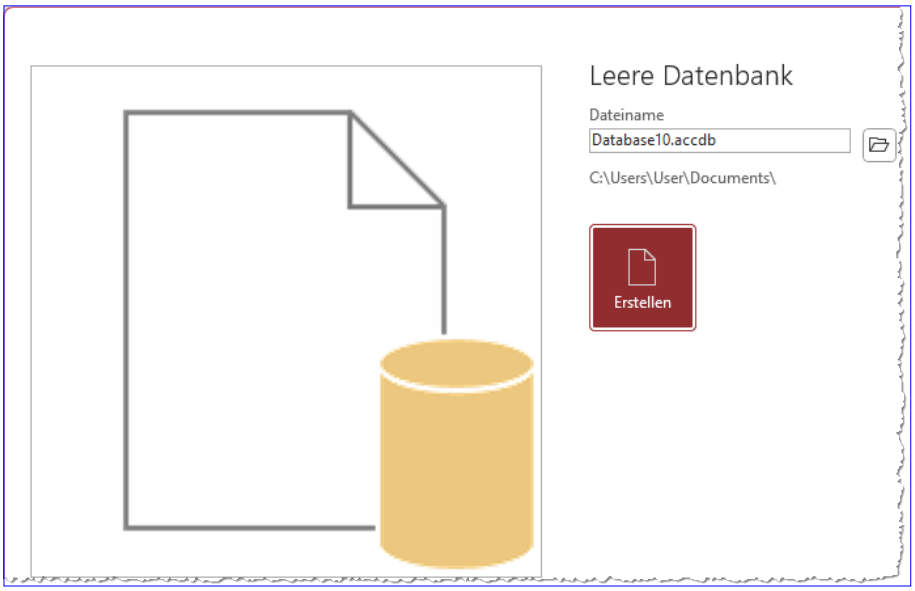

Bild 2: Ein Einsatzzweck des Standarddatenbankordners: Das Anlegen neuer Datenbanken

Zu guter Letzt noch eine Stelle, wo dieser Ordner verwendet wird: Nämlich als Ergebnis der VBA-Funktion **CurDir**.

Manuelles Ändern des Standarddatenbankordners

Wir können diesen Ordner auf verschiedene Arten ändern:

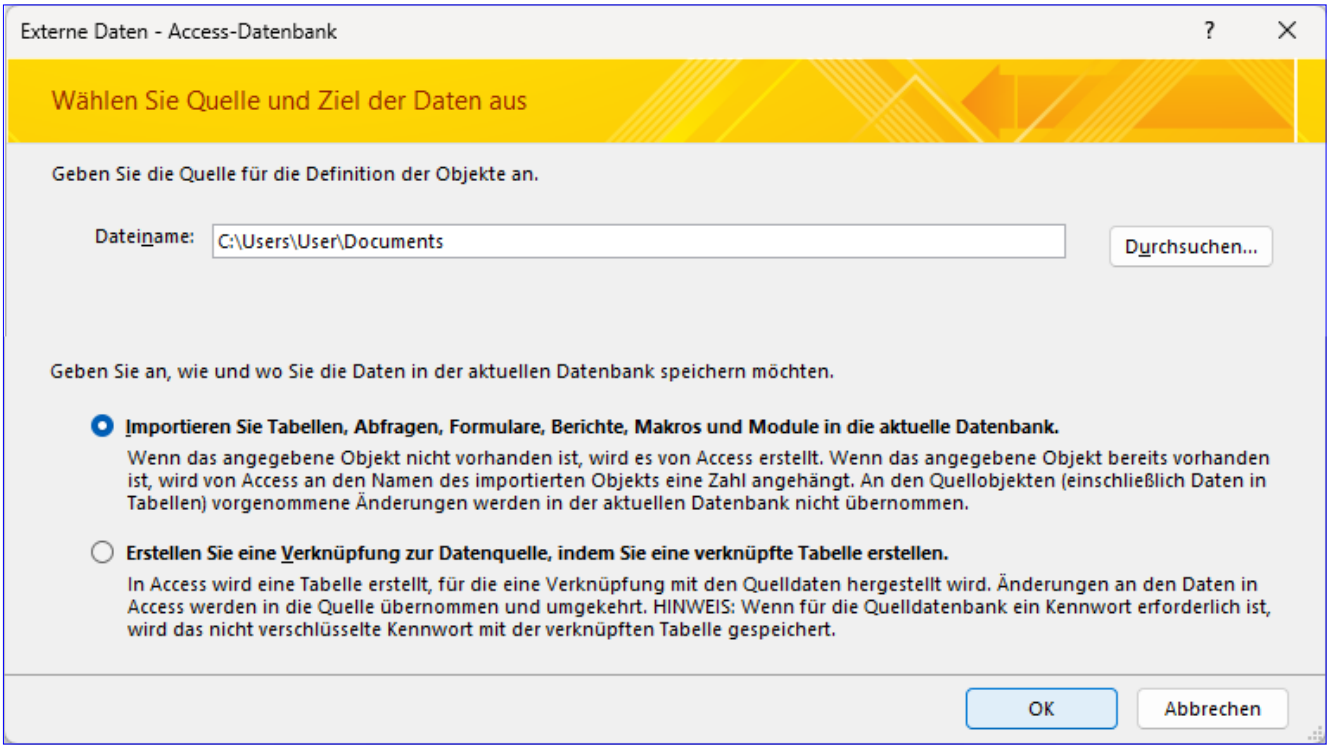

Bild 3: Auch im Dialog **Externe Daten - Access-Datenbank** erscheint der Standarddatenbankordner.

- Die erste Möglichkeit ist der bereits oben erwähnte Optionen-Dialog von Access. Diese Änderung ist dauerhaft und wirkt sich erst nach dem Schließen und erneutem Öffnen von Access aus.

- Die zweite Möglichkeit ist die VBA-Prozedur **ChDir**. Diese wirkt sich jedoch nur auf die Funktion **CurDir** aus und wird beim nächsten Start von Access wieder zurückgesetzt.

- Die dritte Möglichkeit ist eine weitere VBA-Methode namens **SetOption**, die wir gleich vorstellen.

Standarddatenbankordner per SetOption einstellen

Die oben genannte dritte Variante ist die Verwendung der Anweisung **SetOption**. Damit können wir den Standarddatenbankordner wie folgt einstellen:

```
SetOption "Default Database Directory", "c:\"
```

Zum Auslesen des aktuell eingestellten Standarddatenbankordners verwenden wir **GetOption**:

```
Debug.Print GetOption("Default Database Directory")
```

Allerdings wirkt sich auch diese Einstellung nicht unmittelbar auf den Importdialog für Access-Objekte aus, sondern erst nach dem Schließen und erneutem Start von Access.

Also hilft uns dieser Befehl an dieser Stelle auch nicht weiter – auch wenn das die optimale Lösung gewesen wäre. Wir hätten dann einfach den Befehl zum Öffnen des Importdialogs für Access-Objekte im Ribbon so erweitern können, das zusätzlich noch die Zuweisung des gewünschten Pfades erfolgt. So müssen wir uns einen anderen Weg überlegen, wie wir sicherstellen können, dass unser gewünschtes Verzeichnis je nach Anwendung angezeigt wird. Die Ziele sehen dabei wie folgt aus:

- Wenn kein explizites Verzeichnis für die Datenbank festgelegt ist, soll das Verzeichnis der aktuellen Datenbank als Standarddatenbankordner verwendet werden.

