HAUFE.
Ausgabe 01/2017

ACCESS

IM UNTERNEHMEN

RIBBON-KLASSEN

Programmieren Sie das Ribbon komplett ohne den Einsatz von XML und Callback-Funktionen (ab S. 8).

In diesem Heft:

TICKETSYSTEM, TEIL III

Weiter geht es mit der Ticketverwaltung – diesmal mit den Schritten nach Eingang eines Tickets.

SEITE 65

SCHNELLER FILTER

Lernen Sie eine Lösung kennen, mit der Sie mit minimalem Aufwand Datenblätter nach ihren Inhalten filtern.

SEITE 32

ACCESS PER URL STARTEN

Starten Sie eine Access-Anwendung vom Browser aus und übergeben Sie die benötigten Argumente für Formulare und Co.

SEITE 47

Ribbon ohne XML

Mit der Einführung des Ribbons unter Access 2007 hat Microsoft ein neues Menüsystem präsentiert. Allerdings ohne ein einziges Tool zu liefern, mit dem sich das Ribbon programmieren lässt. Mittlerweile gibt es ein paar Tools auf dem Markt, zum Beispiel den Ribbon-Admin. Noch cooler wäre es allerdings, wenn man komplett ohne Tools auskommen könnte. Wie das gelingt, zeigen wir Ihnen in dieser Ausgabe von Access im Unternehmen.

Dazu finden Sie gleich zwei Beiträge im aktuellen Heft. Unter dem Titel **Eigene Ribbons ohne Code** ab S. 2 finden Sie eine Möglichkeit, mit der Sie das Ribbon in einem begrenzten Umfang anpassen können – ohne dass Sie die Tabelle **USysRibbons** zum Speichern der Ribbon-Definition anlegen müssen.

Der zweite Beitrag zum Thema heißt **Ribbonklassen** (ab S. 8) und stellt eine Reihe von Klassenmodulen vor, die das Programmieren des Ribbons komplett per VBA ermöglichen. Damit erhalten Sie ein Objektmodell, mit dem Sie die gängigsten Ribbon-Steuerelemente abbilden können, in diesem Fall das Tab-, Group-, Button- und Separator-Steuerelement. Und das mit Ereignisprozeduren, wie Sie sie von Formularen und Steuerelementen her kennen – und natürlich inklusive der Anzeige von Bilddateien als Icons.

Wer schon einmal Webanwendungen entwickelt hat, die eine Datenbank verwenden und beispielsweise ein Suchformular enthalten, kennt den Begriff **SQL-Injection**. Dabei versucht der Benutzer, die eigentlich für eine Suche verwendete SQL-Anweisung zu manipulieren, indem er dieser zusätzliche Elemente anhängt. Das gelingt in eingeschränktem Maße auch unter Access – zumindest lassen sich so Daten aus anderen Tabellen ermitteln, die möglicherweise nicht für den Benutzer bestimmt sind. Der Beitrag **Sichere Filterausdrücke** zeigt ab S. 20 einige Beispiele dafür, wie Sie Suchtextfelder dazu nutzen können, die angezeigten Ergebnisse zu manipulieren. Natürlich erfahren Sie dort auch, wie Sie Ihre Anwendung gegen solche Manipulationen schützen.

Wenn Sie mit Datenblättern arbeiten, filtern oder sortieren Sie diese sicher oft mit den dafür vorgesehenen Menüs, die sich über einen Klick auf das Symbol rechts in den Spaltenköpfen anzeigen lassen. Eine Funktion, mit der Sie nach dem aktuell markierten Inhalt in einem Feld im Datenblatt filtern können, fehlt allerdings noch. Kein Problem: Wir rüsten diese nach und zeigen Ihnen, wie die Technik dahinter aussieht. Das und mehr lesen Sie im Beitrag **Schneller Filter** ab S. 32.

Passend dazu können Sie nun auch die Inhalte von Nachschlagefeldern im Datenblatt zum Filtern der Inhalte heranziehen: Dazu markieren Sie einfach den kompletten Text oder nur den Ausschnitt, nach dem Sie suchen wollen. Ein Klick auf eine Schaltfläche liefert dann die gewünschten Daten. Mehr dazu lesen Sie unter **Lookup-Kombinationsfelder nach Text filtern** ab S. 26.

In weiteren Beiträgen erfahren Sie zudem, wie Sie Access vom Browser aus durch die Eingabe einer speziellen URL starten können (**Access per URL starten**, ab S. 47). Außerdem stellen wir Ihnen die Möglichkeiten vor, um per VBA schreibend auf die Daten einer SQL Server-Datenbank zuzugreifen (**RDBMS per VBA: Daten bearbeiten**, ab S. 51). Den Abschluss macht eine weitere Folge unserer Reihe **Ticketsystem**, diesmal mit der dritten Folge (ab S. 65).

Und nun: Viel Spaß beim Lesen!

Ihr Michael Forster

Eigene Ribbons ohne Code

Zum Gestalten benutzerdefinierter Ribbons für Ihre Access-Anwendung gibt es zwei unterschiedliche Lösungen. Die eine setzt vollständig auf VBA-Code und die Methode LoadCustomUI, die andere verwendet eine ausgeblendete Tabelle USysRibbons, die Sie jeweils mit den XML-Auszeichnungen für die Anpassungen versehen. Dass jedoch auch noch eine dritte Lösung existiert, ist weitgehend unbekannt.

DocUICustomization

Recherchieren Sie im Netz nach diesem Begriff, so werden Sie nicht viel Erhellendes über ihn finden. Microsoft hat diese Datenbankeigenschaft nicht dokumentiert und andere Entwickler oder Autoren übersahen ihn bisher offenbar. Tatsächlich aber ist er der Schlüssel zu unserer Lösung. Doch fallen wir nicht gleich mit der Tür ins Haus und sehen uns an, wie sie sich in der Beispieldatenbank **RibbonOhne TabelleUndCode.accdb** präsentiert!

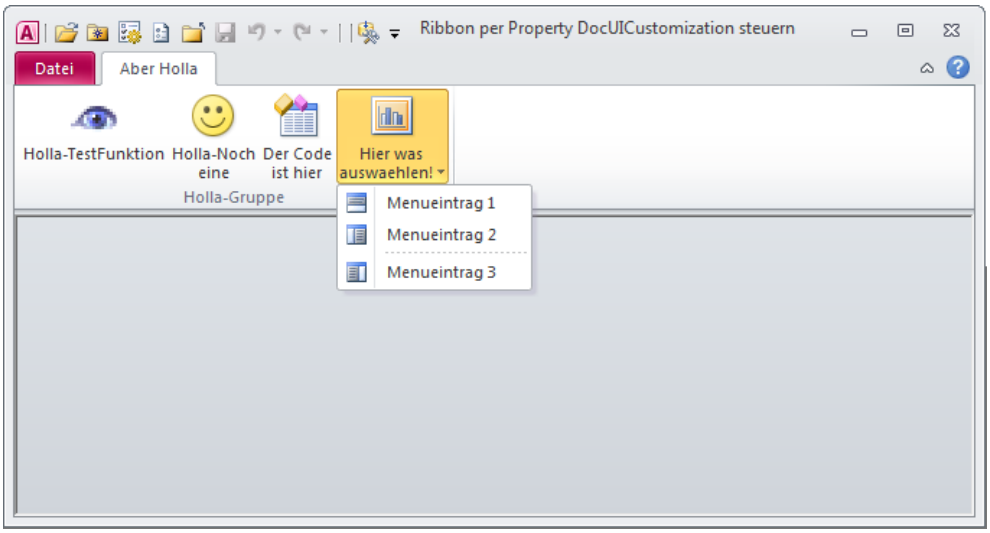

Bild 1: Der minimalistische Ribbon der Beispieldatenbank mit einigen Buttons und einem Menüelement

Nach ihrem Start zeigt sie sich wie in Bild 1. Hier sind unter dem neuen Tab **Aber Hallo!** einige Buttons und ein einfaches Menü untergebracht. Die eingebauten **Tabs** von Access sind ausgeblendet. Der Tab **Datei** allerdings ist vorhanden und aktiviert den **Backstage**-Bereich in gewohnter Weise. Beim Klick auf die Elemente der **Hallo-Gruppe** öffnen sich Meldungsfenster, damit deutlich wird, dass es hier nicht etwa nur um Optik geht, sondern dass wohl funktionierende Callback-Routinen im VBA-Projekt implementiert sind.

Der Code ist hier öffnet automatisch das Modul **mdlOptionen** der Datenbank und gibt die **Click-Callback**-Prozeduren für die Buttons preis. Irgendwelche Routinen zur Modifikation des Ribbon, etwa per **LoadCustomUI**, werden Sie hingegen vergeblich suchen. Auch eine Tabelle **USysRibbons** fehlt und die anderen Tabellen sind gleichfalls unauffällig. Wo also versteckt sich der XML-Code für die Anpassungen?

Setzen Sie dazu die folgende Zeile im VBA-Direktfenster ab:

```
? CurrentDb.Properties("DocUICustomization").Value
```

Damit wird tatsächlich der ganze zuständige XML-Code herausgegeben! Er unterscheidet sich nicht weiter von den sonst gebräuchlichen Codes, die Sie in der Tabelle **USysRibbons** oder über **LoadCustomUI** einsetzen. Warum hüllt sich Microsoft in Schweigen über diese Datenbankeigenschaft? Die mögliche Erklärung folgt.

Ribbon-Anpassungen über die Oberfläche

Seit **Access 2010** haben Sie die Möglichkeit, für eine Datenbank zumindest der Schnellzugriffsleiste eigene Elemente zu spendieren. Öffnen Sie dazu bei geladener Datenbank die Optionen von Access und aktivieren den Eintrag **Symbolleiste für den Schnellzugriff**. Im rechten Bereich des Dialogs findet sich nun die Liste der Elemente der Schnellzugriffsleiste. Das Kombinationsfeld darüber zeigt den unscheinbaren Eintrag **Für alle Dokumente (Standard)**. Doch hier lässt sich auch die aktuelle Datenbank **Für 'xyz.accdb'** auswählen! Bei Aktivierung leert sich die Liste, falls die Datenbank noch mit keinen Anpassungen versehen wurde.

Sie können nun aus dem Pool der Befehle links der Datenbank Elemente hinzufügen (s. Bild 2). Nach dem Speichern über **OK** stehen sie in Zukunft immer für diese Datenbank zur Verfügung. Andere Datenbanken zeigen sie nicht. Ergo müssen diese Anpassungen irgendwo in der Datenbank abgespeichert sein. Und deren Eigenschaft **DocUICustomization** ist eben dieser Ort.

Sie können sich den Inhalt dieser Eigenschaft nach der manuellen Anpassung mit der erwähnten VBA-Zeile ausgeben lassen. Der XML-Code für die abgebildete Anpassung sieht so aus wie in Listing 1. Die Struktur ist die übliche, nur dass hier jedem **XML-Tag** der **Namespace mso** vorangestellt ist, was, wie wir noch sehen werden, eigentlich überflüssig ist.

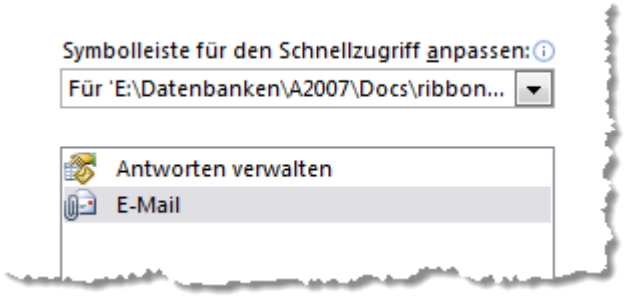

Bild 2: Exklusive Schnellzugriffselemente für geladene Datenbank

Es hat den Anschein, als ob Microsoft Entwicklern die Dokumentation dieser Eigenschaft vorenthält, damit sie der manuellen Anpassung vorbehalten bleibt.

Zuweisung an DocUICustomization

Statt über den **Optionen**-Dialog können Sie nun die Anpassung auch per Code vornehmen, indem Sie der Eigenschaft einen eigenen Wert zuweisen. Listing 2 zeigt ein extrem kurzes Beispiel.

Die Variable **sXML** nimmt zunächst den XML-Code entgegen. Über **Replace** entfernt die nächste Zeile alle **mso:**-Strings aus ihrem Inhalt. Dann wird der geänderte Code der Eigenschaft wieder zugewiesen. Dies wirkt sich noch nicht unmittelbar aus.

Wie bei manuellen Anpassungen auch muss die Datenbank geschlossen und wieder geöffnet werden. Es zeigt sich dann, dass das Entfernen der **mso:**-Präfixe keine negativen Auswirkungen auf das Funktionieren der Anpassungen hat. Das ist indessen für den XML-Kenner auch nicht verwunderlich, da nun als Standard-**Namespace** (**xmlns**) für das XML in der ersten Zeile das Schema direkt deklariert ist, statt über das Präfix **mso**. Beides kommt im Prinzip auf das Gleiche heraus.

```
<mso:customUI xmlns:mso="http://schemas.microsoft.com/office/2006/01/customui">
    <mso:ribbon>
        <mso:qat>
            <mso:documentControls>
                <mso:control idQ="mso:ManageReplies" visible="true"/>
                <mso:control idQ="mso:FileSendAsAttachment" visible="true"/>
            </mso:documentControls>
        </mso:qat>
    </mso:ribbon>
</mso:customUI>
```

Listing 1: Der XML-Code für die Schnellzugriffsanpassungen aus der Eigenschaft **DocUICustomization**

Die spannende Frage ist nun, welche Ribbon-Anpassungen sich alle in **DocUICustomization** abseits der Schnellzugriffsleiste unterbringen lassen.

```
Sub ModifyCustUI()
    Dim sXML As String

    sXML = CurrentDb.Properties("DocUICustomization").Value
    sXML = Replace(sXML, "mso:", vbNullString)
    CurrentDb.Properties("DocUICustomization").Value = sXML
End Sub
```

Listing 2: Modifizieren des XML-Codes für die Schnellzugriffsanpassungen der aktuellen Datenbank

Was geht, was geht nicht?

Bei einer Datenbank, deren Schnellzugriffsleiste Sie noch nicht anpassten, führt der Zugriff auf **DocUICustomization** zu einem Fehler. Es gibt das **Property** schlicht noch nicht. Sie können es so per VBA erzeugen:

```
Dim prp As DAO.Property
Set prp = CurrentDb.CreateProperty( _
    "DocUICustomization",  dbText)
Currentdb.Properties.Append prp
```

Dies legt die Eigenschaft nur an, ohne ihr einen Wert zuzuweisen. Stattdessen könnten Sie auch gleich **prp.Value** vor dem **Append** einen gültigen Anpassungs-XML-Code verabreichen. Später läuft dies etwa so:

```
Dim prp As DAO.Property
Dim sXML As String
sXML = "<customUI ..."
Set prp = CurrentDb.Properties("DocUICustomization")
prp.Value = sXML
```

Der Inhalt von **sXML** ist nun unsere Spielwiese. Es haben sich über **Trial & Error** nach unzähligen Versuchen einige Ergebnisse herausgestellt. Der deklarierte Namespace in Listing 1 ist der des XML-Schemas von **Office 2007**. Tatsächlich kann problemlos auch das ab **Office 2010** gültige Schema angegeben werden:

```
http://schemas.microsoft.com/office/2009/07/customui
```

Immerhin gab es ja in diesem XML-Schema so einige Erweiterungen, wie etwa die Definitionen zum **Backstage**.

Tatsächlich aber funktionieren, wie sich herausstellte, bei Anpassungen über **DocUICustomization** leider eine ganze Menge Dinge nicht:

- Der **Backstage** kann nicht angesprochen werden. Zwar können Sie im XML-Code anstandslos alle hierfür gültigen Auszeichnungen unterbringen, Access ignoriert diese jedoch geflissentlich.

- Dasselbe gilt für die Schnellzugriffsleiste. Hier kann nur die für die Anwendung vorgesehene Anpassung (**documentControls**) vorgenommen werden, nicht aber eine Modifikation der generellen Elemente (**sharedControls**).

- **Callback**-Funktionen werden samt und sonders ignoriert, auch wenn sie angegeben sind. Damit entfällt eine dynamische Steuerung des Ribbons. **Callbacks** wie **onLoad**, **getVisible**, **getImage** et cetera bleiben außen vor. Die einzige mögliche **Callback**-Funktion ist **onAction**, die Sie für Buttons angeben können. Die Funktion muss in Ihrem VBA-Modul allerdings eine von den üblichen **onAction**-Prozeduren abweichende Syntax aufweisen. Später mehr dazu.

- **startFromScratch** steht automatisch immer auf **False**. Ändern können Sie dies nicht. Dadurch können Sie keinen Ribbon von Grund auf neu erstellen, sondern den eingebauten nur modifizieren. Immerhin kann das Attribut **visible** für alle **Tabs** auf **False** gesetzt werden, wodurch alle eingebauten Ribbon-Tabs verschwinden. Nicht gilt dies allerdings für den **Datei-Tab**. Der lässt sich ja generell auch über andere

Objektorientierte Ribbon-Programmierung

Das Ribbon lässt sich normalerweise nur definieren, indem Sie den XML-Code zur Beschreibung des Ribbons festlegen, in einer bestimmten Tabelle speichern und die dortigen Einträge dann als Anwendungsribbon nutzen oder bestimmten Formularen oder Berichten zuweisen. Mit hier vorgestellten Klassen können Sie das Ribbon von nun an ganz einfach per VBA zusammenstellen und etwa Formularen oder Berichten zuweisen.

Wie funktioniert das nun – ein Ribbon anzuzeigen, ohne die Tabelle **USysRibbons** mit Ribbon-Definitionen zu füllen beziehungsweise gänzlich ohne statisch definierte Ribbons? Grundsätzlich ist das ganz einfach: Sie benötigen lediglich einige Objekte aus der Beispieldatenbank zu diesem Beitrag, die Sie über die Importieren-Funktion in die Datenbank importieren, die Sie mit Ribbons ausstatten möchten.

Dabei handelt es sich um die folgenden Objekte:

- **clsStartRibbon**: Klasse, in der Sie das Hauptribbon einer Anwendung definieren
- **Ribbons**: Klasse, mit der Sie die Ribbons einer Anwendung verwalten
- **clsRibbon**: Klasse, die die Eigenschaften und Ereignisse des **Ribbon**-Elements bereitstellt
- **clsTabs**: Klasse, welche die Tabs verwaltet
- **clsTab**: Klasse, welche die Eigenschaften des **Tab**-Elements bereitstellt
- **clsGroups**: Klasse, welche die Gruppen eines **Tab**-Elements verwaltet
- **clsGroup**: Klasse, welche die Eigenschaften und Ereignisse des **Group**-Elements bereitstellt
- **clsControls**: Klasse, welche die Steuerelemente innerhalb einer Gruppe verwaltet
- **clsButton**: Klasse, welche die Funktionen des **Button**-Elements kapselt
- **clsSeparator**: Klasse, welche die Eigenschaften eines Separator-Elements kapselt
- **mdlRibbons**: Stellt einige globale Variablen bereit sowie die Funktionen zur Anzeige von Bilddateien
- **mdlRibbonEnums**: Modul, das einige Enumerationen enthält
- **mdlTools**: Enthält einige allgemeine Funktionen wie etwa zum Ermitteln von GUIDs oder zum Erstellen von formatierten XML-Dokumenten
- **mdlZwischenablage**: Enthält Funktionen, um Daten in die Zwischenablage zu kopieren und von dort abzufragen

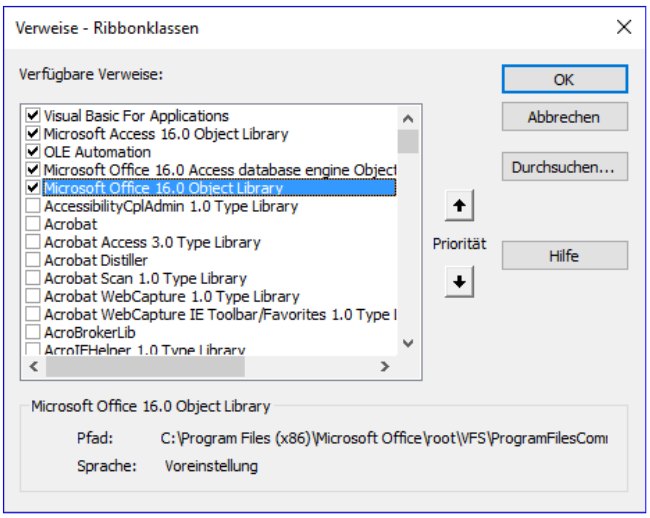

Bild 1: Verweis auf die Office-Bibliothek

Außerdem sind noch drei Schritte nötig:

- Sie fügen den Verweisen der Anwendung den Eintrag **Microsoft Office x.0 Object Library** hinzu (s. Bild 1).
- Sie tragen für die Eigenschaft **Name des Menübands** in den Access-Optionen den Wert **Main** ein (s. Bild 2).

