HAUFE.

Ausgabe 05/2015

ACCESS IM UNTERNEHMEN

KUNDENDATEN ZUSAMMENFÜHREN

Erfahren Sie, wie Sie doppelte Kundendatensätze auffinden und aus zwei Kundendatensätzen einen machen – inklusive der verknüpften Datensätze wie Bestellungen (S. 53).

In diesem Heft:

TABELLENDATEN SICHERN

Sichern Sie die Daten Ihrer Tabellen automatisch und fügen Sie die notwendigen Trigger per Add-In hinzu!

SEITE 2

MAILS AUS OUTLOOK ARCHIVIEREN

Archivieren Sie Outlook-Mails direkt beim Eingang in einer Datenbank und nutzen Sie die komfortable Suchfunktion.

SEITE 31

SUCHE NACH VERKNÜPFTEN DATEN

Durchsuchen Sie die Daten in Unterformularen und lassen Sie die Suchergebnisse im Haupt- und Unterformular anzeigen.

SEITE 15

Aus zwei mach eins

Kundendatenbanken werden aus verschiedenen Quellen gefüttert. Aus dem Webshop, dem Telefonverkauf, aus Onlineportalen wie Amazon, eBay und Co. Wer da nicht aufpasst, hat schnell mehrere Datensätze zu ein und demselben Kunden – daran ändern selbst Validierungen auf die E-Mailadresse oder andere Adressdaten nichts. Eine neue E-Mailadresse, ein Dreher in Vor- und Nachname, schon landet der Kunde in einem neuen Datensatz.

Ist das Kind erst einmal in den Brunnen gefallen, ist es allerdings noch längst nicht zu spät: Mit der in dieser Ausgabe vorgestellten Lösung finden Sie nicht nur die Dubletten in Ihren Kundendatensätzen, sondern gleichen diese auch noch komfortabel ab. Dabei stellen Sie nicht nur aus mehreren Kundeneinträgen die korrekten Daten zusammen und löschen dann die überzähligen Datensätze. Sie legen gleichzeitig fest, welche mit den Kundendaten verknüpften Tabellen berücksichtigt werden sollen. So können Sie beispielsweise die Daten aus einer Tabelle mit Bestellungen von dem zu löschenden Kundendatensatz auf den Kundendatensatz übertragen, den Sie beibehalten möchten. Die komplette Lösung finden Sie im Beitrag **Kundendatensätze zusammenführen** ab S. 53.

Das Löschen von Kundendaten oder auch das Umschreiben von Bestellungsdatensätzen ist nicht risikolos: Haben Sie durch einen unglücklichen Zufall die Daten etwa zweier Michael-Müller-Datensätze zusammengeführt, bei denen es sich tatsächlich um verschiedene Kunden handelte, können Sie die Änderungen kaum ohne größeren Aufwand wiederherstellen. Das verhindern Sie, indem Sie alle Datensätze vor dem Ändern in einer Archivtabelle sichern – das gilt sowohl für die Kundendatensätze als auch für die damit verknüpften Tabellen. Um dies zu erreichen, müssen Sie die Archivtabellen erstellen und die Originaltabellen mit Datenmakros versehen, die beim Ändern oder Löschen der Daten die entsprechenden Einträge zu den Archivtabellen hinzufügen.

Dies schafft bereits eine gewisse Sicherheit, aber es bleibt eine Menge Aufwand, wenn Sie dies für mehrere Tabellen durchführen wollen. Aber keine Sorge: In unserem Beitrag **Änderungshistorie implantieren** ab S. 2 finden Sie ein Add-In, mit dem Sie die gewünschten Tabellen per Mausklick mit den genannten Funktionen ausstatten.

Wenn wir schon bei Kundendaten sind: Vielleicht haben Sie sich auch schon immer eine Möglichkeit gewünscht, in einem Kundenformular, das die Bestellungen und Bestelldetails zu einem Kunden anzeigt, nach den Kunden zu suchen, die einen bestimmten Artikel bestellt haben. Wie dies gelingt, zeigt unser Beitrag **Nach Daten im Unterformular suchen** ab S. 15.

In den vorherigen Ausgaben haben wir uns um das Thema Archivierung von Outlook-Mails gekümmert. Dieses Thema schließen wir mit zwei Beiträgen in der vorliegenden Ausgabe ab. Unter dem Titel **Outlook-Mails in Access archivieren III** erfahren Sie ab S. 22, wie Sie eine Reihe von Outlook-Ordnern festlegen, aus denen regelmäßig Mails gesichert werden sollen, um diese in einer Datenbank bereitzustellen.

Und nachdem Sie dies einmal erledigt haben, möchten Sie vielleicht, dass eingehende E-Mails automatisch in der Datenbank gesichert werden. Wie Sie Outlook dazu bringen, erfahren Sie ab S. 31 im Beitrag **Outlook-Mails nach Empfang archivieren**.

Für den Komfort beim Einsatz von Formularen in der Datenblattansicht haben wir uns auch etwas einfallen lassen: Mit dem Klassenmodul aus **Spaltenbreiten optimieren mit Klasse** (ab S. 39) zeigen Sie die Spalten Ihrer Datenblätter immer in der korrekten Breite an.

Und in **Datenblattereignisse mit Klasse** zeigen wir Ihnen ab S. 44, wie Sie Ereignisse wie einen Klick auf ein beliebiges Feld des Datenblatts auf einfache Weise abbilden können.

Und vergessen Sie nicht, sich die Tipps und Tricks ab S. 67 anzusehen – dort gibt es eine Menge zu entdecken.

Und nun: Viel Spaß bei der Lektüre der neuen Ausgabe!

Ihr Michael Forster

Änderungshistorie implantieren

Die neuen Tabellenereignisse, die mit Access 2010 eingeführt wurden, erlauben die automatische Sicherung von Tabellendaten vor der Durchführung von Änderungen an den Datensätzen. Dazu legen Sie entsprechende Datenmakros an, die durch die Tabellenereignisse ausgelöst werden. Außerdem benötigen Sie eine Tabelle, welche die geänderten Datensätze speichert. Wenn Sie beides für mehrere Tabellen durchführen wollen, ist dies eine Menge Handarbeit. Grund genug, diesen Vorgang zu automatisieren.

Dazu eignet sich natürlich ein Access-Add-In am besten: Dieses können Sie in allen betroffenen Datenbanken starten und die Tabellen mit den zu sichernden Daten präparieren. Dazu soll die Tabelle zunächst kopiert und unter einem passenden Namen gespeichert werden – bei der Tabelle **tblArtikel** beispielsweise unter dem Namen **_tblArtikel_Backup**. Diese Tabelle soll zusätzlich noch zwei Felder enthalten, die das Änderungsdatum oder das Löschdatum aufnehmen.

Die Lösung setzt auf dem Beitrag **Geänderte Daten archivieren** auf (**www.access-im-unternehmen.de/925**).

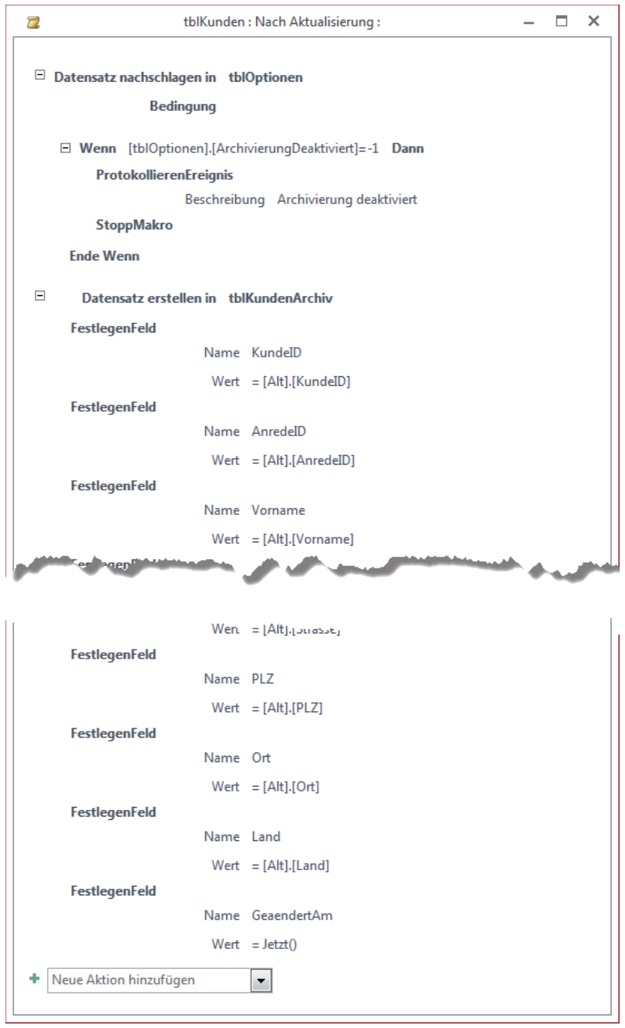

Bild 1: Aufbau des Makros zum Sichern der Daten vor dem Aktualisieren

Aussehen der Makros zum Erstellen des Backups

Das Makro für das Speichern der Daten der Tabelle etwa nach der Aktualisierung eines Datensatzes sieht wie in Bild 1 aus.

Der erste Teil prüft, ob die Option zum Deaktivieren der Sicherung gegebenenfalls deaktiviert ist. Dazu muss eine Tabelle namens **tblOptionen** vorhanden sein. Diese können Sie gegebenenfalls auch unter einem anderen Namen anlegen, da viele Anwendungen bereits über eine solche Tabelle verfügen.

Die folgenden Befehle erstellen einen neuen Datensatz in der jeweiligen Zieltabelle und

stellen die Werte der Felder auf die Werte vor der Änderung ein. Diese Werte liefert jeweils die Tabelle **Alt**.

Schließlich weist das Makro dem Feld **GeaendertAm** noch den Änderungszeitpunkt zu.

Beim Löschen eines Datensatzes sieht es ähnlich aus – der einzige Unterschied besteht darin, dass der Löschzeitpunkt im Feld **GeloeschtAm** gesichert wird.

Einschränkungen

Für das Sichern von Daten per Tabellenereignis gibt es eine Einschränkung bezüglich der Felddatentypen: Von der Sicherung sind nämlich sowohl Memofelder als auch Anlagefelder und mehrwertige Felder ausgeschlossen. Die Backup-Tabellen müssen diese Felder also gar nicht erst aufnehmen und auch das Tabellenereignis berücksichtigt diese nicht.

Deutsch-Englisch

Ein weiteres Problem ist die automatische Übersetzung des Makro-Codes. Wenn Access nämlich ein deutsches Literal erkennt, das in der englischen Version irgendeinem Access-Schlüsselwort entspricht, dann übersetzt Access dieses automatisch.

Sollten Sie also in einem Datenmakro den Feldnamen **[Old].[Beschreibung]** verwenden, wird dieser in **[Old].[Description]** geändert. Das ist unbefriedigend, aber außer durch Umgehung der entsprechenden Feldnamen wohl kaum zu lösen.

Aufgaben des geplanten Add-Ins

Das zu erstellende Add-In soll also folgende Aufgaben erledigen:

- Erstellen einer Tabelle zum Speichern der Optionen, falls noch nicht vorhanden

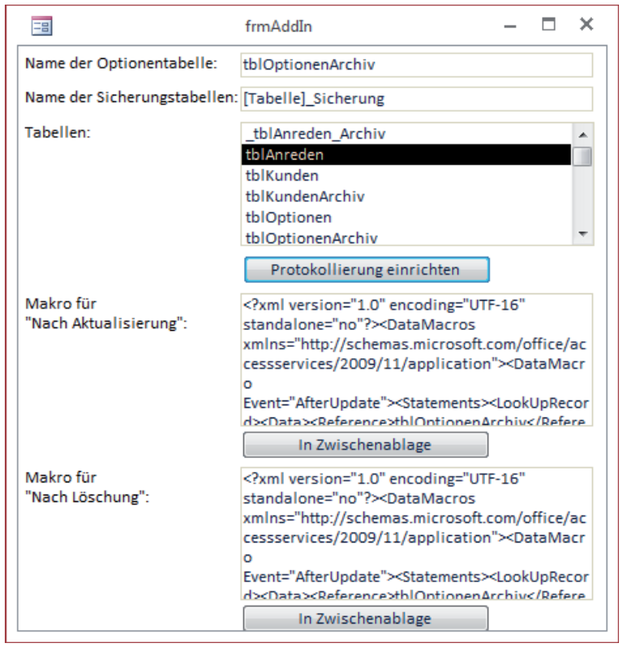

Bild 2: Add-In-Formular zum Anlegen von Sicherungstabellen und zum Generieren der benötigten Makros

- Erstellen der Tabelle zum Sichern der Daten einer Tabelle

- Bereitstellen des Tabellenereignisses und des Datenmakros zum Kopieren der Daten vor der Änderung beziehungsweise vor dem Löschen des Datensatzes.

Das einzige Formular des Add-Ins dazu sieht wie in Bild 2 aus. Es bietet die folgenden Steuerelemente:

- Das oberste Textfeld **txtOptionentabelle** gibt den Namen der Tabelle an, welche die Optionen für das Add-In speichert. Diese wird, wenn das Add-In erstmalig von einer Anwendung aus geöffnet wird, automatisch angelegt. Der Benutzer darf den Namen der Optionentabelle dabei beliebig anpassen.

- Das zweite Textfeld **txtSicherungstabelle** legt das Schema für die Benennung der Archivtabellen fest.

Einziges fixes Element ist der Platzhalter **[Tabelle]**. Dieser wird beim Zusammenstellen des Tabellennamens durch den Namen der Originaltabelle ersetzt.

- Das Listenfeld **lstTabellen** zeigt alle Tabellen der aktuellen Datenbank an. Hier wählen Sie die Tabelle aus, für die eine Archivtabelle erstellt werden soll und für die das Add-In die Datenmakros zum automatischen Archivieren generieren soll.

- Die Schaltfläche **cmdProtokollierungEinrichten** startet die wesentlichen Schritte, nämlich das Anlegen der Archivtabelle sowie die Generierung der beiden Makros.

- Das Textfeld **txtMakroAktualisierung** zeigt das Makro an, das Sie dem Tabellenereignis **Nach Aktualisierung** hinzufügen.

- Das Textfeld **txtMakroLoeschung** liefert das entsprechende Datenmakro für das Ereignis **Nach Löschung**.

- Schließlich gibt es noch zwei Schaltflächen, mit denen Sie die XML-Dokumente für die Datenmakros in die Zwischenablage kopieren und dann für die Makros einsetzen können.

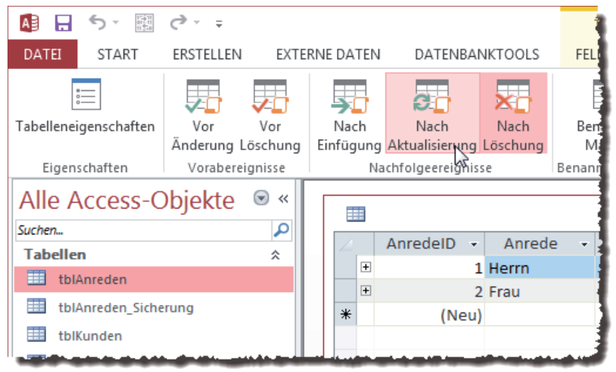

Bild 3: Ribbon-Befehl zum Anzeigen des Datenmakros, das durch das Tabellenereignis **Nach Aktualisierung** ausgelöst wird

Datenmakros kopieren

Wenn Sie die Tabelle ausgewählt und die Schaltfläche **Protokollierung einrichten** angeklickt haben, erhalten Sie die beiden benötigten Datenmakros mit den XML-Dokumenten aus den beiden unteren Textfeldern.

Diese können Sie zur Ansicht oder für Notfälle nutzen, falls das automatische Hinzufügen der Datenmakros zu den Tabellen nicht gelingt.

Und hier ist der Notfallplan: Wenn Sie beispielsweise die Datensätze der Tabelle **tblAnreden** archivieren möchten, erhalten Sie mit der Konfiguration aus dem Screenshot eine neue Tabelle namens **tblAnreden_Archiv**.

Kopieren Sie dann den Inhalt des Textfeldes mit dem Makro für den Fall der Aktualisierung in die Zwischenablage. Öffnen Sie dann die Tabelle, hier **tblAnreden**, beispielsweise in der Datenblattansicht.

Nun wählen Sie im Ribbon den Eintrag **TabellelNachfolgeereignisselNach Aktualisierung** aus (s. Bild 3). Es erscheint ein leeres Makro, das Sie nun einfach markieren und dann per **Strg + C** den Inhalt der Zwischenablage einfügen.

Und siehe da: Der Generator hat tatsächlich einen XML-Code zusammengestellt, der die benötigten Makrobefehle liefert (s. Bild 4).

Auf die gleiche Weise gehen Sie nun noch für das Datenmakro **Nach Löschung** vor. Damit haben Sie der Tabelle zwei Makros hinzugefügt, die dafür sorgen, dass die letzte Version eines Datensatzes vor einer Änderung oder Löschung in der jeweiligen Archivtabelle, hier **tblAnreden_Archiv**, gespeichert wird.

Aber, wie eingangs erwähnt: Normalerweise sollte das Add-In die Datenmakros automatisch hinzufügen.

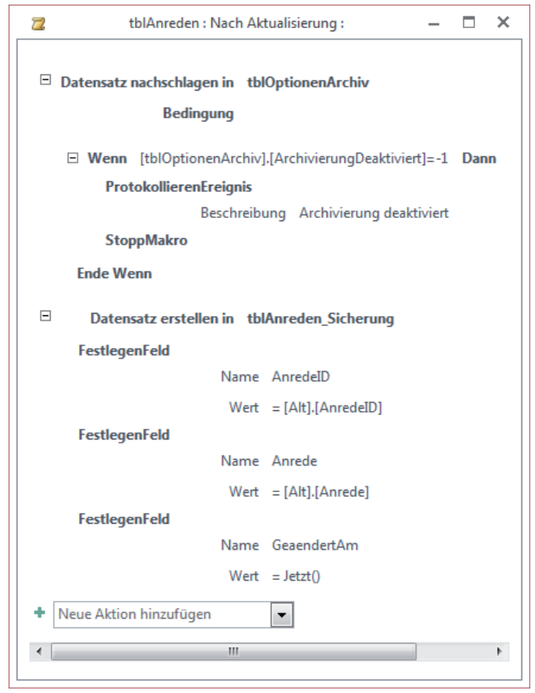

Bild 4: Frisch eingefügtes Datenmakro

Achtung: Ersetzen vorhandener Datenmakros

Beim Testen des Add-Ins ist aufgefallen, dass es beim Überschreiben vorhandener Datenmakros zu Problemen kommen kann. Wir sind so vorgegangen, dass wir den kompletten vorhandenen Inhalt des Datenmakros mit **Strg + A** markiert und diesen dann durch Betätigen der Tastenkombination **Strg + C** durch den Inhalt der Zwischenablage ersetzt haben. Zumindest haben wir das gedacht! Der Inhalt wurde nämlich mitnichten ersetzt, sondern blieb einfach an Ort und Stelle. Also gehen Sie beim Ersetzen des Inhalts eines Datenmakros einfach wie folgt vor: Markieren Sie den kompletten Inhalt mit **Strg + A**, löschen Sie diesen und fügen dann den neuen Inhalt mit **Strg + C** aus der Zwischenablage ein.