- Wir wollen aber auch individuell je Datenbank ein eigenes Verzeichnis festlegen können, das jeweils eingestellt wird, wenn wir diese Datenbank öffnen.

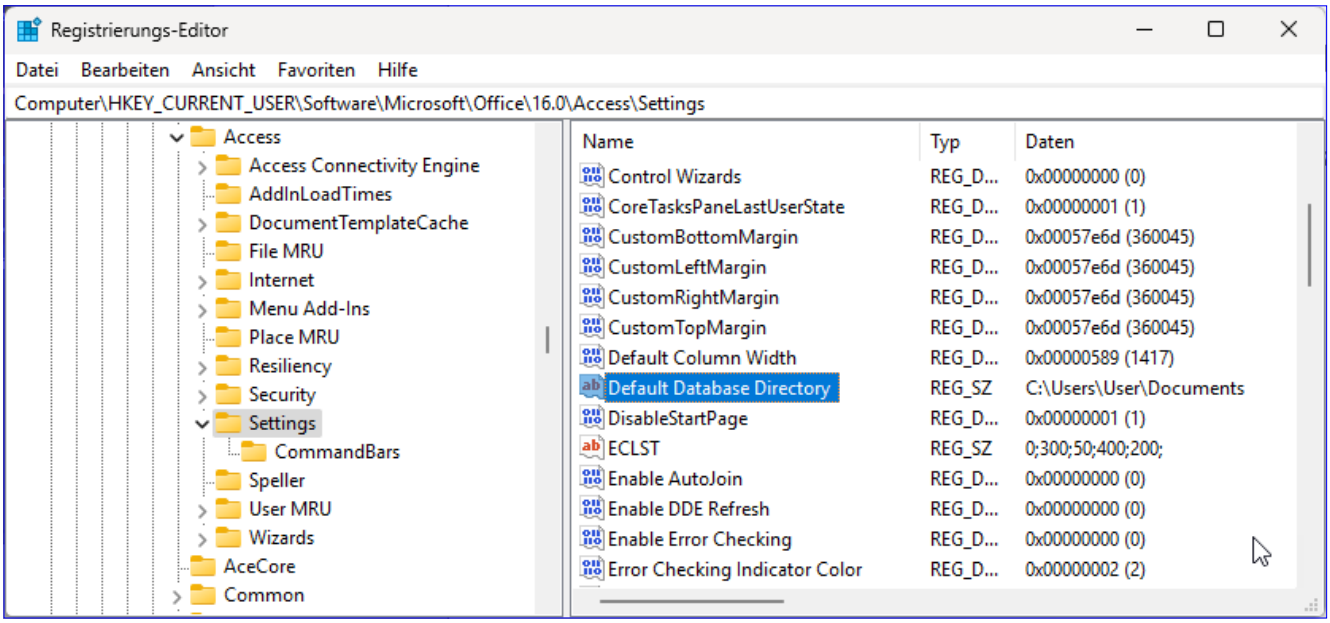

Bild 4: Der Standarddatenbankordner wird in der Registry gespeichert.

Wenn wir zum Beispiel immer wieder Tabellen aus Datenbanken des gleichen Verzeichnisses importieren, können wir dieses Verzeichnis einmalig einstellen.

Aber dann wird dieses Verzeichnis auch für jede andere Datenbank eingestellt. Sollten wir also eine zweite Datenbank verwenden, wo wir auch Tabellen aus einer anderen Datenbank importieren wollen, müssen wir spätestens hier wieder umstellen beziehungsweise immer wieder neu zum gewünschten Ordner navigieren.

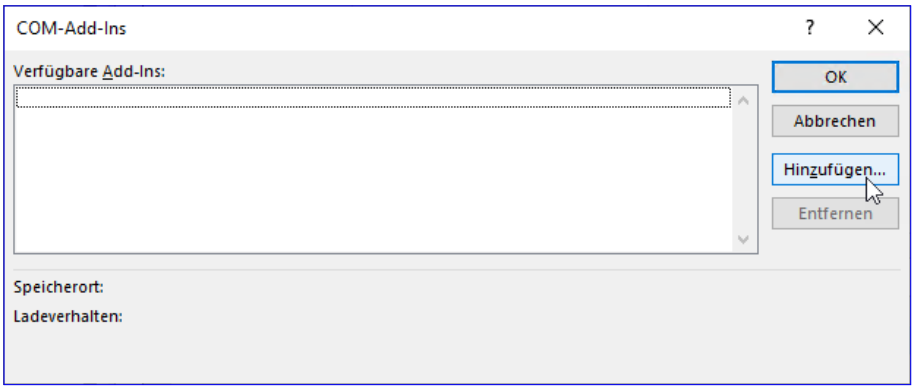

Bild 5: Hinzufügen eines COM-Add-Ins

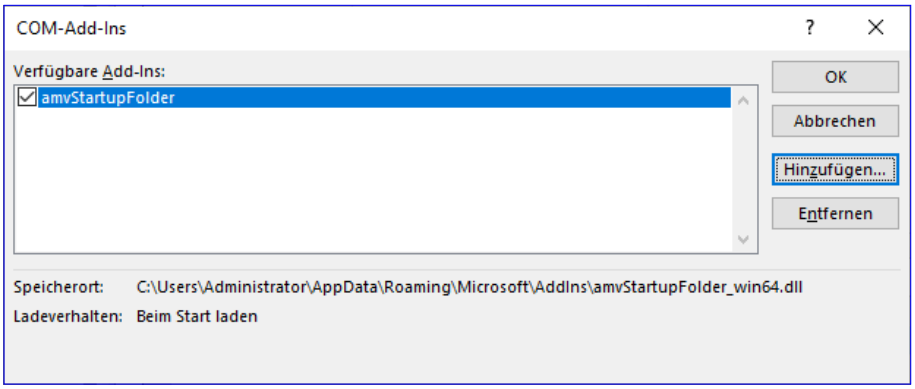

Bild 6: Das neue COM-Add-In im COM-Add-Ins-Dialog

Wo wird der Standarddatenbankordner gespeichert?

Wenn wir die Einstellung **Standarddatenbankordner** über die Access-Optionen ändern, wird diese Änderung in der Registry gespeichert. Den genauen Ort zeigt Bild 4.

Standardverzeichnis für andere Dialoge

Die anderen Dialoge, zum Beispiel zum Einlesen von Textdateien oder Excel-Dateien, berücksichtigen die Einstellung des Standarddatenbankordner übrigens gar nicht. Hier sehen wir auch aktuell keine Möglichkeit, in irgendeiner Weise Einfluss zu nehmen.

Anpassen des Datenbankordners bei Start der Anwendung

Aber wie wollen wir dafür sorgen, dass entweder das aktuelle Datenbankverzeichnis (mit **CurrentProject.Path**) oder ein speziell für die Datenbank festgelegtes Verzeichnis bereits eingestellt ist, wenn wir die entsprechende Datenbank öffnen? Das erledigen wir mit einem COM-Add-In namens **amvStartupfolderr**.

Installation des COM-Add-Ins

Sie finden im Download die entsprechende DLL namens **amvStartupfolder_32.dll** beziehungsweise **amvStartupfolder_64.dll**. Diese installieren Sie, indem Sie die folgenden Schritte durchführen:

- Kopieren Sie die DLL in ein geeignetes Verzeichnis, beispielsweise in das Add-In-Verzeichnis. Wie Sie dieses am einfachsten ermitteln, erfahren Sie in diesem Beitrag: **Add-In-Verzeichnis mit Wizhook ermitteln** (**www.access-im-unternehmen.de/1553**)
- Dann klicken Sie auf **DateiOptionen**, um die Access-Optionen zu öffnen.
- Wechseln Sie hier zum Bereich **Add-Ins** und klicken dann auf die Schaltfläche **Los...** neben dem Kombinationsfeld **Verwalten**.