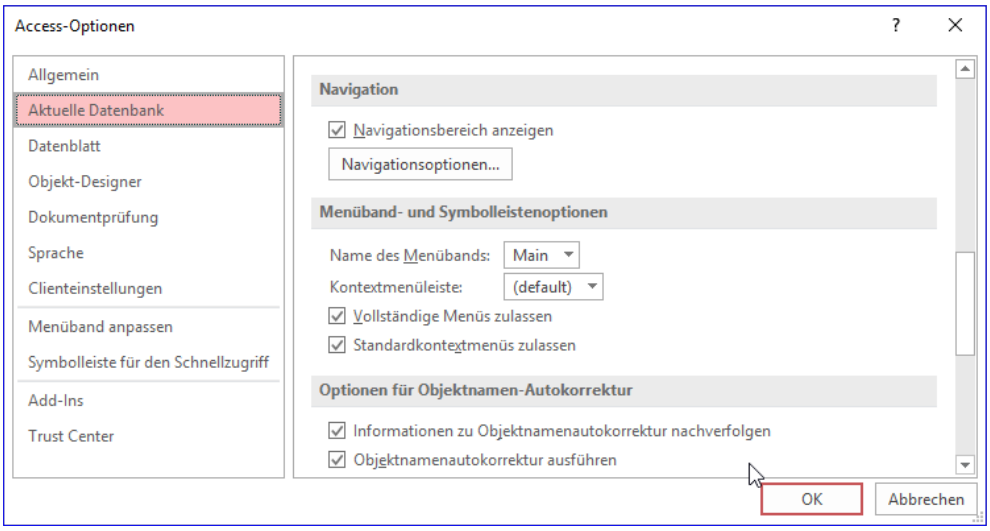

Bild 2: Option, um das Anwendungsribbon einzustellen

- Sie aktivieren die Option **Fehler von Benutzeroberflächen-Add-Ins anzeigen** in den Access-Optionen (s. Bild 3).

Ribbons anzeigen

Zum Anzeigen von Ribbons gibt es auf herkömmlichem Wege zwei Möglichkeiten: Entweder Sie tragen den Namen der gewünschten Ribbon-Definition, die Sie in der Tabelle **USysRibbons** gespeichert haben, für die oben genannte Eigenschaft in den Access-Optionen ein – dann wird das entsprechende Ribbon gleich beim Anwendungsstart angezeigt. Oder Sie möchten, dass das Ribbon mit einem Formular oder Bericht eingeblendet wird.

Dann tragen Sie den Namen des Ribbons für die entsprechende Eigenschaft des Formulars oder Berichts ein.

Wir verwenden in der hier vorgestellten Lösung zwar keine Tabelle namens **USysRibbons** zum Speichern von Ribbon-Definitionen, aber wir können dennoch beide Methoden zum Anzeigen von Ribbons nutzen. Für die erste Methode ist die obige Angabe des Standardribbons nötig, beispielsweise über den Namen **Main**.

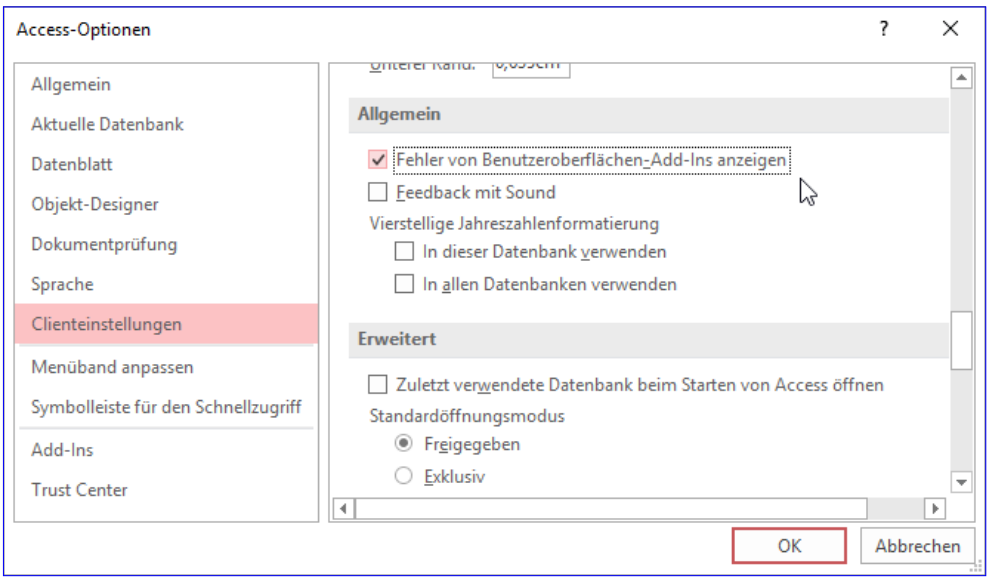

Bild 3: Aktivieren der Anzeige von Ribbon-Fehlern

Außerdem brauchen Sie ein **AutoExec**-Makro, das dafür sorgt, dass das Ribbon auch direkt beim Öffnen der Anwendung angezeigt wird.

Dieses Makro definieren Sie wie in Bild 4. Der einzige Makrobefehl lautet **AusführenCode** und ruft die Funktion **RibbonLaden**

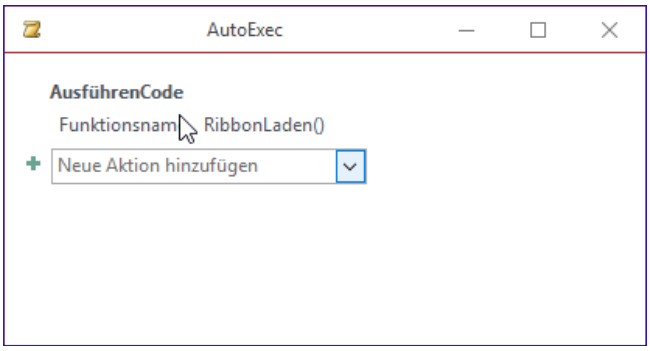

Bild 4: **AutoExec**-Makro, das ein Ribbon einblenden soll

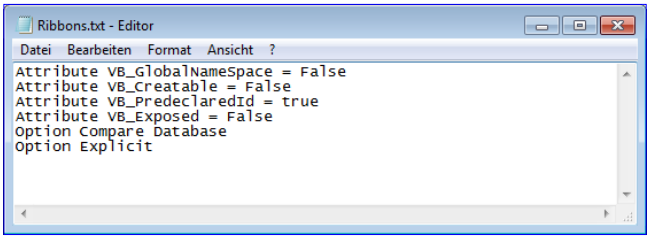

Bild 5: Anpassen der Klasse **Ribbons**, damit diese nicht instanziert werden muss

auf. Diese Funktion ist im Modul **mdlRibbons** definiert und sieht wie folgt aus:

```
Public Function RibbonLaden()
    Startribbon.CreateRibbon
End Function
```

Hier wird also die Methode **CreateRibbon** eines Objekts namens **Startribbon** aufgerufen. Wie Sie weiter oben sehen, haben wir die entsprechende Klasse **Startribbon** bereits in der Beispieldatenbank vorbereitet. Aber wie können wir die Methode **CreateRibbon** dieser Klasse nutzen, ohne dass wir diese instanzieren? Das ist kein Problem: Wie müssen die Klasse nur einmal in eine Textdatei exportieren und dann einige Eigenschaften, die im VBA-Editor nicht sichtbar sind, anpassen. In Bild 5 haben wird dies für die Klasse **Ribbons** durchgeführt, die ebenfalls ohne Instanzierung genutzt werden soll.

Nun wollen wir in der Klasse **StartRibbon** eine Methode namens **CreateRibbon** anlegen, mit der wir ein Ribbon namens **Main** definieren, das dann beim Öffnen der Anwendung gleich angezeigt werden soll.

```
Public Sub CreateRibbon()
    ...
End Sub
```

In dieser deklarieren wir nun zunächst ein Objekt des Typs **clsRibbon**, das unsere Ribbon-Deklaration aufnimmt.

```
Dim objRibbon As clsRibbon
```

Wir wollen als Erstes lediglich ein leeres **Tab**-Element hinzufügen, um zu zeigen, dass die Klassen auch wirklich ihr Werk verrichten. Dazu deklarieren wir auch noch ein Element des Typs **clsTab**:

```
Dim objTab As clsTab
```

Das **clsRibbon**-Element füllen wir dann mit einem Objekt, das die **Add**-Methode der **Ribbons**-Klasse liefert:

```
    Set objRibbon = Ribbons.Add("Main")
```

Dieses Objekt besitzt wie das entsprechende Element in der Ribbon-Definition eine Eigenschaft namens **StartFromScratch**, die wir auf den Wert **True** einstellen:

```
Public Sub CreateRibbon()
    Dim objRibbon As clsRibbon
    Dim objTab As clsTab
    Set objRibbon = Ribbons.Add("Main")
    With objRibbon
        .StartFromScratch = True
        Set objTab = .Tabs.Add("Beispieltab")
        With objTab
            .label = "Beispieltab"
        End With
    End With
    Ribbons.CreateStartRibbon "Main"
End Sub
```

Listing 1: Die Methode **CreateRibbon** füllen Sie mit eigenen Anweisungen.

```
    With objRibbon
        .StartFromScratch = True
```

Dann fügen wir dem **clsRibbon**-Element mit der **Add**-Methode der **Tabs**-Auflistung ein neues **Tab**-Element hinzu. Den Namen übergeben wir als Parameter:

```
Set objTab = .Tabs.Add("Beispieltab")
```

Für dieses Objekt legen wir die Beschriftung mit der Eigenschaft **label** fest:

```
        With objTab
            .label = "Beispieltab"
        End With
    End With
```

Danach rufen wir die Methode **CreateStartRibbon** der Klasse **Ribbons** auf und übergeben dieser den Namen des zu erstellenden Ribbons, also **Main**:

```
    Ribbons.CreateStartRibbon "Main"
End Sub
```

Die vollständige Fassung dieser Prozedur finden Sie in Listing 1.

Anschließend schauen wir uns an, ob diese Art der Ribbon-Definition auch wirklich funktioniert. Dazu schließen Sie die Access-Datei und öffnen diese erneut, damit das **AutoExec**-Makro ausgeführt wird. Es reicht nicht aus, einfach das Makro auszuführen, da das Main-Ribbon möglicherweise zu diesem

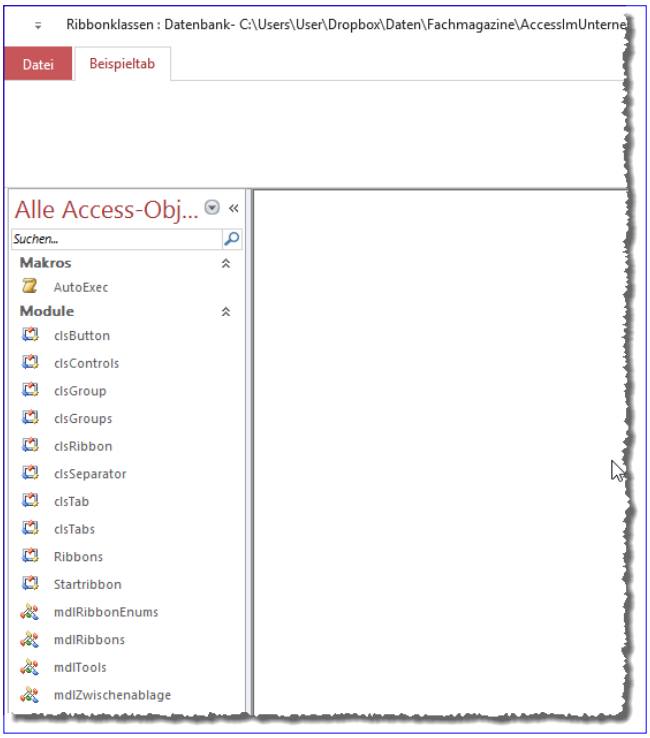

Bild 6: Das erste per VBA-Code erzeugte Ribbon

```
Public Sub CreateRibbon()
    Dim objRibbon As clsRibbon
    Dim objTab As clsTab
    Dim objGroup As clsGroup
    Set objRibbon = Ribbons.Add("Main")
    With objRibbon
        .StartFromScratch = True
        Set objTab = .Tabs.Add("Beispieltab")
        With objTab
            .label = "Beispieltab"
            Set objGroup = .Groups.Add("Beispielgruppe")
            With objGroup
                .label = "Beispielgruppe"
                Set objButton1 = .Controls.Add(msoRibbonControlButton, "Beispielbutton")
                objButton1.label = "Button 1"
                Set objButton2 = .Controls.Add(msoRibbonControlButton, "Beispielbutton1")
                objButton2.label = "Button 2"
            End With
        End With
    End With
    Ribbons.CreateStartRibbon "Main"
End Sub
```

Listing 2: Anlegen eines Ribbons, diesmal mit einer Gruppe und zwei Schaltflächen

Zeitpunkt schon installiert wurde und erneutes Anlegen zu einem Fehler führt (dies können Sie durch erneutes Aufrufen des **AutoExec**-Makros ausprobieren). Beim erneuten Öffnen der Anwendung sollte nun die Ansicht aus Bild 6 erscheinen. Es funktioniert!

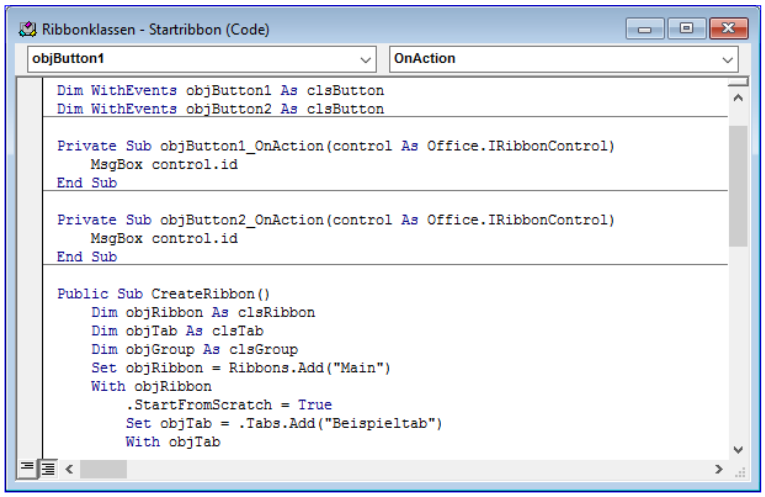

Bild 7: Deklarieren von **Button**-Elementen und Implementieren ihrer Ereignisse

Gruppen und Schaltflächen hinzufügen

Nun gehen wir einen Schritt weiter und wollen dem **Tab**-Element eine Gruppe hinzufügen – und natürlich auch noch **Button**-Elemente. Die einfachste Variante sieht nun wie in Listing 2 aus. Sie sehen hier, dass wir ein weiteres Objekt des Typs **clsGroup** mit dem Variablennamen **objGroup** deklariert haben. Dieses fügen wir dann über die **Add**-Methode der **Groups**-Auflistung des **clsTab**-Objekts zum Ribbon hinzu. Auch dieses Objekt hält eine Eigenschaft namens **label** bereit, mit der Sie die Beschriftung der Gruppe festlegen können. Diese legen wir hier mit dem Ausdruck **Beispielgruppe** fest.

Danach folgen die beiden **Button**-Elemente: Das erste fügen wir mit der Methode **Add** der Auflistung **Controls** des Objekts **objGroup** hinzu. Dabei geben wir als ersten Parameter den Wert **msoRibbonControlButton** an, was bewirkt, dass hier ein **Button**-Element angelegt wird. Mit dem zweiten Parameter übergeben wir den Namen dieses **Button**-Elements. Danach legen wir wiederum mit der Eigenschaft **label** die Bezeichnung des **Button**-Elements fest.

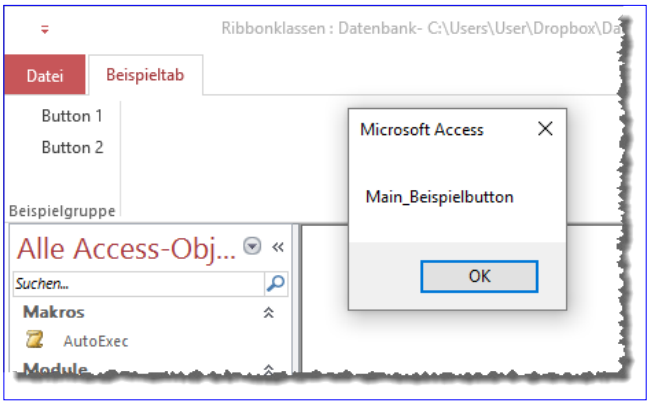

Bild 8: Test der neuen **Button**-Elemente

Wenn Sie aufgepasst haben, fällt Ihnen auf, dass die Prozedur gar keine Deklaration für die beiden Button-Elemente **objButton1** und **objButton2** enthält.

Das hat folgenden Grund: Unter anderem soll die Verwendung dieser Lösung dafür sorgen, dass Sie richtige Ereignisprozeduren für die Steuerelemente des Ribbons hinterlegen können. Das heißt, dass Sie die Ribbon-Elemente, die Ereignisse auslösen, mit dem Schlüsselwort **WithEvents** deklarieren und dafür entsprechende Ereignisprozeduren anlegen können.