Technischer Hintergrund des Add-Ins

In den folgenden Abschnitten schauen wir uns die technischen Hintergründe des Add-Ins an. Dabei arbeiten wir uns anhand des Formulars und des oben beschriebenen Ablaufs vor.

Den Start macht dabei das Ereignis **Beim Laden**, das die Ereignisprozedur aus Listing 1 auslöst. Sie ruft zunächst eine Funktion namens **OptionentabelleFinden** auf und

```
Private Sub Form_Load()
    If OptionentabelleFinden(strOptionentabelle) = True Then
        Me!txtOptionentabelle = strOptionentabelle
    Else
        strOptionentabelle = "tblOptionenArchiv"
        strOptionentabelle = InputBox("Keine Optionentabelle gefunden. Geben Sie den Namen der zu erstellenden 7
                                                                    Tabelle ein.", "Optionentabelle fehlt", strOptionentabelle)
        If Not Len(strOptionentabelle) = 0 Then
            If OptionentabelleAnlegen(strOptionentabelle) = True Then
                Me!txtOptionentabelle = strOptionentabelle
            End If
        End If
    End If
    Me!lstTabellen.RowSourceType = "Value List"
    Me!lstTabellen.RowSource = TabellenEinlesen
    Me!txtSicherungstabelle = DLookup("NameDerArchivtabelle", strOptionentabelle)
End Sub
```

Listing 1: Diese Prozedur wird beim Laden des Formulars ausgelöst.

übergibt dieser einen Rückgabeparameter namens **strOptionentabelle**, der modulweit wie folgt deklariert wird:

```
Dim strOptionentabelle As String
```

Die Funktion hat die Aufgabe, aus den Tabellen der Anwendung diejenige zu identifizieren, die vom Add-In als Optionentabelle definiert wurde. Diese wird nicht am Namen erkannt, sondern anhand einer speziell zu diesem Zweck zu der Tabelle hinzugefügten Eigenschaft namens **Optionentabelle**. Die Funktion **OptionentabelleFinden** schauen wir uns weiter unten an, zunächst reicht es uns zu wissen, dass diese Funktion die Variable **strOptionentabelle** mit dem Namen der Tabelle füllt – oder aber den Wert **False** als Funktionswert zurückliefert, wenn keine passende Tabelle vorhanden ist.

Liegt eine Optionentabelle vor, trägt die Prozedur ihren Namen in das Textfeld **txtOptionentabelle** ein, anderenfalls fragt sie vom Benutzer per **InputBox** den gewünschten Namen für die Optionentabelle ab. Damit ruft die Prozedur eine weitere Funktion namens **OptionentabelleAnlegen** auf und übergibt dieser den Namen der anzulegenden Tabelle. Diese Funktion (siehe weiter unten) legt die Tabelle an und liefert im Erfolgsfall den Wert **True** zurück.

Schließlich stellt die Prozedur die Eigenschaft **Datensatzherkunft** für das Listenfeld **lstTabellen** auf einen Wert ein, den die Funktion **TabellenEinlesen** liefert. Das Ergebnis dieser Funktion ist eine kommaseparierte Liste der Tabellennamen der aktuellen Anwendung. Schließlich liest die Prozedur noch den aktuellen Ausdruck für die Erstellung der Archivtabellen aus dem Feld **NameDerArchivtabelle** aus der Optionentabelle aus und trägt diesen in das Textfeld **txtSicherungstabelle** ein.

Optionstabelle erstellen

In der Regel müssen Sie einer Anwendung immer erst eine Optionentabelle hinzufügen, sofern diese noch nicht vorhanden ist. Dies erledigt die Funktion **OptionentabelleAnlegen** aus Listing 2. Diese erwartet den Namen der zu erstellenden Tabelle als Parameter. Die Funktion erstellt zunächst die Tabelle mit dem angegebenen Namen als neues **TableDef**-Objekt und fügt dieser dann das Feld **ArchivierungDeaktiviert** sowie das Feld **NameDerArchivtabelle** hinzu. Dies geschieht in zwei Schritten – dem eigentlichen Erstellen mit der Methode **CreateField** des **TableDef**-Objekts und dem Anhängen an die **Fields**-Auflistung der Tabelle. Danach hängt die Funktion die Tabelle selbst an die **TableDefs**-Auflistung an.

Damit das Add-In später auf irgendeine Weise die neu erstellte Optionentabelle identifizieren kann, fügen wir dieser noch eine benutzerdefinierte Eigenschaft hinzu. Dazu erstellt die Funktion mit der **CreateProperty**-Funktion eine neue Property namens **Optionentabelle** und stellt den Datentyp auf **Boolean** ein. Danach hängt sie die **Property** an die **Properties**-Auflistung an. Die **Refresh**-Methode aktualisiert die **TableDefs**-Auflistung und die Methode **RefreshDatabaseWindow** sorgt dafür, dass die neue Tabelle gleich im Navigationsbereich erscheint.

Schließlich fügt die Funktion direkt noch einen einzigen Datensatz zur Tabelle hinzu und stellt diesen auf den Wert **False** ein. Die Tabelle sieht nun wie in Bild 5 aus.

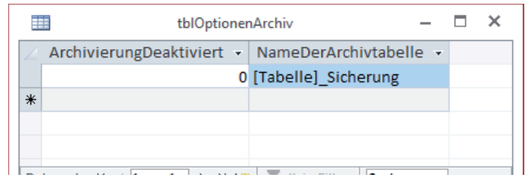

Bild 5: Tabelle zum Speichern einer Option

Optionentabelle auffinden

Natürlich müssen wir dem Add-In auch eine Funktion hinzufügen, mit der dieses beim Starten prüfen kann, ob die benötigte Optionentabelle bereits vorhanden ist. Dies erledigt die Funktion **OptionentabelleFinden** aus Listing 3. Diese erwartet ebenfalls einen Parameter namens **strOptionentabelle**, dieser soll jedoch mit dem Namen der gefundenen Tabelle als Rückgabewert gefüllt werden. Außerdem gibt der Funktionswert selbst einen **Boolean**-Wert zurück, der aussagt, ob eine Optionentabelle gefunden wurde.

Die Funktion **OptionenTabelleFinden** durchläuft alle Tabellen der Datenbank und trägt bei deaktivierter Fehlerbehandlung den Wert der Property **Optionentabelle** der aktuellen Tabelle in die Variable **strTemp** ein. Löst dies

```
Private Function OptionentabelleAnlegen(strOptionentabelle As String) As Boolean
    Dim db As DAO.Database
    Dim tdf As DAO.TableDef
    Dim prp As DAO.Property
    Dim fld As DAO.Field
    Set db = CurrentDb
    Set tdf = db.CreateTableDef(strOptionentabelle)
    Set fld = tdf.CreateField("ArchivierungDeaktiviert", dbBoolean)
    tdf.Fields.Append fld
    Set fld = tdf.CreateField("NamederArchivtabelle", dbText, 255)
    tdf.Fields.Append fld
    db.TableDefs.Append tdf
    Set prp = tdf.CreateProperty("Optionentabelle", dbBoolean, True)
    tdf.Properties.Append prp
    db.TableDefs.Refresh
    Application.RefreshDatabaseWindow
    db.Execute "INSERT INTO " & strOptionentabelle & "(ArchivierungDeaktiviert, NameDerArchivTabelle) 7
                                                    VALUES(0, '[Tabelle]_Archiv')", dbFailOnError
    OptionentabelleAnlegen = True
End Function
```

Listing 2: Anlegen der Optionentabelle

```
Private Function OptionentabelleFinden(strOptionentabelle As String) As Boolean
    Dim db As DAO.Database
    Dim tdf As DAO.TableDef
    Dim strTemp As String
    Dim bolOptionentabelle As Boolean
    Set db = CurrentDb
    For Each tdf In db.TableDefs
        On Error Resume Next
        strTemp = tdf.Properties("Optionentabelle").Name
        bolOptionentabelle = Err.Number = 0 And Not (Left(tdf.Name, 1) = "~")
        On Error GoTo 0
        If bolOptionentabelle = True Then
            strOptionentabelle = tdf.Name
            OptionentabelleFinden = True
            Exit Function
        End If
    Next tdf
End Function
```

Listing 3: Auffinden einer bereits vorhandenen Optionentabelle

Nach Daten im Unterformular suchen

Die Konstellation von Haupt- und Unterformular zur Darstellung von Daten aus 1:n- beziehungsweise m:n-Beziehungen ist bekannt. Einen Datensatz im Hauptformular zu suchen ist auch kein Hexenwerk. Aber wie sieht es aus, wenn wir das Hauptformular nach den Datensätzen filtern wollen, deren verknüpfte Tabelle einen Datensatz mit einem bestimmten Kriterium enthält? Und wenn wir dann noch einen Schritt weitergehen und noch den ersten passenden Datensatz im Unterformular markieren wollen? Wie dies gelingt, zeigt der vorliegende Beitrag.

Die Tabellen der Beispieldatenbank rekrutieren sich wieder eimal aus der Südsturm-Datenbank, unserer angepassten Nordwind-Variante. Im ersten Beispiel schauen wir uns die Kategorien im Hauptformular an und die Artikel einer jeden Kategorie im Unterformular.

Mit einem Suchfeld im Hauptformular wollen wir nach Artikeln filtern. Außerdem soll es zwei Schaltflächen geben, mit denen wir zwischen den Ergebnissen hin- und herblättern können. Das Formular soll dann wie in Bild 1 aussehen.

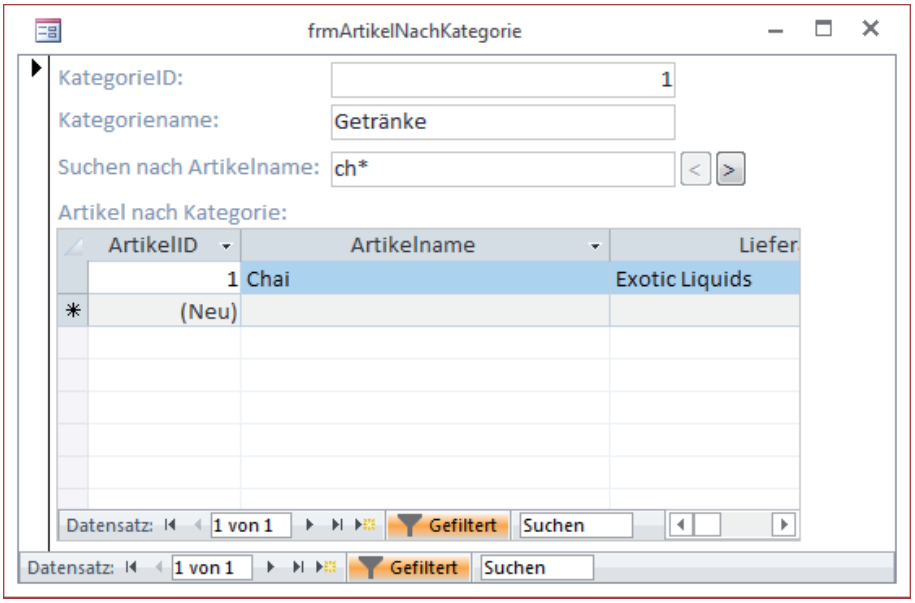

Bild 1: Formular mit Filter nach Artikelname, Variante I

Aufbau des Formulars

Das Hauptformular verwendet die Tabelle **tblKategorien** als Datenherkunft und zeigt die beiden Felder **KategorieID** und **Kategoriename** an. Das Unterformular steuert die Daten der Tabelle **tblArtikel** bei, und zwar in der Datenblattansicht. Damit es jeweils nur die Datensätze anzeigt, die mit dem aktuellen Datensatz der Tabelle **tblKategorien** im Hauptformular verknüpft sind, erhalten die Eigenschaften **Verknüpfen von** und **Verknüpfen nach** des Unterformular-Steuerelements jeweils den Wert **KategorieID**.

Steuerelemente

Die zusätzlichen Steuerelemente im Formular heißen **txtSuchbegriff, cmdVorheriger** und **cmdNaechster**. Wenn der Benutzer einen Begriff in das Textfeld **txtSuchbegriff** eingibt, soll die erste Kombination aus Kategorie und Artikel gefunden werden, deren Artikelname dem im Suchfeld eingegebenen Ausdruck entspricht (Platzhalter wie das Sternchen (*) sind dabei erlaubt).

Die beiden Schaltflächen **cmdVorheriger** und **cmdNaechster** sind beim Laden des Formulars noch deaktiviert. Erst, wenn der Benutzer einen Suchbegriff eingibt, wird geprüft, ob eine der Schaltflächen aktiviert werden soll – oder auch beide. Zeigt das Formular den ersten Tref-

fer an und gibt es noch einen weiteren, soll die Schaltfläche **cmdNaechster** aktiviert werden. Blättert der Benutzer damit zum folgenden Treffer, soll auch die Schaltfläche **cmdVorheriger** aktiviert werden. Die Schaltfläche **cmdNaechster** bleibt dabei verfügbar, bis der Benutzer zum letzten Ergebnis weitergeklickt hat.

Programmierung der Suche

Die Suchfunktion erfordert einen etwas anderen Ansatz als übliche Suchfunktionen. Direkt nach der Eingabe des Suchbegriffs erstellen wir ein Recordset, das die Tabelle **tblArtikel** aufnimmt – mit dem eingegebenen Suchbegriff als Vergleichswert für das Feld **Artikelname**. Die Tabelle liefert für jedes Suchergebnis sowohl die **ArtikelID** also auch die **KategorieID** aus dem entsprechenden Fremdschlüsselfeld. Damit können wir dann also sowohl den gesuchten Datensatz im Unterformular einstellen als auch die dazu passende Kategorie im Hauptformular.

Schaltflächen deaktivieren

Beim Laden des Formulars sollen die beiden Schaltflächen **cmdVorheriger** und **cmdNaechster** zunächst deaktiviert sein. Dazu legen wir für das Ereignis **Beim Laden** die folgende Ereignisprozedur an:

```
Private Sub Form_Load()
    Me!cmdVorheriger.Enabled = False
    Me!cmdNaechster.Enabled = False
End Sub
```

Nach der Eingabe eines Suchbegriffes und dem Auslösen des Ereignisses **Nach Aktualisierung** soll die Prozedur aus Listing 1 ausgelöst werden. Die Prozedur füllt die Variable **db** mit einem Verweis auf das **Database**-Objekt der aktuellen Datenbank. Dann liest sie den Suchbegriff aus dem Textfeld **txtSuchbegriff** in die Variable **strSuchbegriff** ein. Hat der Suchbegriff eine Länge von mehr als null Zeichen, erstellt die Prozedur ein neues Recordset, das alle Datensätze der Tabelle **tblArtikel** enthält, deren Feld Artikelname dem Suchbegriff entspricht. Dieses Recordset speichert die Prozedur in einer Variablen, die im Kopf des Klassenmoduls wie folgt deklariert wird und damit von allen Prozeduren des Moduls aus erreichbar ist:

```
Dim rstErgebnis As DAO.Recordset
```

Wir verwenden den Wert **dbOpenSnapshot** als Parameter, da dies direkt den Wert der Eigenschaft **Recordcount** des Recordsets verfügbar macht. Nachdem dies geschehen ist, ruft die Prozedur zwei weitere Routinen namens **SteuerelementeAktualisieren** und **Filtern** auf. Erstere aktiviert oder deaktiviert die beiden Schaltflächen **cmdVorheriger** und **cmdNaechster** in Abhängigkeit von der Datensatzposition, der zweite filtert die Daten in Haupt- und Unterformular nach dem aktuellen Datensatz des Recordsets **rstErgebnis**.

```
Private Sub txtSuchbegriff_AfterUpdate()
    Dim db As DAO.Database
    Dim strSuchbegriff As String
    Set db = CurrentDb
    strSuchbegriff = Nz(Me!txtSuchbegriff, "")
    If Len(strSuchbegriff) > 0 Then
        Set rstErgebnis = db.OpenRecordset("SELECT ArtikelID, KategorieID FROM tblArtikel WHERE Artikelname LIKE '" _
            & strSuchbegriff & "' ORDER BY tblKategorien.KategorieID, tblArtikel.ArtikelID", dbOpenSnapshot)
        SteuerelementeAktualisieren
        Filtern
    Else
        FilterAufheben
    End If
End Sub
```

Listing 1: Prozedur für das Ereignis **Nach Aktualisierung** des Suchfeldes

Sollte das Textfeld **txtSuchbegriff** keinen Wert enthalten, ruft die Prozedur die Routine **FilterAufheben** auf, was wieder alle Datensätze in Haupt- und Unterformular anzeigt.

Steuerelemente aktualisieren

Die Routine **SteuerelementeAktualisieren** kümmert sich um das Aktivieren und Deaktivieren der beiden Schaltflächen **cmdVorheriger** und **cmdNaechster**. Die erste **If...Then**-Bedingung dieser Routine prüft, ob die aktuelle Position des Datensatzzeigers von **rstErgebnis** kleiner als die Anzahl der Datensätze minus eins ist.