- Im nun erscheinenden Dialog **COM-Add-Ins** klicken Sie auf die Schaltfläche **Hinzufügen...** und wählen die Datei **amvStartupFolder_win32.dll** oder **amvStartup-Folder_win64.dll** (je nach Bitness der Office-Version) aus dem Add-In-Verzeichnis aus (siehe Bild 5).

- Danach wird das neue COM-Add-In in der Liste angezeigt (siehe Bild 6).

Damit ist das COM-Add-In installiert und einsatzbereit.

Einsatz von amvStartupfolder

Das COM-Add-In scheint erst einmal unsichtbar zu sein. Wie können es aber schon testen, indem wir einmal eine neue Access-Anwendung öffnen.

Wichtig: Diese Access-Anwendung muss per Doppelklick auf den Namen der **.accdb**-Datei geöffnet werden! Wenn Sie erst Access starten und dann die Datenbank laden, funktioniert das COM-Add-In nicht. Woran das liegt, erläutern wir später. Das COM-Add-In ist nun noch nicht aktiviert, denn es wird erst beim nächsten Start von Access geladen.

Wenn wir dann zu den Access-Optionen wechseln, sehen wir in der Eigenschaft **Standarddatenbankordner** bereits genau das Verzeichnis, in dem sich die aktuelle Datenbank befindet.

Und auch wenn wir den Import-Assistent für Access starten, zeigt dieser direkt dieses Verzeichnis an.

Damit funktioniert bereits die erste Anforderung, dass das aktuelle Datenbankverzeichnis als **Standarddatenbankordner** verwendet wird.

Standarddatenbankordner individuell einstellen

Damit kommen wir zur zweiten Anforderung. Wir hatten formuliert, dass wir zu jeder Datenbankdatei einen individuellen Standarddatenbankordner definieren können

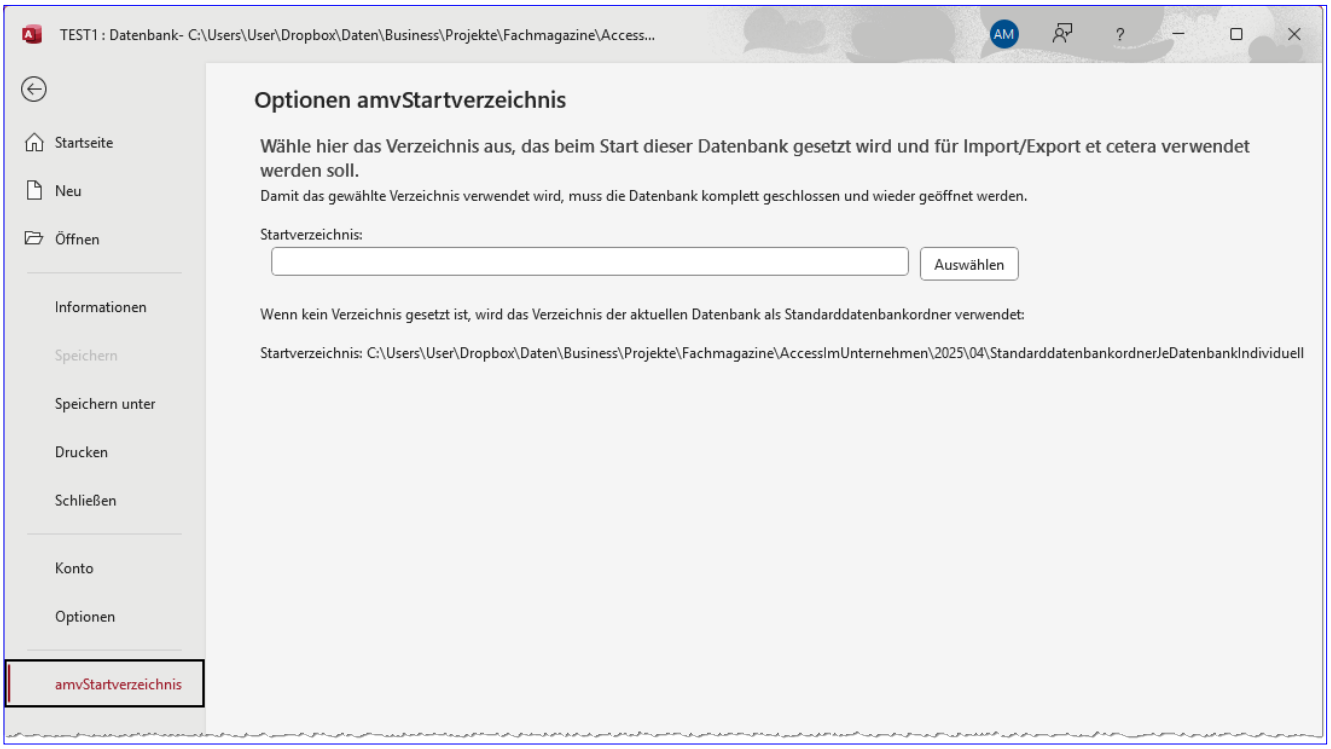

Bild 7: Einstellen des gewünschten Standarddatenbankordners

wollen. Aber wo tun wir das? Dazu öffnen wir den Ribbon-Reiter **Datei** und klicken auf den nun sichtbaren Eintrag **amvStartverzeichnis**.

Danach erscheint der entsprechende Bereich, in dem wir mit dem Button **Auswählen** einen Dialog zum Auswählen des gewünschten Verzeichnisses finden (siehe Bild 7).

Hier finden wir auch gleich den Hinweis, dass bei leerem Feld der aktuelle Ordner der Datenbank als Standarddatenbankordner verwendet wird.

Nachdem wir das Verzeichnis angegeben haben, können wir Access mit dieser Datenbank neu starten. Und siehe da: Wir finden in den Access-Optionen nun den neuen Ordner vor und können diesen auch direkt im Import-Assistenten für Access nutzen.

Das beweist aber noch lange nichts. Wir müssen zunächst noch eine zweite Datenbank mit einem individuellen Standarddatenbankordner versehen. Wenn wir diese öffnen, finden wir wieder eine leere Eigenschaft **Startverzeichnis** im Bereich **amvStartverzeichnis** vor. Das zeigt schon einmal, dass noch kein individueller Standarddatenbankordner festgelegt wurde. Fügen wir also einen passenden Ordner hinzu, der sich von dem der vorherigen Datenbank unterscheidet.

Öffnen wir nun nacheinander diese beiden Datenbanken, finden wir jeweils den dafür festgelegten Standarddatenbankordner vor – ohne selbst etwas dazu beigetragen zu haben.

Beschreibung des COM-Add-Ins amvStartupfolder

Damit kommen wir zum eigentlichen Know-how dieses Beitrags: Die Funktionsweise des COM-Add-Ins. Dieses haben wir, wie immer, mit twinBASIC erstellt. Den Download der aktuellen Version finden Sie hier:

```
https://github.com/twinbasic/twinbasic/releases
```

Das Projekt haben wir dem Download zu diesem Artikel unter dem Namen **amvStartverzeichnis.twinproj** beigelegt. Dieses können Sie nach dem Start von twinBASIC öffnen.