Das gelingt, indem Sie zunächst die Deklaration im Kopf des Klassenmoduls vornehmen – für unsere beiden Schaltflächen also wie folgt:

```
Dim WithEvents objButton1 As clsButton
Dim WithEvents objButton2 As clsButton
```

Dann verwenden Sie die beiden Kombinationsfelder im Kopf des Modulfensters und wählen dort im linken Element den Namen des betroffenen Steuerelements, zum Beispiel **objButton1**, und im rechten den Eintrag **OnAction** aus. Dies legt dann automatisch die passende

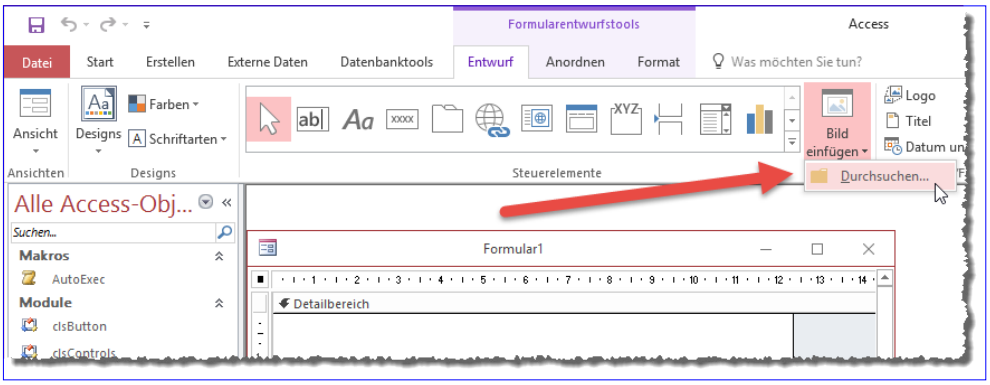

Bild 9: Hinzufügen von Bildern zur Tabelle **MSysResources**

Ereignisprozedur an, die wir mit einer Test-Meldung füllen (s. Bild 7):

```
Private Sub objButton1_OnAction(control As RibbonControl)
    MsgBox control.id
End Sub

Private Sub objButton2_OnAction(control As IRibbonControl)
    MsgBox control.id
End Sub
```

In dieser Prozedur können wir dann gleich den Verweis auf das entsprechende Steuerelement nutzen, der mit dem Parameter **control** geliefert wird.

Dies bietet zum Beispiel mit der Eigenschaft **id** den Namen des angeklickten Steuerelements an, den wir hier gleich per Meldungsfenster ausgeben. Das Ergebnis des aktuellen Stands finden Sie in Bild 8.

Große Schaltflächen mit Bildern

Nun wollen wir natürlich auch die optischen Möglichkeiten des Ribbons nutzen und die Schaltflächen größer und mit Bildern anzeigen. Die dazu benötigten Bilder fügen Sie zur Systemtabelle **MSysResources** hinzu.

Das gelingt am einfachsten, indem Sie ein Formular in der Entwurfsansicht öffnen und dann im Ribbon den Eintrag **EntwurflSteuerelementelBild einfügen** anklicken. Hier erscheint dann ein Eintrag namens **Durchsuchen...**, mit dem Sie einen **Grafik einfügen**-Dialog öffnen können (s. Bild 9).

Damit fügen Sie dann die Bilddateien hinzu – für kleine Schaltflächen in der Größe 16x16, für große Schaltflächen in 32x32. Wenn Sie zuvor nicht das Formular anklicken, werden die Bilddateien noch nicht einmal direkt zum Formular hinzugefügt, sondern nur zur Tabelle **MSysResources**.

Diese sieht dann etwa wie in Bild 10 aus. Die Definition passen wir nun wie folgt an:

```
Set objButton1 = .Controls.Add(7
                msoRibbonControlButton, "Beispielbutton")
With objButton1
    .label = "Button 1"
    .Size = msoRibbonControlSizelarge
    .image = "add"
End With
Set objButton2 = .Controls.Add(7
                msoRibbonControlButton, "Beispielbutton1")
```

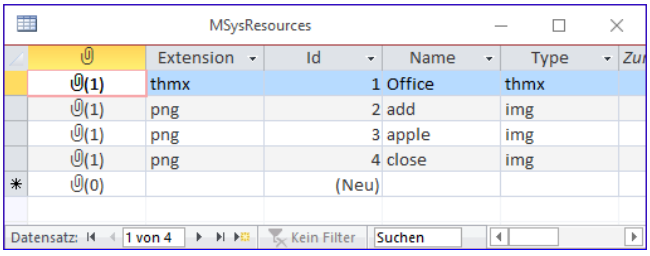

Bild 10: Bilder in der Tabelle **MSysResources**

Bild 11: Anzeige benutzerdefinierter Bilder

Sichere Filterausdrücke

Filterausdrücke können schnell gefährlich werden, zumindest wenn Sie nicht aufpassen. Dafür sorgt die sogenannte SQL-Injection, bei der davon ausgegangen wird, dass der Benutzer einen Ausdruck in die Benutzeroberfläche eingibt, die dann als Teil eines SQL-Ausdrucks verwendet wird und Schaden anrichten könnte. Dieser Beitrag erläutert, was SQL-Injection eigentlich ist und wie Sie sich dagegen wappnen können.

Die SQL-Injection ist eine Methode, aus eigentlich harmlosen SQL-Anweisungen schädliche SQL-Anweisungen zu machen – oder damit an Informationen zu gelangen, die der Benutzer normalerweise nicht erhalten sollte. Dabei ist die Voraussetzung immer ein Eingabefeld in einer Webseite oder, in diesem Falle, in einem Access-Formular, in das der Benutzer beispielsweise einen Vergleichswert für einen Vergleichsausdruck eingeben soll – im einfachsten Fall etwa die ID eines Kunden.

Beispielformular

Als Spielwiese für die folgenden Beispiele verwenden wir ein einfaches Access-Formular namens **frmKunden** samt Unterformular **sfmKunden**, wobei das Unterformular die Daten der Tabelle **tblKunden** anzeigt und das Hauptformular ein Eingabefeld für einen Suchbegriff liefert, der zunächst die Suche nach Werten im Feld **KundeID** ermöglicht (s. Bild 1).

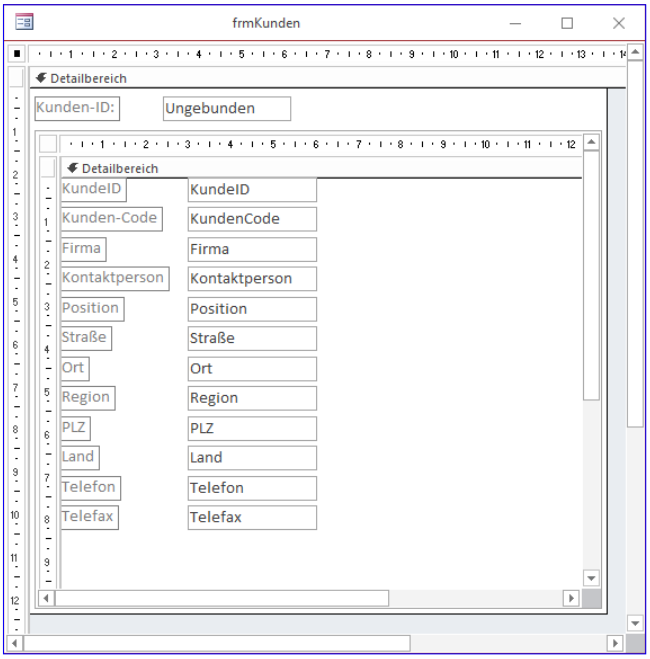

Bild 1: Beispielformular in der Entwurfsansicht

Für das Textfeld **txtKundenID** legen wir die folgende Ereignisprozedur an:

```
Private Sub txtKundenID_AfterUpdate()
    Dim strSQL As String
    strSQL = "SELECT * FROM tblKunden WHERE KundeID = " ⤵
                                        & Me!txtKundenID
    Me!sfmKunden.Form.RecordSource = strSQL
End Sub
```

Diese sorgt dafür, dass eine neue SQL-Anweisung zusammengestellt wird, die den Datensatz der Tabelle **tblKunden** mit einem bestimmten Wert im Feld **KundeID** liefert und diesen der Eigenschaft **RecordSource** des Unterformular zuweist. Dies gelingt auch reibungslos, wenn der Benutzer einfach nur einen Zahlenwert eingibt (s. Bild 2).

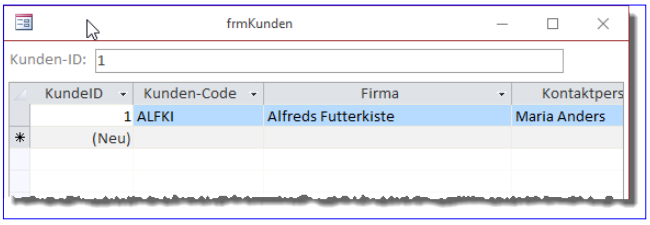

Bild 2: Filtern nach dem Wert **1** im Feld **KundeID**

Ergebnismenge mit UNION erweitern

Was an unserer Methode soll also gefährlich sein – schließlich kann man doch nur Zahlenwerte eingeben und erhält das gewünschte Ergebnis? Mitnichten: Schon die Kenntnis der Datenstruktur (oder ein wenig experimentelle

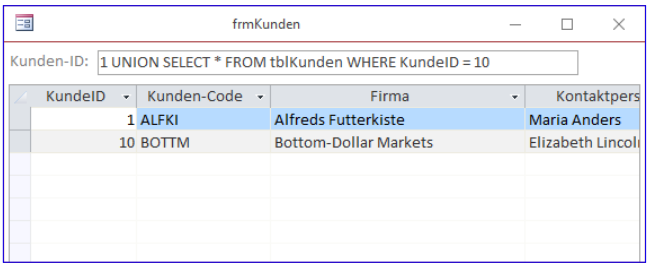

Bild 3: Erweiterung des Ergebnisses per UNION

Arbeit) erlauben es, ein ganz anderes Ergebnis zu erhalten als geplant. Geben Sie beispielsweise einmal den folgenden Ausdruck in das Suchfeld ein:

```
1 UNION SELECT * FROM tblKunden WHERE KundeID = 10
```

Das Ergebnis ist überraschend: Es gibt keinen Fehler, sondern es erscheinen zwei Datensätze (s. Bild 3)! Allzu verwunderlich ist dies jedoch gar nicht, denn der resultierende Ausdruck in der Variablen **strSQL**, den Sie sich einfach im Debugbereich ausgeben lassen können, sieht wie folgt aus:

```
SELECT * FROM tblKunden WHERE KundeID = 1
UNION
SELECT * FROM tblKunden WHERE KundeID = 11
```

Gut, der Schaden ist gering, denn wir greifen so nur auf Datensätze zu, die wir auch anderweitig hätten lesen können. Aber wenn wir keine Bedingung angeben, liefert der resultierende SQL-Ausdruck uns sogar wieder alle Datensätze:

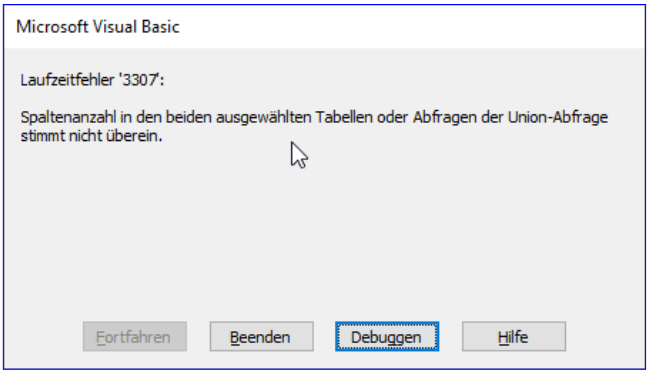

Bild 4: Fehler, wenn nicht alle Seiten der UNION-Abfrage die gleiche Anzahl Felder enthalten

```
SELECT * FROM tblKunden WHERE KundeID = 1
UNION
SELECT * FROM tblKunden
```

Dies könnte interessant werden, wenn der eigentliche Filterausdruck noch dafür sorgt, dass beispielsweise nur die Kunden des aktuell angemeldeten Mitarbeiters angezeigt werden sollen.

Daten anderer Tabellen auslesen

Noch spannender wird es, wenn Sie hinter dem **UNION**-Schlüsselwort nicht auf die gleiche, sondern direkt auf eine andere Tabelle zugreifen, die unter Umständen kritische Daten enthält. Im Beispiel wollen wir einfach einmal probieren, Daten aus der Tabelle **tblArtikel** zusätzlich auszugeben, und testen den folgenden Suchausdruck:

```
1 UNION SELECT ArtikelID as KundeID, Artikelname as Kundencode FROM tblArtikel
```

Dies liefert im ersten Anlauf noch den Fehler aus Bild 4, weil der zweite Teil der **UNION**-Abfrage nicht die gleiche Anzahl Felder wie der erste Teil zurückliefert. Aber was geschieht, wenn wir einfach ein paar fixe Werte entsprechend der Anzahl der Felder des ersten Teils der Abfrage hinzufügen – und so notfalls durch Experimentieren auf die richtige Anzahl Felder kommen? So ergibt sich nach kurzem Test der folgende Ausdruck:

```
1 UNION SELECT ArtikelID as KundeID, Artikelname as Kundencode, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12 FROM tblArtikel
```

Das Ergebnis ist einigermaßen erschreckend: Wir gelangen so tatsächlich an die Informationen der Tabelle **tblArtikel**, die eigentlich gar nicht zur Datenherkunft des Unterformulars gehört (s. Bild 5). Und natürlich könnten wir diese Tabelle auch durch eine Tabelle mit wirklich sensiblen Daten wie etwa der Benutzertabelle ersetzen.

So lassen sich auch leicht die Tabellen einer Datenbank ausgeben – und zwar mit diesem Suchbegriff:

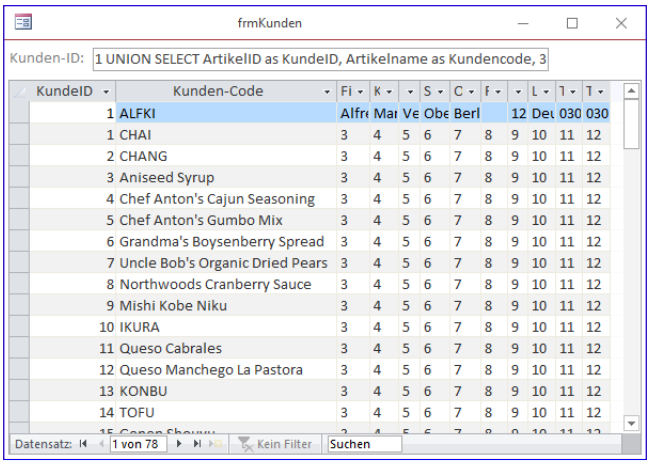

Bild 5: Ermitteln der Daten einer anderen Tabelle

```
1 UNION SELECT Name, Type , 3, 4, 5, 6, 7, 8, 9, 10, 11,
12 FROM MSysObjects WHERE Type = 1
```

Gefährliche Strings

Gibt es ebenfalls Probleme, wenn man ein Suchfeld für die Eingabe einer Zeichenkette verwendet? Ja. Dazu fügen wir dem Formular ein Textfeld namens **txtFirma** hinzu, das nach dem Aktualisieren die folgende Ereignisprozedur auslöst:

```
Private Sub txtFirma_AfterUpdate()
    Dim strSQL As String
    strSQL = "SELECT * FROM tblKunden _
                WHERE Firma LIKE '" & Me!txtFirma & "'"
    Debug.Print strSQL
    Me!sfmKunden.Form.RecordSource = strSQL
End Sub
```

Geben Sie hier etwa **A*** ein, liefert dies alle Einträge der Tabelle **tblKunden**, deren Feld **Firma** einen Wert enthält, der mit **A** beginnt. Wie können wir hier beispielsweise mit dem **UNION**-Schlüsselwort auf die Daten einer zweiten Tabelle wie etwa **MSysObjects** zugreifen? Dazu fügen wir einfach hinter dem Vergleichswert ein schließendes Hochkomma ein und hängen die **UNION**-Klausel hinten an:

```
A*' UNION SELECT Name, Type, 3,4,5,6,7,8,9,10,11,12 FROM
MSysObjects
```

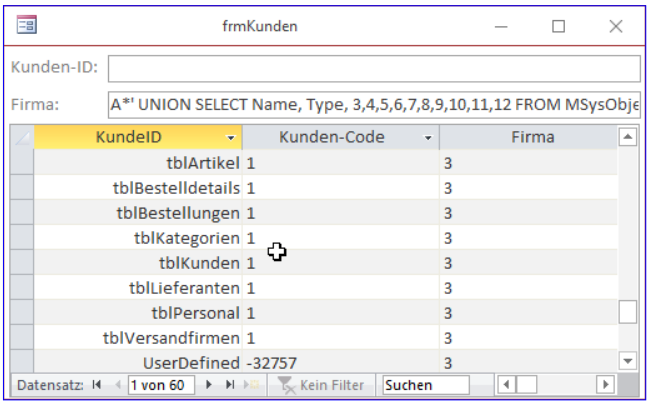

Bild 6: Ermitteln der Daten einer anderen Tabelle, hier die Tabellennamen

Dies liefert uns, wie in Bild 6 zu erkennen, wie gewünscht alle Tabellennamen der Datenbank. Der resultierende SQL-Ausdruck endet nun zwar mit einem unerwünschten Hochkomma, doch das hindert Access nicht an seiner Interpretation:

```
SELECT * FROM tblKunden WHERE Firma LIKE 'A*'
UNION
SELECT Name, Type, 3,4,5,6,7,8,9,10,11,12
FROM MSysObjects'
```

Würde der SQL-Ausdruck wie folgt zusammengesetzt, also mit einem Sternchen am Ende, würde dies allerdings einen Fehler auslösen:

```
strSQL = "SELECT * FROM tblKunden WHERE Firma LIKE '" _
                                    & Me!txtFirma & "*'"
```

Der resultierende SQL-Ausdruck würde nämlich dann wie folgt aussehen:

```
SELECT * FROM tblKunden WHERE Firma LIKE '' UNION SELECT
Name, Type, 3,4,5,6,7,8,9,10,11,12 FROM MSysObjects*'
```

Auch dies können wir jedoch umgehen, wenn wir folgenden Suchausdruck eingeben:

```
' UNION SELECT Name, Type, 3,4,5,6,7,8,9,10,11,12 FROM
MSysObjects WHERE '
```

Lookup-Kombinationsfelder nach Texten filtern

Wenn Sie die Datensätze eines Unterformulars in der Datenblattansicht filtern wollen, gelingt die Eingabe in Text-, Zahlen- und Datumsfelder recht einfach. Wenn Sie jedoch ein Suchfeld für die Werte eines Lookup-Kombinationsfeldes programmieren wollen, stoßen Sie schnell an die Grenzen. Sie können die Feldinhalte nämlich nicht einfach mit den in den Feldern angezeigten Werten vergleichen, denn diese stammen ja aus den Lookup-Tabellen, mit denen solche Steuerelemente gefüllt werden. Dieser Beitrag zeigt, wie auch das Filtern nach den Werten in Kombinationsfeldern zum Kinderspiel wird.

Als Erstes zeigen wir Ihnen, wie es nicht gelingt und was somit das Problem beim Filtern nach den Inhalten von Lookup-Kombinationsfeldern ist. In unserem Beispielformular haben wir ein Textfeld zur Eingabe des Suchbegriffs, eine Suchen-Schaltfläche namens **cmdSuchen** sowie ein Unterformular namens **sfmArtikel_Lookupfilter** angelegt (s. Bild 1).