Minus eins deshalb, weil **AbsolutePosition** für den ersten Datensatz den Wert **0** liefert. In diesem Fall aktiviert die Routine die Schaltfläche **cmdNaechster**, anderenfalls wird sie deaktiert. Bei der Schaltfläche **cmdVorheriger** sieht es ähnlich aus: Die **If...Then**-Bedingung prüft, ob **AbsolutePosition** größer **0** ist. Falls ja, kann der Benutzer noch einen Datensatz nach vorn blättern und die Schaltfläche **cmdVorheriger** wird aktiviert:

```
Private Sub SteuerelementeAktualisieren()
    If rstErgebnis.AbsolutePosition _
            < rstErgebnis.RecordCount - 1 Then
        Me!cmdNaechster.Enabled = True
    Else
        Me!cmdNaechster.Enabled = False
    End If
    If rstErgebnis.AbsolutePosition > 0 Then
        Me!cmdVorheriger.Enabled = True
    Else
        Me!cmdVorheriger.Enabled = False
    End If
End Sub
```

Filtern der Artikel

Die Routine **Filtern** sorgt für die Anzeige des jeweils aktuellen Datensatzes des Recordsets **rstErgebnis** (s. Listing 2). Dabei prüft die Routine zunächst, ob das Recordset mindestens einen Datensatz enthält. Falls ja, stellt sie die Eigenschaft **Filter** des Hauptformulars auf einen Ausdruck ein, bei dem der Wert von **KategorieID** der Datenherkunft dem Wert dieses Feldes im aktuelle Datensatz des Recordsets entspricht und aktiviert den Filter durch Setzen von **FilterOn** auf **True**.

```
Private Sub Filtern()
    If Not rstErgebnis.RecordCount = 0 Then
        With Me
            .Filter = "KategorieID = " & rstErgebnis!KategorieID
            .FilterOn = True
        End With
        With Me!sfmArtikelNachKategorie.Form
            .Filter = "ArtikelID = " & rstErgebnis!ArtikelID
            .FilterOn = True
        End With
    Else
        With Me
            .Filter = "1=2"
            .FilterOn = True
        End With
    End If
End Sub
```

Listing 2: Prozedur zum Filtern der Daten in Haupt- und Unterformular

Auf die gleiche Weise filtert es das Unterformular so, dass nur der Artikel aus dem aktuellen Datensatz des Recordsets erscheint. Liefert **rstErgebnis** keinen Datensatz, stellt die Prozedur den Filter für das Hauptformular auf den Ausdruck **1=2** ein, was keine Datensätze liefert. Dementsprechend bleibt auch das Unterformular leer.

Aufheben des Filters

Leert der Benutzer das Textfeld **strSuchbegriff** und löst das Ereignis **Nach Aktualisierung** des Textfeldes aus, ruft die Prozedur **txtSuchbegriff_AfterUpdate** wie oben erwähnt die Prozedur **FilterAufheben** auf.

Diese leert die Eigenschaft **Filter** sowohl des Unterformulars als auch des Hauptformulars. Dabei ist die Reihenfolge wichtig – Sie müssen erst den Filter im Unterformular aufheben und dann den im Hauptformular, anderenfalls zeigt das Formular die Datensätze im Unterformular nicht korrekt an:

```
Private Sub FilterAufheben()
    Me!sfmArtikelNachKategorie.Form.Filter = ""
    Me.Filter = ""
    Set rstErgebnis = Nothing
    Me!cmdVorheriger.Enabled = False
    Me!cmdNaechster.Enabled = False
End Sub
```

Außerdem leert die Prozedur das Recordset **rstErgebnis** und deaktiviert die Schaltflächen **cmdVorheriger** und **cmdNaechster**.

Funktion der Schaltflächen

Die Schaltfläche **cmdNaechster** soll beim Anklicken das nächste Suchergebnis liefern, also den folgenden Datensatz der Recordsets **rstErgebnis**. Dazu bewegt die Prozedur den Datensatzzeiger mit der Methode **MoveNext** zum folgenden Datensatz und ruft dann die beiden Routinen **SteuerelementeAktualisieren** und **Filtern** auf, um sowohl die Aktivierung der Steuerelemente zu prüfen als auch Haupt- und Unterformular nach dem aktuellen Suchergebnis zu filtern:

```
Private Sub cmdNaechster_Click()
    rstErgebnis.MoveNext
    SteuerelementeAktualisieren
    Filtern
End Sub
```

Die Prozedur, die durch einen Klick auf die Schaltfläche **cmdVorheriger** ausgelöst wird, erledigt die gleiche Aufgabe, springt aber zum vorherigen Datensatz des Recordsets mit den Suchergebnissen:

```
Private Sub cmdVorheriger_Click()
    rstErgebnis.MovePrevious
    SteuerelementeAktualisieren
    Filtern
End Sub
```

Die Prozeduren sorgen in dieser Form dafür, dass Haupt- und Unterformular jeweils nur einen Datensatz anzeigen. Das mag für bestimmte Anwendungszwecke passen, aber nicht für alle – also stellen wir eine Alternative vor.

Variante II: Suchergebnis aktivieren statt filtern

Die zweite Variante soll das Haupt- und Unterformular nicht nach dem aktuell gefundenen Datensatz filtern, sondern weiterhin alle verfügbaren Datensätze anzeigen. Allerdings soll der jeweilige Datensatz im Unterformular markiert werden.

Formular und Unterformular sind gleich aufgebaut, allerdings heißt das Formular nun **frmArtikelNachKategorien_Markieren** und das Unterformular **sfmArtikelNachKategorien_Markieren** (s. Bild 2).

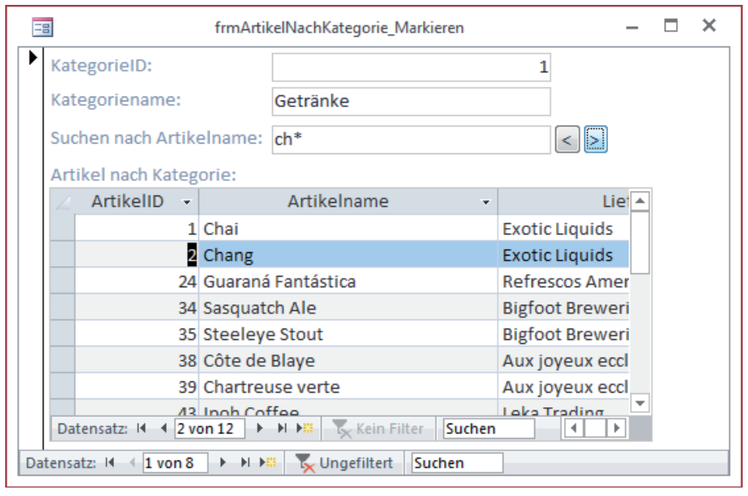

Bild 2: Formular mit Filter nach Artikelname, Variante II


```
Private Sub txtSuchbegriff_AfterUpdate()
    Dim db As DAO.Database
    Dim strSuchbegriff As String
    Set db = CurrentDb
    strSuchbegriff = Nz(Me!txtSuchbegriff, "")
    If Len(strSuchbegriff) > 0 Then
        Set rstErgebnis = db.OpenRecordset("SELECT ArtikelID, KategorieID FROM tblArtikel WHERE Artikelname LIKE '" _
            & strSuchbegriff & "' ORDER BY KategorieID, ArtikelID", dbOpenSnapshot)
        SteuerelementeAktualisieren
        Markieren
    Else
        MarkierungAufheben
    End If
End Sub
```

Listing 3: Aktionen nach dem Eingeben des Suchbegriffs

Nach der Eingabe des Suchbegriffs sollen Haupt- und Unterformular gar nicht gefiltert werden, sondern nur die jeweiligen Datensätze markiert werden. Auch hier verwenden wir ein Recordset zum Ermitteln der Suchergebenisse:

```
Dim rstErgebnis As DAO.Recordset
```

Beim Laden des Formulars deaktivieren wir wieder die beiden Schaltflächen:

```
Private Sub Form_Load()
    Me!cmdVorheriger.Enabled = False
    Me!cmdNaechster.Enabled = False
End Sub
```

Nach der Eingabe des Suchbegriffs wird die Prozedur aus Listing 3 ausgelöst. Diese sieht so ähnlich aus wie die aus dem vorherigen Beispiel, aber sie ruft die Routine Markieren statt Filtern auf. Und bei Aktualisierung bei leerem Textfeld erfolgt der Aufruf der Routine **MarkierungAufheben**, nicht mehr von **FilterAufheben**.

Fund markieren

Die Prozedur **Markieren** prüft zunächst, ob das Recordset mit der Ergebnisliste überhaupt einen Eintrag enthält. Falls ja, ruft sie die **FindFirst**-Methode des **Recordset**-Objekts des Hauptformulars auf, um die richtige Kategorie auszuwählen (s. Listing 4).

Dabei übergibt sie ein Kriterium, das den Wert des Feldes **KategorieID** des Recordsets enthält. Das Hauptformular zeigt so schon einmal den richtigen Datensatz an.

Nun soll das Unterformular noch den aktuellen Datensatz der Ergebnisliste aus **rstErgebnis** liefern. Deshalb verwenden wir auch hier die **FindFirst**-Methode des

```
Private Sub Markieren()
    If Not rstErgebnis.RecordCount = 0 Then
        Me.Recordset.FindFirst "KategorieID = " & rstErgebnis!KategorieID
        Me!sfmArtikelNachKategorie_Markieren.Form.Recordset.FindFirst "ArtikelID = " & rstErgebnis!ArtikelID
    Else
        MsgBox "Keine passenden Daten gefunden"
        MarkierungAufheben
    End If
End Sub
```

Listing 4: Diese Prozedur markiert den nächsten Eintrag des Ergebnis-Recordsets.

entsprechenden Recordset-Objekts – nur diesmal mit dem Kriterium **"ArtikelID = " & rstErgebnis!ArtikelID**. Dadurch landet der Datensatzzeiger im Unterformular genau bei dem gesuchten Datensatz.

Markierung aufheben

Sollen die Markierungen wieder entfernt werden, was beispielsweise nach dem Leeren und Aktualisieren des Textfeldes **txtSuchbegriff** geschehen soll, wird die folgende Prozedur aufgerufen:

```
Private Sub MarkierungAufheben()
    Me.Recordset.MoveFirst
    Me!sfmArtikelNachKategorie_Markieren.Form.Recordset.MoveFirst
    Set rstErgebnis = Nothing
    Me!cmdVorheriger.Enabled = False
    Me!cmdNaechster.Enabled = False
End Sub
```

Diese Prozedur springt wieder zum ersten Datensatz des Hauptformulars und dann zum ersten Datensatz des Unterformulars. Das Recordset mit der Ergebnisliste wird geleert und die Schaltflächen zum Vor- und Zurückblättern werden deaktiviert.

In Ergebnissen blättern

Das Verhalten beim Blättern in den gefundenen Datensätzen sieht natürlich ganz anders aus: Wenn etwa eine Kategorie gleich mehrere Treffer liefert, bleibt die Kategorie im Hauptformular erhalten und es werden beim Anklicken der Schaltfläche **cmdNaechster** nacheinander die einzelnen Funde im Unterformular markiert:

```
Private Sub cmdNaechster_Click()
    rstErgebnis.MoveNext
    SteuerelementeAktualisieren
    Markieren
End Sub
```

Genauso läuft es natürlich auch andersherum:

```
Private Sub cmdVorheriger_Click()
    rstErgebnis.MovePrevious
    SteuerelementeAktualisieren
    Markieren
End Sub
```

Die Prozedur zum Aktivieren oder Deaktivieren der beiden Schaltflächen **cmdNaechster** und **cmdVorheriger** entspricht genau dem vorherigen Beispiel, daher führen wir diese hier nicht nochmals auf.

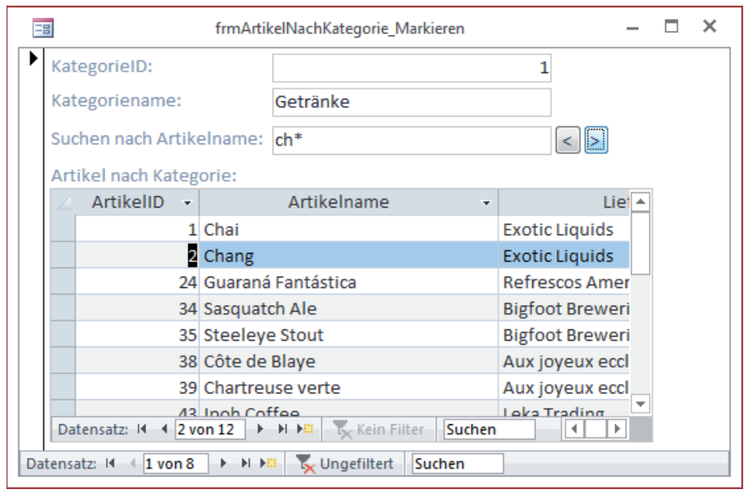

Bild 3: Suche nach Daten aus einer m:n-Beziehung

Nach Daten aus m:n-Beziehungen suchen

Im letzten Beispiel schauen wir uns eine Konstellation aus Haupt- und Unterformular an, bei der Sie ein Feld durchsuchen, das über die im Unterformular angezeigte m:n-Beziehung mit dem Datensatz im Hauptformular verknüpft ist. Das Beispielformular finden Sie in Bild 3.

Die Prozeduren übernehmen wir weitgehend aus den vorherigen Beispielen. Einen wichtigen Unterschied finden Sie jedoch in der Prozedur, die nach der Eingabe eines Suchbegriffs ausgelöst wird (s. Listing 5). Diese stellt eine etwas andere SQL-Abfrage zusammen. Diese berücksichtigt die beiden verknüpften Tabellen **tblBestelldetails** und **tblArtikel**, weil wir ja die Daten des Formulars nach dem Namen des Artikels durchsuchen wollen.

```
Private Sub txtSuchbegriff_AfterUpdate()
    Dim db As DAO.Database
    Dim strSuchbegriff As String
    Set db = CurrentDb
    strSuchbegriff = Nz(Me!txtSuchbegriff, "")
    If Len(strSuchbegriff) > 0 Then
        Set rstErgebnis = db.OpenRecordset("SELECT tblBestelldetails.BestellungID, tblBestelldetails.BestelldetailID, 
            tblBestelldetails.ArtikelID FROM tblBestelldetails INNER JOIN tblArtikel ON tblBestelldetails.ArtikelID 
                = tblArtikel.ArtikelID WHERE tblArtikel.Artikelname LIKE '" & strSuchbegriff & "' 
                ORDER BY tblBestelldetails.BestelldetailID", dbOpenSnapshot)
        SteuerelementeAktualisieren
        Markieren
    Else
        MarkierungAufheben
    End If
End Sub
```

Listing 5: Zusammenstellen der Ergebnisabfrage nach dem Eingeben eines Suchbegriffs

Der zweite Unterschied findet sich in der Prozedur, welche die gefundenen Daten in Haupt- und Unterformular markiert. Hier haben wir noch zwei Zeilen hinzugefügt, die dafür sorgen, dass der Datensatz im Unterformular auch noch komplett hervorgehoben wird. Dafür verschieben wir zunächst den Fokus auf das Unterformular und rufen dann die Methode **RunCommand** mit dem Parameter **acCmdSelectRecord** auf, um die komplette Spalte zu markieren:

```
Private Sub Markieren()
    If Not rstErgebnis.RecordCount = 0 Then
        Me.Recordset.FindFirst "BestellungID = " _
            & rstErgebnis!BestellungID
        Me!sfmArtikelNachBestellung.Form.Recordset.FindFirst "ArtikelID = " & rstErgebnis!ArtikelID
        Me!sfmArtikelNachBestellung.SetFocus
        RunCommand acCmdSelectRecord
    Else
        MsgBox "Keine passenden Daten gefunden"
        MarkierungAufheben
    End If
End Sub
```

Zusammenfassung und Ausblick

Dieser Beitrag hat drei Varianten für die Suche nach Daten der Datenherkunft des Unterformulars in Haupt-/Unterformularkonstellationen geliefert.

Die erste zeigt jeweils nur die Kombination aus Master-/Detaildatensatz an, die zum aktuellen Suchergebnis passt.

Die zweite schränkt die Datenherkunft von Haupt- und Unterformular nicht ein, sondern springt zu den gefundenen Datensätzen.

Die dritte liefert ein etwas komplexeres Beispiel, da die Daten des Unterformulars aus einer m:n-Beziehung stammen.

Outlook-Mails nach Empfang archivieren

In der Beitragsreihe »Outlook-Mails in Access archivieren« haben wir gezeigt, wie Sie die E-Mails aus kompletten und auch untergeordneten Outlook-Ordnern in eine Access-Datenbank importieren. Das ist die Lösung für den Start der Archivierung. Interessant wird es erst, wenn Sie diese Daten direkt nach dem Eingang in Outlook archivieren. Der vorliegende Beitrag zeigt, wie Sie Outlook so erweitern, dass jede in vorgegebenen Ordnern eingehenende E-Mail direkt in unsere Access-Mail-Archiv weitergeleitet wird.

Um das vorgegebene Ziel zu erreichen, sind einige Änderungen am VBA-Projekt von Outlook erforderlich. Sprich: Wir verwenden in diesem Beitrag Access nur zur Bereitstellung der Datenbank, in die wir die Outlook-Mails exportieren möchten. Diese Datenbank haben wir in der Beitragsreihe **Outlook-Mails in Access archivieren** (**www.access-im-unternehmen.de/985**, **990** und **997**) ausführlich vorgestellt.