Das Hauptmodul enthält wie üblich die Implementierung der Schnittstellen **IDTExtensibility2** und **IRibbonExtensibility**, die wir in der Klasse deklarieren. Außerdem deklarieren wir hier noch eine Variable für die Ribbon-Anpassung:

```
[ClassId("DC8E7154-FAC8-43B7-88E5-A93527539223")]
Class amvStartupfolder
    Implements IDTExtensibility2
    [WithDispatchForwarding]
    Implements IRibbonExtensibility

    Private objRibbon As IRibbonUI

    ... weitere Elemente ...

End Class
```

An der Stelle **... weitere Elemente ...** stehen die nachfolgend beschriebenen Elemente.

Implementierung der Schnittstelle IDTExtensibility2

Als Erstes implementieren wir die Schnittstelle **IDTExtensibility2** wie in Listing 1. Diese enthält verschiedene Ereignisprozeduren, die von Access zu den entsprechenden Zeitpunkten ausgeführt werden. Interessant für uns ist zunächst das Ereignis **OnConnection**. Hier nehmen wir mit dem Parameter **Application** einen Verweis auf die aufrufende Instanz entgegen, also die Access-Anwendung.

Diesen Verweis speichern wir in der Variablen **objAccess**, die wir im Modul **mdlGlobal** hinterlegt haben – neben einer weiteren namens **strStartupfolder**:

```
Module mdlGlobal
```

Verpackungsprozesse mit Access: Formulare

Ein Kunde hatte neulich die Herausforderung, dass er Produkte bestehend aus einzelnen Bauteilen ausliefern möchte und dabei erfassen muss, welche Bauteile in welcher Menge in den verschiedenen Versandkartons landen. Der Verpackungsvorgang im Lager erfolgt individuell und in einer Weise, dass die Mitarbeiter die einzelnen Bauteile so in Kartons füllen, dass die Kartons optimal genutzt werden. Während des Verpackens sollen sie aufzeichnen, welcher Karton welche Bauteile enthält, damit diese beim Empfänger so ausgepackt werden können, wie die Bauteile benötigt werden. In diesem Beitrag beschreiben wir zunächst, wie das Datenmodell für dieses Vorhaben aussieht und erstellen basierend darauf die notwendigen Formulare. Schließlich geben wir auch noch die Stücklisten für die verschiedenen Kartons per Bericht aus.

Bestellungen eingeben

Die Grundlage für diese Lösung sind Bestellungen. Diese bearbeiten wir in dem einfachen Formular aus Bild 1. Hier können wir im Hauptformular den Kunden auswählen und das Bestelldatum eingeben.

Außerdem wählen wir im Unterformular die Produkte aus, die dieser Bestellung zugeordnet werden sollen und fügen die gewünschte Menge hinzu.

Um Preise und andere Informationen kümmern wir uns in diesem Fall aus Gründen der Übersichtlichkeit nicht.

Wenn eine Bestellung eingegeben wurde, können wir mit dem Button **Verpacken** zum nächsten Formular wechseln, dass die eigentliche Funktion dieses Beitrags abbildet.

Vorab noch die Information, dass jedes Produkt aus verschiedenen Bauteilen in unterschiedlicher Menge besteht, die im Hintergrund in Tabellen gespeichert sind.

Auf die genauen Zusammenhänge im Datenmodell gehen wir später ein.

Erfassung der verpackten Bauteile

Das Formular **frmVerpackung** ist das eigentliche Arbeitstier (siehe Bild 2). Hier sehen wir verschiedene Bereiche:

- Ganz oben finden wir nochmal die Daten zu der jeweiligen Bestellung.
- Darunter sehen wir die einzelnen Positionen der Bestellung, also die Produkte.
- Das Listenfeld links unten zeigt die Bauteile, die zu den Produkten dieser Bestellung gehören. Wenn das

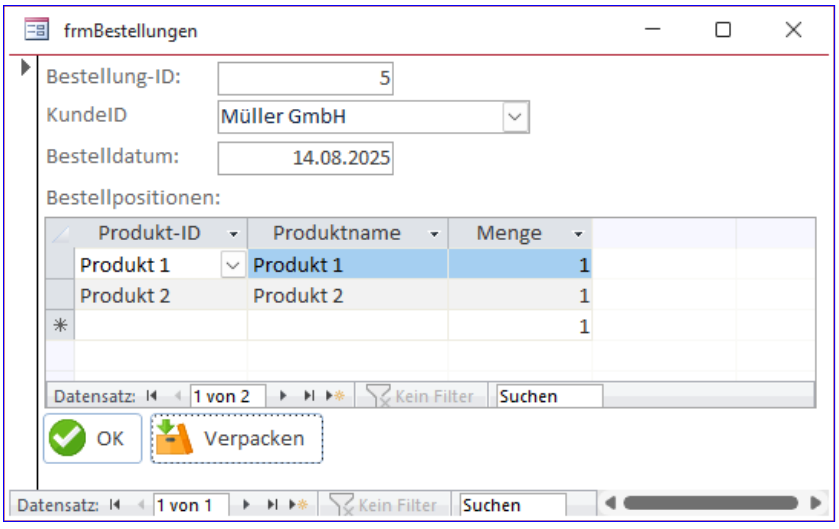

Bild 1: Einfaches Bestellformular

Formular geöffnet wird, zeigt diese Liste zunächst die Bauteile für alle Produkte der Bestellung an. Wenn wir jedoch auf eines der Produkte im Listenfeld darüber klicken, werden nur noch die Bauteile zu diesem Produkt angezeigt. Außerdem sehen wir dort die gesamte Menge des jeweiligen Bauteils, die Anzahl der gepackten Exemplare und die Restmenge, also die nicht gepackten Bauteile.