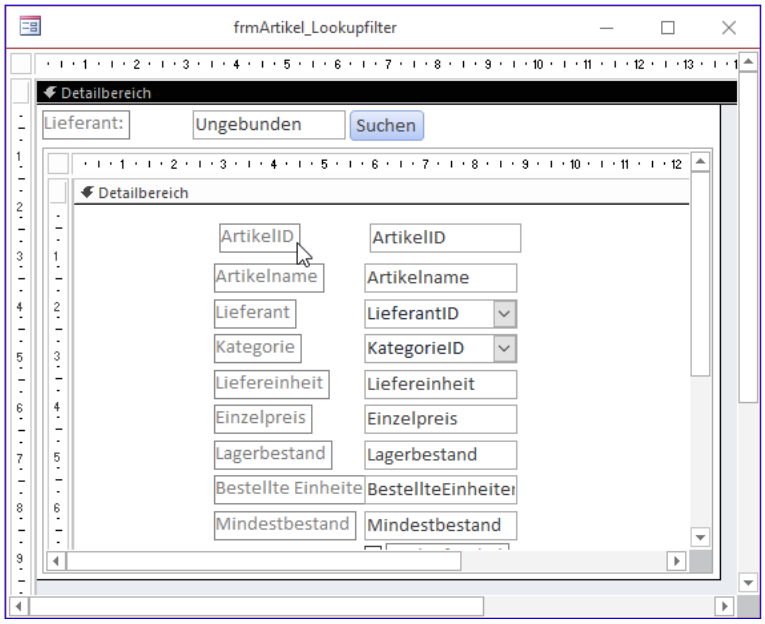

Bild 1: Aufbau des Beispielformulars

In einem ersten, naiven Ansatz wollen wir die Suche wie für ein normales Feld des Datentyps String durchführen. Die dazugehörige Prozedur sieht dann wie in Listing 1 aus.

Das Ergebnis sieht wie in Bild 2 aus: Es liefert keinerlei Datensätze, obwohl das Kombinationsfeld doch einige Werte anzeigt, die mit dem Suchbegriff **E*** beginnen. Eine Analyse des resultierenden Filterausdrucks lässt eine Vorahnung aufkommen. Das Feld **LieferantID** enthält doch nur Zahlenwerte?

```
LieferantID LIKE 'E*'
```

```
Private Sub cmdSuchen_Click()
    Dim strLieferant As String
    strLieferant = Me!txtLieferant
    Me!sfmArtikel_Lookupfilter.Form.Filter = "LieferantID LIKE '" & strLieferant & "'"
    Me!sfmArtikel_Lookupfilter.Form.FilterOn = True
End Sub
```

Listing 1: Erster Ansatz zum Filtern der angezeigten Werte im Kombinationsfeld

Es ist also klar: Auch, wenn das Kombinationsfeld die Hersteller anzeigt, sind dies nicht die Werte des an das Steuerelement gebundenen Feldes **LieferantID**. **LieferantID** gibt lediglich an, mit welchem Datensatz der Tabelle **tblLieferanten** der aktuelle Datensatz der Tabelle **tblArtikel** verknüpft ist. Das Feld enthält also, wenn wie in diesem Fall referenzielle Integrität definiert ist, mit Sicherheit nur Zahlenwerte.

Sie könnten nun den Wert **1** als Suchbegriff in das

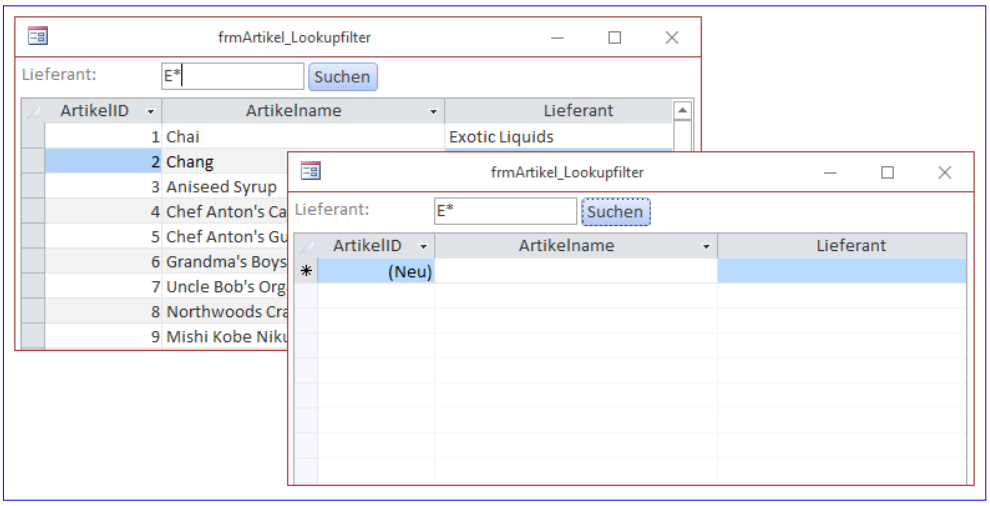

Bild 2: Erster, erfolgloser Anlauf

Textfeld **txtFirma** eingeben. Das Ergebnis wäre für den Benutzer eher noch verwirrender als das vorherige: Es erscheinen nämlich alle Datensätze mit dem Lieferanten **Exotic Liquids**.

Dieser hat wiederum, wie Sie vielleicht erahnen, den Wert **1** im Feld **LieferantID** der Tabelle **tblLieferanten**, also zeigt das Unterformular nach dem Filtern alle Datensätze der Tabelle **tblArtikel** an, die diesem Lieferanten zugeordnet sind.

Noch verwirrender wird es, wenn Sie beispielsweise den Wert **1*** eingeben. Dann erscheint das Ergebnis aus Bild 3.

Der Benutzer würde dies nun gar nicht mehr einordnen können, aber wir wissen: Das sind alle Artikel, deren Feld **LieferantID** einen Wert enthält, der mit **1** beginnt.

Filtern nach dem angezeigten Wert

Damit wenden wir uns dem eigentlichen Ziel dieses Beitrags zu: Wir wollen ja nicht nach den Fremdschlüsselwerten filtern, sondern nach dem Wert, der im Kombinationsfeld angezeigt wird. Hier gibt es verschiedene Ansätze, die wir uns nun ansehen.

Im Grunde ist es ganz einfach: Wir müssen lediglich die Abfrage in der Ereignisprozedur der Schaltfläche **cmdSuchen** anpassen, um das Ergebnis aus Bild 4 zu erhalten! Die entsprechende Zeile sieht dann wie folgt aus:

```
Me!sfmArtikel_Lookupfilter.Form.Filter = "LieferantID IN
(SELECT LieferantID FROM tblLieferanten WHERE Firma LIKE
'" & strLieferant & "')"
```

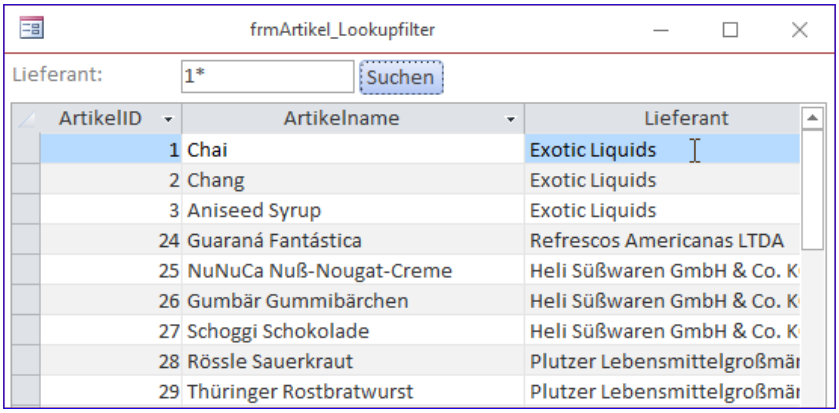

Bild 3: Alle Artikel, die zu einem Lieferanten gehören, der mit einer **LieferantID** wie **1*** beginnt

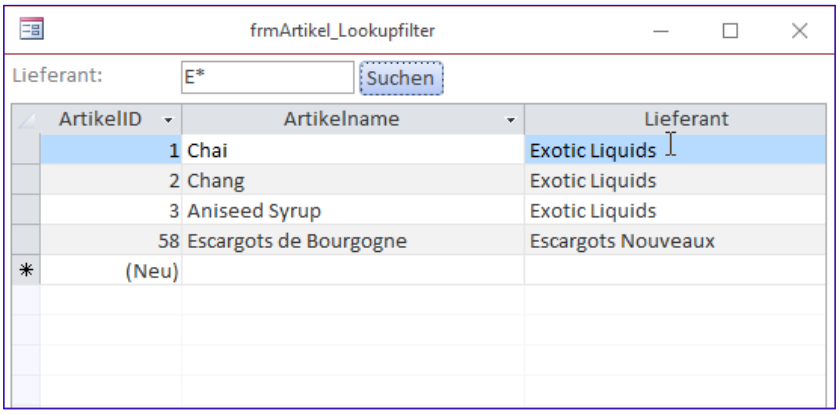

Bild 4: Diese Artikel wollen wir sehen.

Der nackte SQL-Ausdruck etwa nach Eingabe des Suchausdrucks **E*** sieht wie folgt aus:

```
LieferantID IN (
    SELECT LieferantID
    FROM tblLieferanten
    WHERE Firma LIKE 'E*'
)
```

Was bedeutet das nun? Wir verwenden hier immer noch das Feld **LieferantID** als Vergleichsfeld. Allerdings unterscheidet sich der Rest gravierend von der vorherigen Version.

Der erste Unterschied ist, dass wir nicht den Operator **LIKE** (oder auch das Gleichheitszeichen) verwenden, sondern den **IN**-Operator.

Dieser erwartet eine Menge von einem oder mehreren Vergleichswerten, die immer in Klammern angegeben werden müssen. Die Vergleichswerte können als kommaseparierte Liste angegeben werden, also etwa so:

```
IN (1, 2, 3)
```

In diesem Fall ist die Menge der Vergleichswerte jedoch dynamisch und von den Werten der Tabelle **tblLieferanten** abhängig, daher nutzen wir in Klammern eine **SELECT**-Abfrage:

```
SELECT LieferantID FROM tblLieferanten WHERE Firma LIKE 'E*'
```

Die Abfrage liefert alle Werte des Feldes **LieferantID** der Tabelle **tblLieferanten**, deren Feld **Firma** mit dem Buchstaben **E** beginnt (**LIKE E***) – hier steckt also nun ein weiterer Vergleichsoperator, der diesmal festlegt, mit welchem Suchbegriff die im Kombinationsfeld angezeigten Werte verglichen werden sollen.

Dies ist die einfachste Lösung, um ein Lookup-Kombinationsfeld nach den angezeigten Werten zu filtern.

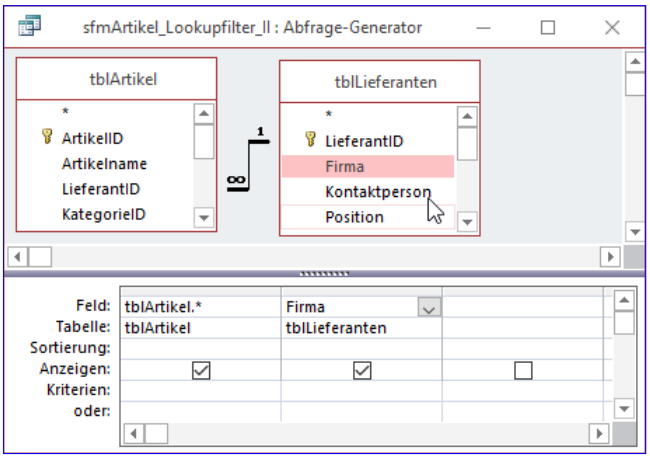

Bild 5: Erweiterte Datenherkunft des Formulars

Variante mit erweiterter Datenherkunft

Die zweite Variante setzt ein wenig Vorarbeit voraus, dafür ist die Gestaltung des Filter-Kriteriums wesentlich einfacher. Wir verwenden hier im Formular **frmArtikel_Lookupfilter_II** die Abfrage aus Bild 5 als Datenherkunft des Unterformulars.

Der Unterschied zum Einsatz der reinen Tabelle **tblArtikel** ist, dass wir die Tabelle **tblLieferanten** noch hinzugefügt und aus dieser Tabelle das Feld **Firma** in das Entwurfsraster gezogen haben.

Im Entwurf des Unterformulars müssen wir allerdings keine Änderungen vornehmen, denn wir wollen die **Firma** aus der Tabelle **tblLieferanten** ja nicht direkt in einem Textfeld, sondern nach wie vor als Kombinationsfeld anzeigen.

Hinweis: Der Name des Unterformular-Steuerelements im Formular **frmArtikel_Lookupfilter_II** lautet nach wie vor **sfmArtikel_Lookupfilter**, aber es zeigt das neue Unterformular **frmArtikel_Lookupfilter_II** an.

Die Ereignisprozedur, die durch die Schaltfläche **cmdSuchen** ausgelöst wird, sieht nun so aus:

```
Private Sub cmdSuchen_Click()
    Dim strLieferant As String
```

Schneller Filter

Formulare in der Datenblattansicht bieten alle Filter- und Sortiermöglichkeiten, die das Benutzerherz begehrt. Allerdings sind diese nicht unbedingt immer schnell erreichbar – hier und da könnte es noch ein wenig fixer gehen. Ein Beispiel ist ein Filter, der nur die Datensätze anzeigt, die den Wert des aktuell markierten Feldes im jeweiligen Feld enthalten. Wenn Sie also etwa eine Reihe von Artikeln anzeigen, die einer bestimmten Kategorie angehören und schnell nur noch die Artikel dieser Kategorie sehen wollen, benötigen Sie dazu mehrere Mausklicks. Dieser Beitrag zeigt, wie Sie verschiedene Suchen mit einem einfachen Klick auf eine Schaltfläche erledigen.

Die Datenblattansicht von Access-Formularen bietet eine Reihe von Möglichkeiten, schnell nach Daten zu suchen oder diese zu sortieren.

Dazu klicken Sie einfach auf das nach unten zeigende Dreieck rechts im Spaltenkopf der jeweiligen Spalte. Hier sehen Sie auf Anhieb zwei Einträge zum Sortieren in verschiedenen Richtungen oder zum Selektieren verschiedener, im aktuellen Feld enthaltener Werte (s. Bild 1).

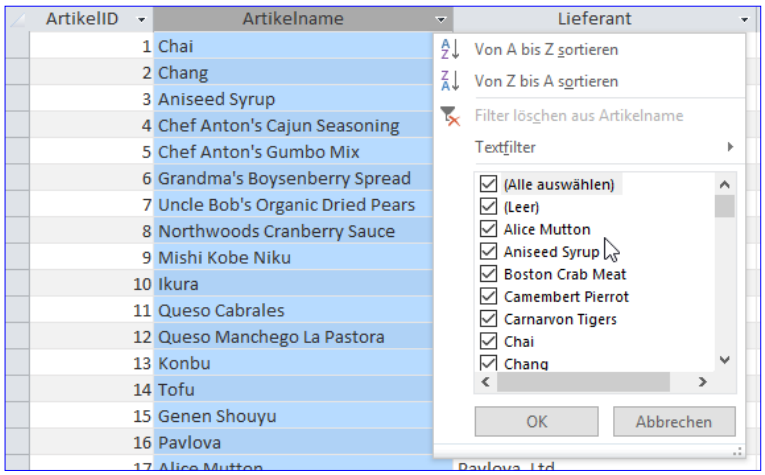

Bild 1: Filtern nach allen vorhandenen Werten

Der Untereintrag **Textfilter** liefert etwa für Textfelder weitere Möglichkeiten: Hier können Sie beispielsweise nach Datensätzen suchen, deren markiertes Feld einen benutzerdefinierten Wert enthält. Bei Textfeldern können hier etwa die Vergleichsoperatoren **Gleich**, **Nicht gleich**, **Beginnt mit**, **Beginnt nicht mit**, **Enthält** und weitere verwendet werden (s. Bild 2). Andere Felddatentypen halten dem Datentyp entsprechende Vergleichskriterien bereit.

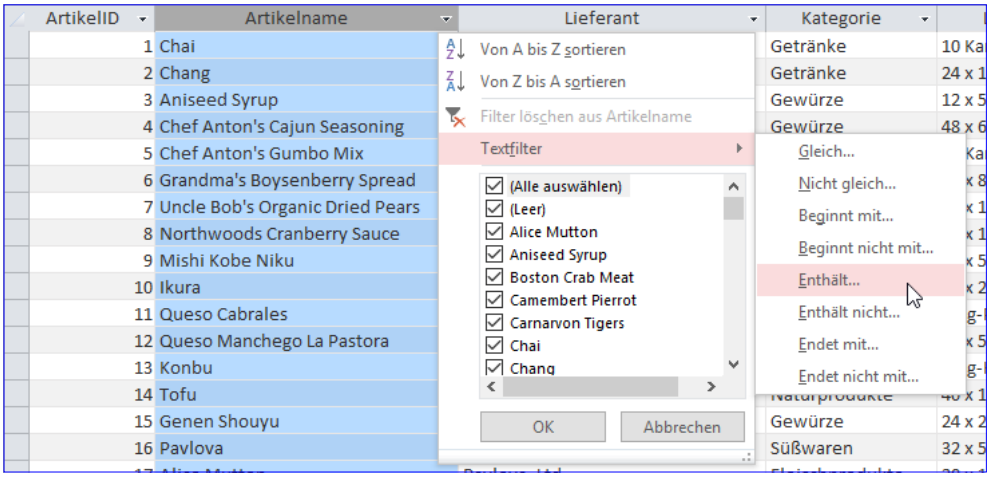

Bild 2: Filtern nach benutzerdefinierten Vergleichswerten

Datensätze mit gleichem Feldwert finden

Was aber hier fehlt, ist die einfache Möglichkeit, schnell alle Einträge anzuzeigen, die den gleichen Wert im zurzeit markierten Feld aufweisen wie der aktuelle Datensatz. Und genau diese Funktion wollen wir nun nachrüsten. Für das Feld Artikelna-

me macht dies natürlich recht wenig Sinn, aber beim Lieferanten oder bei der Kategorie finden sich schnell Einsatzmöglichkeiten.

Warum nicht beim Artikelnamen? Nun: Dabei handelt es sich um ein Feld mit einem eindeutigen Index. Da nur jeweils ein Datensatz mit dem aktuellen Wert vorhanden ist, macht es wenig Sinn, danach zu filtern ... außer natürlich, wenn Sie etwa aus Gründen der Übersicht nur diesen einen Datensatz anzeigen möchten. Also nehmen wir diese einfache Variante einfach mit hinzu.

Später wollen wir jedoch gerade für Textfelder noch eine schnelle Filterfunktion hinzufügen, mit der Sie sogar alle Datensätze anzeigen können, die den aktuell markierten Wert enthalten.

Der Filter soll dann wie im Beispiel aus Bild 3 funktionieren. Der Benutzer markiert den Wert, nach dem die Daten gefiltert werden sollen, und klickt auf die Schaltfläche **Schneller Filter**. Daraufhin werden alle Datensätze ausgeblendet, deren Inhalt im betroffenen Feld nicht mit dem Vergleichswert übereinstimmt. Eine weitere Schaltfläche soll den Filter wieder deaktivieren.

Schaltfläche zum schnellen Filtern

Beginnen wir doch einfach mit einer Schaltfläche, die wir **cmdSchnellerFilter** nennen und mit der Beschriftung **Schneller Filter** versehen. Diese soll die Datenherkunft des Unterformulars in dem Formular, in dem sich die Schaltfläche befindet, nach dem Wert des zuvor markierten Feldes filtern. Wie sich herausstellt, ist dies gar nicht so einfach, denn wir finden erst gar nicht heraus, welches Feld gerade überhaupt markiert war.