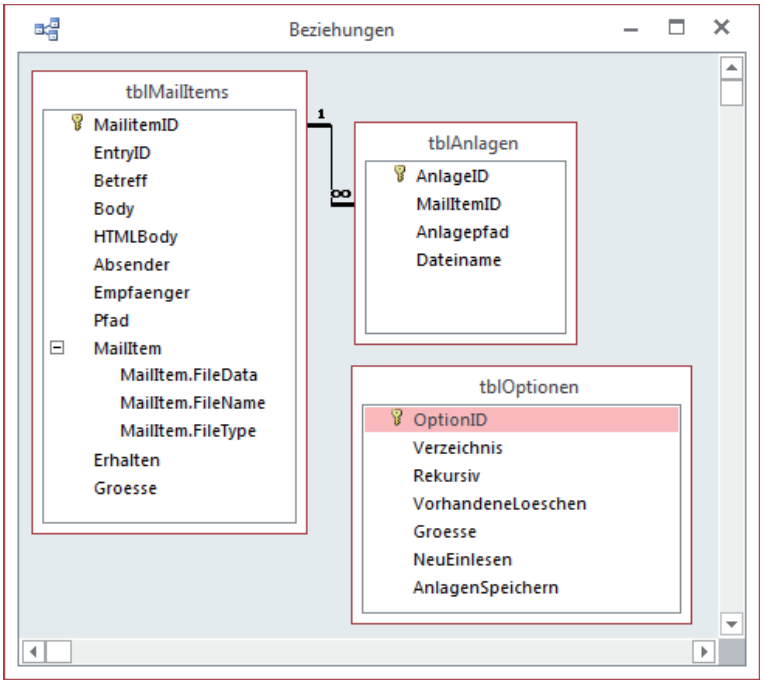

Bild 1: Tabellen der Access-Seite der Lösung

Von dort benötigen wir drei Tabellen (siehe auch Bild 1):

- **tblOptionen**: Enthält die vom Benutzer definierten Outlook-Ordner, die beim Archivieren von E-Mails berücksichtigt werden sollen, und gibt beispielsweise den Ordnerpfad und die Einbeziehung von Unterordnern an.
- **tblMailItems**: Ziel des Exports. Hier werden die Metadaten der E-Mails gespeichert. Die eigentlichen Mails, also die .msg-Dateien, landen dort in einem eigenen Anlagefeld oder aber im Dateisystem.
- **tblAnlagen**: Speichert, sofern so konfiguriert, die Anlagen der E-Mails. Alternativ landen diese in der Verzeichnisstruktur.

Outlook vorbereiten

Damit Sie von Outlook aus auf die Elemente der Access-Objektbibliothek sowie auf die DAO-Bibliothek zugreifen können (in diesem Fall die neuere Version der DAO-Bibliothek, nämlich die **Microsoft Office x.0 Access Database Engine Object Library**), legen Sie im VBA-Projekt von Outlook zunächst zwei Verweise an. Der **Verweise**-Dialog (VBA-Editor, **Extras|Verweise**) sollte danach etwa wie in Bild 2 aussehen.

Den VBA-Editor öffnen Sie unter Outlook übrigens über die Tastenkombination **Alt + F11**. Die Kombination **Strg + G** funktioniert im Gegensatz zu Access nicht.

Der VBA-Editor präsentiert, sofern der Projekt-Explorer sichtbar ist, unter dem Ordner **Microsoft Outlook Objekte** die Klasse **ThisOutlookSession**. In dieser Klasse

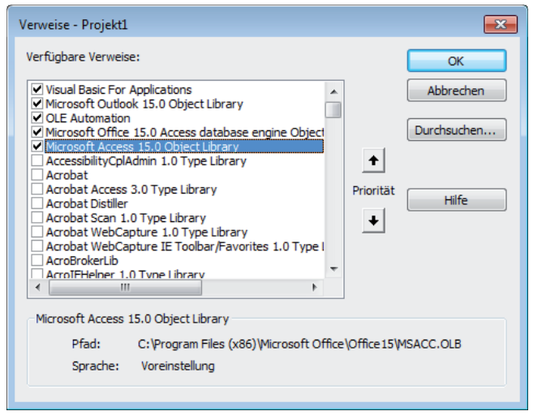

Bild 2: Verweise für den Export nach Access

landen die ersten Codezeilen dieser Lösung. Die erste lautet wie folgt und deklariert die Konstante zum Speichern des Verzeichnisses für die Ziel-Datenbank. Dazu ermitteln Sie zunächst den Speicherort der Datenbank, in der Sie die eingehenden E-Mails speichern wollen – zum Beispiel **c:\Outlook\Outlook_III.accdb**. Legen Sie dann die folgende Zeile im Kopf der Klasse an:

```
Const cStrExportdatenbank As String = _
                "C:\Outlook\Outlook_III.accdb"
```

Außerdem deklarieren wir hier noch eine **Collection**, deren Sinn wir weiter unten erläutern werden:

```
Dim colFolders As Collection
```

Neue Mails abfangen

Bevor wir weitermachen, benötigen wir ein paar theoretische Grundlagen. Wenn eine Mail abgerufen wird, landet diese in einem der Mail-Ordner in Outlook. Welcher das ist, hängt von den festgelegten Regeln ab. Normalerweise ist der Ordner Posteingang das Ziel einer neu eingetroffenen E-Mail.

Sie können jedoch festlegen, dass E-Mails mit bestimmten Eigenschaften direkt in einem anderen Ordner landen – beispielsweise solche mit bestimmten E-Mail-Adressen oder solche, die einen bestimmten Text im Betreff enthalten.

Um dies festzulegen, benötigen Sie eine sogenannte Regel. Zum Definieren dieser Regel suchen Sie sich am besten eine E-Mail, welche die betroffene Eigenschaft aufweist. Klicken Sie mit der rechten Maustaste auf die E-Mail und wählen Sie den Kontextmenü-Eintrag **Regeln\|Regel erstellen** aus. Nun erscheint der Dialog aus Bild 3. Dieser zeigt bereits einige möglichen Ausdrücke zum Formulieren der Regel an. In diesem Fall wollen wir, dass alle E-Mails, die wie die aktuelle das Schlüsselwort **Outlooktest** im Betreff enthalten, von der Regel betroffen sind.

Diese Mails sollen in einen bestimmten Outlook-Ordner verschoben werden, daher aktivieren Sie unten die Option **Element in Ordner verschieben**. Nach einem Klick auf **Ordner auswählen** erscheint nun der Dialog aus Bild 4, wo Sie den Zielordner festlegen. Ist diese Aufgabe erledigt, können Sie sich bereits eine Test-E-Mail mit genau diesem Betreff zusenden. Sie wird dann beim Abrufen wie erwartet in den angegebenen Ordner verschoben.

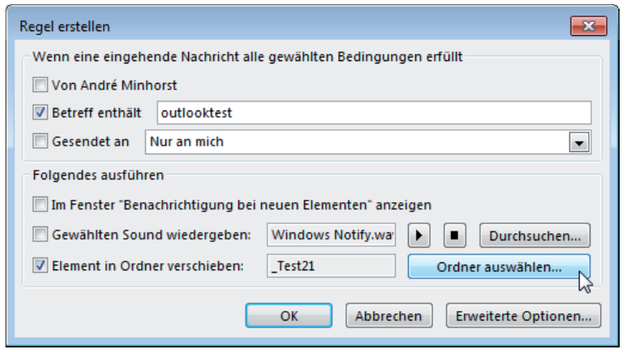

Bild 3: Anlegen einer neuen Regel für den Posteingang

Maileingang erkennen

Nun landet eine Mail also in einem von mehreren Ordnern, die wir in der Datenbank in der Tabelle **tblOptionen** festgelegt haben. All diese Ordner wollen wir nun ständig auf den Eingang von E-Mails überprüfen.

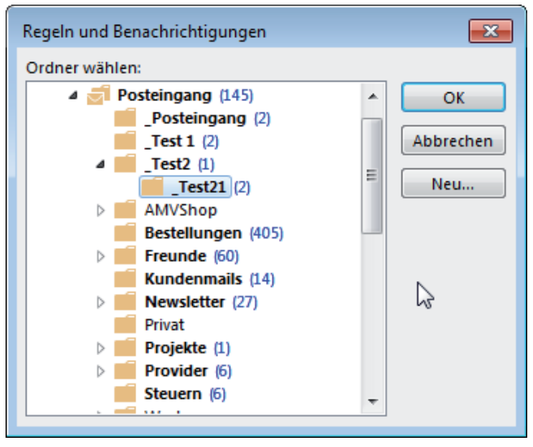

Bild 4: Auswählen des Zielordners

Dies gelingt nur, wenn wir herausfinden, welches Ereignis beim Maileingang ausgelöst wird, und dieses für die entsprechenden Elemente implementieren. Das Ganze muss natürlich dynamisch ausgelegt werden, da ja zuvor noch nicht feststeht, welche Ordner der Benutzer überhaupt für die Überprüfung vorgesehen hat.

Um es vorwegzunehmen: Es ist nicht das **Folder**-Objekt, das beim Maileingang ein Ereignis auslöst, sondern das **Items**-Objekt, also das Objekt, das die in einem Folder enthaltenen Objekte auflistet.

Andererseits können Sie Ereignisprozeduren für Objekte wie etwa einen Mailordner oder eine Liste wie **Items** nur innerhalb von Klassenprozeduren implementieren. Also wollen wir zu einer Technik greifen, die wir bereits in einigen anderen Beiträgen umgesetzt haben. Wir werden eine Klasse erstellen, welche einen Verweis auf ein **Items**-Objekt aufnehmen kann. Diese implementiert genau für dieses **Items**-Objekt, nennen wir es **objItems**, das Ereignis **ItemAdd**. Dieses Ereignis liefert ein **Item**-Objekt, das in unserem Fall in der Regel den Typ **MailItem** aufweisen sollte. Dies ist schon der Verweis auf die E-Mail, der wir nun die für das Archivieren in der Datenbank benötigten Daten entnehmen können.

Nun müssen wir nur noch ein oder mehrere dieser Klassen instanzieren, und zwar eine für jeden Ordner, der in der Tabelle **tblOptionen** der Archivdatenbank für die E-Mails enthalten ist. Diese sollen nach dem Erstellen nicht im Nirvana verschwinden, daher verwenden wir ein **Collection**-Objekt, um diese Objekte zu speichern – eben jenes, das wir weiter oben bereits deklariert haben.

Ordner-Klassen instanzieren

Die Klassen mit den zu überwachenden Ordnern sollen direkt beim Start von Outlook erstellt werden. Dazu legen Sie eine Prozedur namens **Application_Startup** in der Klasse **ThisOutlookSession** an (s. Listing 1). Diese wird gleich beim Öffnen von Outlook ausgeführt – vorausgesetzt, Sie haben die Ausführung von Makros nicht aus Sicherheitsgründen deaktiviert.

Die Prozedur erstellt zunächst eine Referenz auf das **Database**-Objekt der Zieldatenbank zum Exportieren der E-Mails. Wir benötigen zwei besondere Variablen: **objFolderArchiv** hat den Typ **clsFolderArchiv**. Dies ist die Klasse, die wir gleich im Anschluss für jeden zu beobachtenden Mailordner erstellen und die das mit der anderen Variablen **objFolder** referenzierte **Folder**-Objekt entgegennehmen wird.

Um in Access auf die aktuell geöffnete Datenbank zuzugreifen, würden wir die Funktion **CurrentDb** verwenden, aber von Outlook aus ist ein anderer Befehl nötig, nämlich die **OpenDatabase**-Methode des **DBEngine**-Objekts. Dieser übergeben Sie den Namen der gewünschten Datenbankdatei, den wir ja bereits für die Konstante **cStrExportdatenbank** definiert haben.

Im Anschluss öffnet die Prozedur ein Recordset auf Basis der Tabelle **tblOptionen**, die ja die Outlook-Ordner enthält, deren E-Mails archiviert werden sollen. Die

Spaltenbreiten optimieren mit Klasse

Wenn Sie Daten in der Datenblattansicht von Unterformularen anzeigen, stoßen Sie immer wieder auf das Problem, dass die Spaltenbreiten nicht gleich zu Beginn optimal an die Inhalte der Spalten angepasst werden – also an die Breite der angezeigten Daten. Immerhin kann der Benutzer die Breite in der Regel selbst anpassen. Praktischer aber wäre es, wenn die Spaltenbreiten gleich beim Anzeigen der Daten optimiert würden. Der vorliegende Beitrag zeigt eine kleine Klasse, mit der Sie dies bewerkstelligen können.

Für die Demonstration der in diesem Beitrag vorgestellten Klasse eignet sich am besten ein Unterformular in der Datenblattansicht, das seine Daten nach der Auswahl etwa eines Filters im Hauptformular ändert, wodurch eine erneute Optimierung der Spaltenbreiten erforderlich wird. Dazu legen wir ein Unterformular namens **sfmArtikel** an, das mit Feldern der Tabelle **tblArtikel** wie in Bild 1 gefüllt ist und seine Daten in der Datenblattansicht präsentiert.

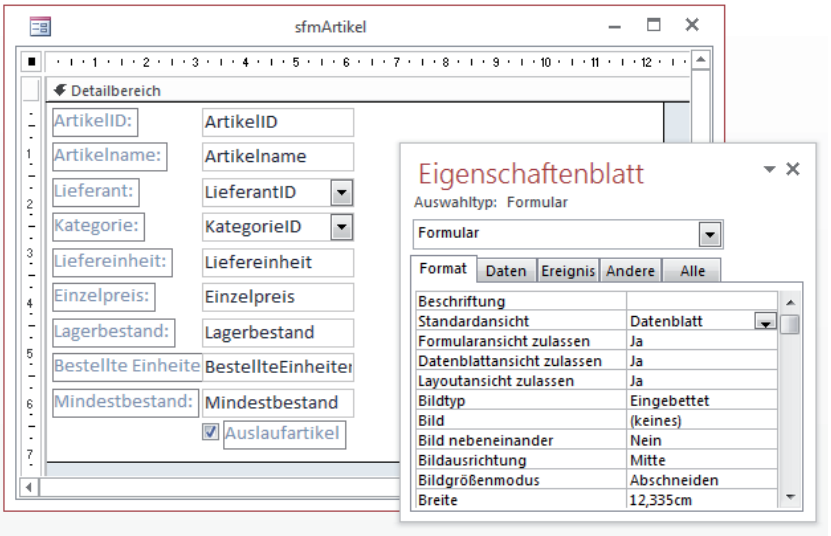

Bild 1: Vorbereitung für das automatische Anpassen der Spaltenbreiten

Dieses Formular fügen wir in einem weiteren Formular namens **frmArtikeNachKategorie** ein, dem wir als Datenherkunft die Tabelle **tblKategorien** hinzufügen. Das Formular soll die beiden Felder **KategorieID** und **Kategorie** anzeigen. Ziehen Sie das Unterformular **sfmArtikel** in den Entwurf. Dadurch sollten die beiden Eigenschaften **Verknüpfen von** und **Verknüpfen nach** automatisch auf den Wert **KategorieID** eingestellt werden (s. Bild 2).

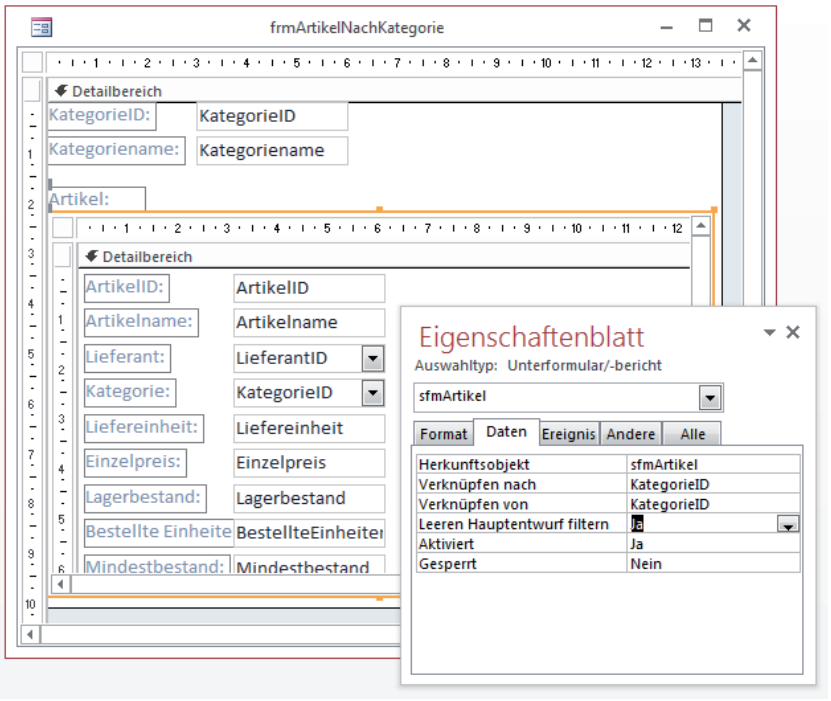

Bild 2: Verknüpfung zwischen Haupt- und Unterformular

Öffnen Sie das Hauptformular nun in der Formularansicht, kann es sein, dass die Spaltenbreiten entweder viel zu breit oder zu schmal für die anzuzeigenden Inhalte eingestellt sind (s. Bild 3). Dies wollen wir ändern, indem wir die Spal-

tenbreiten zu jeder Aktualisierung der Datenherkunft des Unterformulars neu anpassen.

Dazu sind, wenn wir die fertige Klasse, welche diese Aufgabe erledigt, als vorhanden voraussetzen, nur wenige Handgriffe nötig.

Automatische Spaltenoptimierung einbauen

Damit die Spalten automatisch angepasst werden, sind nur drei Schritte nötig:

- Hinzufügen der Klasse **clsColumnWidths**
- Schreiben einiger Zeilen Code ins Klassenmodul des Hauptformulars
- Hinzufügen des Klassenmoduls zum Unterformular

Klasse nutzen

Nachdem Sie die Klasse **clsColumnWidths** zum VBA-Projekt der Datenbank hinzugefügt haben, legen Sie eine neue Ereignisprozedur für das Ereignis **Beim Laden** des Hauptformulars an. Diese füllen Sie wie folgt:

```
Private Sub Form_Load()
    Set objColumnWidths = New clsColumnWidths
    With objColumnWidths
        Set .DataSheetForm = Me!sfmArtikel.Form
       .OptimizeColumnWidths
    End With
End Sub
```

Außerdem benötigen Sie noch eine Deklarationszeile für das Objekt **objColumnWidths**, die Sie im Kopf des Klassenmoduls einfügen:

```
Dim objColumnWidths As clsColumnWidths
```

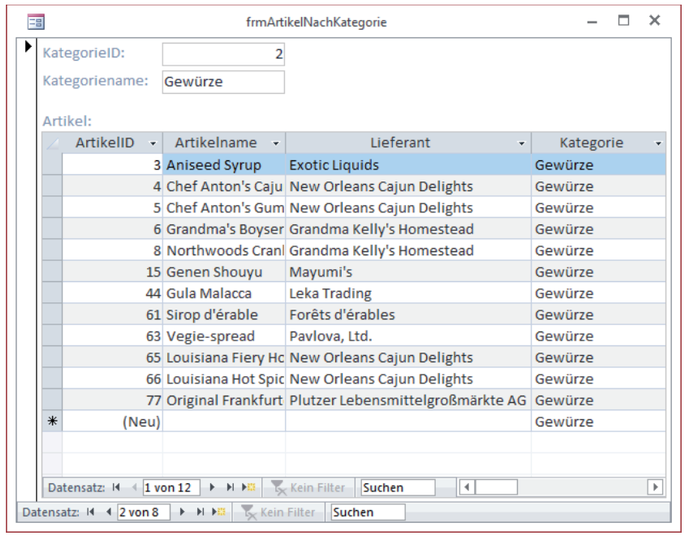

Bild 3: Unglückliche Darstellung der Spalten

Die Prozedur instanziert zunächst ein Objekt auf Basis der Klasse **clsColumnWidths**. Danach übergibt sie diesem Objekt über seine Eigenschaft **DataSheetForm** einen Verweis auf das Unterformular, das mit der Funktion zum automatischen Anpassen der Spaltenbreiten ausgestattet werden soll. Schließlich wird die Methode **OptimizeColumnWidths** zum Optimieren der Spalten einmalig aufgerufen, damit das Unterformular gleich optimiert wird. Wenn Sie das Formular nun öffnen, ohne den oben genannten dritten Schritt durchzuführen, tut sich nichts – die Spaltenbreiten behalten die vorherigen Einstellungen und werden nicht optimiert.