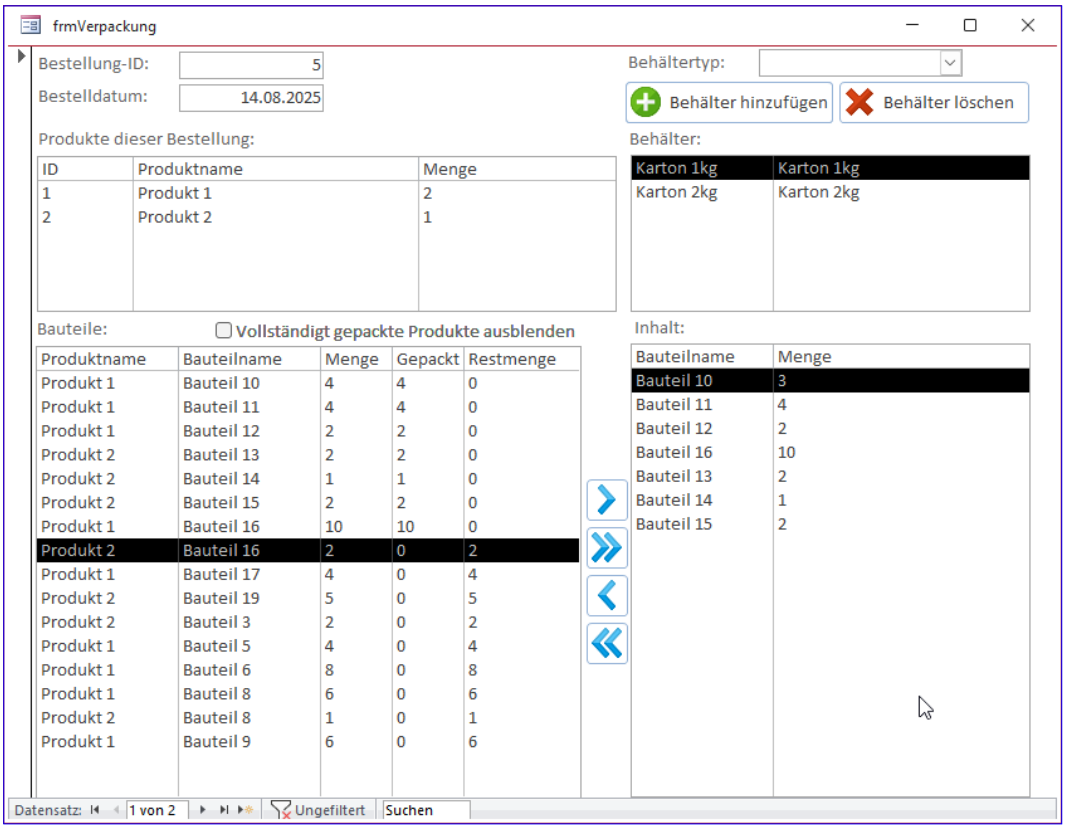

Bild 2: Formular zum Protokollieren der Verpackung

- Rechts finden wir Steuerelemente, mit denen wir einen Behältertyp auswählen und diesen zur aktuellen Bestellung hinzufügen können. Wir wählen einen Behälter aus und klicken auf **Behälter hinzufügen**, um diesen der Bestellung zuzuordnen. Dabei können wir dem Behälter noch einen individuellen Namen zuweisen (siehe Bild 3). Der angelegte Behälter landet schließlich im Listenfeld der Behälter zu dieser Bestellung. Außerdem können wir mit der Schaltfläche **Behälter löschen** auch den aktuell markierten Behälter löschen.

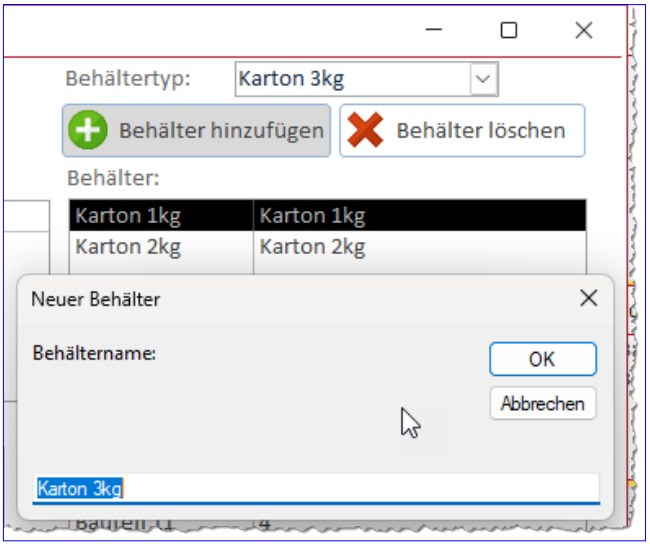

Bild 3: Anlegen von Behältern

- Unten rechts sehen wir die dem jeweiligen Behälter zugeordneten Bauteile. Die Zuordnung erfolgt über die Pfeil-Schaltflächen. Mit dem einfachen Pfeil nach rechts ordnen wir ein Stück des aktuell markierten Bauteils zum aktiven Behälter zu. Mit dem doppelten Pfeil weisen wir dem Behälter alle verbleibenden Bauteile zu.

- Mit dem einfachen und dem doppelten Pfeil nachh links können wir ein Bauteil oder alle Bauteile aus der Liste **Inhalt** entfernen.

- Wenn wir Veränderungen am Inhalt eines Behälters vornehmen, wird die dortige Menge angepasst. Außerdem wird auch die Liste **Bauteile** aktualisiert. Wenn wir beispielsweise ein Bauteil zu den gepackten Bauteilen hinzufügen, wird der Wert in der Spalte **Gepackt** um **1** erhöht und die **Restmenge** um **1** vermindert.

Auf diese Weise sieht der Mitarbeiter genau, wie viele Exemplare eines Bauteils noch gepackt werden müssen.

Nach dem Packen eines oder mehrerer Bauteile in einen Behälter braucht der Mitarbeiter nur das entsprechende Bauteil in der Liste **Bauteile** zu markieren und entweder entsprechend der Menge der gepackten Bauteile mehrfach auf die Schaltfläche mit den einfachen Pfeil nach rechts zu klicken, oder er packt alle Exemplare eines Bauteils in einen Behälter und klickt direkt auf die Schaltfläche mit den zwei Pfeilen.

Auch wenn er bereits ein einzelnes Bauteil von mehreren mit einem Klick auf die Schaltfläche mit dem einfachen Pfeil zu einem Behälter hinzugefügt hat, kann er alle verbleibenden mit einem Klick auf die Schaltfläche mit dem doppelten Pfeil dem aktuell gewählten Behälter zuordnen.

Datenmodell der Anwendung

Bevor wir in die Programmierung des Formulars einsteigen, wollen wir uns kurz das Datenmodell ansehen (siehe Bild 4). Die Tabellen, die mit dem Formular **frmBestellungen** verwaltet werden, heißen **tblKunden**, **tblBestellungen** und **tblBestellungProdukte**.

Daneben gibt es noch weitere Stammdaten. Die Tabelle **tblBauteile** enthält alle Bauteile, die über die Tabelle **tblProdukteBauteile** dem jeweiligen Produkt zugewiesen werden. Das Feld **Menge** dieser Tabelle gibt an, wie viele Exemplare eines Bauteils zu einem Produkt gehören.

Genau so, wie wir den Produktnamen bei Auswählen einer Bestellung in die Tabelle **tblBestellungProdukte** schreiben, um diese unabhängig von den sich gegebenenfalls ändernden Stammdaten zu machen, wollen wir auch die Bauteile zu jeder Bestellposition in einer eigenen Tabelle historisieren.

Dazu verwenden wir die Tabelle **tblBestellungProdukteBauteile**. Sie ist einerseits mit der Tabelle **tblBestellungProdukte** verknüpft und andererseits mit der Tabelle **tblBauteile**. Damit halten wir fest, woher die Bauteile