Unsere erste Idee war es nämlich, das Steuerelement mit dem besagten Filtervergleichswert einfach über den Ausdruck **Screen.PreviousControl.Name** zu ermitteln und diesen per **Debug.Print** im Direktbereich auszugeben. Dazu haben wir die **Beim Klicken**-Ereignisprozedur der Schaltfläche **cmdSchnellerFilter** wie folgt ausgestattet:

```
Private Sub cmdSchnellerFilter_Click()
    Debug.Print Screen.PreviousControl.Name
End Sub
```

Das Ergebnis lieferte aber leider nicht das gesuchte Steuerelement, sondern den Namen des Unterformulars:

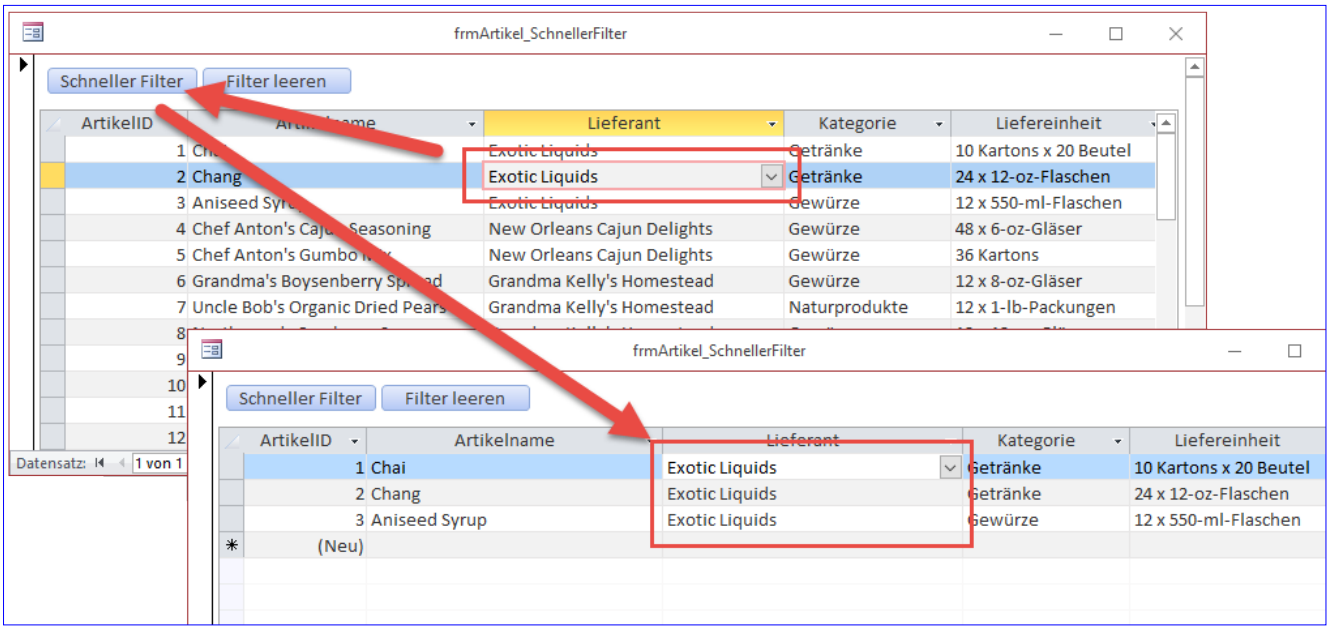

Bild 3: So soll der Filter nach einem Feldwert arbeiten.

sfmArtikel_SchnellerFilter

Zuletzt aktives Feld ermitteln

Wir stehen nun also vor dem Problem, zwar zum Zeitpunkt den Inhalt des zuletzt aktivierten Feldes im Unterformular zu benötigen, dieses aber nicht mehr zu kennen.

Es gibt nun diverse Möglichkeiten, die gewünschte Information zu erhalten. Eine davon lautet, irgendwo eine Variable vorzuhalten, die wir mit einem Verweis auf das jeweils aktive Steuerelement des Unterformulars füllen und dann beim Mausklick auf die Schaltfläche **cmdSchnellerFilter** über diese Variable auf das Feld und seinen Inhalt zugreifen. Das ist allerdings mit einigem Aufwand verbunden, wenn wir es auf dem einfachen Weg erledigen. Dieser sieht vor, eine Variable zum Speichern des zuletzt verwendeten Feldes im Klassenmodul des Hauptformulars zu deklarieren. Außerdem legen wir für jedes Steuerelement im Unterformular eine Ereigniseigenschaft namens **Bei Fokuserhalt** an und hinterlegen dafür eine Ereignisprozedur, welche einen Verweis auf das jeweilige Steuerelement in die Variable im Klassenmodul des Hauptformulars einträgt.

Mit der Ereignisprozedur **Beim Klicken** der Schaltfläche **cmdSchnellerFilter** können Sie dann aus der Variablen den Wert ermitteln und nach dem Feld, an welches das Steuerelement aus der Variablen gebunden ist, filtern.

Wir müssen nur für jedes betroffene Steuerelement im Unterformular eine entsprechende Ereignisprozedur für das Ereignis **Bei Fokuserhalt** hinterlegen. Und ebenso für alle Steuerelemente der Unterformulare in anderen Formularen, die Sie mit der Funktion ausstatten möchten.

Lösung mit Klasse

Nun ist Access im Unternehmen aber weniger bekannt dafür, den Leser mit Fleißarbeit auszustatten. Wir suchen eher nach einer Lösung, die der Leser in wenigen Minuten implementieren kann. Also greifen wir, wie schon ein paar Mal geschehen, auf Klassenmodule zurück, in denen wir die gewünschte Funktionalität unterbringen. Das Klassenmodul des Hauptformulars soll nur mit wenigen Zeilen Code ausgestattet werden, die zum größten Teil in der Ereignisprozedur **Form_Load** landen. Insgesamt sieht der benötigte Code wie folgt aus. Als Erstes benötigen wir eine Objektvariable, welche den Verweis auf die gleich noch erläuterte Klasse **clsFastFilter** aufnimmt:

```
Dim objFastFilter As clsFastFilter
```

Als Nächstes folgt dann die Ereignisprozedur **Form_Load**, die wir folgt aussieht:

```
Private Sub Form_Load()
    Set objFastFilter = New clsFastFilter
    With objFastFilter
        Set .Subform = Me!sfmArtikel_SchnellerFilter.Form
        Set .FastFilterButton = Me!cmdSchnellerFilter
    End With
End Sub
```

Sie erstellt zunächst ein neues Objekt auf Basis der Klasse **clsFastFilter** und speichert den Verweis darauf in der Variablen **objFastFilter**. Dann weist sie den beiden Eigenschaften **Subform** und **FastFilterButton** Verweise auf das Unterformular mit den zu durchsuchenden Datensätzen (Achtung: **<Unterformularname>.Form** liefert die richtige Referenz) und auf die Schaltfläche zu, welche den Filter erstellen soll. Zur Abrundung fügen Sie noch eine Ereignisprozedur für die Schaltfläche mit der Beschriftung **Filter leeren** hinzu, welche schlicht den Filter des Unterformulars leert und somit wieder alle Datensätze anzeigt:

```
Private Sub cmdFilterLeeren_Click()
    Me!sfmArtikel_SchnellerFilter.Form.Filter = ""
End Sub
```

Wichtige Vorbereitung

Wenn Sie Klassen erstellen, die Objekte wie etwa Formulare oder die enthaltenen Steuerelemente referenzieren und deren Ereignisse implementieren wollen, muss für das

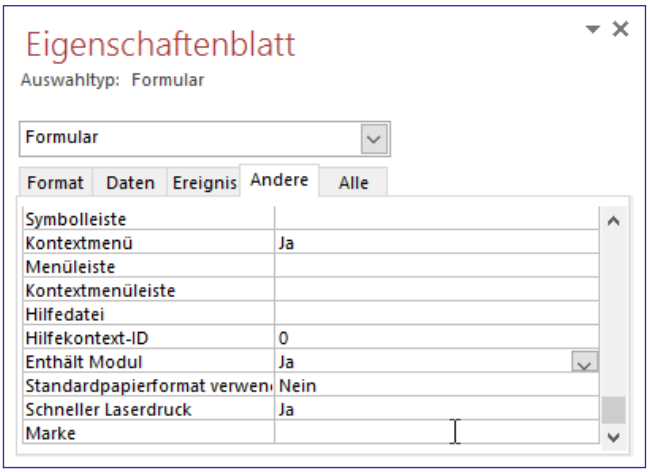

Bild 4: Hinzufügen eines Klassenmoduls per Eigenschaft

jeweilige Formular (und somit auch für Unterformulare) auch ein Klassenmodul vorliegen! In unserem Fall haben wir etwa für das Unterformular mit dem Datenblatt noch kein Klassenmodul angelegt. Dies erfolgt automatisch, sobald Sie für eine der Ereigniseigenschaften des Formulars ein Ereignis anlegen und dieses über die Schaltfläche mit den drei Punkten im VBA-Editor öffnen. Sie können dies aber auch durch einfaches Einstellen der Eigenschaft **Enthält Modul** erledigen. Diese Eigenschaft finden Sie im Reiter **Andere** des jeweiligen Formulars (s. Bild 4).

Die Klasse clsFastFilter

Diese Klasse ist die Steuerzentrale der Lösung. Sie nimmt die Verweise auf die beteiligten Elemente entgegen, also das Unterformular sowie die Schaltfläche zum Auslösen des Filters. Das Unterformular wird mit der folgenden Variablen referenziert, die im Kopf des Klassenmoduls deklariert wird:

```
Private m_Subform As Form
```

Die Schaltfläche zum Auslösen des Filters landet per Verweis in dieser Variablen:

```
Private WithEvents m_FastFilterButton As CommandButton
```

Die Variable ist mit dem Schlüsselwort **WithEvents** deklariert, was dafür sorgt, dass wir in diesem Klassenmodul Ereignisprozeduren für das Steuerelement implementieren können. Desweiteren benötigen wir noch zwei weitere Variablen. Die erste ist eine Collection und nimmt die Instanzen der Wrapper-Klassen auf, von denen wir für jedes filterbare Steuerelement eine erstellen und in die Collection schreiben:

```
Private colControls As Collection
```

Außerdem brauchen wir noch die besagte Variable, welche das zuletzt durch den Benutzer angeklickte Steuerelement im Unterformular aufnimmt:

```
Private m_CurrentControl As Control
```

Damit die Wrapper-Objekte, die jeweils eines der gebundenen Steuerelemente im Unterformular aufnehmen, einen Verweis auf das zuletzt durch den Benutzer angeklickte Element in die Variable **m_CurrentControl** schreiben können, stellen wir eine **Property Set**-Methode in der Klasse **clsFastFilter** bereit, die wie folgt aussieht:

```
Public Property Set CurrentControl(ctl As Control)
    Set m_CurrentControl = ctl
End Property
```

Damit wir die Funktion, welche die Schaltfläche **cmdFastFilter** auslöst, auch in der Klasse **clsFastFilter** unterbringen können und diese nicht in jedem neuen Formular erneut schreiben müssen, füllen wir die lokale Variable **m_FastFilterButton** über die **Property Set**-Prozedur **FastFilterButton** mit einem Verweis auf die jeweilige Schaltfläche. Die Prozedur sieht so aus:

```
Public Property Set FastFilterButton(cmd As CommandButton)
    Set m_FastFilterButton = cmd
    cmd.OnClick = "[Event Procedure]"
End Property
```

Sie erwartet einen Verweis auf die Schaltfläche als Parameter und trägt diese in die Variable **m_FastFilterButton**

ein. Außerdem legt sie noch fest, dass in diesem Klassenmodul eine Implementierung des Ereignisses **OnClick** vorliegen könnte und beim Auslösen entsprechend berücksichtigt werden soll.

Die Hauptarbeit in der Klasse übernimmt die **Property Set**-Methode **Subform**, mit welcher die **Form_Load**-Ereignisprozedur der Klasse **clsFastFilter** das zu verwendende Unterformular zuweist. Sie erwartet das Formular als Parameter und sieht wie folgt aus:

```
Public Property Set Subform(frm As Form)
    Dim ctl As Control
    Dim objFastFilterControl As clsFastFilterControl
    Dim strControlSource As String
    Set m_Subform = frm
    Set colControls = New Collection
    For Each ctl In m_Subform.Controls
        strControlSource = ""
        On Error Resume Next
        strControlSource = ctl.ControlSource
        On Error GoTo 0
        If Len(strControlSource) > 0 Then
            Set objFastFilterControl = _
                New clsFastFilterControl
            With objFastFilterControl
                Set .Control = ctl
                Set .MyParent = Me
                colControls.Add objFastFilterControl
            End With
        End If
    Next ctl
End Property
```

Die Prozedur stellt zunächst die Variable **m_Subform** auf das übergebene Formular ein. Dann instanziert sie eine neue Collection und speichert diese in der Variablen **colControls**. Schließlich durchläuft sie alle Steuerelemente des mit **frm** angegebenen Unterformulars. In der dafür verwendeten **For Each**-Schleife prüft die Prozedur, ob es sich beim aktuell durchlaufenen Steuerelement überhaupt um ein gebundenes Steuerelement handelt. Andere Steuerelemente wie etwa Bezeichnungsfelder brauchen wir gar nicht zu berücksichtigen. Dazu leert die Prozedur eine Variable namens **strControlSource** und versucht dann, diese bei deaktivierter Fehlerbehandlung mit dem Wert der Eigenschaft **ControlSource** des aktuellen Steuerelements aus **ctl** zu füllen. Enthält **strControlSource** danach keine leere Zeichenkette, handelt es sich um ein gebundenes Steuerelement und es kann in Form des Wrapper-Objekts auf Basis der Klasse **clsFastFilterControl** referenziert und zur Collection **colControls** hinzugefügt werden. Dann erstellt die Prozedur ein

```
Private Sub m_FastFilterButton_Click()
    Dim rst As DAO.Recordset
    Dim fld As DAO.Field
    Dim strControlSource As String
    strControlSource = m_CurrentControl.ControlSource
    Set rst = m_Subform.Recordset
    Select Case rst.Fields(strControlSource).Type
        Case dbText
            m_Subform.Filter = strControlSource & " = '" _
                & Replace(m_CurrentControl.Value, "'", "''") & "'"
            m_Subform.FilterOn = True
       Case dbDate
            m_Subform.Filter = strControlSource & " = " & CDbl(m_CurrentControl.Value)
            m_Subform.FilterOn = True
        Case Else
            m_Subform.Filter = strControlSource & " = " & m_CurrentControl.Value
            m_Subform.FilterOn = True
    End Select
End Sub
```

Listing 1: Implementierung des **Beim Klicken**-Ereignisses der Filter-Schaltfläche

neues Objekt auf Basis von **clsFastFilterControl**, füllt dessen Eigenschaft **Control** mit einem Verweis auf das aktuelle Steuerelement und die Eigenschaft **MyParent** mit einem Verweis auf sich selbst, also die Instanz der Klasse **clsFastFilter**. Wozu dies nötig ist, erfahren Sie gleich bei der Beschreibung der Klasse **clsFastFilterControl**. Nun müssen wir nur noch dafür sorgen, dass das Wrapper-Objekt, das ja lokal innerhalb der **Property Set**-Prozedur deklariert wurde und anderenfalls mit dem Ende der Prozedur erlöschen würde, nicht im Nirwana verschwindet. Dazu fügt die Prozedur es zur Collection **colControls** hinzu.

Gehen wir an dieser Stelle vereinfachend davon aus, dass die Wrapper-Objekte beim Anklicken durch den Benutzer dafür sorgen, dass ein Verweis auf das angeklickte gebundene Steuerelement in der Variablen **m_CurrentControl** landet.

Dann können wir uns die Ereignisprozedur ansehen, die durch einen Mausklick auf die mit der Variablen **m_FastFilterButton** referenzierte Schaltfläche ausgelöst wird. Die Prozedur sieht wie in Listing 1 aus. Sie liest zunächst den Namen des Feldes, das dem mit der Variablen **m_CurrentControl** referenzierten und zuletzt durch den Benutzer angeklickten Steuerelement angehört, in die Variable **strControlSource** ein.

Dann füllt sie eine **Recordset**-Variable namens **rst** mit dem Recordset des zu filternden Unterformulars. Sie ermittelt dann für das Element der **Fields**-Auflistung mit dem Namen aus **strControlsource** den Datentyp und gleicht diesen in einer **Select Case**-Bedingung mit verschiedenen Werten ab. Wir haben hier nur **dbText** für Textfelder, **dbDate** für Datumsfelder und alle übrigen Datentypen untersucht, obwohl hier noch weitere Unterscheidungen möglich wären. Im Falle des Wertes **dbText** stellt die folgende Anweisung einen Vergleichsausdruck zusammen, der aus dem Feldnamen aus **strControlSource**, dem Gleichheitszeichen und dem in Hochkommata eingefassten Wert des Steuerelements aus **m_CurrentControl** besteht, also etwa **Artikelname = 'Chai'**. Dieser Vergleichsausdruck landet in der Eigenschaft **Filter** des Unterformulars. Das Einstellen einer weiteren Eigenschaft namens **FilterOn** auf den Wert **True** sorgt schließlich dafür, dass der Filter auch auf die aktuellen Daten angewendet wird.

Wichtig ist an dieser Stelle noch der Hinweis auf das eventuelle Auftreten von Hochkommata oder Anführungszeichen innerhalb der Zeichenkette. Diese führen beim Zusammensetzen des Filterausdrucks zu Fehlern, was nicht geschieht, wenn Sie diese verdoppeln – in diesem Fall durch entsprechenden Einsatz der **Replace**-Funktion.

Im Falle eines Datums würde, wenn wir dieses einfach wie eine Zahl behandeln, ein Ausdruck wie **Auslaufdatum = 1.1.2016** verwendet, was zu einem Fehler führt. Daher wandeln wir den Wert des Datumsfeldes zuvor mit der **CDbl**-Funktion in einen **Double**-Wert um, der dann keinen Fehler mehr auslöst.

Für alle übrigen Datentypen soll einfach das Feld mit dem jeweiligen Wert verglichen werden.

Die Klasse clsFastFilterControl

Es fehlt noch ein genauerer Blick auf die Klasse **clsFastFilterControl**. Diese wird für jedes gebundene Steuerelement im Unterformular genau einmal instanziert, denn wir wollen ja für jedes dieser Steuerelemente das Ereignis **Bei Fokuserhalt** implementieren, um dort das aktuelle Steuerelement in der Variablen **m_CurrentControl** des Objekts basierend auf der Klasse **clsFastFilter** zu speichern.