Sie müssen also unbedingt das Klassenmodul zum Unterformular **sfmArtikel** hinzufügen – auch wenn Sie gar keine Ereignisprozeduren für das Unterformular benötigen. Dies erledigen Sie, indem Sie die Eigenschaft **Enthält Modul** des Unterformulars auf den Wert **Ja** einstellen. Danach sollten die Spaltenbreiten etwa wie in Bild 4 erscheinen.

Das Objekt **objColumnWidths** haben wir modulweit deklariert, damit es auch nach der Ausführung der Ereignisprozedur **Form_Load** noch erhalten bleibt. Auf diese Weise wird es nun regelmäßig ausgelöst, wenn der Benutzer den Datensatz im Unterformular wechselt oder das Unterformular gefiltert wird. Dies geschieht beispielsweise auch, wenn Sie den Kategorie-Datensatz im Hauptformular wechseln. Sie können einmal durch die Datensätze scrollen und betrachten, wie die Spaltenbreiten jeweils an die angezeigten Daten angepasst werden.

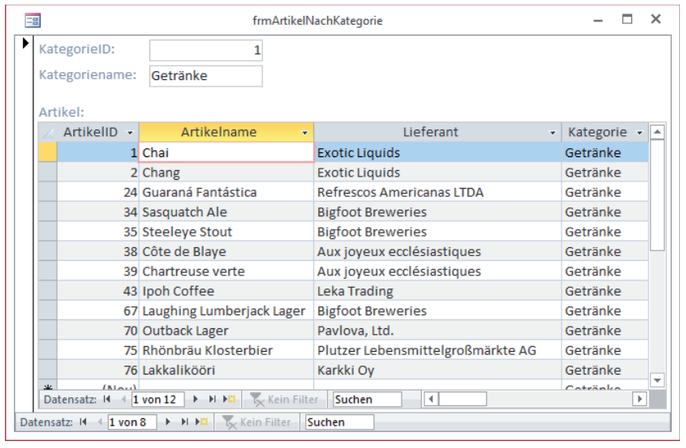

Bild 4: Spalten mit optimierten Breiten

Durch die modulweite Deklaration können Sie außerdem von jeder anderen Prozedur aus die Methode **OptimizeColumnWidths** aufrufen, um die Spaltenbreiten zu aktualisieren.

Anpassung nur nach Wunsch

Mit der Eigenschaft **OptimizeAutomatically** können Sie der Klasse auch mitteilen, dass die automatische Optimierung nicht erwünscht ist:

```
objColumnWidths.OptimizeAutomatically = False
```

In diesem Fall stoßen Sie die Optimierung der Spaltenbreiten ausschließlich über die Methode **OptimizeColumnWidths** an. **OptimizeAutomatically** hat standardmäßig den Wert **True**.

Warum OptimizeColumnWidths beim Start?

Die automatische Optimierung erfolgt immer dann, wenn eines der Ereignisse **Form_Current** (**Beim Anzeigen**) oder **Form_ApplyFilter** (**Bei Filter**) ausgelöst wird. Allerdings wurde **Form_Current** für den ersten Datensatz im Unterformular bereits ausgelöst, bevor wir im **Form_Load**-Ereignis die Klasse instanzieren und einstellen können.

Daher müssen Sie die Methode **OptimizeColumnWidths** nach dem Instanzieren einmal direkt aufrufen.

Verwendung der Klasse für mehrere Formulare

Wenn Sie mehrere Unterformulare mit der Funktion der Klasse ausstatten wollen, ist dies kein Problem.

Deklarieren Sie einfach für jedes Unterformular eine eigene Objektvariable:

```
Dim objCWUnterformular1 As clsColumnWidths
Dim objCWUnterformular1 As clsColumnWidths
```

Dann schreiben Sie die entsprechenden Anweisungen für beide Objekte in das **Form_Load**-Ereignis des Formulars mit den Unterformularen:

```
Private Sub Form_Load()
    Set objCWUnterformular1 = New clsColumnWidths
    With objCWUnterformular1
        Set .DataSheetForm = Me!sfmArtikel.Form
       .OptimizeColumnWidths
    End With
   Set objCWUnterformular1 = New clsColumnWidths
    With objCWUnterformular1
        Set .DataSheetForm = Me!sfmArtikel.Form
       .OptimizeColumnWidths
    End With
End Sub
```

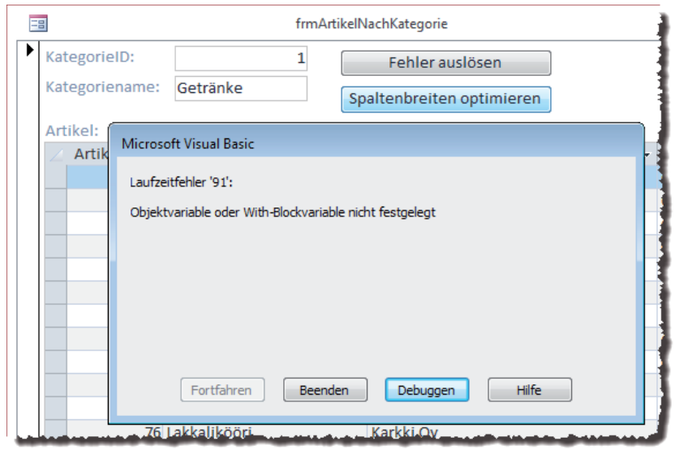

Bild 5: Fehler beim Versuch, ein durch einen Fehler geleertes Objekt zu nutzen

Im Falle eines Fehlers ...

Wenn bei der Arbeit mit dem Formularen ein nicht behandelter Fehler auftritt, werden Objektvariablen wie **objColumnWidths** gelöscht. Der Zugriff darauf führt dann zu einem Fehler. Um dies zu testen, haben wir dem Beispielformular eine Schaltfläche zum Auslösen eines Fehlers hinzugefügt:

```
Private Sub cmdFehler_Click()
    MsgBox 1 / 0
End Sub
```

Wenn Sie diesen Fehler ausgelöst haben, führt das nachfolgende Aufrufen der Methoden des Objekts **objColumnWidths** zu einem Fehler.

Auch dazu haben wir eine Schaltfläche hinzugefügt, die folgendes Ereignis auslöst:

```
Private Sub cmdSpaltenbreitenOptimieren_Click()
    objColumnWidths.OptimizeColumnWidths
End Sub
```

Bild 5 zeigt die Fehlermeldung nach Betätigung der beiden Schaltflächen an.

Dies lässt sich am einfachsten verhindern, indem Sie Fehler entsprechend behandeln.

Aufbau der Klasse clsColumnWidths

Die Klasse **clsColumnWidths** erstellen Sie, indem Sie über den Menüeintrag **EinfügenlKlassenmodul** des VBA-Editors ein neues Klassenmodul hinzufügen und dieses unter dem Namen **clsColumnWidth** speichern. Dann fügen Sie zunächst zwei Variablen zum Klassenmodul hinzu:

```
Dim WithEvents m_Form As Form
Dim m_OptimizeAutomatically As Boolean
```

Die erste soll einen Verweis auf das Formular aufnehmen, das in der Datenblattansicht erscheint und dessen Spalten automatisch optimiert werden sollen. Die zweite nimmt einen Boolean-Wert auf, mit dem die instanziierende Routine festlegt, ob die Spaltenbreiten automatisch beim Datensatzwechsel oder beim Filtern oder nur nach expliziter Aufforderung optimiert werden sollen.

Nach dem Instanzieren der Klasse weist die aufrufende Routine der Eigenschaft **DataSheetForm** einen Verweis auf das zu behandelnde Formular zu.

Dazu stellt die Klasse die folgende **Property Set**-Prozedur zur Verfügung:

```
Public Property Set DataSheetForm(frm As Form)
    Set m_Form = frm
    m_Form.OnCurrent = "[Event Procedure]"
    m_Form.OnApplyFilter = "[Event Procedure]"
End Property
```

Diese schreibt das mit **frm** übergebene Formular-Objekt in die Variable **m_Form** und legt außerdem mit den beiden Eigenschaften **OnCurrent** und **OnApplyFilter** fest, dass die Klasse auf diese beiden Ereignisse des mit **m_Form** referenzierten Formulars lauschen soll.

Diese beiden Ereignisse müssen wir natürlich auch implementieren. Dazu legen Sie die folgenden beiden Ereignisprozeduren an:

```
Private Sub m_Form_Current()
    If m_OptimizeAutomatically Then
        OptimizeColumnWidths
    End If
End Sub

Private Sub m_Form_ApplyFilter(Cancel As Integer, 
                               ApplyType As Integer)
    If m_OptimizeAutomatically Then
        OptimizeColumnWidths
    End If
End Sub
```

Beide Prozeduren prüfen den Inhalt der Variablen **m_OptimizeAutomatically**. Hat diese den Wert **True**, rufen die Ereignisprozeduren die Routine **OptimizeColumnWidths** auf. Diese sieht wie folgt aus:

```
Public Sub OptimizeColumnWidths()
    Dim ctl As control
    m_Form.RowHeight = 1
    For Each ctl In m_Form.Controls
        Select Case ctl.ControlType
            Case acTextBox, acComboBox, acCheckBox, 
                                                acListBox
                If Len(ctl.ControlSource) > 0 And 
                        ctl.ColumnHidden = False Then
                    ctl.ColumnWidth = -2
                End If
            Case Else
        End Select
    Next ctl
    m_Form.RowHeight = -1
End Sub
```

Die Prozedur stellt im Wesentlichen die Eigenschaft **ColumnWidth** eines der Steuerelemente Textfeld, Kontrollkästchen, Kombinationsfeld oder Listenfeld auf den Wert **-2** ein, was bewirkt, dass die Breite der jeweiligen Spalte für die aktuell sichtbaren Inhalte optimiert wird. Was aber, wenn der Benutzer nach unten scrollt und dort noch Einträge auftauchen, die nicht in die so eingestellte Spalte passen?

Dazu nutzt die Prozedur einen Trick: Sie stellt die Zeilenhöhe mit **RowHeight = 1** auf den minimalen Wert ein, sodass alle Zeilen direkt sichtbar sind und dementsprechend mit optimiert werden. Dann durchläuft die Prozedur alle Felder und stellt den Wert von **ColumnWidth** entsprechend ein. Schließlich wird die Zeilenhöhe mit der Einstellung auf den Wert **-1** wieder zurückgesetzt.

Automatisch oder nicht?

Fehlt noch die Eigenschaft **OptimizeAutomatically**: Diese stellen Sie über die gleichnamige **Property Let**-Prozedur ein:

```
Public Property Let OptimizeAutomatically(bol As Boolean)
    m_OptimizeAutomatically = bol
End Property
```

Damit die automatische Optimierung standardmäßig aktiviert ist, nutzen wir das **Initialize**-Ereignis der Klasse, um **m_OptimizeAutomatically** auf den Wert **True** einzustellen:

```
Private Sub Class_Initialize()
    m_OptimizeAutomatically = True
End Sub
```

Datenblattereignisse mit Klasse

Wenn Sie Ereignisse auslösen möchten, sobald der Benutzer an irgendeine Stelle eines Datensatzes in der Datenblattansicht klickt, müssen Sie theoretisch für jedes einzelne Steuerelement eine entsprechende Ereignisprozedur anlegen. Bei Formularen mit vielen Feldern kann das recht mühselig werden. Daher stellt dieser Beitrag eine Klasse vor, der Sie das Unterformular in der Datenblattansicht übergeben und die Ihnen Ereignisse für die gängigen Ereignisse wie Klick, Doppelklick et cetera bereitstellt. Diese müssen Sie dann nur noch einfach im Hauptformular implementieren.

Die hier vorgestellte Klasse soll folgende Funktionen bieten:

- Erfassung von Klick-, Doppelklick-, Maus auf- und Maus ab-Ereignissen
- Möglichkeit der Implementierung an einer Stelle im Hauptformular, wobei mehrere Informationen übergeben werden sollen – nämlich das Steuerelement, das angeklickt wurde, sowie optional der Primärschlüsselwert des Datensatzes, der angeklickt wurde. Alternativ kann auch der Wert jedes beliebigen anderen Feldes übergeben werden.
- Übergabe des markierten Datensatzes auch dann, wenn der Benutzer auf den Datensatzmarkierer klickt
- Optionale Markierung des kompletten Datensatzes beim Anklicken oder Doppelklicken

Im Beispiel aus Bild 1 werten wir beispielsweise den Primärschlüsselwert, die Aktion (also **Click**, **DblClick**, **MouseUp** oder **MouseDown**) und den Namen des angeklickten Steuerelements aus und zeigen diesen in den drei Textfeldern des Formulars an.

Um dies zu realisieren, sind, sobald die beiden Klassen **clsDatasheetForm** und **clsDatasheetControl** einmal in das Projekt importiert wurden, nur einige wenige Zeilen Code erforderlich.

Dabei deklarieren wir zunächst eine Variable, welche ein Objekt auf Basis der Klasse **clsDatasheetForm** aufnimmt, und zwar unter dem Namen **objDatasheetForm**. Außerdem erstellen wir eine Ereignisprozedur, die durch das Ereignis **Beim Laden** des Formulars ausgelöst wird. Diese sieht wie in Listing 1 aus (dort finden Sie auch die

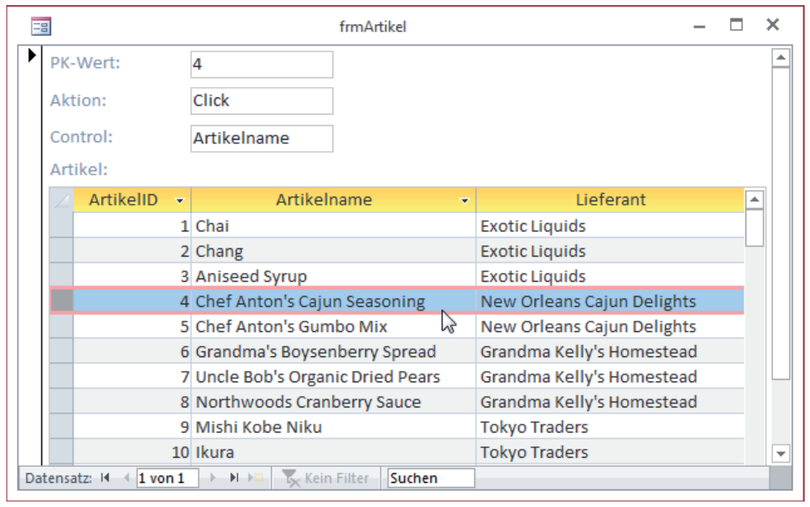

Bild 1: Unterformular mit Ereignisprozeduren im Hauptformular

Deklaration des Objekts **objDatasheetForm**). Die Prozedur erstellt eine neue Instanz der Klasse **clsDatasheetForm** und speichert den Verweis auf das neue Objekt in der Variablen **objDatasheetForm**.

```
Dim WithEvents objDatasheetForm As clsDataSheetForm

Private Sub Form_Load()
    Set objDatasheetForm = New clsDatasheetForm
    With objDatasheetForm
        Set .DatasheetForm = Me!sfmArtikel.Form
        .PrimaryKey = "ArtikelID"
        .SelectRowOnClick = True
    End With
End Sub
```

Listing 1: Ereignisprozedur, welche die Datenblattereignisse zugreifbar macht

Für diese stellt sie nun drei Eigenschaften ein:

- **DatasheetForm** nimmt einen Verweis auf das Unterformular entgegen, dessen Ereignisse abgegriffen und an das Klassenmodul des Hauptformulars weitergeleitet werden sollen.
- **PrimaryKey** ist eine optionale Eigenschaft, der Sie den Namen eines Feldes übergeben können, dessen Wert beim Anklicken eines Datensatzes zurückgeliefert werden soll.
- **SelectRowOnClick** ist eine **Boolean**-Eigenschaft, mit der Sie festlegen, ob beim Anklicken eines der Steuerelemente des Datensatzes gleich der komplette Datensatz im Unterformular markiert werden soll.

Damit ist die Arbeit noch nicht getan. Schließlich müssen wir noch die Ereignisse definieren, die im Klassenmodul des Hauptformulars ausgelöst werden sollen, wenn ein Benutzer auf ein Feld eines Datensatzes im Unterformular klickt.

Dazu wählen Sie im VBA-Codefenster im linken Kombinationsfeld den Eintrag **objDatasheetForm** aus und im rechten einen der vier Einträge **Click**, **DblClick**, **MouseDown** oder **MouseUp** – je nachdem, welches Ereignis Sie implementieren möchten. Der VBA-Editor legt dann automatisch den Prozedurrumpf für das jeweilige Ereignis an – genau so, als ob Sie ein Ereignis eines Formulars oder Steuerelements über das Eigenschaftsfenster der Entwurfsansicht anlegen (s. Bild 2).