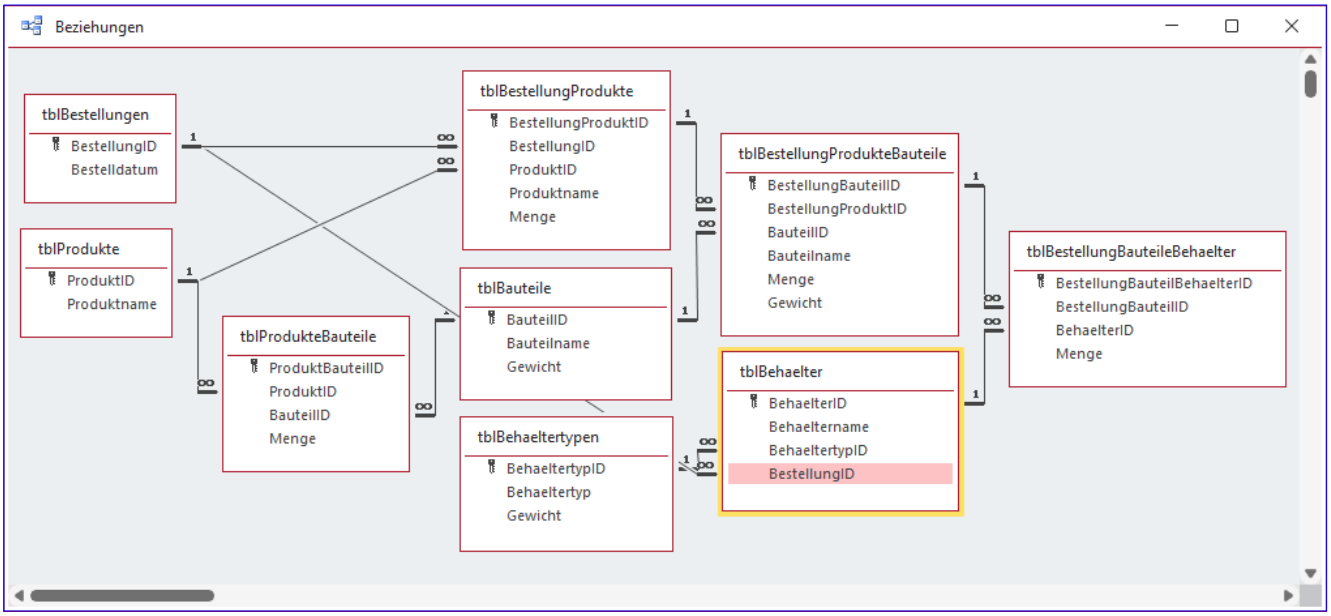

Bild 4: Datenmodell der Anwendung

ursprünglich stammen. In den Feldern **Bauteilname**, **Menge** und **Gewicht** speichern wir die Eigenschaften zum Zeitpunkt dieser Bestellung. Die Menge wird berechnet aus dem Produkt des Feldes **Menge** der Tabelle **tblBestellungProdukte** und dem gleichnamigen Feld der Tabelle **tblProdukteBauteile**.

Damit kommen wir zu einem weiteren Teil des Datenmodells, der die Zuordnung der Bauteile einer Bestellung zu den dafür angelegten Behältern speichert. Die Behälter werden in der Tabelle **tblBehaelter** angelegt. Hier können wir über das Feld **BehaeltertypID** den Typ des Behälters aus der Tabelle **tblBehaeltertypen** auswählen. Jeder Behälter wird außerdem noch der jeweiligen Bestellung zugeordnet.

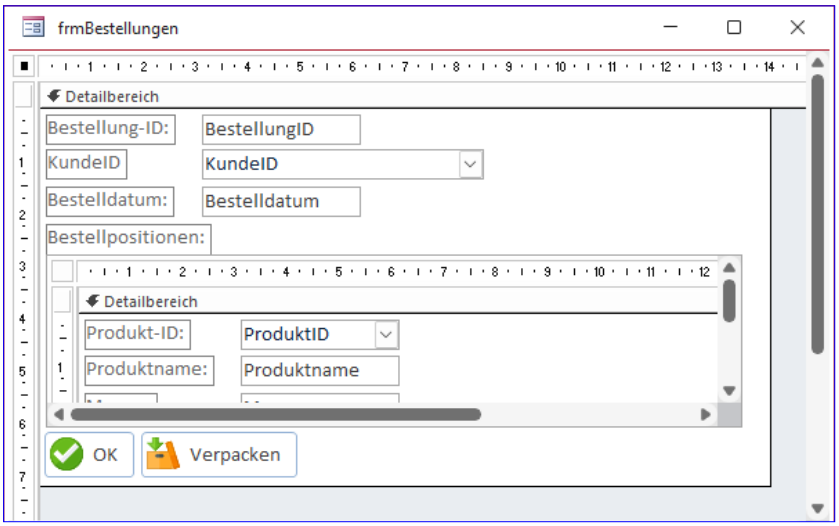

Bild 5: Entwurfsansicht des Formulars **frmBestellungen**

Die Tabelle **tblBestellungBauteileBehaelter** schließlich speichert, wie viele Exemplare eines Bauteils eines Produkts in einem der Behälter gespeichert sind.

Löschweitergaben im Datenmodell

Wir haben bei den meisten mit referenzieller Integrität definierten Beziehungen auch die Löschweitergabe aktiviert. So werden beispielsweise beim Löschen von Bestellungen auch automatisch die damit verknüpften Produkte, Bauteile et cetera gelöscht. Auch beim Löschen von Behältern sollen die Zuordnungen von Bauteilen gelöscht werden.

Programmierung des Formulars frmBestellungen

Wenn der Benutzer im Formular **frmBestellungen** ein Produkt im Unterformular auswählt, wird der Name des Produkts in das Feld **Produktname** der Tabelle **tblBestellungProdukte** eingetragen.

Damit stellen wir sicher, dass bei späteren Änderungen des Produktnamens der zum Zeitpunkt der Bestellung gültige Produktname in den Bestelldaten beibehalten wird. Das Formular sehen wir in Bild 5 in der Entwurfsansicht.

Wenn der Benutzer das Produkt ausgewählt hat, soll der Produktname aus der Tabelle **tblProdukte** in die Tabelle **tblBestellungProdukte** eingetragen werden. Das erledigen wir in der **Vor Aktualisierung**-Ereignisprozedur:

```
Private Sub ProduktID_BeforeUpdate(Cancel As Integer)
    Me.Produktname = DLookup("Produktname", _
        "tblProdukte", "ProduktID = " & Me.ProduktID)
End Sub
```

Nachdem der Benutzer die Menge für die aktuelle Position eingetragen hat und den Datensatz gespeichert hat, tragen wir die Daten in die Tabelle **tblBestellungProdukteBauteile** ein.

Das erledigen wir mit der Prozedur, die durch das Ereignis **Nach Aktualisierung** des Unterformulars ausgelöst wird (siehe Listing 1). Die Prozedur erstellt zwei Recordsets. **rstBauteileQuelle** enthält eine Abfrage, die alle Bauteile enthält, die zu dem im Unterformular hinzugefügten Produkt gehören. Der zweite enthält die Tabelle **tblBestellungProdukteBauteile**, also die Tabelle, in die wir die Bauteile für die aktuelle Bestellposition eintragen wollen.

In der folgenden **Do While**-Schleife durchlaufen wir alle Datensätze des ersten Recordsets und tragen die benötig-

```
Private Sub Form_AfterUpdate()
    Dim db As DAO.Database
    Dim rstBauteileQuelle As DAO.Recordset
    Dim rstBauteileZiel As DAO.Recordset

    Set db = CurrentDb
    Set rstBauteileQuelle = db.OpenRecordset("SELECT tblProdukteBauteile.ProduktID, tblProdukteBauteile.Menge, " _
        & "tblBauteile.BauteilID, tblBauteile.Bauteilname, tblBauteile.Gewicht FROM tblBauteile " _
        & "INNER JOIN tblProdukteBauteile ON tblBauteile.BauteilID = tblProdukteBauteile.BauteilID WHERE ProduktID = " _
        & Me.ProduktID, dbOpenDynaset)
    Set rstBauteileZiel = db.OpenRecordset("SELECT * FROM tblBestellungProdukteBauteile WHERE 1=2", dbOpenDynaset)

    Do While Not rstBauteileQuelle.EOF
        With rstBauteileZiel
            .AddNew
            !BestellungProduktID = Me.BestellungProduktID
            !BauteilID = rstBauteileQuelle!BauteilID
            !Bauteilname = rstBauteileQuelle!Bauteilname
            !Menge = rstBauteileQuelle!Menge * Me.Menge
            !Gewicht = rstBauteileQuelle!Gewicht
            .Update
        End With
        rstBauteileQuelle.MoveNext
    Loop
End Sub
```

Listing 1: Eintragen der Produkt- und Bauteildaten für die aktuelle Bestellung

ten Werte aus den Tabellen **tblBauteile** und **tblProdukteBauteile** in die Tabelle **tblBestellungProdukteBauteile** ein. Dabei werden alle Daten übernommen – mit Ausnahme des Feldes **Menge**. Hier multiplizieren wir die Menge der Produktposition mit der Menge der Bauteile für dieses Produkt.