Die Klasse **clsFastFilterControl** soll zunächst einmal einen Verweis auf das erstellende Objekt speichern, hier also das Objekt auf Basis der Klasse **clsFastFilter**. Dazu verwenden wir die folgende Variable:

```
Private m_Parent As clsFastFilter
```

Die benötigte **Property Set**-Methode sieht so aus:

```
Public Property Set MyParent(obj As clsFastFilter)
    Set m_Parent = obj
End Property
```

Außerdem wollen wir bei den Steuerelementen des Datenblatts unter den Steuerelementtypen Textfeld, Kombinationsfeld und Kontrollkästchen unterscheiden (andere Steuerelemente machen im Datenblatt auch keinen Sinn). Daher deklarieren wir die folgenden drei Variablen innerhalb der Klasse, da wir ja noch nicht wissen, welchen Steuerelementtyp die Klasse aufnehmen soll:

```
Private WithEvents txt As TextBox
Private WithEvents cbo As ComboBox
Private WithEvents chk As CheckBox
```

Nun fehlt noch die **Property Set**-Methode, die das zu untersuchende Steuerelement entgegennimmt und einige weitere Schritte durchführt. Diese sieht wie folgt aus:

```
Public Property Set Control(ctl As Control)
    Select Case ctl.ControlType
        Case acTextBox
            Set txt = ctl
            txt.OnGotFocus = "[Event Procedure]"
        Case acComboBox
            Set cbo = ctl
            cbo.OnGotFocus = "[Event Procedure]"
        Case acCheckBox
            Set chk = ctl
            chk.OnGotFocus = "[Event Procedure]"
    End Select
End Property
```

Die Prozedur prüft zunächst anhand der Eigenschaft **ControlType**, um welchen Typ es sich handelt. Im Falle von **acTextBox** füllt sie beispielsweise die Variable **txt** mit einem Verweis auf das mit dem Parameter **ctl** gelieferte Steuerelement. Außerdem stellt sie die Eigenschaft **OnGotFocus** auf den Wert **[Event Procedure]** ein, was im Eigenschaftsfenster des Formulars dem Einstellen der Eigenschaft **Bei Fokuserhalt** auf **[Ereignisprozedur]** entspricht. Für die beiden anderen Typen, die mit den Konstanten **dbComboBox** und **dbCheckBox** geprüft werden, verläuft dies ähnlich. Schließlich müssen wir noch das Ereignis **Bei Fokuserhalt** für die drei deklarierten Objektvariablen für die drei Steuerelementtypen **Textfeld**, **Kombinationsfeld** und **Kontrollkästchen** implementieren, was wir wie folgt erledigen:

```
Private Sub cbo_GotFocus()
    Set m_Parent.CurrentControl = cbo
End Sub

Private Sub chk_GotFocus()
    Set m_Parent.CurrentControl = chk
End Sub

Private Sub txt_GotFocus()
    Set m_Parent.CurrentControl = txt
End Sub
```

In diesen drei Prozeduren weisen wir jeweils der Eigenschaft **CurrentControl** des in **m_Parent** gespeicherten Objekts (also unsere Instanz der Klasse **clsFastFilter**) eine Referenz auf das auslösende Steuerelement zu, die dann wiederum in der Variablen **m_CurrentControl** der Klasse **clsFastFilter** landet. Damit ist die Programmierung der grundlegenden Funktion bereits erledigt: Wir können das Datenblatt nun nach den Werten einzelner Felder filtern.

Filtern nach »Enthält ...«

Für den Einsatz mit Textfeldern wäre es noch interessant, wenn wir auch Vergleiche mit Teilausdrücken erlauben würden. Wie wäre es etwa, wenn Sie nur ein paar Buchstaben des Feldinhalts eines Textfeldes markieren und danach in allen Datensätzen suchen wollten? Das wäre schon praktisch, also machen wir uns an die Arbeit. Zu diesem Zweck fügen wir dem Formular eine weitere Schaltfläche namens **cmdSchnellerFilterTeilausdruck** hinzu. Die Prozedur **Form_Load** erweitern wir entsprechend um die Zuweisung dieser Schaltfläche zur gleich

Access per URL starten

Ein interessanter Anwendungsfall ist das Starten einer Access-Anwendung über den Aufruf einer URL etwa im Browser – zum Beispiel von einer Internetanwendung aus, die einen speziell vorbereiteten Link enthält. Die Betätigung dieses Links soll dann Access starten, die gewünschte Datenbank öffnen und gegebenenfalls sogar noch einen oder mehrere Parameter an die Anwendung übergeben. Wie das gelingt, schauen wir uns im vorliegenden Beitrag an.

Im konkreten Fall schickte uns ein Leser die Anforderung, bei der eine Internetanwendung eine Möglichkeit bietet, über eine Schaltfläche beziehungsweise einen Link eine eigene URL anzugeben, die bei Betätigung geöffnet wird.

Lässt sich dies realisieren, und wenn ja, wie? Die Lösung ist nicht besonders nahe liegend, denn wir können die Anwendung ja nicht über einen herkömmlichen Batch-Befehl starten, wie es etwa über eine Dateiverknüpfung erfolgt.

Der Trick ist, dass wir einen eigenen Datei-Handler definieren, wie er normalerweise in die URL-Leiste des Browsers eingegeben wird, also beispielsweise **http:** oder **ftp:**. Dem lassen wir dann die Daten folgen, die als Parameter an die aufzurufende Access-Datenbank übergeben werden sollen. Aber wie definieren, dass etwa bei Übergabe des Handlers **accessdb:** auch wirklichen eine Access-Datenbank geöffnet wird, und welche dies ist? Dies legen wir in der Registry fest, und zwar etwa wie in Bild 1. Wir fügen also dem Zweig HKEY_CLASSES_ROOT ein neues Element namens **accessdb** hinzu. Dafür legen wir als Standardwert **URL:accessdb Protocol** fest und ein weiteres Element namens **URL Protocol**, das aber leer bleibt. Darunter folgen drei weitere verschachtelte Elemente, nämlich **shell**, **open** und **command**. Letzteres erhält die auszuführende Batch-Anweisung, in diesem Fall den Aufruf der Anwendung **MSACCESS.EXE** und die Angabe der zu öffnenden Datenbankdatei mit Pfad. Danach folgt noch der Ausdruck **/cmd %1**, den wir später erläutern:

```
"C:\Program Files (x86)\Microsoft Office\root\Office16\
MSACCESS.EXE" "C:\Users\User\Dropbox\Daten\URLTest.accdb"
/cmd %1
```

Um diese Einträge automatisch per Doppelklick auf eine Datei namens **accessdb.reg** zur Registry hinzuzufügen, füllen Sie diese Datei mit dem Code aus Listing 1.

Wenn Sie nun im Browser die Adresse **accessdb:** gefolgt von einem beliebigen Ausdruck eingeben (oder auch nur **accessdb:**), wird die Access-Datenbank geöffnet und in Access angezeigt. Beim ersten Aufruf kann es vorkom-

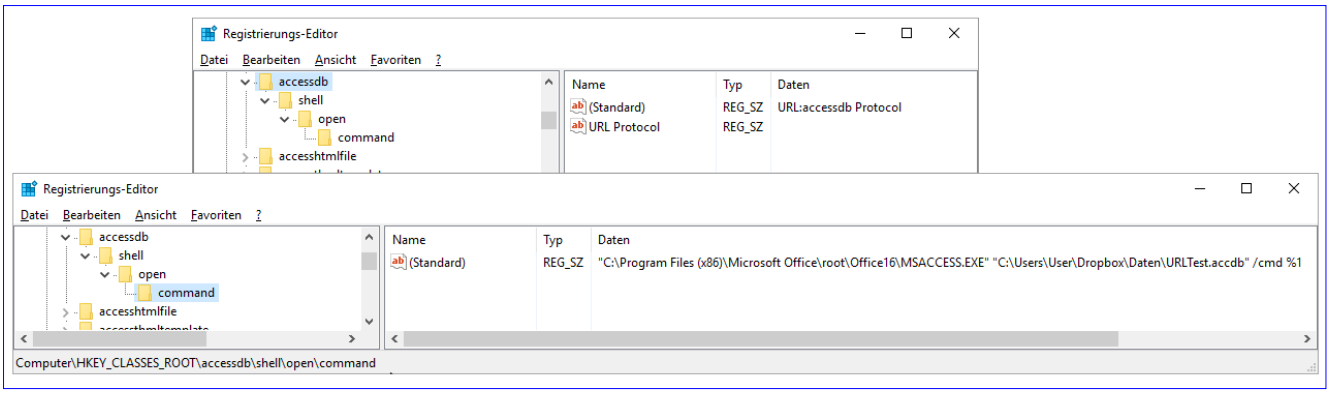

Bild 1: Einstellungen in der Registry

men, dass die Meldung aus Bild 2 erscheint und Sie diese zunächst noch bestätigen müssen. Wenn Sie hier die Option **Auswahl für accessdb-Links speichern** aktivieren, erscheint diese Meldung anschließend nicht mehr.

```
Windows Registry Editor Version 5.00

[HKEY_CLASSES_ROOT\accessdb]
@="URL:accessdb Protocol"
"URL Protocol"=""

[HKEY_CLASSES_ROOT\accessdb\shell]

[HKEY_CLASSES_ROOT\accessdb\shell\open]

[HKEY_CLASSES_ROOT\accessdb\shell\open\command]
@="\"C:\\Program Files (x86)\\Microsoft Office\\root\\Office16\\MSACCESS.EXE\"
\"C:\\Users\\User\\Dropbox\\Daten\\URLTest.accdb\" /cmd %1"
```

Listing 1: Inhalt der Datei **accessdb.reg** zum Anlegen der benötigten Registrierungswerte

Dies funktionert wie beschrieben mit Firefox. Unter Microsoft Edge muss die Meldung jedes Mal erneut bestätigt werden, Chrome will Access gar nicht öffnen. Der Internet Explorer bietet auch die Möglichkeit, die Einstellung zu speichern. Wer die Registry bearbeitet, um vom Browser aus eine Access-Datenbank zu starten, wird sich aber vermutlich auch mit der Auswahl eines der funktionierenden Browser anfreunden können.

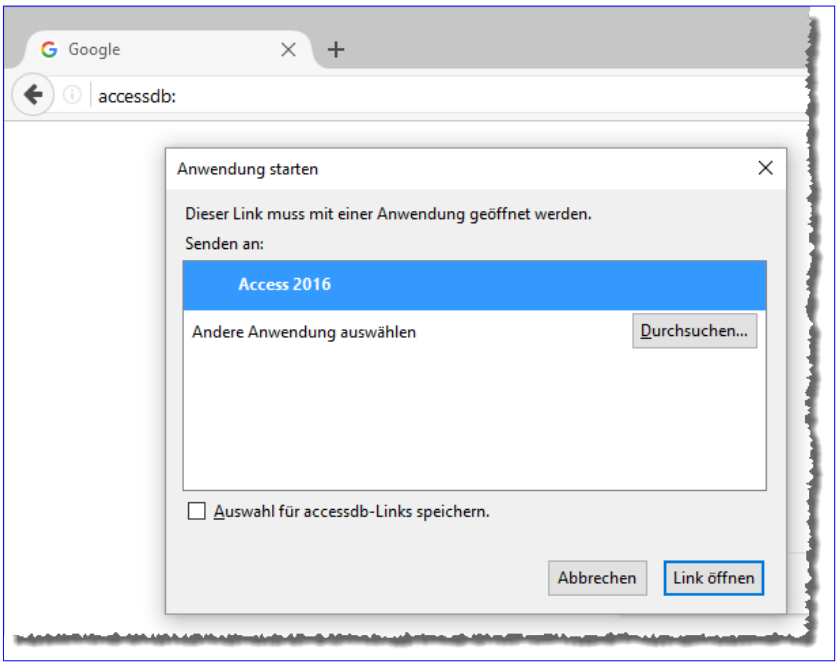

Bild 2: Warnmeldung beim Ausführen von Access via Browser

Parameter übergeben

Nun kommen wir zum interessanten Teil, denn wir wollen ja nicht nur einfach irgendeine Access-Datenbank öffnen, sondern gegebenenfalls auch noch Parameter übergeben.

Die Vorbereitungen, um diese Parameter zu übergeben, haben wir in der Registry bereits getroffen. Dort haben wir nämlich den Ausdruck **/cmd %1** an den Batch-Aufruf angehängt. **%1** entspricht dabei der kompletten URL. Wenn Sie die Datenbank also mit dem Link **accessdb:test** öffnen, wird **accessdb:test** auch als **cmd**-Parameter übergeben.

Den **cmd**-Parameter können Sie in der geöffneten Anwendung dann mit der Funktion **Command()** auslesen. Wenn Sie die Anwendung also mit dem Link geöffnet haben, wechseln Sie testweise zum Direktbereich des VBA-Editors (**Strg + G**) und geben dort den folgenden Befehl ein:

```
Debug.Print Command()
  accessdb:test
```

Den Parameter wollen Sie natürlich nun noch professioneller auswerten – beispielsweise, wenn der Aufruf die Angabe etwa einer Datensatznummer enthält und die

RDBMS-Zugriff per VBA: Daten bearbeiten

Im Beitrag »RDBMS-Zugriff per VBA: Verbindungen« haben wir die Grundlage für den Zugriff auf SQL Server-Datenbanken geschaffen. Zudem zeigt der Beitrag »RDBMS-Zugriff per VBA: Daten abfragen«, wie Sie die Daten einer SQL Server-Datenbank ermitteln. Im vorliegenden Teil dieser Beitragsreihe erfahren Sie nun, wie Sie die Daten einer SQL Server-Datenbank bearbeiten.

Aktionsabfragen

Aktionsabfragen sind Abfragen, die Daten ändern – also Lösch-, Aktualisierungs- und Anfügeabfragen. In reinen Access-Datenbanken führen Sie solche Abfragen aus, indem Sie diese mit dem Abfrage-Entwurf erstellen und direkt ausführen oder per VBA aufrufen oder indem Sie die gewünschte Abfrage als SQL-Ausdruck per Code zusammenstellen und dann mit der **Execute**-Methode des **Database**-Objekts ausführen. Für SQL Server-Daten gibt es die folgenden Arten der Ausführung:

- Erstellen einer Aktionsabfrage in Access, die sich auf die Daten einer per ODBC verknüpften Tabelle des SQL Servers bezieht,
- Erstellen einer Pass-Through-Abfrage, welche die Aktionsabfrage enthält und diese direkt an den SQL Server übermittelt,
- Erstellen einer gespeicherten Prozedur, welche die Aktionsabfrage enthält und die notwendigen Parameter entgegen nimmt – also beispielsweise die ID eines zu löschenden Datensatzes –, und die über eine Pass-Through-Abfrage aufgerufen wird.

Wenn es um die Performance geht, ist die erste Variante die langsamste, die zweite Version ist etwas schneller und die dritte Version ändert die Daten in der Regel am schnellsten. Aus diesem Grund schauen wir uns nachfolgend lediglich die zweite und die dritte Variante an.

Datensatz löschen per SQL

Bei der ersten Variante legen Sie eine Pass-Through-Abfrage mit der auszuführenden **DELETE**-Anweisung an (s. Bild 1).

Dazu sind folgende Schritte nötig:

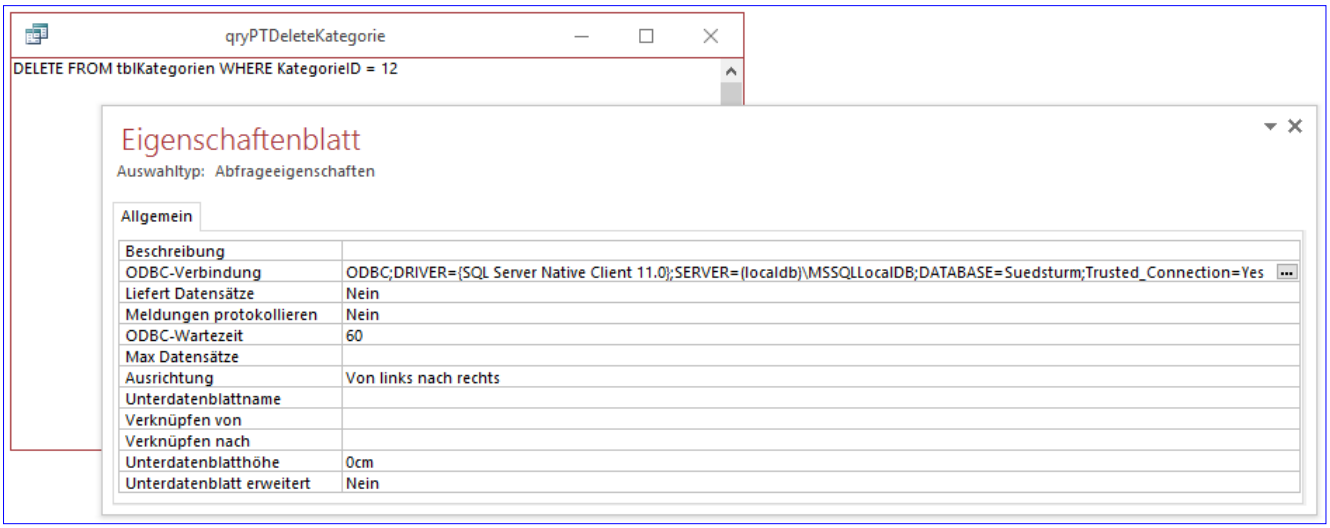

Bild 1: Die neue PassThrough-Abfrage zum Löschen eines Datensatzes im SQL Server

- Erstellen einer neuen, leeren Abfrage und Schließen des Dialogs **Tabelle anzeigen**
- Wechseln des Abfragetyps auf **Pass-Through**
- Einstellen der Eigenschaft **ODBC-Verbindung** auf die gewünschte Verbindungszeichenfolge (hier **ODBC;DRIVER={SQL Server Native Client 11.0};SERVER=(localdb)\MSSQLLocalDB;DATABASE=Suedsturm;Trusted_Connection=Yes**)
- Einstellen der Eigenschaft **Liefert Datensätze** auf **Nein**
- Eintragen der **DELETE**-Anweisung

Die **DELETE**-Anweisung soll in unserem Fall wie folgt lauten:

```
DELETE FROM tblKategorien WHERE KategorieID = 12
```

Die Abfrage können Sie dann per VBA mit einer einzigen Anweisung ausführen:

```
CurrentDb.Execute "qryPTDeleteKategorie"
```

Sie können auch die Variante mit dem **QueryDefs**-Objekt verwenden:

```
CurrentDb.QueryDefs("qryPTDeleteKategorie").Execute
```

Damit haben Sie allerdings noch nicht viel gewonnen: Die Anweisung löscht ja nur genau den Datensatz, dessen ID Sie als Kriterium angegeben haben. Immerhin haben wir aber bereits eine Abfrage erstellt, die den richtigen Typ aufweist, die Verbindungszeichenfolge enthält und deren Eigenschaft **Liefert Datensätze** auf **Nein** eingestellt ist. Diese nutzen wir nun, um gezielt einen bestimmten Datensatz zu löschen. Die folgende Prozedur (wie auch die weiteren Beispiele im Modul **mdlRDBMSZugriff_DatenBearbeiten**) erwartet den Primärschlüsselwert des zu löschenden Datensatzes als Parameter:

```
Public Sub KategorieLoeschen_PT(lngKategorieID As Long)
    Dim db As DAO.Database
    Dim qdf As DAO.QueryDef
    Set db = CurrentDb
    Set qdf = db.QueryDefs("qryPTDeleteKategorie")
    qdf.SQL = "DELETE FROM dbo.tblKategorien " _
                & "WHERE KategorieID = " & lngKategorieID
    qdf.Execute
    Set qdf = Nothing
    Set db = Nothing
End Sub
```

Mit dieser Prozedur referenzieren wir die soeben erstellte Abfrage **qryPTDeleteKategorie** und ändern die enthaltene SQL-Anweisung so, dass diese als Kriterium den per Parameter übergebenen Primärschlüsselwert enthält.

Danach führen wir die geänderte Abfrage mit der **Execute**-Anweisung aus. Der Aufruf dieser Prozedur sieht etwa so aus:

```
KategorieLoeschen_PT 104
```

Diese Variante hat noch folgende Nachteile:

- Die an den SQL Server übergebene SQL-Anweisung wird dynamisch zusammengesetzt. Wenn sich die SQL-Anweisung dabei von einer bereits verwendeten unterscheidet, also etwa ein anderer Parameterwert zum Einsatz kommt, muss der Ausführungsplan neu erstellt werden.
- Die Verbindungszeichenfolge ist in der Abfrage gespeichert. Wenn sich diese ändert, muss sie in jeder Abfrage angepasst werden.
- Wir erfahren nicht, ob die Aktion erfolgreich war und wie viele Datensätze gelöscht wurden.