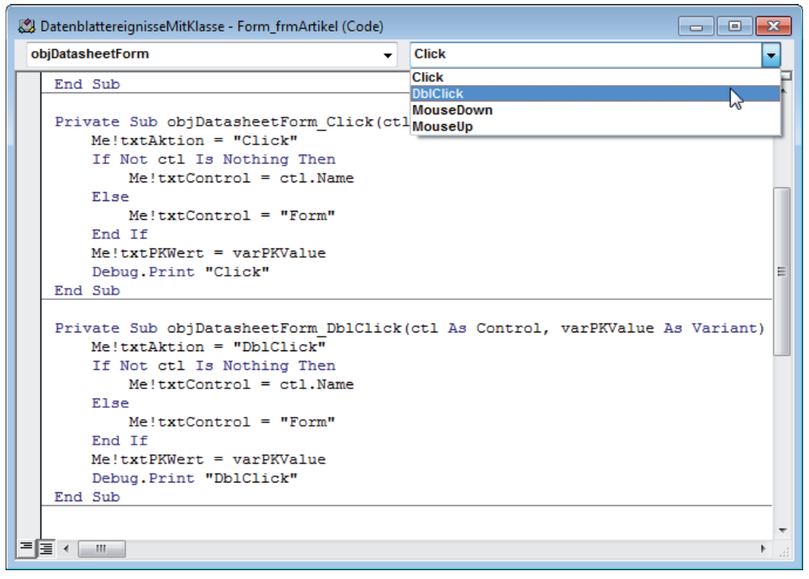

Bild 2: Anlegen einer Ereignisprozedur für das Objekt **objDatasheetForm**

In dieser Liste sehen Sie, dass **objDatasheetForm** vier Ereignisse anbietet – **Click**, **DblClick**, **MouseDown** und **MouseUp**.

Wir wollen alle vier implementieren, um Spielmaterial für unsere Beispielan-

Kundendatensätze zusammenführen

Wer eine Kundendatenbank pflegt, wird früher oder später Dubletten in seiner Datenbank vorfinden. Sei es, weil Kunden sich mit neuer E-Mail und neuer Adresse erneut im Onlineshop anmelden und von dort importiert werden oder weil man bei der Suche nach einem vorhandenen Konto für einen Kunden wegen eines Tippfehlers keinen Treffer landet – langfristig lassen sich doppelte Kundendatensätze nicht verhindern. Aber das ist kein Problem: In Datenbanken lässt sich zum Glück alles nachträglich ändern. Wie dies bei Kundendaten und den damit verknüpften Daten wie etwa Bestellungen funktioniert, erklärt dieser Beitrag.

Das Problem nach der Erstellen eines doppelten Kundendatensatzes lautet: Wie mache ich aus den beiden Kundendatensätzen einen, und vor allem: Wie sorge ich dafür, dass die Daten, die mit dem zu löschenden Kundendatensatz verknüpft sind, mit dem verbleibenden Kundendatensatz zusammengeführt werden?

Es wäre ja leicht, wenn man einfach den »alten« Kundendatensatz löschen könnte und dann Ruhe hätte. Aber in der Regel erstellt man ja einen Kundendatensatz erst, wenn für diesen auch eine Bestellung vorliegt. Wenn wir ein Datenmodell wie in Bild 1 zugrunde legen, bei dem ein Bestelldatensatz auf der einen Seite mit dem bestellenden Kunden, auf der anderen Seite mit der Tabelle Bestelldetails verknüpft wird, ist klar: Wir können nicht einfach den alten Datensatz löschen, sondern müssen auch alle Daten der Tabelle **tblBestelldetails** auf den Kundendatensatz übertragen, der beibehalten werden soll. Das bedeutet eigentlich nur, dass der Wert des Fremdschlüsselfeldes **KundeID** mit dem entsprechenden Primärschlüsselwert

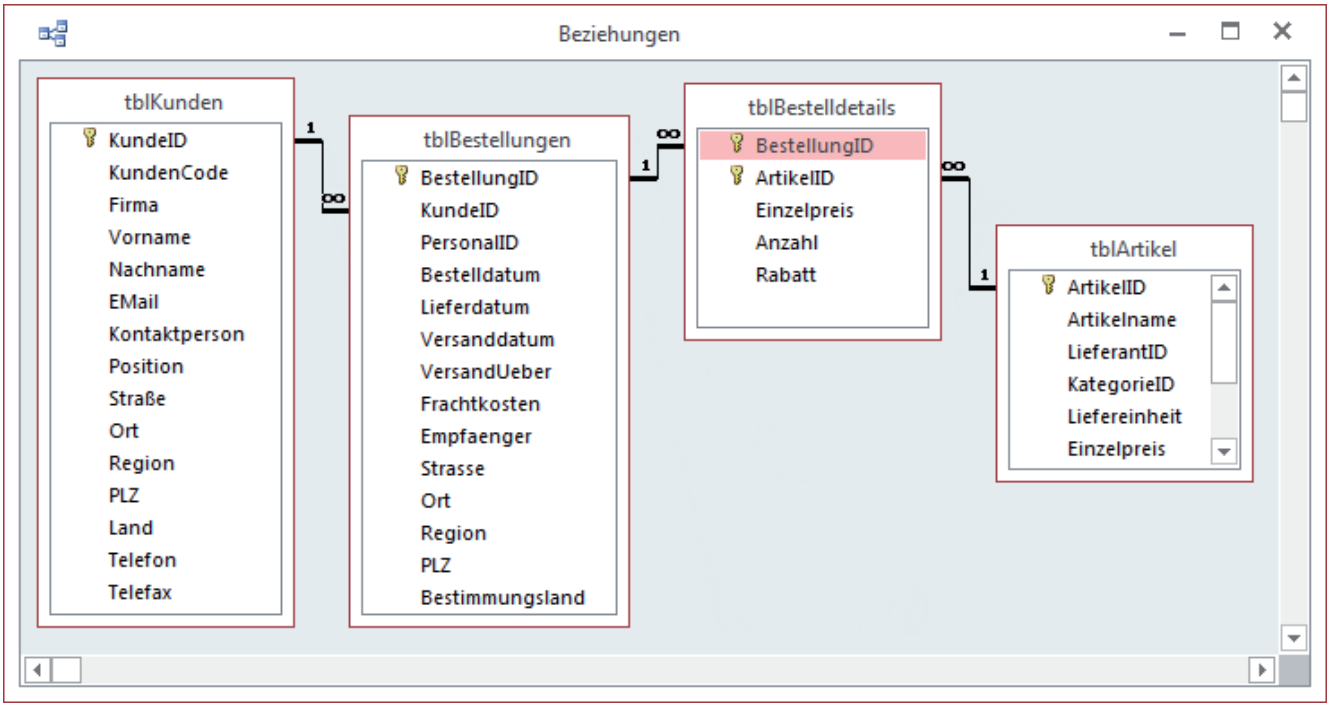

Bild 1: Tabellen der Beispieldatenbank

des neuen Kundendatensatzes gefüllt werden muss. Das allein lässt sich mit einer einfachen **UPDATE**-Abfrage erledigen. Danach noch den nicht mehr benötigten Kundendatensatz löschen oder als inaktiv markieren, schon ist man fertig.

Um solche Änderungen nachher wieder rückgängig machen oder zumindest nachvollziehen zu können, sollten Sie die geänderten oder gelöschten Daten archivieren. Wenn Sie Access ab Version 2010 verwenden, können Sie dies etwa gemäß dem Beitrag **Geänderte Daten archivieren** (**www.access-im-unternehmen.de/925**) erledigen. Beim SQL Server würden Sie entsprechende Trigger nutzen oder in die gespeicherten Prozeduren, welche die Daten ändern, passende Anweisungen zum Sichern der Datensätze einbringen.

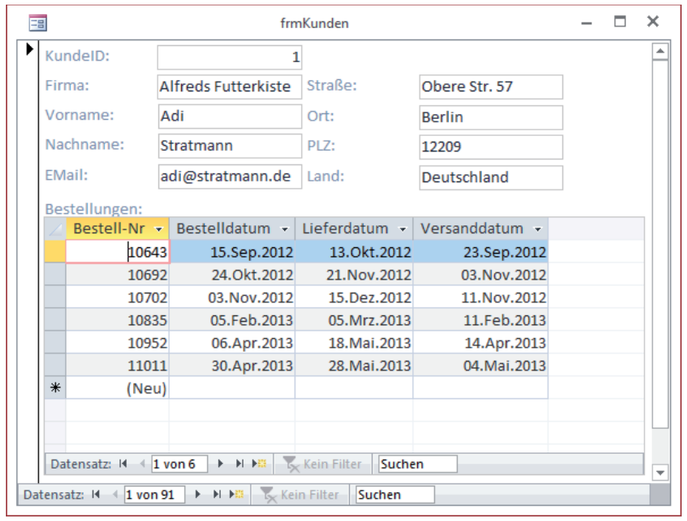

Bild 2: Formular zur Anzeige der Bestellungen eines Kunden

Benutzeroberfläche

Nun hätten wir die Theorie bereits erledigt. Wie aber sieht die Praxis aus? Otto Normalverbraucher kann leider meist nicht mal eben eine **UPDATE**-Anweisung ins Direktfenster schmeißen, sodass wir für die gewünschte Funktion eine entsprechende Benutzeroberfläche bereitstellen müssen. Nehmen wir doch ein herkömmliches Formular mit Unterformular als Basis, wie es zur Darstellung von Daten aus zwei Tabellen einer 1:n-Beziehung aussieht. Im Beispiel aus Bild 2 finden Sie die Tabelle **tblKunden** im Hauptformular und die Bestellungen des aktuell angezeigten Kunden aus **tblBestellungen** im Unterformular vor.

Im Gegensatz zum Standard-Bestellformular, das eine Bestellung samt Bestellpositionen anzeigt, fallen hier die Bestelldetails unter den Tisch. Wir können diese mit einem zweiten Unterformular nachreichen, das die Bestellpositionen zu der jeweils im ersten Unterformular ausgewählten Bestellung anzeigt. Dies ist allerdings unnötig, denn wir wollen ja nur die Bestellungen des nicht mehr benötigten Duplikats eines Kunden auf einen anderen Kunden übertragen. Die Bestellpositionen sind ja ohnehin mit der Tabelle **tblBestellungen** verknüpft und werden quasi »mit übertragen«.

Gibt es nun eine sinnvolle Variante, um diesen Datensatz mit einer Dublette zusammenzuführen? Nun, eigentlich nicht – denn diese müssten wir ja auf jeden Fall erst einmal ausfindig machen. Gelegentlich wird sich vielleicht ein Kunde melden, der vielleicht Kundennewsletter an die beiden unterschiedlichen E-Mail-Adressen seiner beiden Accounts erhält und somit das Vorhandensein eines Duplikats aufdecken. In der Regel sollten Sie sich allerdings, je nach der Anzahl der zu verwaltenden Kunden, von Zeit zu Zeit selbst auf die Suche nach Duplikaten machen.

Aufmerksame und langjährige **Access im Unternehmen**-Leser werden jetzt aufhorchen: War da nicht mal was? Ja, genau: Im Beitrag **Duplikatsuche in Access** (**www.access-im-unternehmen.de/744**) haben wir ein Formular vorgestellt, mit dem Sie flexibel Duplikate in Ihren Datenbeständen finden können. Die Lösung aus diesem Beitrag werden wir für unsere Zwecke nutzen und entsprechend aufbohren.

Integration der Lösung in eigene Datenbanken

Wenn Sie die Lösung in einer eigenen Datenbank nutzen möchten, müssen Sie zunächst die folgenden Objekte aus der Beispieldatenbank in Ihre Datenbank importieren:

- **frmDuplikatmanager**
- **sfmDuplikatfelder**
- **sfmFlex**
- **frmDuplikatdetails**
- **tblDuplikatfelder**
- **clsDatasheetForm**
- **clsDatasheetControl**
- **clsColumnWidths**
- **mdlTools**

Damit erhalten Sie schon einmal das Formular aus Bild 3 mit allen benötigten Unterformularen, Modulen und Klassen. Außerdem fügen Sie so eine Tabelle hinzu, welche die Konstellation für das Auffinden der Duplikate speichert.

Außerdem müssen Sie das Formular **frmDuplikatdetails** noch an die Gegebenheiten der Zieldatenbank anpassen. Das Formular sieht in der Beispieldatenbank im Entwurf wie in Bild 4 aus. Dieses Formular soll einen der zusammenzufassenden Datensätze mit den notwendigsten Informationen für den Abgleich darstellen. Das Hauptaugenmerk liegt dabei darauf, dass die verknüpften Daten angezeigt werden, die beim Zusammenführen zweier (oder auch mehrerer) Duplikate berücksichtigt werden sollen. Auf diese Weise kann der Benutzer sich nochmals

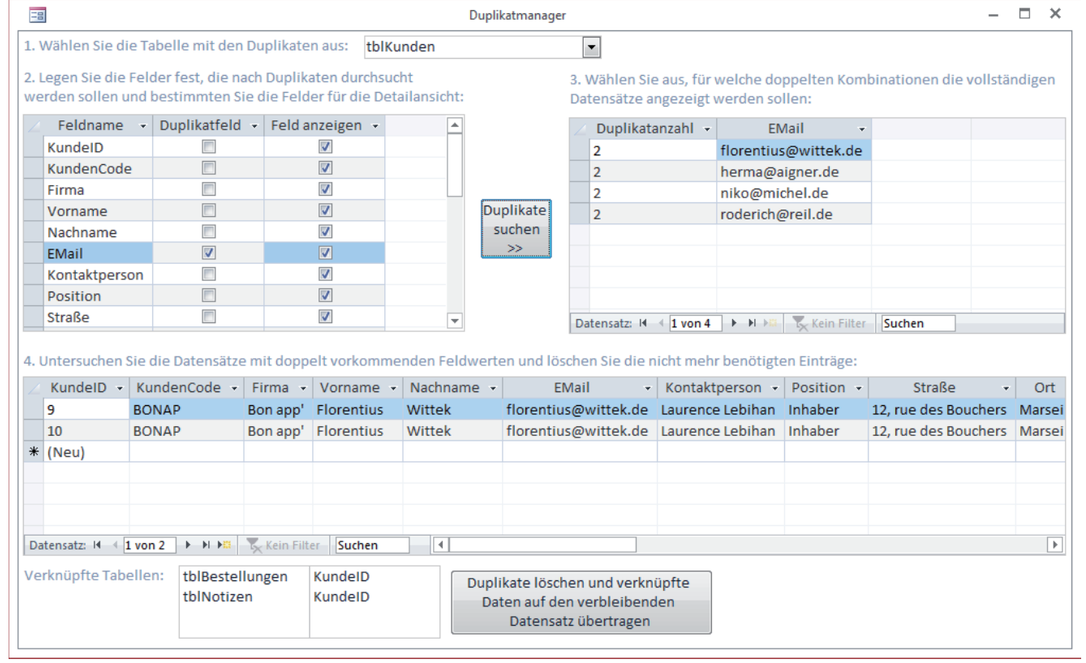

Bild 3: Das Formular zum Ermitteln und Abgleichen der Duplikate einer Tabelle

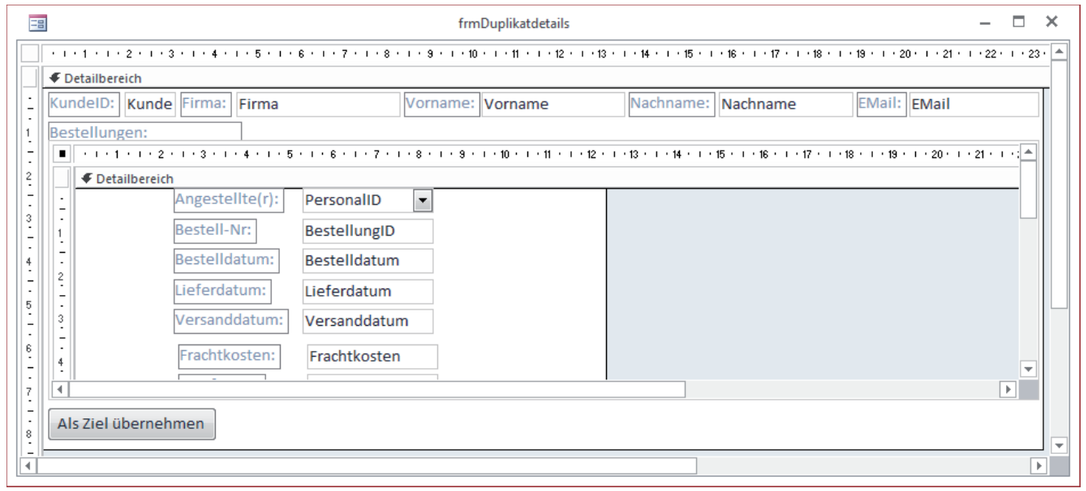

Bild 4: Formular zur Anzeige der Details zu einem der Duplikate

versichern, dass dort auch die richtigen Daten zusammengeführt werden.

Das Formular soll mehrfach geöffnet werden. Mit einem Klick auf die Schaltfläche **Als Ziel übernehmen** übernimmt der Benutzer dann den entsprechenden Datensatz als Zieldatensatz für das Zusammenführen der Daten – die bis dahin geöffneten Detailformulare werden dann geschlossen.

Ablauf der Zusammenführung zweier Datensätze

Das Formular **frmDuplikatmanager** bietet in einem Kombinationsfeld alle Tabellen der aktuellen Datenbank zur Auswahl an. Wenn der Benutzer eine Tabelle ausgewählt hat, erscheinen alle Felder im linken, oberen Unterformular. Dort finden Sie neben der Spalte mit den Feldnamen noch zwei weitere Spalten – eine mit den für die Duplikatsuche zu verwendenden Felder und eine mit den Feldern, die im Ergebnis angezeigt werden sollen.

Nachdem der Benutzer diese festgelegt hat (im Screenshot sollen nur die E-Mail-Adressen abgeglichen werden und alle Felder in der Ergebnisliste erscheinen), klickt er auf die Schaltfläche **Duplikate suchen**. Findet die Lösung mindestens ein Duplikat, zeigt es die Anzahl der gefundenen Exemplare sowie den Wert des Vergleichsfeldes im Unterformular rechts oben an.

Hier kann der Benutzer nun wiederum auf einen Eintrag klicken und so alle Duplikate zu diesem Eintrag im unteren Unterformular einblenden. Dies hat den Vorteil, dass Sie direkt prüfen können, ob sich die Inhalte der übrigen Felder unterscheiden. Wenn Sie sich entschieden haben, welcher der Datensätze beibehalten werden soll, können Sie in diesem gegebenenfalls Korrekturen vornehmen oder Informationen aus den zu löschenden Datensätzen übernehmen.