Programmierung des Formulars frmVerpackung

Da wir nun in der Tabelle **tblBestellungProdukteBauteile** alle Informationen über die zu verpackenden Bauteile haben, können wir uns dem Formular zum Verwalten des Packvorgangs zuwenden. Dieses heißt **frmVerpackung** und sieht in der Entwurfsansicht wie in Bild 6 aus.

Das Formular selbst verwendet die Tabelle **tblBestellungen** als Datensatzquelle. Es zeigt die beiden Felder **BestellungID** und **Bestelldatum** dieser Tabelle an.

Alle übrigen Daten werden überwiegend in Listenfeldern angezeigt. Einzige Ausnahme ist das Kombinationsfeld zur Auswahl des Behältertyps für das Anlegen eines neuen Behälters zur aktuellen Bestellung. Schauen wir uns also diese Listenfelder an.

Listenfeld zur Anzeige der Produkte einer Bestellung

Das erste Listenfeld heißt **lstProdukte** und soll alle Produkte einer Bestellung inklusive Menge anzeigen. Damit das Listenfeld beim Anzeigen eines Datensatzes immer die zur aktuellen Bestellung gehörenden Produkte anzeigt, weisen wir die Datensatzherkunft im Ereignis **Beim Anzeigen** des Formulars jeweils neu zu.

Dabei verwenden wir einen SQL-Ausdruck, der die benötigten Daten der Tabelle **tblBestellungProdukte** liefert

und diese nach der aktuell im Formular angezeigten Bestellung filtert. Außerdem erledigen wir dort noch weitere Schritte, indem wir Prozeduren zum Aktualisieren der übrigen Listenfelder und der Schaltflächen aufrufen (siehe Listing 2).

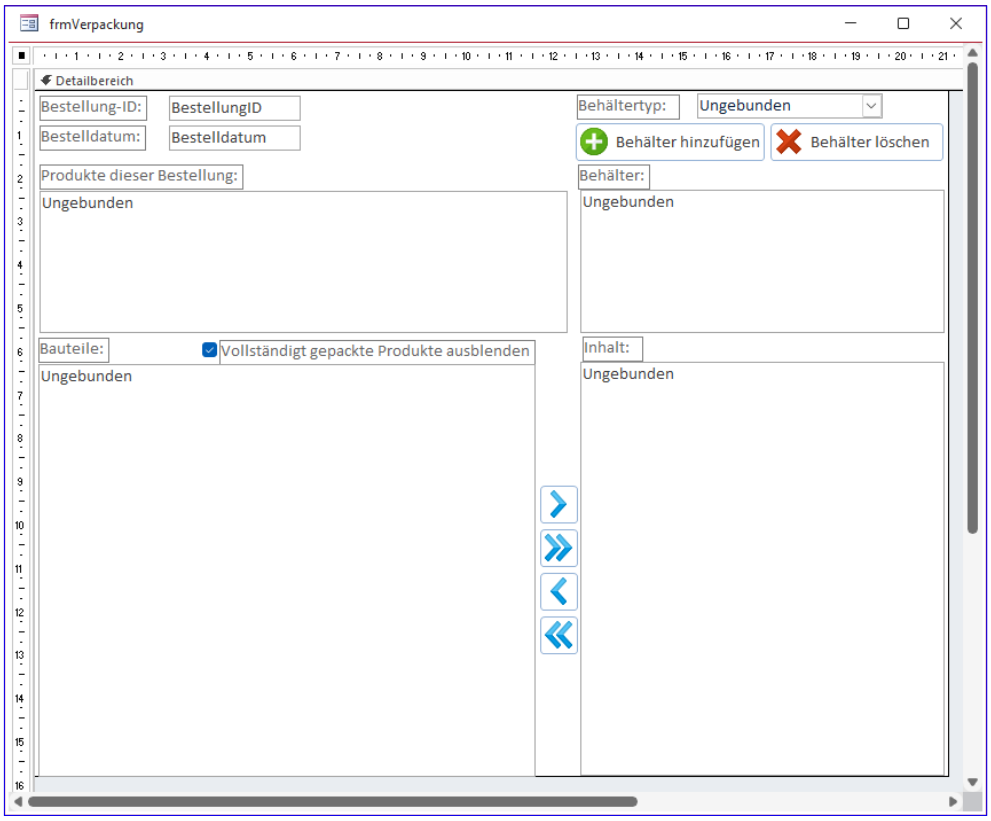

Bild 6: Entwurf des Formulars **frmVerpackung**

Wozu aber wollen wir dieses Listenfeld überhaupt nutzen? Es soll dazu dienen, die Anzeige der Bauteile im Listenfeld **lstBauteile** nach dem jeweils ausgewählten Produkt zu filtern.

Wenn das Formular geöffnet wird, soll dieses Listenfeld zunächst alle Bauteile anzeigen, die zu den Produkten der Bestellung gehören. Vielleicht sollen die Bauteile zu einem Produkt aber jeweils in eigenen Behältern gepackt werden. Dann kann der Mitarbeiter auf das jeweilige Produkt klicken und erst einmal nur diese Bauteile packen, bevor er mit dem folgenden Produkt fortfährt.

Also haben wir auch zwei Prozeduren, die das Listenfeld **lstBauteile** füllen – eine trägt alle Bauteile der aktuellen Bestellung ein, die andere ermittelt alle Bauteile zum aktuell markierten Produkt.

Wenn wir normalerweise einen Eintrag in einem Listenfeld anklicken, können wir diesen nicht wieder abwählen – außer, wir wählen einen anderen Eintrag aus. Wir wollen aber auch wieder alle Bauteile der aktuellen Bestellung anzeigen können, obwohl wir vielleicht gerade ein Produkt gewählt haben, um nur dessen Bauteile anzuzeigen. Also

```
Private Sub Form_Current()
    Me.lstProdukte.RowSource = "SELECT BestellungProduktID, ProduktID AS ID, Produktname, Menge " _
        & "FROM tblBestellungProdukte WHERE BestellungID = " & Me.BestellungID
    Call lstBauteileAktualisieren
    Call lstBehaelterAktualisieren
    Call lstInhaltAktualisieren
    Call SchaltflaechenAktualisieren
End Sub
```

Listing 2: Füllen der Listenfelder beim Anzeigen eines Datensatzes

müssen wir uns einen kleinen Trick überlegen, um den aktuell markierten Eintrag im Listenfeld **lstProdukte** wieder abzuwählen.