In den folgenden beiden Abschnitten kümmern wir uns um diese Nachteile.

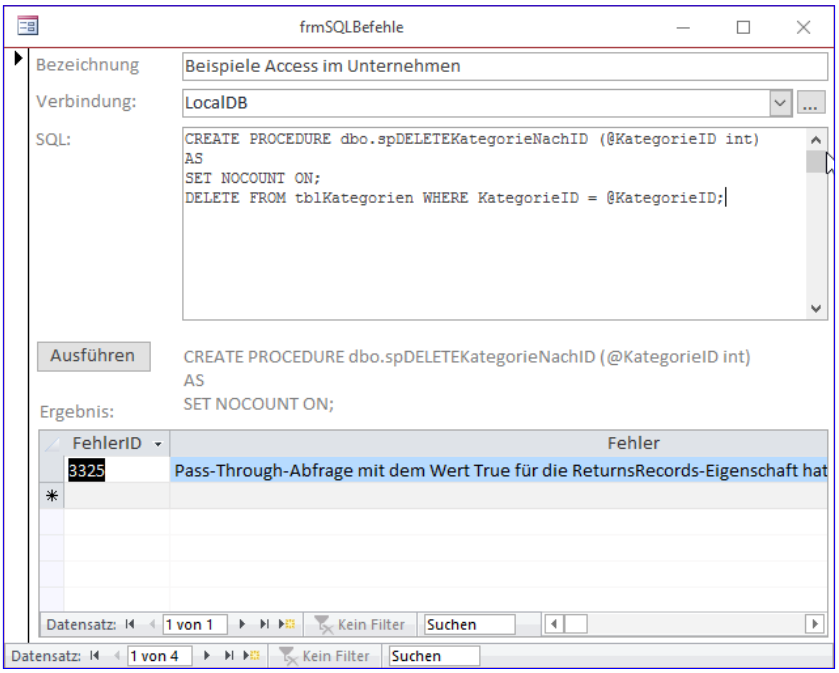

Bild 2: Anlegen einer gespeicherten Prozedur per Access-Formular

Datensatz löschen per gespeicherter Prozedur

Als Erstes sorgen wir dafür, dass der SQL Server unabhängig vom übergebenen Parameter nur einen Ausführungsplan für die Abfrage erstellt, speichert und bei weiteren Aufrufen wiederverwendet. Dazu erstellen wir eine gespeicherte Prozedur, und zwar mit folgendem SQL-Skript:

```
CREATE PROCEDURE dbo.spDELETEKategorieNachID (@KategorieID int)
AS
SET NOCOUNT ON;
DELETE FROM tblKategorien
WHERE KategorieID = @KategorieID;
```

Dieses Skript können Sie, wenn Sie es von Access aus ausführen möchten, in das Formular **frmSQLBefehle** eingeben und dann mit der **Ausführen**-Schaltfläche ausführen (s. Bild 2). Ob die gespeicherte Prozedur erfolgreich angelegt wurde, können Sie mit der folgenden Anweisung, ebenfalls in diesem Formular abgesetzt, prüfen:

```
SELECT * FROM Suedsturm.information_schema.routines
WHERE routine_type = 'PROCEDURE'
```

Die gespeicherte Prozedur **spDELETEKategorieNachID** erwartet den Primärschlüsselwert des zu löschenden Datensatzes als Parameter. Wenn Sie die gespeicherte Prozedur direkt vom Abfragefenster des SQL Servers aus ausführen wollten, würden Sie dies mit folgender Anweisung erledigen:

```
EXEC dbo.spDELETEKategorieNachID 105
```

Sie können auch diese Abfrage im Formular **frmSQLBefehle** absetzen, aber es gibt noch eine andere Variante – zum Beispiel für den Fall, dass Sie diese gespeicherte Prozedur per Code aufrufen wollen.

Also erstellen Sie zunächst eine neue Abfrage, wandeln diese in eine Pass-Through-Abfrage um und legen den SQL-Ausdruck aus Bild 3 fest.

In dieser Abfrage müssen Sie nun natürlich ebenfalls den Primärschlüsselwert des zu löschenden Datensatzes als Parameter angeben. Dies erledigen Sie ähnlich wie oben:

```
Public Sub KategorieLoeschen_PT_SP(lngKategorieID As Long)
    Dim db As DAO.Database
    Dim qdf As DAO.QueryDef
    Set db = CurrentDb
    Set qdf = db.QueryDefs("qryPTDeleteKategorie")
    qdf.SQL = "EXEC dbo.spDELETEKategorieNachID " _
                                        & lngKategorieID
    qdf.Execute
    Set qdf = Nothing
    Set db = Nothing
End Sub
```

Der Aufruf sieht beispielsweise wie folgt aus:

```
KategorieLoeschen_PT_SP 106
```

Dies ändert zunächst den SQL-Ausdruck der Abfrage **ptKategorieLoeschen** wie folgt:

```
EXEC dbo.spDELETEKategorieNachID 106
```

Dieser Aufruf wird direkt an den SQL Server gesendet, der dann die gespeicherte Prozedur **spDELETEKategorieNachID** mit dem angegebenen Parameter ausführt und den entsprechenden Datensatz löscht.

Pass-Through-Abfrage mit dynamischer Verbindungszeichenfolge

Nun soll noch die Verbindungszeichenfolge direkt aus der Tabelle **tblVerbindungszeichenfolgen** bezogen werden (Erläuterungen zu dieser Tabelle siehe **RDBMS-Zugriff per VBA: Verbindungen**, **www.access-im-unternehmen.de/1054**). Dazu übergeben Sie der VBA-Prozedur noch die ID der Verbindungszeichenfolge als weiteren Parameter. Dieser Parameter wird an die in dem oben erwähnten Beitrag erläuterte Funktion **VerbindungszeichenfolgeNachID** übergeben, die dann die Verbindungszeichenfolge zurückliefert. Das Ergebnis landet direkt in der Eigenschaft **Connect** des **QueryDef**-Objekts, was dem Zuweisen der Verbindungszeichenfolge zur Eigenschaft **ODBC-Verbindung** entspricht. Die Prozedur finden Sie in Listing 1. Auch hier noch ein Beispielaufruf:

```
KategorieLoeschenNachID_PT_SP_Connection 107, 9
```

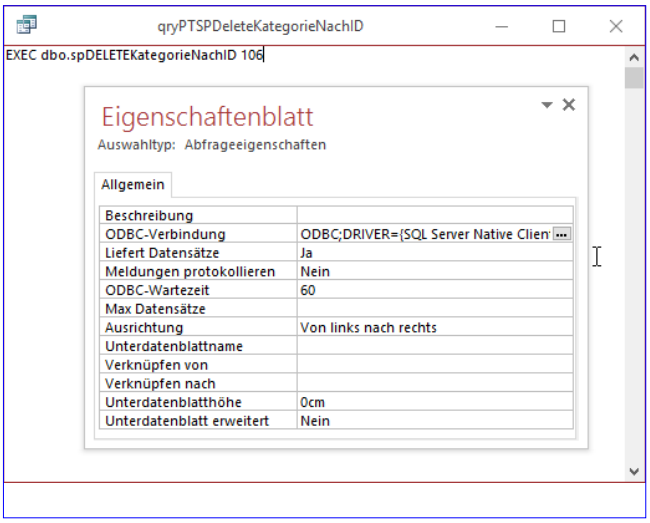

Bild 3: Aufruf einer gespeicherten Abfrage per Passthrough-Abfrage

Dies löscht den Datensatz mit dem Wert **107** im Feld **KategorieID** und verwendet die Verbindungszeichenfolge mit dem Wert **9** im Feld **VerbindungszeichenfolgeID** der Tabelle **tblVerbindungszeichenfolgen**.

Sie können die Verbindungszeichenfolge natürlich auch mit der Funktion **Standardverbindungszeichenfolge** ermitteln. Dazu ersetzen Sie die Zeile mit der **Connect**-Eigenschaft wie folgt:

```
qdf.Connect = Standardverbindungszeichenfolge
```

Oder Sie übergeben die Standardverbindungszeichenfolge beim Aufruf:

```
Public Sub KategorieLoeschenNachID_PT_SP_Connection(lngKategorieID As Long, lngVerbindungszeichenfolgeID As Long)
    Dim db As DAO.Database
    Dim qdf As DAO.QueryDef
    Set db = CurrentDb
    Set qdf = db.QueryDefs("qryPTDELETEKategorie")
    qdf.Connect = VerbindungszeichenfolgeNachID(lngVerbindungszeichenfolgeID)
    qdf.SQL = "EXEC dbo.spDELETEKategorieNachID " & lngKategorieID
    qdf.Execute
    Set qdf = Nothing
    Set db = Nothing
End Sub
```

Listing 1: Aufruf einer gespeicherten Prozedur mit dynamischer Verbindungszeichenfolge

```
KategorieLoeschenNachID_PT_SP_Connection 108, 7
                                    Standardverbindungszeichenfolge
```

Löschen mit Bestätigung

Schließlich möchten Sie vielleicht noch wissen, ob der Löschvorgang überhaupt erfolgreich war beziehungsweise wie viele Datensätze von der Aktionsabfrage betroffen waren. T-SQL bietet mit der Funktion **@@ROWCOUNT** ein Mittel, um die Anzahl der von der zuletzt ausgeführten Abfrage betroffenen Datensätze zu ermitteln. Dies bezieht sich auf die Aktionsabfragen der aktuellen Verbindung. Die folgende gespeicherte Prozedur löscht wie in den obigen Beispielen einen Datensatz mit dem übergebenen Wert für das Feld **KategorieID**, gibt aber als Ergebnis die Anzahl der betroffenen Datensätze zurück:

```
CREATE PROCEDURE dbo.spDELETEKategorieNachIDMitErgebnis
@KategorieID INT
AS
SET NOCOUNT ON;
DELETE FROM tblKategorien WHERE KategorieID = @KategorieID
SELECT @@ROWCOUNT AS RecordsAffected;
```

Wenn Sie diese gespeicherte Prozedur im Abfragefenster im **SQL Server Management Studio** aufrufen, sieht das Ergebnis wie in Bild 4 aus. Um dieses Ergebnis von Access aus zu nutzen, ist eine kleine Änderung am Entwurf der Pass-Through-Abfrage nötig.

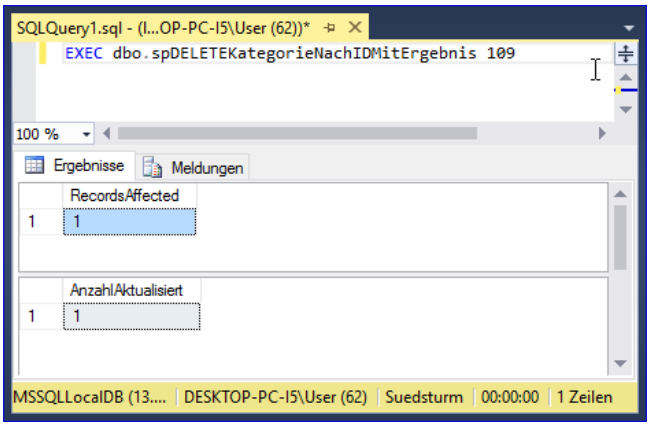

Bild 4: Ergebnis einer gespeicherten Prozedur im SQL Server Management Studio

Wir haben die Abfrage von oben unter dem Namen **qrySPDELETEKategorieNachIDMitErgebnis** kopiert und die Eigenschaft **Liefert Datensätze** auf den Wert **Ja** eingestellt (s. Bild 5). Anderenfalls liefert die Abfrage das Ergebnis der **SELECT**-Abfrage mit der Anzahl der betroffenen Datensätze nicht zurück!

Führen Sie diese Abfrage direkt aus, liefert sie das Ergebnis aus Bild 6.

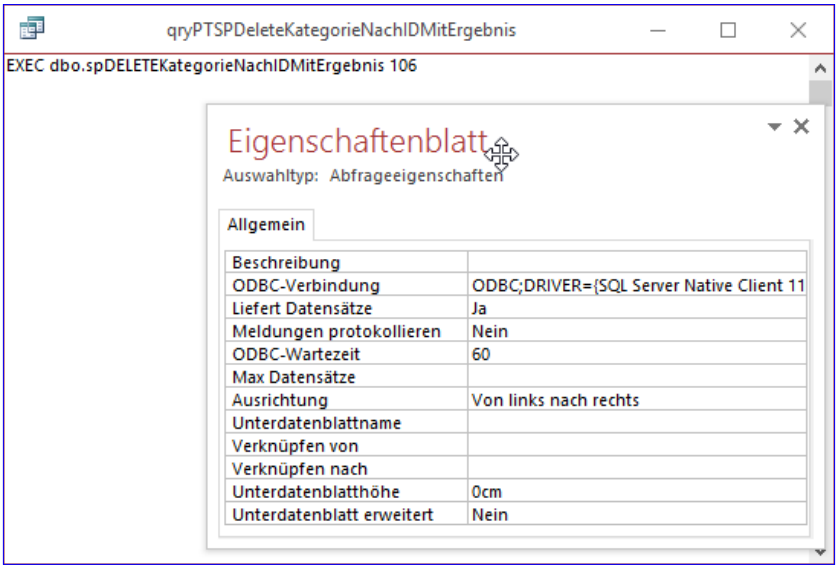

Bild 5: Entwurf der Passthrough-Abfrage zum Löschen eines Datensatzes mit Rückgabe der betroffenen Datensätze

Dies ist ein Ergebnis, mit dem wir auch unter VBA arbeiten können. Die Prozedur aus Listing 2 verwendet wieder die **KategorieID** und ermittelt die Verbindungszeichenfolge mit der Funktion **Standardverbindungszeichenfolge**. Sie erzeugt wie gewohnt ein **QueryDef**-Objekt auf Basis einer neuen gespeicherten Access-Abfrage namens **spDELETEKategorieNachIDMitErgebnis** und ermittelt die gewünschte Verbindungszeichenfolge. Dann weist sie wie zuvor den neuen SQL-Ausdruck zu, führt die Abfrage aber diesmal nicht mit **Execute** aus. Stattdes-

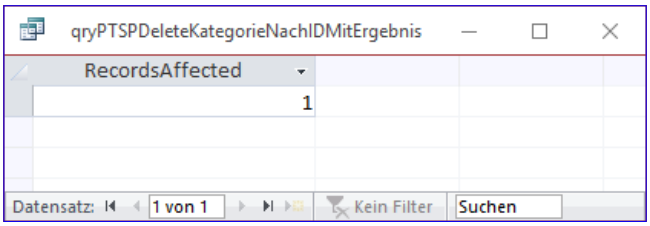

Bild 6: Ergebnis der gespeicherten Prozedur innerhalb einer Pass-Through-Abfrage in Access

sen erstellt sie ein neues **Recordset**-Objekt und füllt es über die **OpenRecordset**-Methode mit dem Ergebnis der gespeicherten Prozedur. Dies erzeugt ein herkömmliches **Recordset**-Objekt, das nur einen Datensatz mit einem Feld enthält – und dieses wird mit **rst!RecordsetAffected** ausgelesen und in einem Meldungsfenster ausgegeben.

Dynamische Aktionsabfrage ohne Rückgabewert

Die bisherigen Ansätze gingen davon aus, dass die Access-Datenbank eine gespeicherte Access-Abfrage mit den wichtigsten Eigenschaften zum Ausführen der gespeicherten Prozedur per Pass-Through-Abfrage enthält. Je mehr solcher Abfragen Sie verwenden, desto unübersichtlicher wird es im Navigationsbereich. Und davon abgesehen ändern wir ohnehin zumindest den SQL-Code jeder Pass-Through-Abfrage, die eine gespeicherte Prozedur mit Parametern ausführt. Dann könnten wir diese auch gleich neu anlegen – der Performance-Unterschied dürfte sich in Grenzen halten. Es gibt jedoch auch die Möglichkeit, ein **QueryDef**-Objekt komplett temporär zu erzeugen.

Was benötigen wir also im Vergleich zur vorherigen Variante? Eigentlich müssen wir nur den Namen der zu verwendenden gespeicherten Prozedur zusätzlich übergeben, die Verbindungszeichenfolge und die Parameter werden ja bereits verarbeitet. Außerdem referenzieren wir nicht über die **QueryDefs**-Auflistung eine bestehende Abfrage, sondern erstellen mit **CreateQueryDef** eine neue – und zwar mit einer leeren Zeichenkette als Name. Die Prozedur sieht nun wie in Listing 3 aus. Sie erstellt mit der **CreateQueryDef**-Methode eine temporäre Abfrage, was wir dadurch erreichen, dass wir eine leere Zeichenfolge als Parameter übergeben.

Für die Parameterliste verwenden wir im Kopf der Prozedur einen Parameter namens **varParameter** des Typs **ParamArray**, dem man beliebig viele durch Kommata getrennte Parameterwerte übergeben kann. Die damit übergebenen Werte setzt die Prozedur in einer **For Each**-Schleife über alle Elemente von **varParameter** zusammen und stellt jeweils ein Komma voran. Das erste Komma wird danach gegebenenfalls abgeschnitten. Die

```
Public Sub KategorieLoeschenNachID_PT_SP_Connection_MitErgebnis(lngKategorieID As Long)
    Dim db As DAO.Database
    Dim qdf As DAO.QueryDef
    Dim rst As DAO.Recordset
    Dim lngAnzahl As Long
    Set db = CurrentDb
    Set qdf = db.QueryDefs("qryPTSPDELETEKategorieNachIDMitErgebnis")
    qdf.Connect = Standardverbindungszeichenfolge
    qdf.SQL = "EXEC dbo.spDELETEKategorieNachIDMitErgebnis " & lngKategorieID
    Set rst = qdf.OpenRecordset(dbOpenSnapshot)
    lngAnzahl = rst!RecordsAffected
    MsgBox "Es wurden " & lngAnzahl & " Datensätze gelöscht."
    Set rst = Nothing
    Set qdf = Nothing
    Set db = Nothing
End Sub
```

Listing 2: Aufruf einer gespeicherten Prozedur mit Rückgabewert

Ticketsystem, Teil III

In unser Ticketsystem haben wir bereits Funktionen integriert, mit denen Sie die Kundenanfragen per Drag and Drop in der Datenbank speichern können. Außerdem haben wir ein Formular hinzugefügt, das alle offenen E-Mails anzeigt und eine einfache Möglichkeit enthält, daraus Tickets zu generieren. Diese wollen wir nun im vorliegenden Teil der Beitragsreihe verarbeiten, und zwar in dafür ausgelegten Formularen. Diese zeigen sowohl eine Übersicht aller Tickets – filterbar nach dem Status und anderen Eigenschaften – als auch den Verlauf eines einzelnen Tickets.

Ticketübersicht

Im Formular **frmTickets** wollen wir eine Übersicht aller Tickets liefern, und zwar nach verschiedenen Kriterien. Die Tickets sollen nach dem Anlagedatum sortiert werden können oder nach dem Kunden. Außerdem benötigen wir natürlich verschiedene Filter, etwa nach dem Kunden oder nach dem Status oder der Priorität. Das Formular soll per Doppelklick auf einen der Einträge die Möglichkeit bieten, das Ticket in der Detailansicht zu öffnen und alle bisher erfolgten Schritte darzustellen.