Nun kommt auch unser Formular für die Anzeige der Duplikatdetails ins Spiel. Der Hauptgrund für die Erstellung der vorliegenden Lösung ist ja, nicht nur einen von mehreren Datensätzen (im Beispiel Kunden) zu übernehmen und die übrigen zu löschen, sondern auch noch die Daten, die mit den zu löschenden Datensätzen verknüpft sind, auf den verbleibenden Datensatz zu übertragen.

Deshalb können Sie mit dem Formular **frmDuplikatdetails** in diesem Beispiel die Kundendaten plus die Bestellungen der Kunden anzeigen, und zwar per Doppelklick auf einen der Einträge im unteren Unterformular.

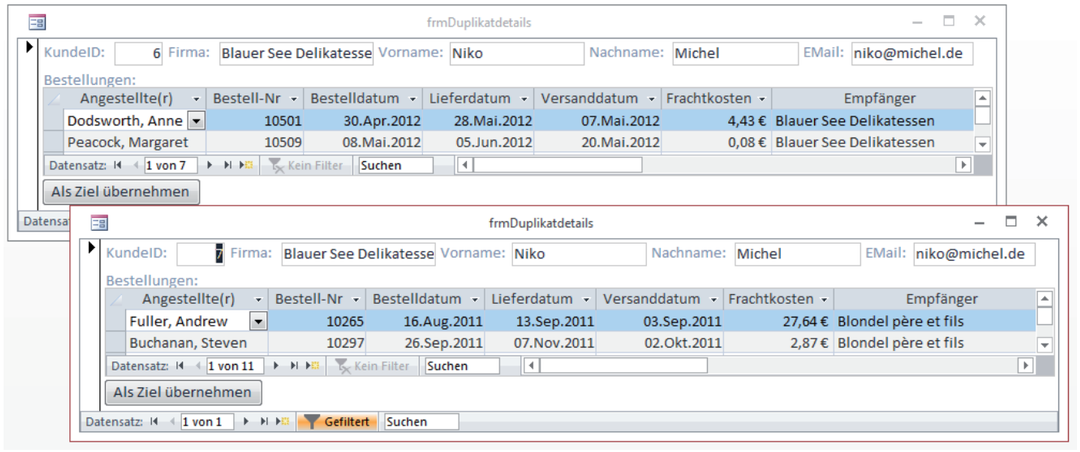

Bild 5: Vergleich zweier Datensätze für den gleichen Kunden

Im Gegensatz zu üblichen Formularen, von denen Sie nur jeweils eine einzige Instanz öffnen, können Sie hier für jeden der in der Liste enthaltenen Kunden ein Detailformular öffnen. In Bild 5 sehen Sie beispielsweise zwei Formulare mit verschiedenen Datensätzen zum gleichen Kunden. Klicken Sie hier auf die Schaltfläche **Als Ziel übernehmen**, werden alle Formulare geschlossen. Außerdem markiert das Formular **frmDuplikatmanager** den zu übernehmenden Datensatz im unteren Unterformular.

Nun folgt der interessante Teil: Das untere Listenfeld des Formulars **frmDuplikatmanager** zeigt alle mit der zusammenzuführenden Tabelle per 1:n-Beziehung verknüpften Tabellen an, in diesem Fall **tblBestellungen** und **tblNotizen** (s. Bild 6).

Sie können nun einen oder mehrere Einträge auswählen, damit die enthaltenen Datensätze auf den zu übernehmenden Kundendatensatz übertragen werden. Um die Duplikate letztlich zusammenzuführen, klicken Sie auf die Schaltfläche rechts neben dem Listenfeld.

Danach sollte der übernommene Datensatz rasch vom unteren Unterformular verschwinden. Über einen Doppelklick auf den verbleibenden Datensatz können Sie sich im Detailformular vergewissern, dass die verknüpften Daten wie gewünscht übernommen wurden.

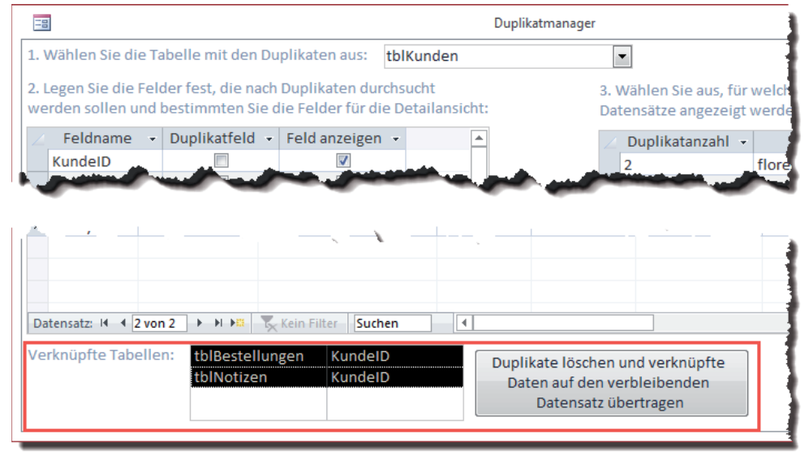

Bild 6: Auswahl der zu übernehmenden Daten aus den verknüpften Tabellen

Sicher ist sicher

Zur Sicherheit sollten Sie solche Aktionen nicht durchführen, ohne zuvor eine Kopie der Datenbank angelegt zu haben. Noch besser wäre es, wenn Sie Access 2010 oder höher verwenden und die geänderten oder gelöschten Datensätze in entsprechenden Archivtabellen sichern. Eine geeignete Lösung finden Sie in den Beiträgen **Geänderte Daten archivieren** auf (**www.access-im-unternehmen.de/925**) und **Änderungshistorie implantieren** (**www.access-im-unternehmen.de/995**).

Aufbau der benötigten Formulare

Die folgenden Abschnitte erläutern die Zusammenhänge zwischen dem Haupt- und den Unterformularen und wie diese gefüllt werden und auf Benutzeraktionen reagieren.

Laden des Formulars

Beim Laden des Formulars **frmDuplikatmanager** müssen einige Aktionen ausgeführt werden, um das Formular vorzubereiten. Dies geschieht in der Ereignisprozedur, die durch das Ereignis **Beim Laden** ausgelöst wird (s. Listing 1).

Diese leert zunächst die Tabelle **tblDuplikatfelder**, welche später mit je einem Datensatz für jedes Feld der zu untersuchenden Tabelle gefüllt wird (s. Bild 7). Danach

```
Private Sub Form_Load()
    Dim db As DAO.Database
    Set db = CurrentDb
    db.Execute "DELETE FROM tblDuplikatfelder", dbFailOnError
    Me!sfmDuplikatfelder.Form.Requery
    Set frm_sfmDuplikate = Me!sfmDuplikate.Form
    frm_sfmDuplikate.OnCurrent = "[Event Procedure]"
    Set objCW_Duplikatfelder = New clsColumnWidths
    Set objCW_Duplikatfelder.DataSheetForm = Me!sfmDuplikatfelder.Form
    Set objCW_Duplikate = New clsColumnWidths
    Set objCW_Duplikate.DataSheetForm = Me!sfmDuplikate.Form
    Set objCW_DuplikateDetail = New clsColumnWidths
    Set objCW_DuplikateDetail.DataSheetForm = Me!sfmDuplikateDetails.Form
    DatasheetFormInstanzieren
    Set colForms = New Collection
End Sub
```

Listing 1: Vorbereitung des Formulars und seiner Elemente

tblDuplikatfelder

FeldID	Feldname	Duplikatfeld	FeldAnzeigen
568	KundeID	☐	☑
569	KundenCode	☐	☑
570	Firma	☐	☑
571	Vorname	☐	☑
572	Nachname	☐	☑
573	EMail	☑	☑
574	Kontaktperson	☐	☑
575	Position	☐	☑
576	Straße	☐	☑
577	Ort	☐	☑
578	Region	☐	☑
579	PLZ	☐	☑
580	Land	☐	☑
581	Telefon	☐	☑
582	Telefax	☐	☑
(Neu)		☐	☐

Datensatz: 1 von 15 | Kein Filter | Suchen

Bild 7: Tabelle zum Speichern der zu verwendenden Felder

aktualisiert sie den Inhalt des Unterformulars **sfmDuplikatfelder**, damit dieses den aktualisierten Inhalt der nun leeren Tabelle **tblDuplikatfelder** anzeigt. Anschließend füllt sie die folgende Variable mit einem Verweis auf das Unterformular **sfmDuplikate**:

```
Dim WithEvents frm_sfmDuplikate As Form
```

Dies geschieht mit dem Schlüsselwort **WithEvents**, weil wir im Klassenmodul des Hauptformulars Ereignisse für dieses Unterformular implementieren wollen – zum Beispiel für die Auswahl eines der enthaltenen Datensätze, um dann alle passenden Duplikate im Unterformular **sfmDuplikatdetails** anzuzeigen. Dabei handelt es sich um das Ereignis **Beim Anzeigen**, wozu wir noch mitteilen müssen, dass das aktuelle Klassenmodul auf solche Ereignisse lauschen soll (**OnCurrent = [Event Procedure]**).

Die drei Unterformulare **sfmDuplikatfelder**, **sfmDuplikate** und **sfmDuplikateDetails** sollen mit optimierter Spaltenbreite angezeigt werden. Dazu verwenden wir die Klasse **clsColumnWidths**, die wir ausführlich im Beitrag **Spaltenbreiten optimieren mit Klasse vorstellen** (siehe **www.access-im-unternehmen.de/998**).

Wir wollen jedes der drei Unterformulare mit der Funktion zur optimalen Anpassung der Spaltenbreiten ausstatten, also legen wir drei Objektvariablen für die entsprechenden Objekte fest:

```
Dim objCW_Duplikatfelder As clsColumnWidths
Dim objCW_Duplikate As clsColumnWidths
Dim objCW_DuplikateDetail As clsColumnWidths
```

Diese instanziert die Prozedur **Form_Load** dann und stellt mit der Eigenschaft **DataSheetForm** jeweils das betroffene Unterformular ein.

Danach ruft sie noch die Prozedur **DatasheetFormInstanzieren** auf, die dem unteren Unterformular einige Funktionen hinzufügt, und instanziert ein **Collection**-Objekt, das wir im Kopf des Klassenmoduls deklarieren:

```
Dim colForms As Collection
```

Den Zweck dieser Collection erläutern wir weiter unten.

Instanzieren der Datenblattfunktionen von sfmDuplikateDetails

Das untere Unterformular namens **sfmDuplikateDetails** soll bei einem Doppelklick auf einen der Datensätze ein Ereignis auslösen, um einen Detaildatensatz in einem eigenen Formular anzuzeigen. Damit der Benutzer dabei nur auf eine beliebige Stelle im Datensatz zu klicken braucht, müssen wir theoretisch für jedes Steuerelement eine **Beim Klicken**-Ereignisprozedur anlegen.

Dies können wir uns jedoch sparen, wenn wir die beiden im Beitrag **Datenblattereignisse mit Klasse** vorgestellten Klassen nutzen. Damit brauchen wir als Erstes nur ein Element mit folgendem Typ zu deklarieren:

```
Dim WithEvents objDS As clsDataSheetForm
```

```
Private Sub DatasheetFormInstanzieren()
    Set objDS = New clsDatasheetForm
    With objDS
        Set .DatasheetForm = Me!sfmDuplikateDetails.Form
        If Len(Me!cboTabellen) > 0 Then
            .PrimaryKey = GetSinglePrimaryKey(Nz(Me!cboTabellen))
            .ZeileBeiKlickMarkieren = False
        End If
    End With
End Sub
```

Listing 2: Funktionen für das Unterformular **sfmDuplikateDetails** einrichten

Danach benötigen wir noch die Anweisungen aus der Routine aus Listing 2. Diese instanziert das Objekt auf Basis von **clsDatasheetForm** und weist diesem das Unterformular **sfmDuplikateDetails** als Formular zu. Außerdem stellt sie die Eigenschaft **PrimaryKey** auf den Primärschlüsselnamen der zu untersuchenden Tabelle ein, den wir mit der Funktion **GetSinglePrimaryKey** einlesen (siehe **Primärschlüsselfelder ermitteln**, **www.access-im-unternehmen.de/1004**. Mit **ZeileBeiKlickMarkieren = False** legen wir fest, dass beim Anklicken nicht die komplette Zeile des Datensatzes markiert werden soll.

Auswahl der Tabelle

Das Hauptformular **frmDuplikatmanager** verwendet das Kombinationsfeld **cboTabellen**, um dem Benutzer die Auswahl der zu untersuchenden Tabelle zu ermöglichen. Diese füllen wir mit der folgenden Abfrage:

```
SELECT MSysObjects.Name FROM MSysObjects WHERE Name Not
Like 'MSys*' And Name Not Like 'USys*' And Type=1
```

Nach dem Auswählen eines der Einträge löst das Ereignis **Nach Aktualisierung** des Kombinationsfeldes die Prozedur aus Listing 3 aus. Die Prozedur leert die Tabelle **tblDuplikatfelder**, welche die Konfiguration für die Duplikatsuche speichert, und füllt diese dann neu, indem sie in einer **For Each**-Schleife alle Felder der zu untersuchenden Tabelle durchläuft und für jedes einen neuen Datensatz zur Tabelle **tblDuplikatfelder** hinzufügt. Dann aktualisiert sie das Unterformular **sfmDuplikatfelder** und optimiert

Titel des aktiven Fensters ermitteln

Wenn Sie den Titel des aktuellen Fensters, also beispielsweise des Access-Fensters, ermitteln möchten (s. Bild 1), benötigen Sie zwei API-Funktionen und eine VBA-Funktion, welche die beiden zum Ermitteln des Fenstertitels nutzt.

Bild 1: Titel des Access-Fensters

Diese Elemente haben wir in der Beispieldatenbank zu diesem Beitrag im Modul **mdlAPI** untergebracht (s. Listing 1). Die Funktion **GetActiveWindowTitle** ermittelt zunächst das Handle des mit der API-Funktion **GetForegroundWindow** ermittelten Fensters. Um nun mit der Funktion **GetWindowText** den Fenstertitel zu ermitteln, müssen wir zunächst eine String-Variable mit 255 Leerzeichen füllen.

Die Funktion ersetzt dann die Leerzeichen durch den tatsächlichen Fenstertitel, behält aber die nicht ersetzten Leerzeichen bei. Das ist aber kein Problem, denn das Ergebnis der Funktion **GetWindowText** ist die Anzahl der zum zweiten Parameter der Funktion hinzugefügten Zeichen.

Der erste Parameter erwartet übrigens das Fenster-Handle des zu untersuchenden Fensters, der dritte die voreingestellte Größe der übergebenen String-Variablen.

Das Funktionsergebnis speichert die Prozedur in der Variablen **nResult**. Damit kann die letzte Anweisung dem Funktionswert von **GetActiveWindowTitle** per **Left**-Funktion die relevanten Zeichen der Variablen **sTitle** übergeben. Ein Aufruf der Funktion etwa im Direktfenster des VBA-Editors liefert dann den Titel des aktuellen Fensters – in diesem Fall natürlich den des VBA-Fensters:

```
? GetActiveWindowTitle
Microsoft Visual Basic for Applications - TippsUndTricks [Aktiv] - [mdlAPI (Code)]
```

Um den Titel des Access-Fensters zu ermitteln, müssen Sie die Funktion etwa beim Anklicken einer Schaltfläche in einem Formular im Access-Hauptfenster aufrufen.

```
Private Declare Function GetForegroundWindow Lib "user32" () As Long
Private Declare Function GetWindowText Lib "user32" Alias "GetWindowTextA" (ByVal hwnd As Long, _
    ByVal lpString As String, ByVal cch As Long) As Long
Public Function GetActiveWindowTitle() As String
    Dim nHWnd As Long
    Dim sTitle As String
    Dim nResult As Long
    nHWnd = GetForegroundWindow()
    sTitle = Space$(255)
    nResult = GetWindowText(nHWnd, sTitle, Len(sTitle))
    GetActiveWindowTitle = Left$(sTitle, nResult)
End Function
```

Listing 1: Ermitteln des Titels des aktuellen Fensters

Access-Titel per VBA ändern

Den Titel des Access-Fensters können Sie manuell oder per VBA ändern. Dieser Beitrag stellt beide Methoden vor.

Access-Titel ändern

Sicher schon an vielen Stellen beschrieben, aber hier der Vollständigkeit halber nochmals aufgeführt: Wenn Sie den Titel des Access-Fensters für die aktuell geöffnete Anwendung ändern möchtern, können Sie die entsprechende Eigenschaft in den Access-Optionen einstellen. Im Optionen-Dialog der moderneren Access-Versionen ab 2007 finden Sie die Eigenschaft zum Einstellen des Anwendungstitels im Bereich **Aktuelle Datenbank** unter **Anwendungsoptionen** (s. Bild 1).