Dazu wollen wir beim Auswählen eines Produkts prüfen, ob dieses bereits zuvor selektiert war. Dies erfordert eine Variable außerhalb der Prozedur, um den zuletzt selektierten Eintrag zu speichern:

```
Private varProduktSelektiertVorher As Variant
```

Für das Ereignis **Nach Aktualisierung** des Listenfeldes **lstProdukte** hinterlegen wir nun die folgende Ereignisprozedur:

```
Private Sub lstProdukte_AfterUpdate()
    If Me.lstProdukte = Nz(varProduktSelektiertVorher, 0) Then
        Call lstBauteileAktualisierenNachBestellungID
        varProduktSelektiertVorher = Null
        Me.lstProdukte = Null
    Else
        Call lstBauteileAktualisierenNachProduktID
        varProduktSelektiertVorher = Me.lstProdukte
    End If
End Sub
```

Diese prüft, ob der aktuell angeklickte Eintrag dem in **varProduktSelektiertVorher** gespeicherten Eintrag entspricht. Falls ja, will der Benutzer offensichtlich gerade den gewählten Listeneintrag abwählen. Also stellen wir das Listenfeld auf den Wert **Null** ein, um die Selektion aufzuheben und leeren auch die Variable **varProduktSelektiertVorher**. Außerdem rufen wir die Prozedur **lstBauteileAktualisierenNachBestellungID** auf, um im Listenfeld **lstBauteile** alle Bauteile anzuzeigen, die zur aktuellen Bestellung gehören.

Wenn der Benutzer ein Produkt ausgewählt hat, das zuvor nicht selektiert war, rufen wir die Prozedur **lstBauteileAktualisierenNachProduktID** auf, um im Listenfeld **lstBauteile** alle Bauteile der aktuellen Bestellung anzuzeigen. Außerdem legen wir den Wert von **varProduktSelektiertVorher** auf den aktuell markierten Eintrag fest.

Listenfeld zur Anzeige der Bauteile einer Bestellung

Das Listenfeld **lstBauteile** soll, wie zuvor beschrieben, entweder die Bauteile für die aktuelle Bestellung anzeigen

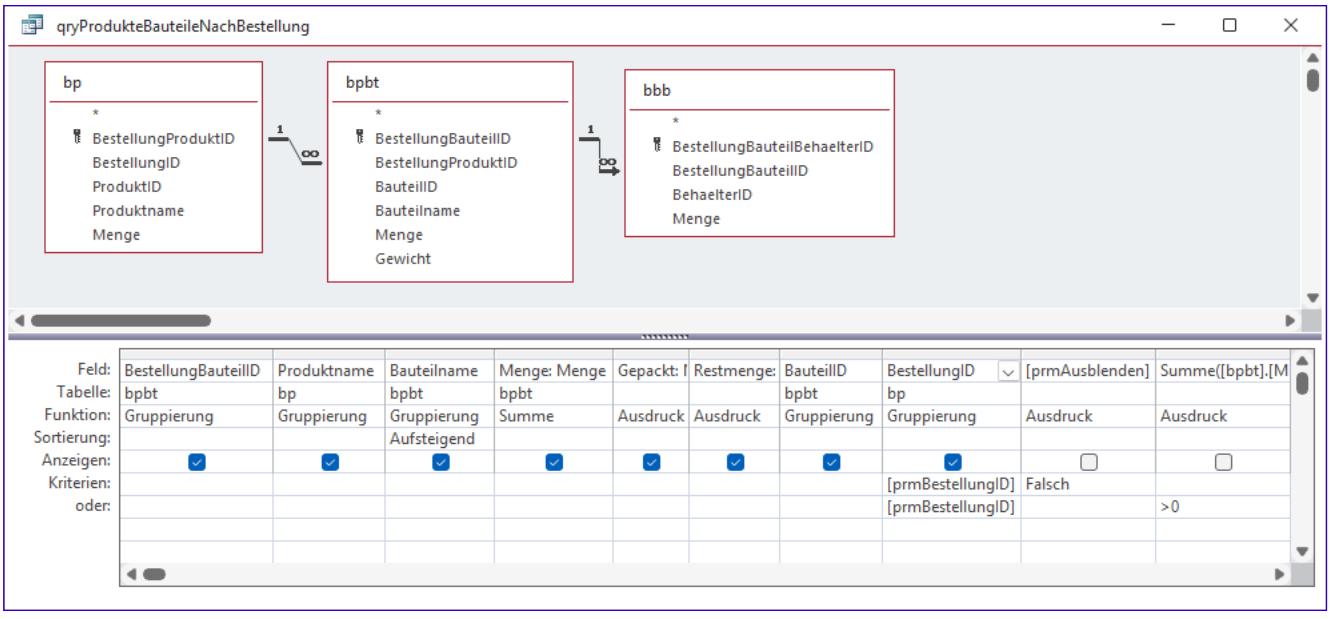

Bild 7: Entwurf der Abfrage **qryProdukteBauteileNachBestellung**

Mit Screen auf Formulare, Controls und Co. zugreifen

Wer Formulare programmiert, möchte immer wieder mal Informationen über das Formular oder die enthaltenen Steuerelemente im Direktbereich ausgeben. Die Screen-Klasse liefert die passenden Eigenschaften dazu: Damit können wir das aktuelle Formular, die aktuelle Datenbank und auch das aktuelle Steuerelement referenzieren.

Beim Entwickeln von Formularen benötigt man immer wieder Informationen über Steuerelemente, die man nur bekommt, wenn das Formular in der Formularansicht angezeigt wird. Dann kann man aber nicht einfach auf das entsprechende Element klicken und die Eigenschaften oder Werte im Eigenschaftenblatt nachlesen.

Es besteht die Möglichkeit, die Elemente mit Ausdrücken wie den folgenden über den Direktbereich des VBA-Editors zu referenzieren:

```
Debug.Print Forms!frmKunden!sfmProjekte.Form.ProjektID
```

Für diese Vorgehensweise müssen wir auch den Namen von Formularen, Unterformularen und Steuerelementen kennen. Einfacher geht es mit den Funktionen der **Screen**-Klasse. Wir geben einfach **Screen** gefolgt von einem Punkt ein und finden die Eigenschaften aus Bild 1 vor.

Mit **ActiveControl** holen wir uns einen Verweis auf das aktuelle Steuerelement und können uns beispielsweise den Wert ausgeben lassen:

```
Debug.Print ActiveControl.Value
```

Bild 1: Eigenschaften der **Screen**-Klasse

Das ist zum Beispiel praktisch bei Kombinationsfeldern, die nicht ihren eigentlichen Wert, sondern nur den angezeigten Wert liefern.

ActiveDatasheet liefert uns einen Verweis auf das aktive Datenblatt, sofern gerade ein Feld innerhalb eines Datenblatts markiert ist. Das gelingt für Tabellen, Abfragen und auch Formulare – allerdings nicht für Formulare in der Datenblattansicht, die als Unterformulare verwendet werden.

Mit **ActiveForm** erhalten wir einen Verweis auf das Formular, das aktuell den Fokus hat. Damit können wir beispielsweise den Namen ausgeben, falls wir diesen nicht mehr in der Titelleiste anzeigen:

```
Debug.Print Screen.ActiveForm.Name
```

Und auch auf den aktuell im Fokus befindlichen Bericht können wir zugreifen, und zwar mit **ActiveReport**. Dazu nutzen wir:

```
Debug.Print Screen.ActiveReport
```

Weiter unten in der IntelliSense-Liste finden wir noch einen interessanten Eintrag, nämlich **PreviousControl**. Damit holen wir einen Verweis auf das zuvor fokussierte Steuerelement. Das ist zum Beispiel praktisch, wenn man auf eine Schaltfläche klickt und in der Ereignisprozedur herausfinden möchte, welches Steuerelement zum Zeitpunkt des Klicks den Fokus hatte. Beim Klick auf die Schaltfläche liefert **Screen.ActiveControl** nämlich schon den Verweis auf die Schaltfläche.