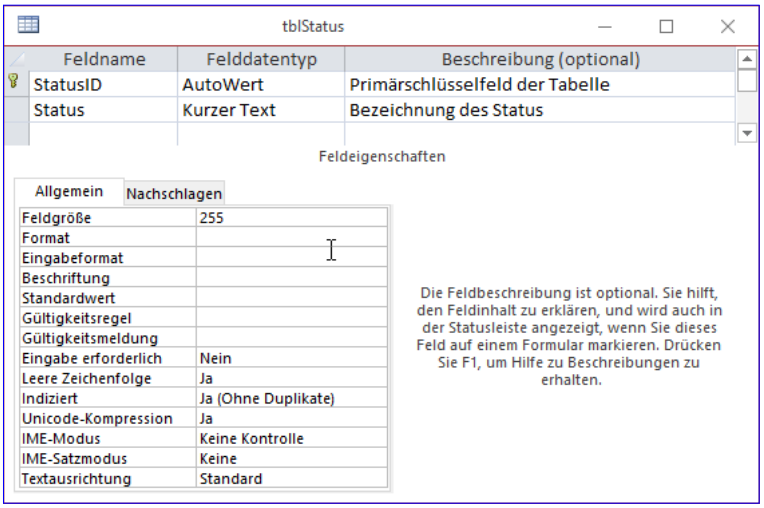

Bild 1: Entwurf der Tabelle **tblStatus**

Um keine eigene Programmierung etwa für die Sortierung vornehmen zu müssen, nutzen wir zur Anzeige der Tickets einfach ein Unterformular in der Datenblattansicht. Dieses bietet genügend Möglichkeiten zum Sortieren und Filtern der Einträge. Schnelle Filter wollen wir hinzufügen für den Status von Tickets sowie für die Priorität. Außerdem wollen wir eine Schaltfläche zum schnellen Aufheben der Sortierung und des Filters hinzufügen. Schließlich soll ein Ticket natürlich schnell per Doppelklick geöffnet werden können.

Zusätzliche Tabellen

Um den Status und die Priorität eines Tickets erfassen zu können, wollen wir der Tabelle **tblTicktes** zwei neue Felder hinzufügen. Diese sollen als Fremdschlüsselfelder ausgelegt werden und zwei weitere neue Tabellen referenzieren.

Die erste Tabelle heißt **tblStatus** und soll die verschiedenen Statuswerte erfassen (Status ist übrigens tatsächlich der Plural von Status, daher diese Tabellenbezeichnung). Sie finden den Entwurf dieser Tabelle in Bild 1. Neben dem Primärschlüsselfeld **StatusID** enthält die Tabelle noch das mit einem eindeutigen Index versehene Feld **Status**.

Die zweite neue Tabelle heißt **tblPrioritaeten** und ist fast genauso aufgebaut wie die Tabelle **tblStatus**. Sie enthält neben dem Primärschlüsselfeld **PrioritaetID** ebenfalls ein weiteres Feld zum Erfassen der Bezeichnung der Inhalte, in diesem Fall namens **Prioritaet**. Auch dieses Feld haben wir mit einem eindeutigen Index versehen (s. Bild 2).

Die beiden Fremdschlüsselfelder der Tabelle **tblTickets** richten wir als Nachschlagefelder ein. Dazu wählen Sie

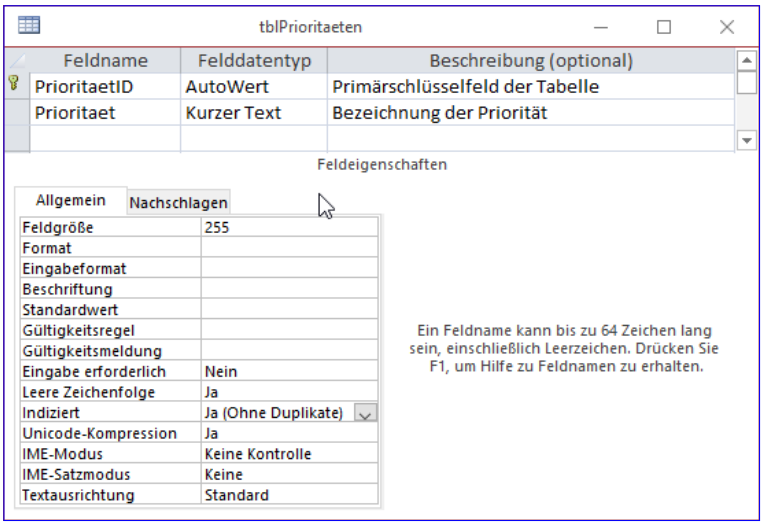

Bild 2: Entwurf der Tabelle **tblPrioritaeten**

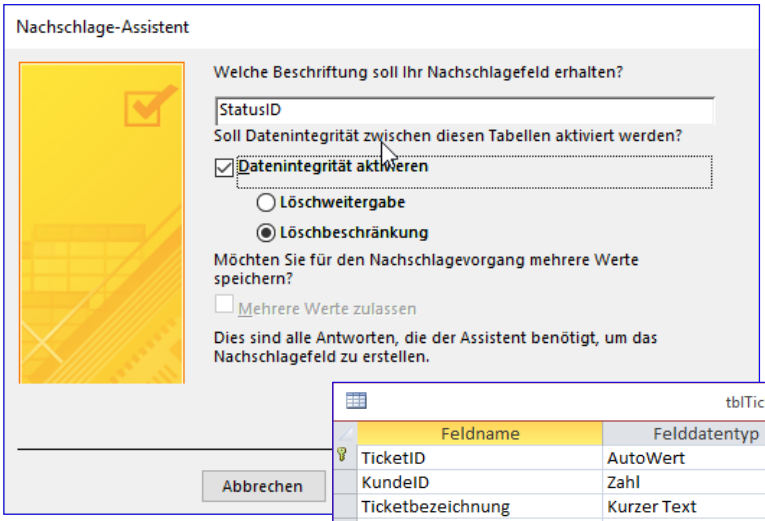

Bild 3: Letzter Schritt beim Einrichten eines Nachschlagefeldes

zunächst für das Feld **StatusID** den Datentyp **Nachschlage-Assistent** aus und starten somit den gleichnamigen Assistenten. Wählen Sie hier die Tabelle **tblPrioritaeten** aus, selektieren Sie die beiden Felder **StatusID** und **Status**, legen Sie keine aufsteigende Sortierung für irgendeines der Felder fest und aktivieren Sie die Option **Schlüsselspalte ausblenden**.

Schließlich aktivieren Sie die referenzielle Integrität durch Markieren der Option **Datenintegrität aktivieren** (s. Bild 3).

Die Tabelle **tblTickets** sieht danach im Entwurf wie in Bild 4 aus, sodass wir Status und Priorität bequem per Nachschlagefeld auswählen können. Die Einrichtung von Nachschlagefeldern hat außerdem den praktischen Vorteil, dass Sie später, wenn Sie die Felder dieser Tabelle in den Entwurf von Formularen ziehen, direkt Kombinationsfelder für solche Felder erhalten.

Unterformular sfmTickets einrichten

Genau davon profitieren wir gleich im nächsten Schritt. Wir erstellen ein neues Formular und speichern es unter dem Namen **sfmTickets** ab. Dieses soll das Unterformular eines weiteren Formulars namens **frmTickets** werden und die

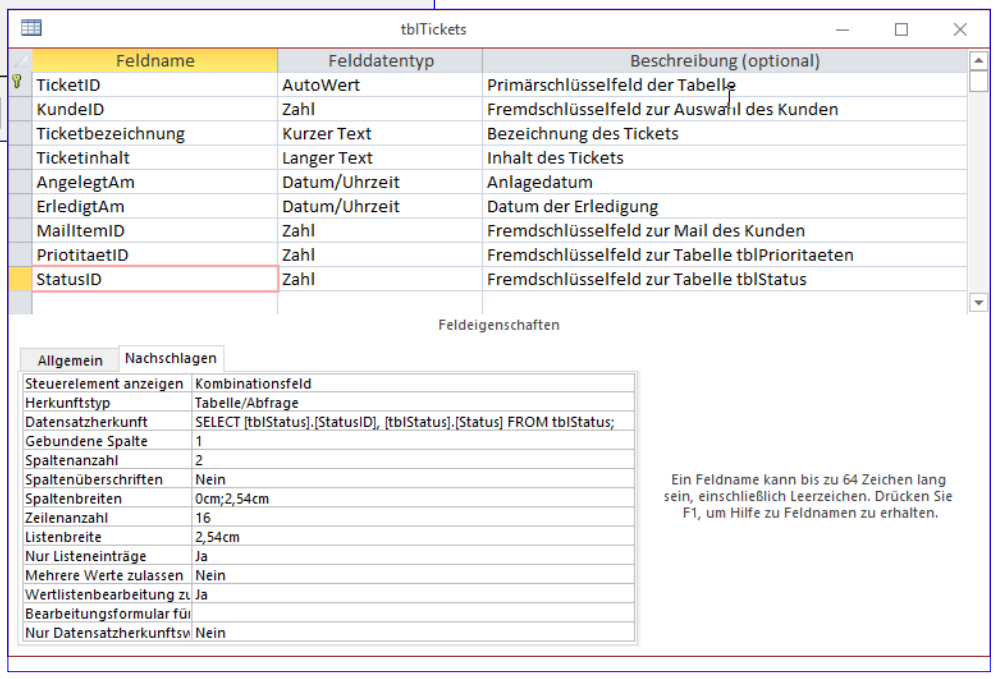

Bild 4: Entwurf der soeben erweiterten Tabelle **tblTickets**

Tickets in der Datenblattansicht übersichtlich anzeigen.

Legen Sie als Datenherkunft des Formulars die Tabelle **tblTickets** fest. Ziehen Sie dann aus der Feldliste alle Felder der Tabelle in den Detailbereich des Formularentwurfs (s. Bild 5).

Nun stellen Sie noch die Eigenschaft **Standardansicht** des **Formulars** auf Datenblatt ein und schließen das Formular.

Nun legen Sie das Formular **frmTickets** an und ziehen das Unterformular **sfmTickets** in den Detailbereich des Entwurfs. Stellen Sie die Eigenschaften **Horizontaler Anker** und **Vertikaler Anker** jeweils auf den Wert **Beide** ein.

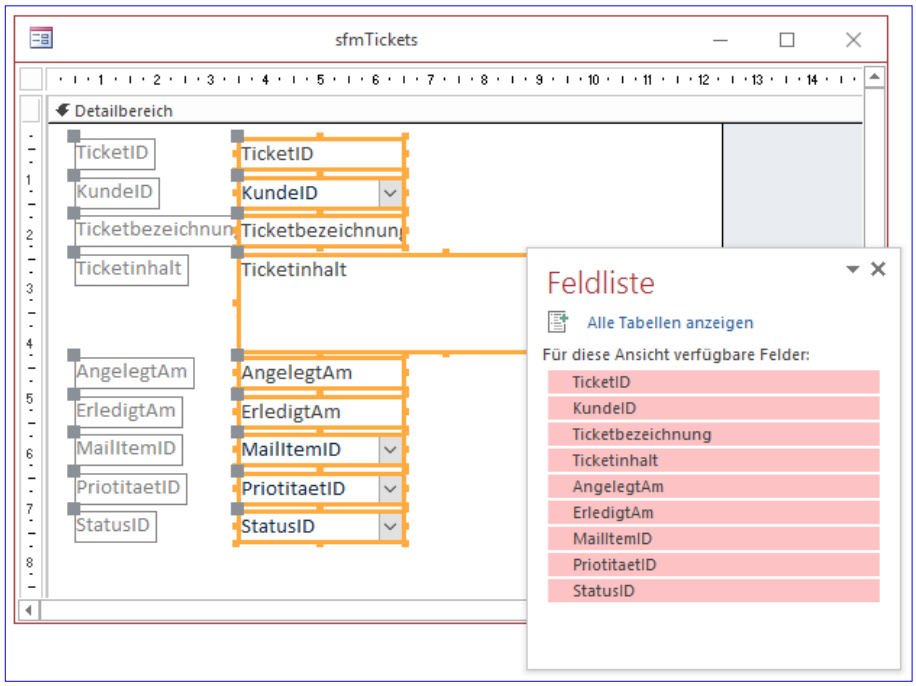

Bild 5: Entwurf des Unterformulars **sfmTickets**

Für die optische Gestaltung des Hauptformulars legen Sie noch die Eigenschaften **Navigationsschaltflächen**, **Datensatzmarkierer**, **Bildlaufleisten** und **Trennlinien** auf den Wert **Nein** fest.

Filter für die Ticketübersicht steuern

Nun nutzen wir die beiden Klassen **clsFastFilter** und **clsFastFilterControl**, die wir im Beitrag **Schneller Filter** vorgestellt haben (**www.access-im-unternehmen.de/1072**). Importieren Sie diese beiden Klassen in die aktuelle Datenbank.

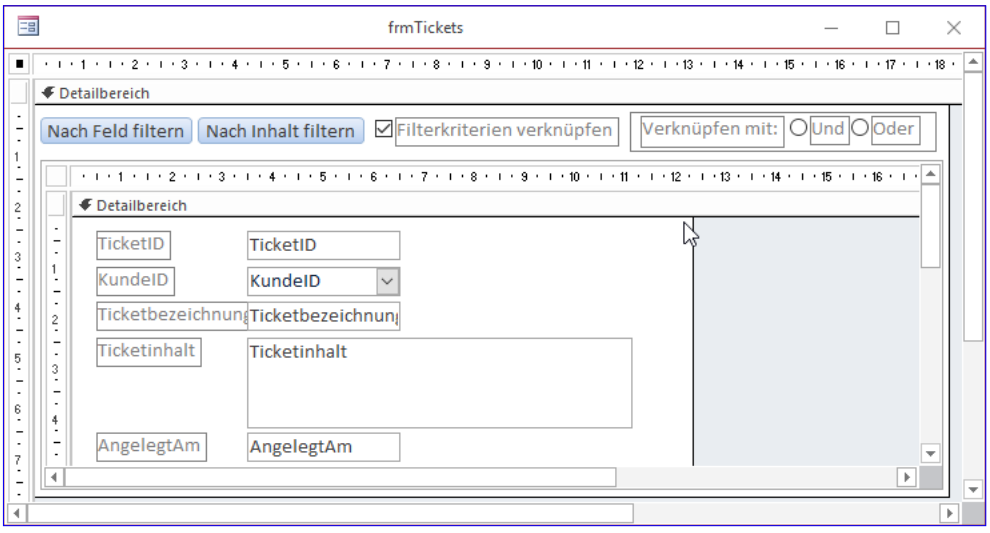

Bild 6: Entwurf des Formulars **frmTickets**

Nun statten Sie das Formular **frmTickets** mit einigen Steuerelementen aus, mit denen der Benutzer die Daten filtern kann. Sie benötigen zwei Schaltlfächen namens **cmdNachFeldFiltern** und **cmdNachInhaltFiltern**, ein Kontrollkästchen namens **chkFilterkriterienVerknuepfen** und eine Optionsgruppe mit dem Namen **ogrVerknuepfenMit** (s. Bild 6). Für das

Kontrollkästchen stellen Sie als Standardwert **False** ein, für die Optionsgruppe den Wert **1**. Außerdem stellen Sie für die Optionsgruppe die Eigenschaft **Aktiviert** aus **Nein** ein. Schließlich stellen Sie noch die Eigenschaft **Enthält Modul** des Unterformulars auf den Wert **Ja** ein.

Nun fügen wir den Code zum Initialisieren der Filterfunktionen hinzu. Das Klassenmodul des Formulars **frmTickets** sieht danach wie folgt aus:

```
Dim objFastFilter As clsFastFilter

Private Sub Form_Load()
    Set objFastFilter = New clsFastFilter
    With objFastFilter
        Set .Subform = Me!sfmTickets.Form
        Set .FastFilterButton = Me!cmdNachFeldFiltern
        Set .FastFilterContainsButton = _
                                Me!cmdNachInhaltFiltern
        Set .CombineFilterCheckbox = _
                            Me!chkFilterkriterienVerknuepfen
        Set .ANDOrOROptiongroup = Me!ogrVerknuepfenMit
    End With
End Sub
```

Nach dem ersten Test, bei dem wir einen Teil des Feldes **Kunde** markieren und die Schaltlfäche **cmdNachInhaltFiltern** anklicken, stellen wir allerdings fest, dass dies nicht reibungslos funktioniert.

Der Hintergrund ist, dass das Kombinationsfeld **KundeID** eine Datensatzherkunft mit einem berechneten Feld verwendet:

```
SELECT KundeID, [Nachname]
& ", " & [Vorname] AS Kunde
FROM tblKunden;
```

Dies wird durch die Klasse **clsFastFilterControl** nicht abgedeckt. Wir haben probiert, dies so anzupassen, dass auch berechnete Felder aufgelöst werden können, aber dies gelingt nicht, weil es keine Möglichkeit gibt, den berechneten Ausdruck über das Objektmodell zu ermitteln. Dies zeigt im Übrigen auch sehr schön, dass auch eine ausgefeilte Lösung nur selten alle möglichen Konstellationen berücksichtigen kann ...

Also müssen wir uns einen kleinen Workaround einfallen lassen und die Klasse **clsFastFilterControl** etwas anpassen – und zwar so, dass wir in Fällen wie dem hier vorliegenden selbst Hand anlegen können. In diesem Fall ist das Problem, dass wir den Namen des Feldes der Datensatzherkunft des Kombinationsfeldes ermitteln wollen, dessen Inhalt im Kombinationsfeld angezeigt wird. Dieser lautet aber schlicht **Kunde** und entspricht dem Alias-Namen des berechneten Feldes, das eigentlich den Ausdruck **[Nachname] & ", " & [Vorname]** enthält.

An diesen kommen wir aber programmatisch nicht heran (außer, wir würden den SQL-Ausdruck auseinandernehmen – aber dieser kann ja auch beliebig komplex werden, also lassen wir gleich die Finger davon ...). Deshalb tragen wir diesen Ausdruck einfach in die Eigenschaft **Marke** (englisch und unter VBA: **Tag**) des Kombinationsfeldes ein (s. Bild 7).

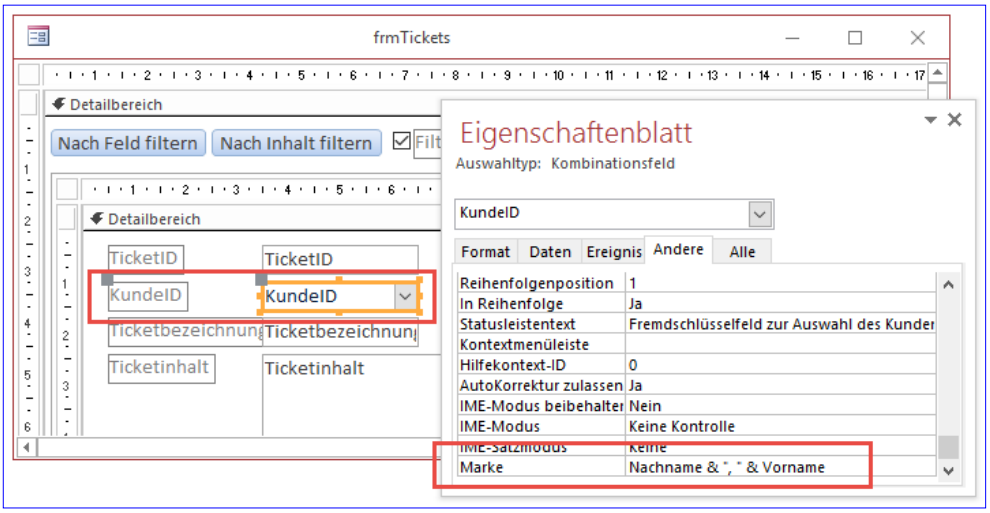

Bild 7: Eintragen des angezeigten Ausdrucks des Kombinationsfeldes in der Eigenschaft **Marke**