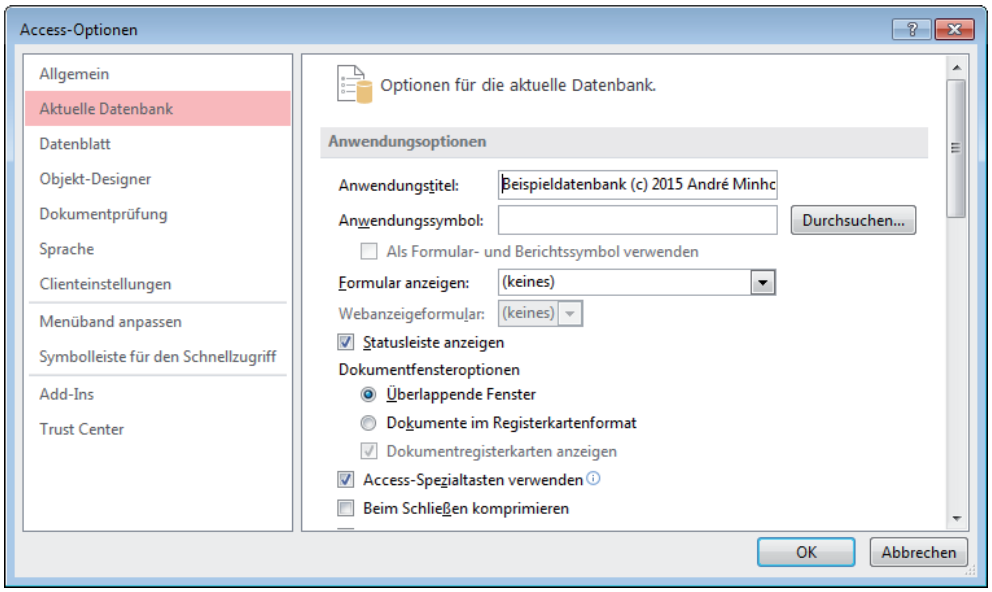

Bild 1: Titel des Access-Fensters einstellen

Access-Titel per VBA ändern

Sie können den Titel auch per VBA ändern. Allerdings wird diese Änderung erst beim nächsten Öffnen der Datenbank wirksam. Dazu verwenden Sie die Eigenschaft **AppTitle** der **Properties**-Auflistung des **CurrentDb**-Objekts von Access:

```
CurrentDb.Properties("AppTitle") = "Tipps und Tricks"
```

Anschließend müssen Sie noch die Methode **RefreshTitleBar** des **Application**-Objekts aufrufen:

```
Application.RefreshTitleBar
```

Auf diese Weise könnten Sie auch dynamisch Informationen in der Titelleiste anzeigen. Allerdings muss der Titel zuvor bereits einmal über den Optionen-Dialog geändert worden sein. Sonst existiert die Eigenschaft **AppTitle** noch nicht und der Zugriff darauf löst den Fehler **3270** aus. Mit der folgenden Prozedur ändern Sie den Anwendungstitel aber zuverlässig – der auftauchende Fehler wird gegebenenfalls behandelt und die fehlende Property nachgetragen:

```
Public Sub Fenstertitel(strTitel As String)
    Dim db As DAO.Database
    Dim prp As DAO.Property
    Set db = CurrentDb
    On Error Resume Next
    db.Properties("AppTitle") = strTitel
    If Err.Number = 3270 Then
        Set prp = db.CreateProperty("AppTitle", dbText, _
            strTitel)
        db.Properties.Append prp
    End If
    On Error GoTo 0
    Application.RefreshTitleBar
End Sub
```

Access-Fenster in den Vordergrund holen

Wenn Sie von Access aus einmal ein anderes Fenster aufrufen, möchten Sie vielleicht sicherstellen, dass der Blick nach dem Abschließen der dort durchgeführten Arbeiten wieder bei der Access-Anwendung landet. Wenn Sie etwa von Access aus den Dialog zum Auswählen eines Outlook-Ordners auswählen und das Outlook-Fenster ist zu diesem Zeitpunkt bereits geöffnet, dann gerät nach dem Schließen des Dialogs das Outlook-Fenster in den Vordergrund.

Um das Access-Fenster wieder nach vorn zu holen, sind nun zwei Schritte nötig: Als Erstes müssen Sie wie im Beitrag **Titel des aktiven Fensters ermitteln** (**www.access-im-unternehmen.de/1001**) beschrieben den Titel des Access-Fensters ermitteln und diesen etwa in einer Variablen speichern. Zweitens müssen Sie das so bezeichnete Fenster wieder in den Vordergrund holen.

Dabei hilft Ihnen die eingebaute Funktion **AppActivate**. Sie erwartet als Parameter lediglich den Titel des zu aktivierenden Fensters. Die folgende kleine Beispielprozedur ermittelt den Titel des aktuellen Access-Fensters und speichert diesen in der Variablen **strFenster**. Dann ruft sie den Outlook-Dialog mit der Hilfsfunktion **VerzeichnisWaehlen** auf. Danach sollte, wenn das Outlook-Fenster zuvor bereits geöffnet war, Outlook im Vordergrund bleiben. Schließlich holt **AppActivate** mit dem Fenstertitel aus **strFenster** das Access-Fenster wieder in den Vordergrund:

```
Private Sub cmdOutlook_Click()
    Dim strFenster As String
    strFenster = GetActiveWindowTitle
    Me!txtVerzeichnis = VerzeichnisWaehlen
    AppActivate strFenster
End Sub
```

Wenn Sie dies ausprobieren möchten, finden Sie der Vollständigkeit halber hier noch die Funktion **VerzeichnisWaehlen**:

```
Public Function VerzeichnisWaehlen() As String
    Dim objOutlook As Outlook.Application
    Dim objMAPI As Outlook.NameSpace
    Dim objFolder As Outlook.Folder
    Set objOutlook = New Outlook.Application
    Set objMAPI = objOutlook.GetNamespace("MAPI")
    Set objFolder = objMAPI.PickFolder
    If Not objFolder Is Nothing Then
        VerzeichnisWaehlen = objFolder.FolderPath
    End If
End Function
```

Diese Funktion öffnet den Dialog **Ordner auswählen**. Wenn das Outlook-Fenster bereits geöffnet ist, wird das Outlook-Fenster aktiviert und der Dialog vor diesem angezeigt (s. Bild 1). Schließt der Benutzer den Dialog, behält das Outlook-Fenster dennoch den Fokus. Erst die **AppActivate**-Anweisung holt den Fokus zurück zu Access.

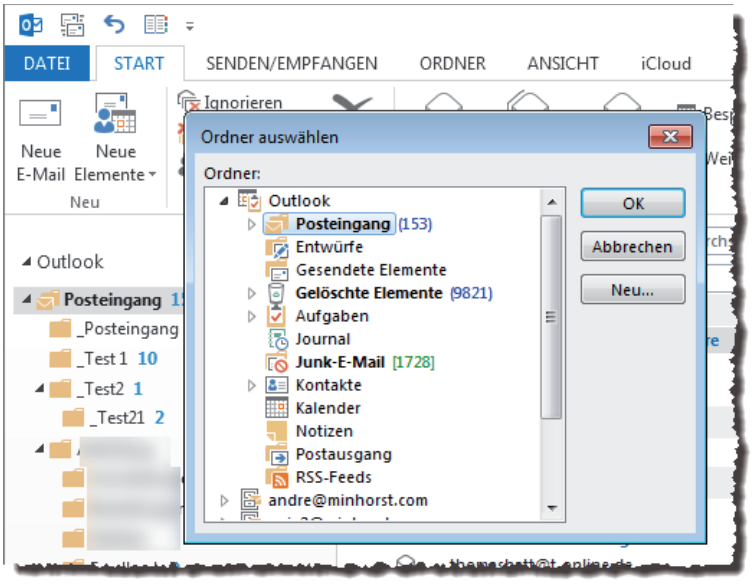

Bild 1: Ordner auswählen-Dialog vor dem Outlook-Fenster

Verknüpfte Tabellen ermitteln

Wenn Sie herausfinden wollen, welche Tabellen über ein Fremdschlüsselfeld mit einer gegebenen Tabelle verknüpft sind, benötigen Sie ein paar Zeilen DAO-Code. Die hier vorgestellte Funktion erwartet den Namen der zu untersuchenden Tabelle und enthält zwei weitere Parameter, die zur Rückgabe der Ergebnisse vorgesehen sind.

Da wir nicht nur die verknüpften Tabellen, sondern auch noch die Namen der Fremdschlüsselfelder der Tabellen ermitteln wollen, können wir hier nicht mit einem einfachen Rückgabewert arbeiten, sondern benötigen eben diese zwei Rückgabeparameter.

Beide Parameter sind als Arrays ausgelegt, da ja auch einmal mehrere Tabellen über ein Fremdschlüsselfeld mit einer Tabelle verknüpft sein können.

In diesem Fall durchläuft die Funktion alle Elemente der **Relations**-Auflistung der Datenbank. Dabei vergleicht sie den Namen der Tabelle der einen Seite der Relation mit dem Namen der zu untersuchenden Tabelle. Stimmen beide überein, folgen weitere Schritte.

Die Funktion erweitert die Arrays dann um ein Element und trägt den Namen der verknüpften Tabelle (aus der Eigenschaft **ForeignTable** des **Relation**-Objekts) und den Namen des Fremdschlüsselfeldes (aus der Eigenschaft **ForeignName**) als neue Elemente in die beiden Arrays **strVerknuepfteTabellen** und **strFremdschluesselfelder** ein:

```
Public Function VerknuepfteTabellen(strTabelle As String, _
        strVerknuepfteTabellen() As String, _
        strFremdschluesselfelder() As String)
    Dim db As DAO.Database
    Dim rel As DAO.Relation
    Dim i As Integer
    Set db = CurrentDb
    For Each rel In db.Relations
        If rel.Table = strTabelle Then
            ReDim Preserve strVerknuepfteTabellen(i)
            ReDim Preserve strFremdschluesselfelder(i)
            strVerknuepfteTabellen(i) = rel.ForeignTable
            strFremdschluesselfelder(i) = _
                rel.Fields(0).ForeignName
            i = i + 1
        End If
    Next rel
    Set db = Nothing
End Function
```

Der Aufruf dieser Prozedur sieht beispielsweise so aus:

```
Public Sub Test_VerknuepfteTabellen()
    Dim strVerknuepfteTabellen() As String
    Dim strFremdschluesselfelder() As String
    Dim i As Integer
    VerknuepfteTabellen "tblKunden", _
        strVerknuepfteTabellen, strFremdschluesselfelder
    For i = LBound(strVerknuepfteTabellen) To _
            UBound(strVerknuepfteTabellen)
        Debug.Print strVerknuepfteTabellen(i), _
            strFremdschluesselfelder(i)
    Next i
End Sub
```

Hier deklarieren wir zwei Arrays für die Rückgabeparameter. Den Namen der zu untersuchenden Tabelle übergibt die Prozedur als ersten Parameter, die beiden Arrays danach.

Das Ergebnis durchläuft die Prozedur in einer **For...Next**-Schleife über die Elemente der Arrays und gibt diese im Direktfenster aus.

Primärschlüsselfelder ermitteln

Für die eine oder andere Anforderung benötigen Sie das Primärschlüsselfeld beziehungsweise die Primärschlüsselfelder einer Tabelle. Dieser Beitrag liefert zwei Funktionen, mit denen dies möglich ist.

Einzelnes Primärschlüsselfeld ermitteln

Gelegentlich wollen Sie das Primärschlüsselfeld einer Tabelle ermitteln. Dies erledigen Sie mit den Methoden der DAO-Bibliothek. Die folgende Funktion namens **GetSinglePrimaryKey** ermittelt einen einzelnen Primärschlüssel.

```
Public Function GetSinglePrimaryKey(strTable As String) As String
    Dim db As DAO.Database
    Dim tdf As DAO.TableDef
    Dim idx As DAO.Index
    Set db = CurrentDb
    Set tdf = db.TableDefs(strTable)
    For Each idx In tdf.Indexes
        If idx.Primary Then
            If idx.Fields.Count = 1 Then
                GetPrimaryKey = idx.Fields(0).Name
            End If
            Exit For
        End If
    Next idx
End Function
```

Listing 1: Ermitteln des Primärschlüssels einer Tabelle

Sie funktioniert nicht für Tabellen mit zusammengesetzten Primärschlüsseln. Die Funktion erwartet den Namen der zu untersuchenden Tabelle als Parameter und liefert den Namen des Primärschlüsselfeldes zurück.

Sie erstellt ein **TableDef**-Objekt auf Basis der übergebenen Tabelle und durchläuft alle Elemente der Indexes-Auflistung dieser Tabelle. Dabei prüft sie, ob die Eigenschaft **Primary** den Wert **True** hat. In diesem Fall handelt es sich um einen Primärschlüssel-Index. Wenn dieser Index nur ein Feld der Tabelle enthält, trägt die Funktion den Namen dieses Feldes als Rückgabewert in die Variable **GetPrimaryKey** ein (s. Listing 1).

Alle Primärschlüssel ermitteln

Manche Tabellen verwenden zusammengesetzte Primärschlüssel. Dies ist nicht besonders praktisch, da Sie beispielsweise bei einer Verknüpfung mit einer solchen Tabelle immer direkt zwei Fremdschlüsselfelder bereitstellen müssen (s. Bild 1).

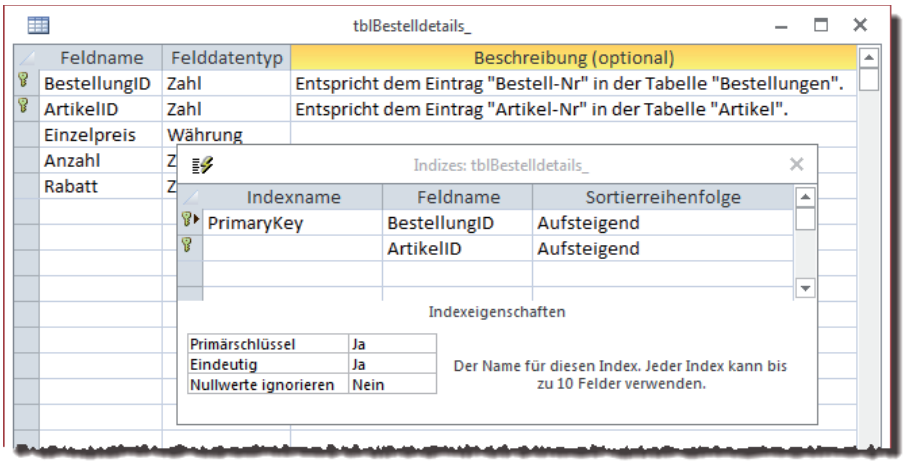

Bild 1: Beispiel für einen zusammengesetzten Primärschlüssel

Sinnvoller wäre es, ein zusätzliches Feld als Primärschlüsselfeld bereitzustellen und die bis eigentlich als zusammengesetzten Primärschlüssel vorgesehenen Felder als zusammengesetzten, eindeutigen Index zu definie-

Primärschlüssel in Verknüpfungstabelle nachrüsten

Einige Verknüpfungstabellen für die Herstellung von m:n-Beziehungen verwenden ihre beiden Fremdschlüsselfelder als zusammengesetzten Primärschlüssel. Das ist in manchen Fällen unpraktisch, zum Beispiel, wenn Sie mal die Datensätze dieser Tabelle referenzieren wollen – Sie müssten dann immer gleich zwei Fremdschlüsselfelder in der referenzierenden Tabelle angeben.

Also zeigen wir Ihnen, wie Sie einer solchen Tabelle ein Primärschlüsselfeld hinzufügen. Dies gelingt so:

- Tabelle im Entwurf öffnen
- Zusammengesetzten Primärschlüssel entfernen (s. Bild 1)
- Neues Feld als zukünftiges Primärschlüsselfeld hinzufügen, aber noch nicht als Primärschlüssel festlegen
- Für die beiden ehemaligen Primärschlüsselfelder einen zusammengesetzten, eindeutigen Index festlegen (s. Bild 2). Damit gewährleisten Sie, dass weiterhin keine Kombination der Werte für die beiden Fremdschlüsselfelder doppelt vorkommt.

Wenn Sie nun versuchen, das neue Feld als Primärschlüsselfeld zu definieren und die Tabelle speichern, erhalten Sie die Fehlermeldung **Null-Wert in Index oder Primärschüssel nicht möglich**.

Das war vorhersehbar: Ein Primärschlüsselfeld darf nur eindeutige Werte und Werte ungleich Null enthalten, aber unser Primärschlüsselfeld enthält ja noch gar keine Werte. Bild 3 zeigt, dass die Spalte **BestelldetailID** noch komplett leer ist – und somit denkbar ungeeignet für die Umwandlung in ein Primärschlüsselfeld.

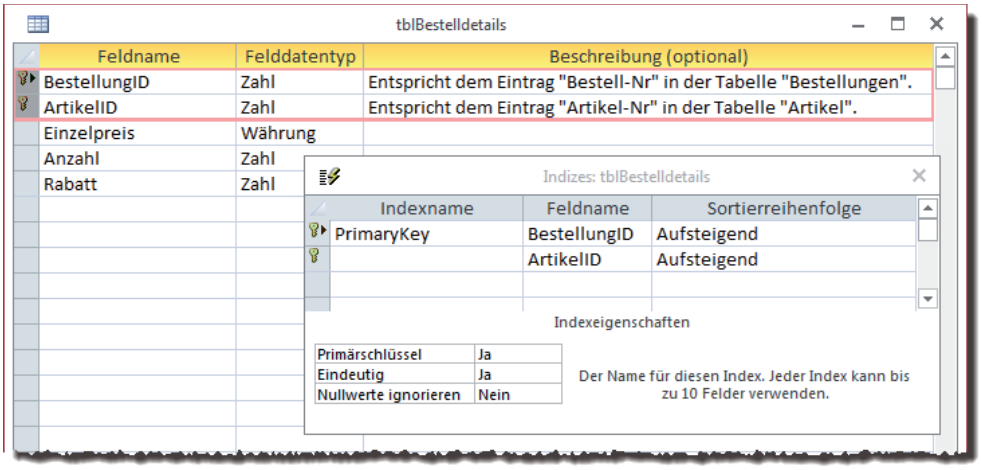

Bild 1: Tabelle mit zusammengesetztem Primärschlüssel

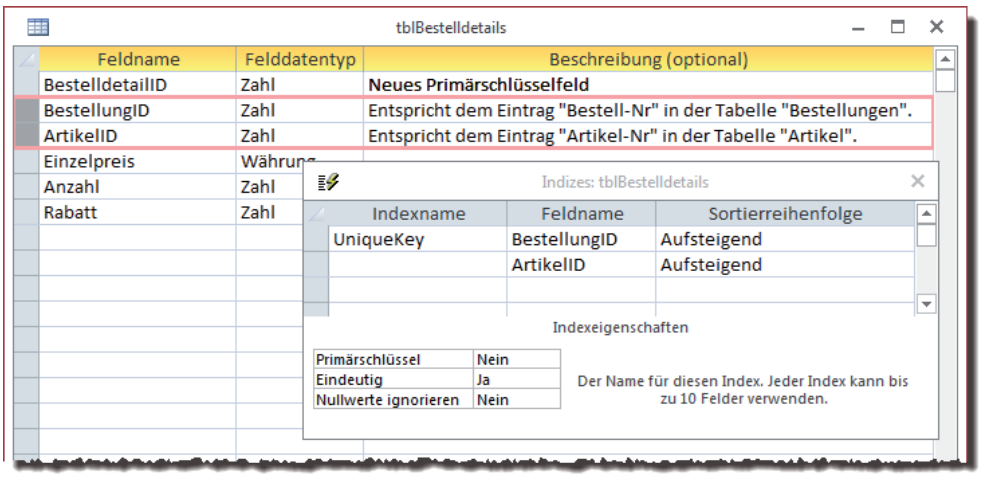

Bild 2: Ehemalige Primärschlüsselfelder als eindeutigen Index definieren